Received February 25, 2020, accepted March 9, 2020, date of publication March 17, 2020, date of current version April 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2981411

A Layer-Partitioning Approach for Faster Execution of Neural Network-Based Embedded Applications in Edge Networks

DARREN SAGUIL AND AKRAMUL AZIM, (Senior Member, IEEE)
Department of Electrical, Computer and Software Engineering, Ontario Tech University, Oshawa, ON L1G 0C5, Canada

Corresponding author: Akramul Azim (akramul.azim@uoit.ca)

This work was supported in part by the Ontario Tech University Grant and in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Grant.

ABSTRACT As embedded systems become more prominent in society, 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 entirely rely on networks that 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%.

INDEX TERMS Layer-partitioning, edge networks, neural network models.

I. 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 [11]. 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, forecasting [8] 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) [26]. 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 [14]. 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 [20], 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 edge networks or cloud computing to host their NN models. Edge networks are preferred to host this data over cloud computing due to their predictable networks latency, whereas cloud data centers are usually too far away to provide a reliable network latency for cyber-physical functionalities [5]. Edge networks, however, provide the same functionality to a lesser extent using nearby fog nodes, also known as cloudlets. Although this reduces the variance in network latency, there is still much to be improved in these systems.

Complete reliance on edge networks may expose embedded devices with NN functionality to new issues concerning

The associate editor coordinating the review of this manuscript and approving it for publication was Yongming Li.

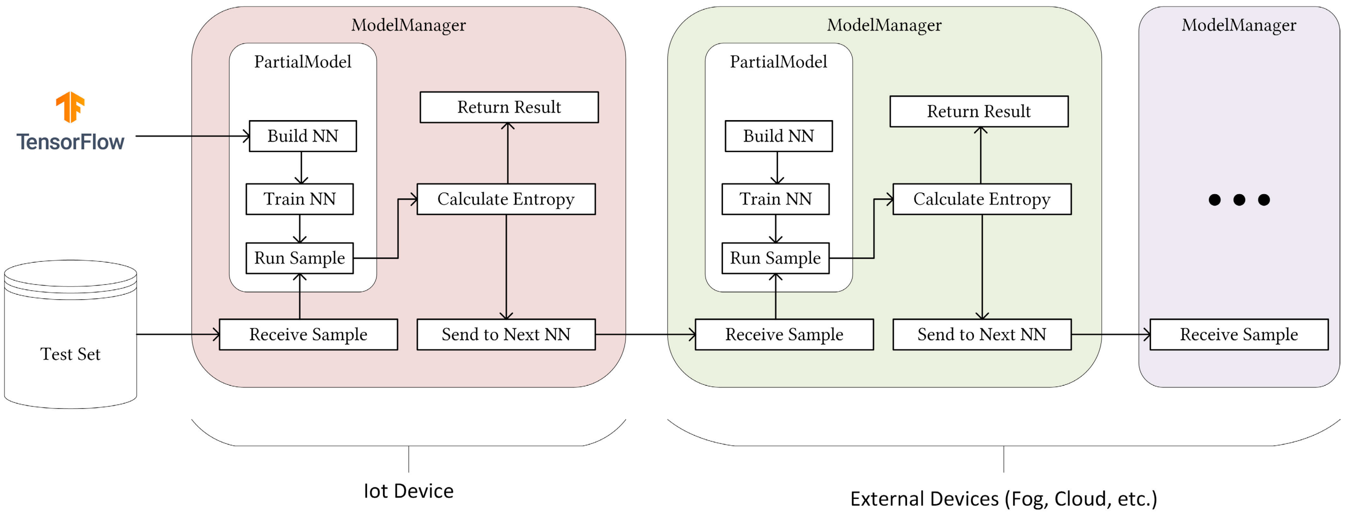

FIGURE 1. Model of the proposed solution. It consists of a NN model encapsulated inside of a manager which handles its inputs and outputs.

networking. First, without a reliable connection to wireless networks, or if their wired connection were to be severed, these devices would lose some or all of their functionality [25]. Even with a wireless connection, their communications may lose quality at unpredictable times, making the entire system unpredictable. Secondly, by frequently sending data over networks, these devices add themselves to the attack surface of the whole system. For example, the Hospira LifeCare PCA3 and PCA5 computerized infusion pumps were recalled due to a US Federal Drugs Administration report showing that these devices could be remotely accessed by potential attackers [2]. Lastly, data transmission over networks are one of the most energy-consuming tasks of embedded systems such that it might be more energy efficient to run NN's on the device itself [4].

Although embedded systems have increased in power over time, they may still not have the resources to run NN's independently. To consistently meet real-time deadlines, embedded devices may have to run smaller and simpler NN models, thus reducing the prediction accuracy of the model itself [36]. Also, the extra resources and development costs to run NN models may be used to fulfill other unrelated but more important requirements. As discussed in [6], a lower and upper bound on accuracy can be derived from different domains. Therefore, we propose to find possible improvements to the status quo.

These improvements include 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.

To find these possible improvements, a simulated IoT environment was prepared along with several NN models. We then executed each NN models using several datasets to measure the execution time of each layer on both an embedded system and a fog node. From this, we determined if the model should be offloaded to the fog node at all (i.e. the embedded system runs the model at a rate faster than the network latency added to the fog node execution time). After measuring the execution time of each layer on both devices, we then plotted splitting points in each of the models and implemented them in the following experiment phase.

In the following phase, we 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. By performing these experiments, the following contributions were made:

- Developed a framework that allowed users to create several neural networks and chain them together, thus creating a distributable NN model. This framework was built off of TensorFlow 2.0 and also provided entropy calculations and layer-by-layer timing analysis.
- Analyzed several datasets on several different NN models to determine which of their characteristics affected the runtime on embedded systems the most.
- Created several distributed NN models based on previous results to analyze their improvements over purely offloading all inputs to the fog node.

The rest of the paper is organized as follows. Section II describes the technology in which this experiment was based on. Section III describes the framework built to perform this experiment. Section IV explains the materials and workflow of the experiment. Section V explains the results of the experiment, and Section VII describes works related to this experiment, as well as their findings.

II. SYSTEM MODEL AND BACKGROUND

The system model presented in Figure 1 was created to implement the proposed solution. The system featured a class to create, train, and run samples through a NN model. It also featured secondary class to contain the NN model, feed it inputs, and send manage its outputs. These classes were called `PartialModel` and `ModelManager` respectively. When combined, they can create shallow NN's that can be chained together. Each of these smaller NN's can then be placed on devices on the IoT hierarchy.

A. INTERNET OF THINGS HIERARCHY

The hierarchy of IoT is displayed in Figure 2. It shows that end devices, such as cameras and printers, can be connected to networks that consist of fog and cloud computers through the use of embedded systems. The Harvard Business Bureau states that connection to external devices gives these objects new functionality, such as the ability to monitor their environment, be remotely controlled, automate their functions, and provide data for optimization [14]. However, these functions are better suited to fog nodes, as the distance between an end device to a cloud data-center may introduce too much latency for the "seamless" integration of objects into the network [23].

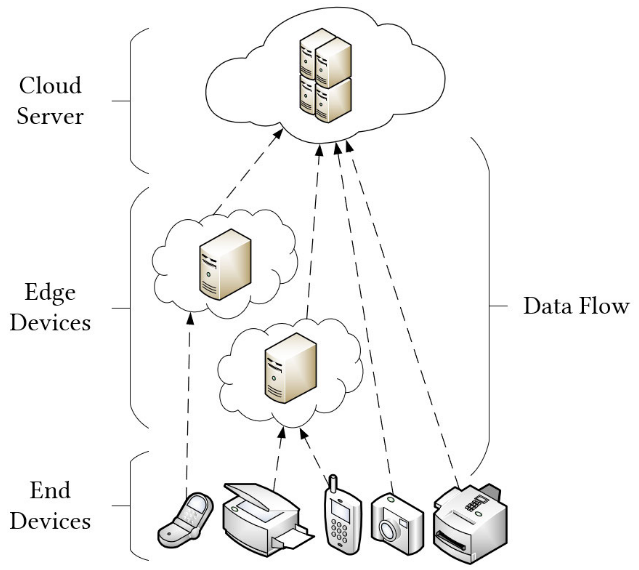

FIGURE 2. The hierarchy of an IoT network.

Fog nodes are a type of edge device that can provide the services of cloud servers, but at a "grounded" level [37]. They are placed within closer proximity to end devices, such that they can serve them with lower latency for real-time applications, such as healthcare.

B. NEURAL NETWORKS

NN's are a subset of classification algorithms that are capable of providing classifications from a stream of data from IoT networks. These classification algorithms are comprised of an input layer and an output layer, as well as numerous hidden layers in between them. The function of these layers can vary from matrix multiplication layers to matrix reshaping layers (e.g., flattening a matrix into a single-dimension array). Every layer consists of neurons that hold the data being analyzed as the layers perform their function on them. In a Feed-Forward NN, the data inside each neuron is the output from the previous layer; the data is passed forward through all the layers until the output where a classification is made.

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* [36]. 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* [21], a machine learning model used for image processing and classification.

C. MULTILAYER NETWORK USABILITY ANALYSIS

In modern ages, artificial intelligence is transforming different real-time application domains such as the gaming industry. To enhance the realism and excitement in the virtual reality (VR) games, it uses multilayer deep learning approaches. The multilayer learning approaches teach the game agent to behave more intelligently through machine-human interaction in real-time. For example, convolution neural and LSTM (Long Short-Term Memory) networks are used to detect the objects and movements accurately [38]. In addition, the wireless sensor networks (WSN) or IoT networks offer a promising computational platform for parallel and distributed multilayer perceptron (MLP) neural networks [33].

To receive high performance in complex applications, deep neural networks use efficient algorithms that ensures high accuracy in the output. The deep neural networks are suitable to model with nonlinear data with a large number of inputs. They split the tasks of the application into a layered network and distribute the tasks as simple elements to multiple layers. Their main advantages are that they are easy to use, and they can approximate any input/output map [29]. The more you use the training data the more accurate result can be achieved. However, meeting the high accuracy is not always the best measure for assessing the training model. We need to consider the recall and precision values which show the percentage of total relevant results. Moreover, extreme training in the multilayer approach requires a lot of effort as well.

In multilayer neural network approach, the network is hard to train and it requires to tune a lot of parameters like the number of hidden neurons, the number of hidden layers, error function, learning rate, initial weight, window size, weight decay, weight updates, and input normalization [24]. The principal disadvantages are that they train slowly, and require lots of training data. In [29], the authors claim that it needs three times more training samples than network weights. It is computationally very expensive and time-consuming to train the model. As a result, offloading a multilayer neural network model to an edge network can reduce the computation load and improve response time significantly.

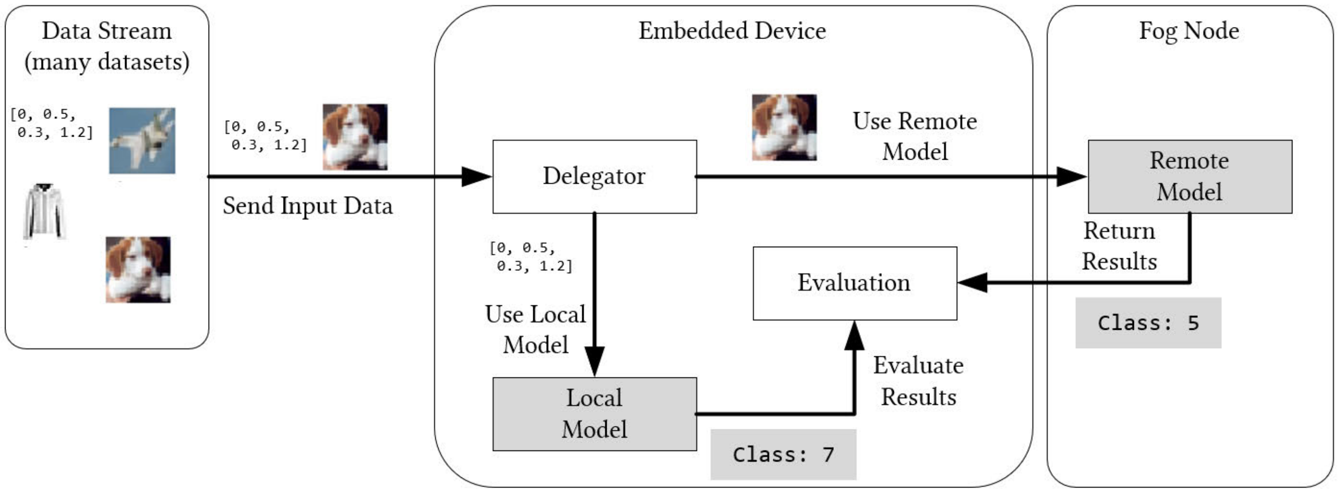

FIGURE 3. **The setup of the task distribution for the Delegation Phase.**

III. PROPOSED FRAMEWORK

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 that can be affected when attempting to reduce the communication costs, such as the size of data transmissions and the shape of the input samples. However, while lowering the communication cost was the main goal, it was still imperative that the prediction accuracy of the status quo (i.e. hosting all machine learning tasks on external hardware) must remain unaffected.

The contribution in this paper is that we have 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. More specifically, the following main contributions and novelty have been achieved:

- 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.

To reach this, we developed an NN framework that can create machine learning models and distribute or split it amongst devices. Then, the framework was used to measure the accuracy and performance of the models in two different phases: The Delegation Phase and the Splitting Phase.

A. FRAMEWORK 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.

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

B. FRAMEWORK ARCHITECTURE

1) DELEGATION PHASE FRAMEWORK ARCHITECTURE

This phase of the research examines the status quo and any possible improvements without making significant changes to the models. This phase, depicted in Figure 3, consists of a data stream sending samples to an Embedded Device. Instead

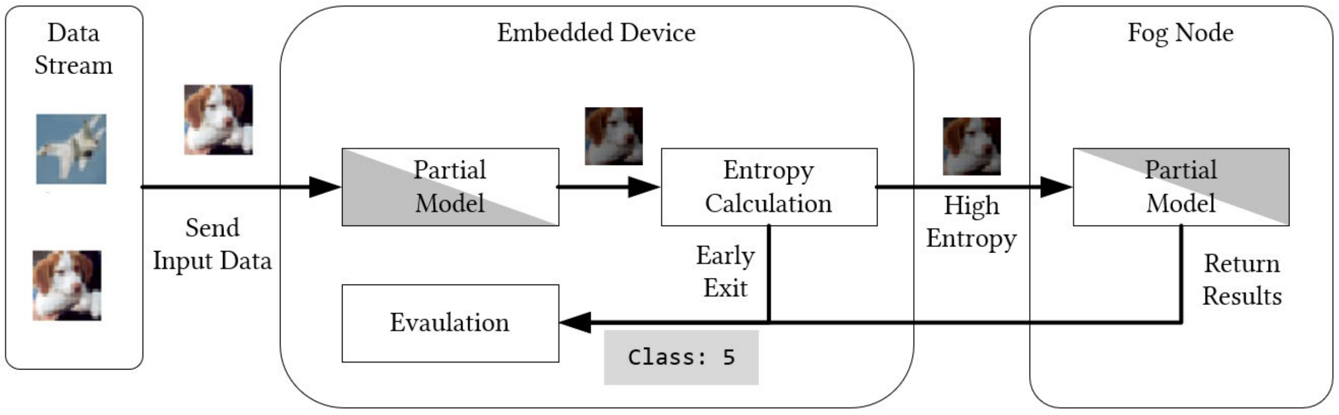

FIGURE 4. **Setup of the task distribution for the Splitting Phase.**

of sending the sample data directly to the fog node, it is instead sent first to a delegator. As both the Embedded Device and the Fog Node both contain a copy of the same model, the delegator 108 can choose that model to send the input sample to.

Before the delegator was implemented, a validation set was passed through the models in two different ways: only performing local inferences, only using remote inferences. This was to compare the performance of embedded systems to the status quo. After the validation set was processed, the following measurements were made: the WCET for local inference ($WCET_D$), the upper-bound of network latency (T_L), and the WCET for inference on the fog node (T_F).

Using these measurements, a WCET threshold was created; any input that could have made the local model run longer than this threshold should have their machine learning tasks hosted on the remote device. The equation for this threshold can be seen in Equation 1.

$$WCET_T = 2 * T_L + T_F \tag{1}$$

A simple algorithm using this threshold was implemented as the delegator to show one of its possible uses. For example, in a resource-constrained device, such as a mobile phone or a security camera with a microphone, may rely on more than one machine learning model to process its data. The algorithm shown in Algorithm III-B.1 was implemented to delegate the input samples for these devices. The delegator sends the samples to their respective models, and decide where their inference should have taken place.

2) SPLITTING PHASE FRAMEWORK ARCHITECTURE

The splitting phase took a different approach; it took multiple models and chained them together as if they were a single model. Another key difference in this phase is that the local machine is always guaranteed to execute its local model; the only decision the delegator made was to determine if the result is sufficient enough for the system. Every sub-model within the model had an exit layer where they can finish the inference in the case of a confident result. Each of these

Algorithm 1 WCET-Based Delegation Algorithm

```
0: procedure CheckOffloadStatus($D$)
0:     for every $d \in D$ do
0:         if $WCET_d \geq WCET_T$ then
0:             $sendToFogNode(d)$
0:         else
0:             $queueToLocalProcessors(d)$
0:         end if
0:     end for
0: end procedure=0
```

sub-models were then distributed amongst different devices. The full implementation is shown in Figure 4.

C. LAYER-BY-LAYER CONSTRUCTION

To complete this phase, 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 the ability to customize their options. The 6 layers implemented were: Convolutional, Dense, Pooling, Activation, Flatten, and Dropout.

1) CONVOLUTIONAL

These layers use a filter with a learned kernel that convolves around 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 [21]. However, convolutional layers have high complexity; their complexity can be seen in Equation 2 [17], 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.

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

When an image is passed through a convolutional layer, it will be broken down into more images based on how many times the filter was used. Each of these new images will have a lower resolution, but will also be focused on a smaller part of the image, allowing for finer analysis on the image.

2) 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 multiplying complexity and large number of neurons. This complexity is seen in Equation 3, where n is the length of the input data and m is the length of the output data.

$$O(nm) \tag{3}$$

The dense layers implemented in this framework only accepted flattened data (i.e. data with only a single dimension in shape). When an array of length n is passed through the dense layer, it will be multiplied by the kernel, which is a matrix with a length of m and height of n. It may also add a bias of length m, thus resulting in an output of length m. The values inside kernel and bias are both learned during training.

3) POOLING

The above layers may produce too many features that 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. This could potentially reduce the execution time of data transmission as well by reducing the amount of data needed to be sent. They can either have max or average pooling functions.

4) ACTIVATION

Activation layers provide many different non-linear functions, such as rectified-linear (relu), sigmoid, softmax, and argmax. These functions run over all the individual neurons in the previous layer, altering the data but retaining the shape of the input. For example, a softmax function can be used to determine the output of a DNN by normalizing the array. It will make the sum of the array equal to one, making each index of the array hold a probability that the data is of the class at that index (e.g. a value of 0.5 at index 3 indicates that there is a 50% chance that the input was class 4(.

5) FLATTEN

Reduces the shape of the input layer to a single dimension, though the data itself remains the same. This is used since the exit layers implemented in this framework only accept inputs of a single dimension.

6) DROPOUT

Randomly makes neurons ignored during training. This is to prevent overfitting a model (i.e. training a model to only classify data from a specific set, rather than new sets it has not seen before).

D. CHAINING AND LOSS AGGREGATION

After the models were created, they were placed within a ModelManager class that 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. 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. In this equation, L'_k is the loss to be used by the 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.

$$L'_k = \sum_{l=1}^{k} w_l L_l \tag{4}$$

Every model also uses this same loss to train the variables in its exit layer. This means every model has the same exit layer shape, but different values inside of the exit layer. 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 (i.e. the 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 2 Multiple Model Training Algorithm

```
0: procedure ModelTraining($data, label$)
0:     $x, prediction \leftarrow model_0(data)$
0:     $loss \leftarrow 0$
0:     $i \leftarrow 1$
0:     while $i < length(model)$ do
0:         $x, prediction \leftarrow model_i(x)$
0:         $loss \leftarrow loss + calculateLoss(prediction, label)$
0:         $optimize(model_i, loss)$
0:         $i \leftarrow i + 1$
0:     end while
0: end procedure=0
```

E. 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 5 [36].

$$\eta(x) = -\sum_{i=1}^{|C|} \frac{x_1 \log x_i}{\log |C|} \tag{5}$$

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 that offers the best accuracy can be chosen as the threshold.

F. 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 and the output is passed to the output layer. Otherwise, the output is transferred to the next model. This threshold algorithm, along with the chaining algorithm can be seen in Algorithm 3.

Algorithm 3 Confidence Testing Algorithm

```
0: procedure EntropyCheck($data$)
0:     $x, prediction \leftarrow model_0(data)$
0:     $i \leftarrow 1$
0:     while $entropy(y) < T$ & $i < length(model)$ do
0:         $x, prediction \leftarrow model_i(x)$
0:         $i \leftarrow i + 1$
0:     end while
0: end procedure=0
```

G. 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 training computations on the fog node. The weights of the trained models were then distributed to the embedded device.

Finding the splitting points in each of the models is done by finding the areas with the largest differences in consecutive layer runtimes. Next, the layer that the split follows is manually chosen by their function; pooling and dropout layers are preferred as the splitting point due to their low data output and quick execution on embedded devices. If there are multiple possible splitting points, then the earlier one was chosen. Flatten layers are ignored when finding the splitting points as they are automatically done in early exit layers if needed.

IV. EXPERIMENTS

Using the framework and workflows described in Section III, we devised the following experiment using several datasets and machine learning models. This experiment was run on the following pieces of hardware: A raspberry Pi 3 Model B as the embedded system, and a laptop with a 7th Generation Intel® Core i5 Processor acting as the fog node. It had 2 cores that supported 4 threads each. The experiments were both carried out over their respective CPUs. The framework was implemented using Python 3.7.4 and TensorFlow 2.0. Networking was done using Python's built-in socket class. The data analysis frameworks Numpy, Pandas, Scipy/SKlearn, and MatPlotLib were also used in the handling of large amounts of data.

A. DATASETS USED

Four datasets were used in this experiment; each set provided ample data to create testing and training sets. Every dataset used varied in content and dimensionality, ranging from 1D arrays of data to full-colour 2D images. These datasets are the Wall-Following Robot Navigation dataset [16], Fashion-MNIST dataset [40], and the CIFAR-10 dataset [22].

1) WALL-FOLLOWING ROBOT NAVIGATION DATASET

This dataset which was provided by Fireire *et al.* was used as a case study for this experiment. It consisted of inputs from ultrasound sensors placed on a self-moving robot and outputs which consisted of which of four directions the robot should turn (i.e. Slight-Right-Turn, Sharp-Right-Turn, Move-Forward, and Slight-Left-Turn). An important aspect of this dataset was that it provided 3 alternate datasets with the same outputs, but a differing number of inputs. These alternate datasets were 2-inputs (front and left sensors), 4 inputs (front, back, left, and right sensors), and 24-inputs (all sensors placed on the robot). These three datasets were used to create 3 sets of training and testing data to determine the impact of dimensionality on the runtime and accuracy of the models.

2) FASHION-MNIST DATASET

The MNIST dataset was used to benchmark NN's in many ML related experiments. It provides 28 × 28 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 [40], 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, the classes provide much finer differences to find;

FIGURE 5. Example Images from the Fashion-MNIST data set).

the classes pullover (2), coat (4), and shirt (6) seem visually similar but can be separated using ML models.

3) CIFAR-10 DATASET

The CIFAR-10 dataset was the last dataset to be analyzed in this experiment. This dataset provided inputs of 32 × 32 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 6 where every image is visually distinct, even those where the images are in the same class.

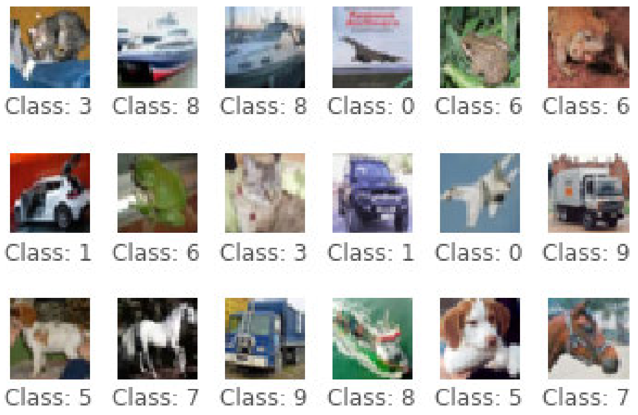

FIGURE 6. Example Images from the CIFAR-10 data set.

B. MODELS USED

The models used in this 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 7.

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 1.

TABLE 1. Prediction accuracy for every dataset on every model they were executed on.

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%

C. EXPERIMENTAL STRUCTURE

The experiment was ran in the delegation phase and the splitting phase as mentioned in Section III using a testing set from each dataset. During the delegation phase, 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 1. 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, 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.

V. RESULTS AND ANALYSIS

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

A. DELEGATION PHASE

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 2 shows the total runtimes of every model with every dataset.

For the first experiment, all the datasets were run on Model 0. It was determined that this model should not be

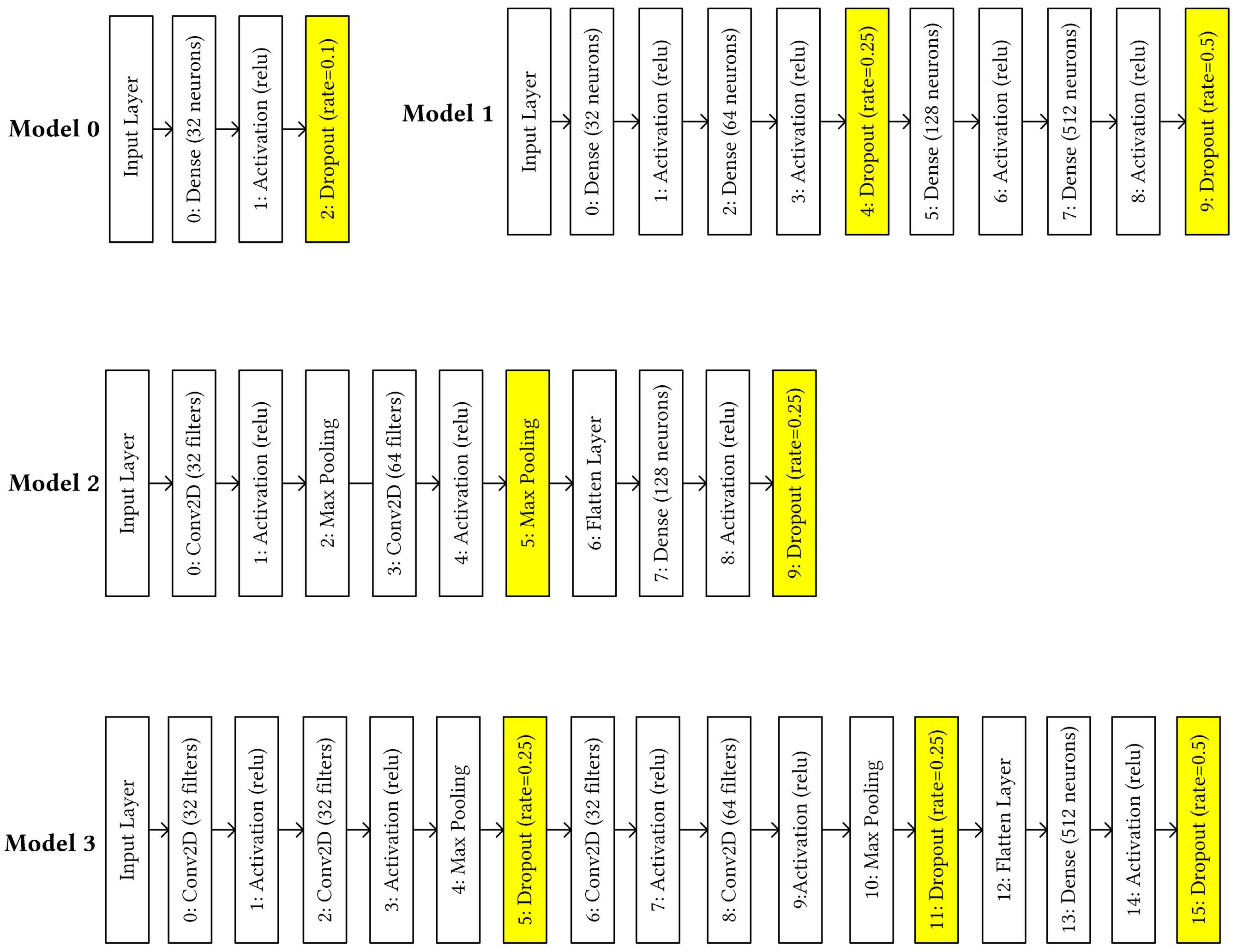

FIGURE 7. Structure of the NN models used in this experiment. The highlighted layers indicate that they have an early exit.

TABLE 2. Total execution times of each model with every dataset.

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

split since the runtime of each dataset on the model did not exceed the threshold provided by Equation 1. 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 8. 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 2. This means that completely offloading or splitting this model is a viable option when improving the total runtime. To find the splitting point, we look at Figure 9, 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 2 shows that the embedded system runtime surpasses the WCET threshold for all datasets. This is because of a major bottleneck in the model. Figure 10 shows that the major bottleneck for this model is layer 7, a Dense layer with 128 neurons. Since layer 6 is a flatten

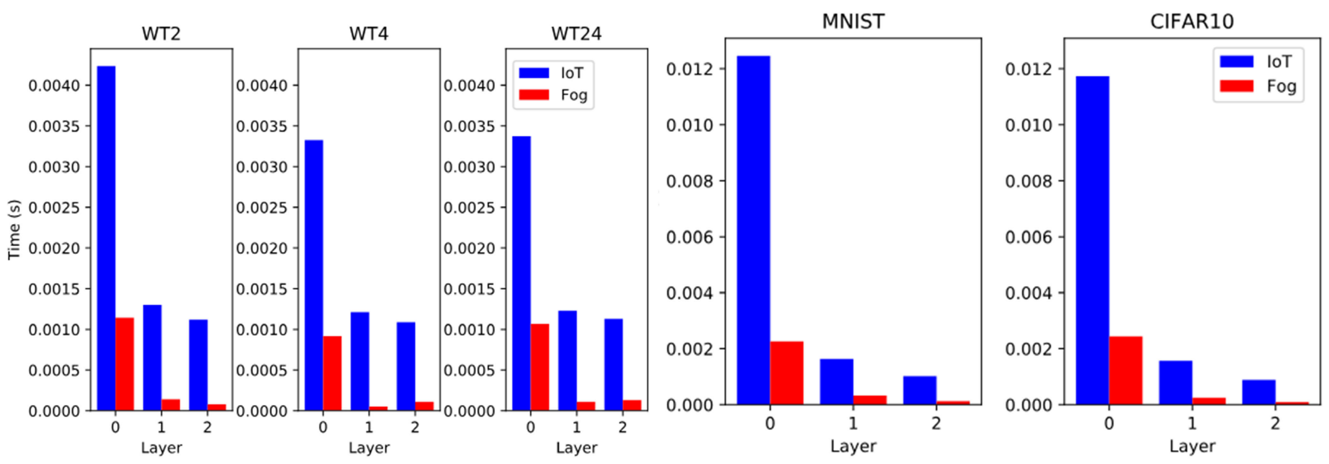

FIGURE 8. Layer time results for all of the datasets on Model 0.

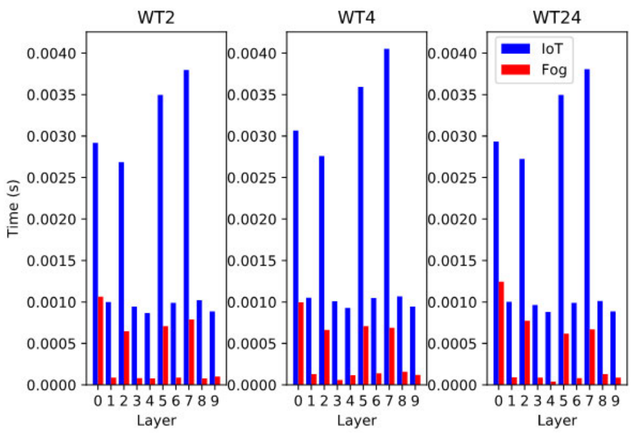

FIGURE 9. Layer time results of the Wall-Turn datasets on Model 1.

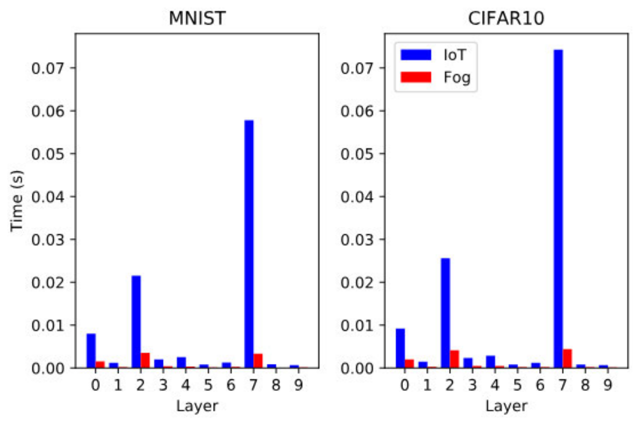

FIGURE 10. Layer time results of the image datasets on Model 2.

layer, the splitting point was chosen to be layer 5 instead. Another key observation can be made from Figure 10; 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 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. Figure 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.

B. 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 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 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.

When increasing the entropy threshold of the model, we decrease the number of messages being sent between devices, thus saving time and energy. Figure 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,

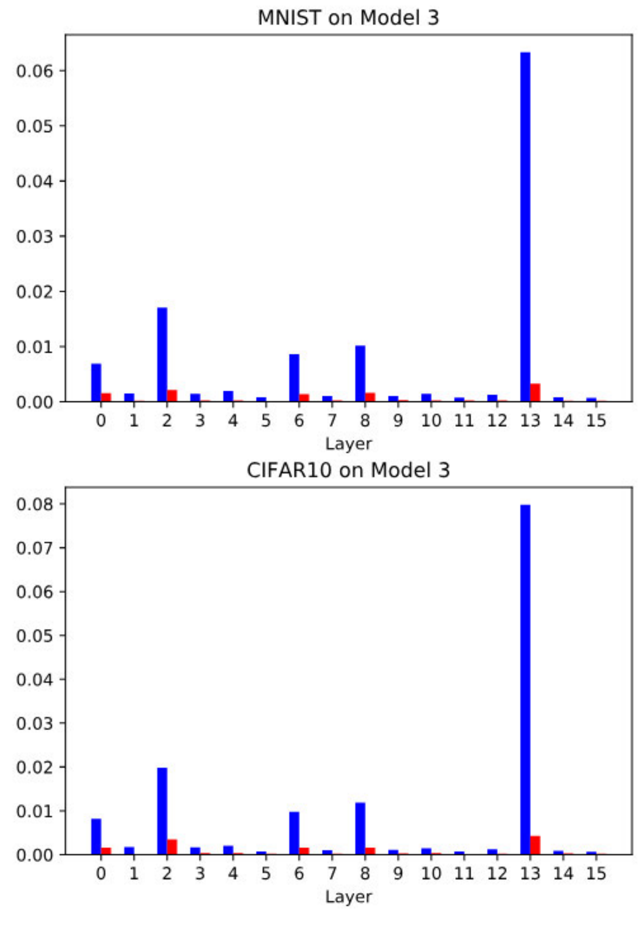

FIGURE 11. Layer time results of the image datasets on Model 3.

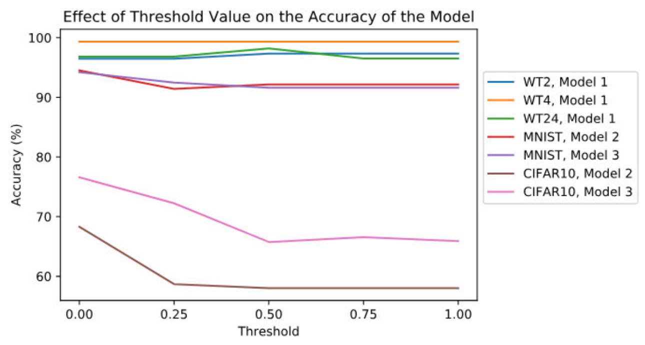

FIGURE 12. Effect of changing the entropy threshold on the accuracy of the split model.

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.

VI. DISCUSSION

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,

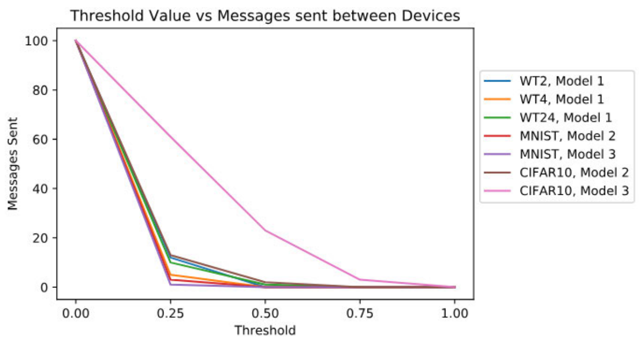

FIGURE 13. Effects of increasing the entropy threshold on the messages sent between devices.

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.

Many applications may have different accuracy requirements, for which we need to reconfigure and build the neural network models separately. To achieve a particular accuracy, it may not be necessary to train the multilayer model with full phases. Therefore, the proposed approach suggests making an early exit when any application meets its certain accuracy. A lower bound and upper bound on accuracy requirement can be derived for an application under different performance requirements, which we could consider a potential future work in this area of research. Many soft real-time applications (e.g., virtual reality games) can be trained up to a certain level instead of training the model for a longer time. For example. virtual reality (VR) games demand to generate the 3D images in real-time based on the training data. This becomes quite challenging as the VR applications require a faster response at the same time of training a model based on new data. In such a scenario, our proposed layer partitioning approach can be easily applicable because of providing a faster response time over a slightly reduced accuracy. The tolerable accuracy bound can be derived by analyzing the acceptable error rate under different situations.

VII. RELATED WORKS

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 [3], 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 [15], also known as cloudlets [31]. Although this reduces the variance in network latency, there is still much to be improved in these systems.

In this section, we describe possible improvements to externally hosting machine learning models in edge networks. The possible improvements tackled areas such as the method of transfer between devices, the format in which data is transferred between devices, and the distribution of machine learning models between devices. However, we must first look at the feasibility of running machine learning models locally on embedded systems.

One such study of locally hosting ML models was performed by Zidek *et al.* in [41]. 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 experiments showed that training a fault-detecting vision system is 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. A key observation from this study was that numerous machine learning models can already be deployed on an embedded device and run with an acceptable accuracy. However, this is only for a small dataset; analysis of larger and more diverse datasets will need more complex algorithms such as a DNN which may need external machines to host it.

The transfer of data between devices is one of the most time and energy-consuming processes in machine learning in an edge network [4]. Therefore, it is important to look at ways to improve the efficiency or minimize the frequency of data transmissions. The research done by Azar *et al.* in [4] shows that the former is possible by applying lossy compression to the data being sent. Although some data is lost in the compression algorithm, the overall accuracy of the model was, on average, unaffected but the battery life of the device could have been extended by 27%. Another possible improvement was examined by Jeong *et al.* in [20]. 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 also made another important contribution: they implemented partial inferences to the execution.

Partial inferences are a novel development in DNN's. It 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 [21], 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. This means that, by using a partial inference, data transmissions can be performed faster. Partial inferences can also be used to secure privacy; without a copy of the model, any data intercepted will be difficult for any attackers to read [20].

Partial inferences also allows for another novel advancement: early exits. Early exits allow DNN's to make a prediction at an earlier layer. 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, where the later layers are executed on higher devices on the hierarchy. This concept is labeled as the Distributed Deep Neural Network, or DDNN [36]. 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 [25].

In [34], the authors investigate different techniques to make an early prediction of the accuracy of machine learning applications considering the initial error rates of sample training data. Their proposed method trains an ensemble of heterogeneous classifiers to estimate an upper bound of accuracy accurately for future training data in the target domain. Similarly, the analysis of [19], [32] and [35] show an improvement in the prediction of cloud-enhanced decision making to ensure maintenance, productivity, anomaly detection, monitoring [27] and product quality. In a source domain, a classifier may perform well with the same distribution of training and test data, but it may not perform well in the target domain with a different distribution. Ben-David *et al.* [6] propose a cross-domain transfer where the target domain has a different distribution of training data. They investigate the target and source error to understand the divergence between the two domains.

Regardless of machine learning-based applications, we have found different works that propose different architectures [30] on distributing tasks for faster execution. One of them is the Lambda architecture that consists of three layers: batch, speed, and serving layers. Although the Lambda Architecture [12] provides a general-purpose approach for faster response with low latency, it has some complexity in implementing different layers. The support and maintenance for batch and speed layers are quite hard in complex distributed systems. In another study [1], the authors propose a mechanism to offload machine learning-based non-critical tasks in the cloud to avoid overloads. The survey papers [18], [39] on mobile edge networks demonstrate why network functions and contents are computed on edge networks.

In another study [28], the authors show a comparison between single and multiple hidden layer network. The experimental results show that the single hidden layer network provides faster convergence to approximate linear and quadratic functions. Moreover, the multilayer network training brings different issues like slow convergence, time-consuming, and local minima [9]. To get an improved result, we require to reconfigure the model data, network weights, the sensitivity of multilayer perceptrons [10]. However, to visualize and analyze the total representation of multilayer networks, Manlio De Domenico *et al.* [13] present an open-source software called muxViz that provides the insight of the networks including the algorithms. In addition, a new multilayer graph edge bundling technique [7] is proposed to visualize the multilayer networks as a graph format.

VIII. CONCLUSION

As NN's become more commonplace, it is essential that when they are used in conjunction with IoT, they run at a quick, reliable, and predictable speed. Unfortunately, there are many issues that 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.

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.

ACKNOWLEDGMENT

This work was supported in part by the Ontario Tech University Grant and in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Grant. The authors would like to thank Md. Al Maruf for assisting them to improve the quality of the article.

REFERENCES

[1] M. A. Maruf and A. Azim, "Extending resources for avoiding overloads of mixed-criticality tasks in cyber-physical systems," *IET Cyber-Phys. Syst., Theory Appl.*, vol. 5, no. 1, pp. 60–70, Mar. 2020.

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

[3] N. Auluck, A. Azim, and K. Fizza, "Improving the schedulability of real-time tasks using fog computing," *IEEE Trans. Services Comput.*, to be published.

[4] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, "An energy efficient IoT data compression approach for edge machine learning," *Future Gener. Comput. Syst.*, vol. 96, pp. 168–175, Jul. 2019.

[5] D. L. Beaty, D. Quirk, and J. Jaworski, "Fog computing," *Ashrae J.*, vol. 60, no. 1, pp. 68–74, Jan. 2018.

[6] S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan, "A theory of learning from different domains," *Mach. Learn.*, vol. 79, nos. 1–2, pp. 151–175, May 2010.

[7] R. Bourqui, D. Ienco, A. Sallaberry, and P. Poncelet, "Multilayer graph edge bundling," in *Proc. IEEE Pacific Visualizat. Symp. (PacificVis)*, Apr. 2016, pp. 184–188.

[8] B. Cannas, A. Fanni, L. See, and G. Sias, "Data preprocessing for river flow forecasting using neural networks: Wavelet transforms and data partitioning," *Phys. Chem. Earth, A/B/C*, vol. 31, no. 18, pp. 1164–1171, Jan. 2006.

[9] W. Cao, X. Wang, Z. Ming, and J. Gao, "A review on neural networks with random weights," *Neurocomputing*, vol. 275, pp. 278–287, Jan. 2018.

[10] J. Y. Choi and C.-H. Choi, "Sensitivity analysis of multilayer perceptron with differentiable activation functions," *IEEE Trans. Neural Netw.*, vol. 3, no. 1, pp. 101–107, Jan. 1992.

[11] G. Crespo-Perez and A. Ojeda-Castro, "Convergence of cloud computing, Internet of Things, and machine learning: The future of decision support systems," *Int. J. Sci. Technol. Res.*, vol. 6, no. 7, pp. 131–136, 2017.

[12] N. Marz and J. W. Henning, *Big Data, Principles and Best Practices of Scalable Real-Time Data Systems*. New York, NY, USA: Manning Publications, 2014.

[13] M. De Domenico, M. A. Porter, and A. Arenas, ''MuxViz: A tool for multilayer analysis and visualization of networks,'' *J. Complex Netw.*, vol. 3, no. 2, pp. 159–176, Jun. 2015.
[14] S. Earley, ''Analytics, machine learning, and the Internet of Things,'' *IT Prof.*, vol. 17, no. 1, pp. 10–13, Jan. 2015.
[15] K. Fizza, N. Auluck, A. Azim, M. A. Maruf, and A. Singh, ''Faster OTA updates in smart vehicles using fog computing,'' in *Proc. 12th IEEE/ACM Int. Conf. Utility Cloud Comput. Companion-UCC Companion*, Dec. 2019, pp. 59–64.
[16] A. L. Freire, G. A. Barreto, M. Veloso, and A. T. Varela, ''Short-term memory mechanisms in neural network learning of robot navigation tasks: A case study,'' in *Proc. 6th Latin Amer. Robot. Symp. (LARS)*, Oct. 2009, pp. 1–6.
[17] K. He and J. Sun, ''Convolutional neural networks at constrained time cost,'' 2014, *arXiv:1412.1710*. [Online]. Available: http://arxiv.org/abs/1412.1710
[18] X. Huang, R. Yu, J. Kang, and Y. Zhang, ''Distributed reputation management for secure and efficient vehicular edge computing and networks,'' *IEEE Access*, vol. 5, pp. 25408–25420, 2017.
[19] P. Jahnke, *Machine Learning Approaches for Failure Type Detection and Predictive Maintenance*, Darmstadt, Germany: Technische Univ. Darmstadt, 2015.
[20] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, ''Computation offloading for machine learning Web apps in the edge server environment,'' in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 1492–1499.
[21] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, ''Neurosurgeon: Collaborative intelligence between the cloud and mobile edge,'' in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA, 2017, pp. 615–629.
[22] University of Toronto, Toronto, ON, Canada. *Dataset*. Accessed: Mar. 25, 2020. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html
[23] P. Kulkarni and T. Farnham, ''Smart city wireless connectivity considerations and cost analysis: Lessons learnt from smart water case studies,'' *IEEE Access*, vol. 4, pp. 660–672, 2016.
[24] X. L. Dencelin, and T. Ramkumar, ''Analysis of multilayer perceptron machine learning approach in classifying protein secondary structures,'' *Biomed. Res.*, 2016.
[25] S. Leroux, S. Bohez, E. De Coninck, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, ''The cascading neural network: Building the Internet of smart things,'' *Knowl. Inf. Syst.*, vol. 52, no. 3, pp. 791–814, Sep. 2017.
[26] D. Maevsky, A. Bojko, E. Maevskaya, O. Vinakov, and L. Shapa, ''Internet of Things: Hierarhy of smart systems,'' in *Proc. 9th IEEE Int. Conf. Intell. Data Acquisition Adv. Comput. Syst., Technol. Appl. (IDAACS)*, vol. 2, Sep. 2017, pp. 821–827.
[27] M. A. Maruf and A. Azim, ''Software-based monitoring for calibration of measurement units in real-time systems,'' in *Proc. 44th Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Oct. 2018, pp. 2941–2946.
[28] T. Nakama, ''Comparisons of single-and multiple-hidden-layer neural networks,'' in *Proc. Int. Symp. Neural Netw.* Berlin, Germany: Springer, 2011, pp. 270–279.
[29] G. Panchal, A. Ganatra, Y. Kosta, and D. Panchal, ''Behaviour analysis of multilayer perceptrons with multiple hidden neurons and hidden layers,'' *Int. J. Comput. Theory Eng.*, vol. 3, no. 2, pp. 332–337, 2011.
[30] I. Rebai, Y. BenAyed, and W. Mahdi, ''Deep multilayer multiple kernel learning,'' *Neural Comput. Appl.*, vol. 27, no. 8, pp. 2305–2314, Nov. 2016.
[31] D. Saguil and A. Azim, ''Time-efficient offloading for machine learning tasks between embedded systems and fog nodes,'' in *Proc. IEEE 22nd Int. Symp. Real-Time Distrib. Comput. (ISORC)*, May 2019, pp. 79–82.
[32] B. Schmidt and L. Wang, ''Cloud-enhanced predictive maintenance,'' *Int. J. Adv. Manuf. Technol.*, vol. 99, nos. 1–4, pp. 5–13, Oct. 2018.
[33] G. Serpen and Z. Gao, ''Complexity analysis of multilayer perceptron neural network embedded into a wireless sensor network,'' *Procedia Comput. Sci.*, vol. 36, pp. 192–197, Jan. 2014.
[34] J. E. Smith, P. Caleb-Solly, M. A. Tahir, D. Sannen, and H. Van-Brussel, ''Making early predictions of the accuracy of machine learning applications,'' 2012, *arXiv:1212.1100*. [Online]. Available: http://arxiv.org/abs/1212.1100
[35] G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi, ''Machine learning for predictive maintenance: A multiple classifier approach,'' *IEEE Trans Ind. Informat.*, vol. 11, no. 3, pp. 812–820, Jun. 2015.
[36] S. Teerapittayanon, B. McDanel, and H. T. Kung, ''Distributed deep neural networks over the cloud, the edge and end devices,'' in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 328–339.
[37] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, and L. T. Yang, ''Data mining for Internet of Things: A survey,'' *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 77–97, 1st Quart., 2014.
[38] A. Varangaonkar. (Mar. 28, 2018). *Deep Learning in Games—Neural Networks Set to Design Virtual Worlds*. Accessed: Feb. 19, 2020. [Online]. Available: https://hub.packtpub.com/deeplearning-games-neural-networks-design-virtual-world/
[39] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, ''A survey on mobile edge networks: Convergence of computing, caching and communications,'' *IEEE Access*, vol. 5, pp. 6757–6779, 2017.
[40] Zalando Research, Berlin, Germany. (2017). *Dataset*. Accessed: Mar. 25, 2020. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist
[41] K. Židek, A. Hošovský, and J. Dubják, ''Diagnostics of surface errors by embedded vision system and its classification by machine learning algorithms,'' *Key Eng. Mater.*, vol. 669, pp. 459–466, Oct. 2015.

DARREN SAGUIL received the M.A.Sc. degree in applied sciences from the Department of Electrical, Computer and Software Engineering, Ontario Tech University under the supervision of Dr. Azim. He is a member of the Real-Time Embedded Software (RTEMSOFT) Research Group, Ontario Tech University, Oshawa. His main research focuses on real-time embedded systems, cloud computing, the Internet of Things, and machine learning.

AKRAMUL AZIM (Senior Member, IEEE) is an Assistant Professor with the Department of Electrical, Computer and Software Engineering, and the Head of the Real-Time Embedded Software (RTEMSOFT) Research Group, Ontario Tech University, Oshawa, ON, Canada. His research interests include real-time systems, embedded software, software verification and validation, safety-critical software, and intelligent transportation systems. He is a Professional Engineer of Ontario.