NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SHARED LEARNING AMONG DISTRIBUTED EDGE DEVICES USING CORAL EDGE TPU MACHINE LEARNING ENGINES

by

Erik Dubois

June 2021

Thesis Advisor: Marko Orescanin
Second Reader: Gurminder Singh

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE SHARED LEARNING AMONG DISTRIBUTED EDGE DEVICES USING CORAL EDGE TPU MACHINE LEARNING ENGINES	**5. FUNDING NUMBERS**
6. AUTHOR(S) Erik Dubois	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Internet of Things (IoT) edge devices have small amounts of memory and limited computational power. These resource-constrained devices consist of sensors that generate large amounts of data, making IoT edge devices attractive targets for machine learning models. To take advantage of machine learning models normally requires the data to be transported to a remote device with enough computational power to process these data. The transport of data to a remote node creates a delayed response and is dependent on data transport availability. Besides performance hits to machine learning models on IoT at the edge, any model training on IoT edge devices is nearly impossible. With the introduction of the Coral Tensor Processing Unit (TPU), real-time data processing through machine learning models on IoT edge devices is achievable. This research explores splitting a convolutional neural network (CNN) to expose an intermediate layer for fine-tune training. This study found that it is possible to extract an intermediate layer output from a CNN running on the TPU for fine-tune training on a Raspberry Pi v4 where the fine-tuning is done only on the upper layers of the model. This makes it possible to fine-tune train larger models on a resource restricted device. The model's performance improved 6.7%, from 53.9% to 60.6%.

14. SUBJECT TERMS machine learning, transfer learning, fine-tuning, federated learning, federated averaging, shared learning, Coral TPU	**15. NUMBER OF PAGES** 49
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

SHARED LEARNING AMONG DISTRIBUTED EDGE DEVICES USING CORAL EDGE TPU MACHINE LEARNING ENGINES

Erik Dubois
Major, United States Army
BS, Michigan Tech University, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2021

Approved by: Marko Orescanin
Advisor

Gurminder Singh
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Internet of Things (IoT) edge devices have small amounts of memory and limited computational power. These resource-constrained devices consist of sensors that generate large amounts of data, making IoT edge devices attractive targets for machine learning models. To take advantage of machine learning models normally requires the data to be transported to a remote device with enough computational power to process these data. The transport of data to a remote node creates a delayed response and is dependent on data transport availability. Besides performance hits to machine learning models on IoT at the edge, any model training on IoT edge devices is nearly impossible. With the introduction of the Coral Tensor Processing Unit (TPU), real-time data processing through machine learning models on IoT edge devices is achievable. This research explores splitting a convolutional neural network (CNN) to expose an intermediate layer for fine-tune training. This study found that it is possible to extract an intermediate layer output from a CNN running on the TPU for fine-tune training on a Raspberry Pi v4 where the fine-tuning is done only on the upper layers of the model. This makes it possible to fine-tune train larger models on a resource restricted device. The model's performance improved 6.7%, from 53.9% to 60.6%.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
- A. MOTIVATION....1
- B. RESEARCH FOCUS....2
- C. RESEARCH QUESTIONS....4
- D. CONTRIBUTIONS....4
- E. THESIS ORGANIZATION....5

II. TECHNICAL BACKGROUND....7
- A. EDGE DEVICE CAPABILITIES AND LIMITATIONS....7
- B. CORAL EDGE TENSOR PROCESSING UNIT ACCELERATOR....7
- C. FINE-TUNING MACHINE LEARNING MODELS....8
- D. SPLIT MACHINE LEARNING MODELS....9
- E. FEDERATED LEARNING....10
- F. APPLICATIONS....11
- G. SUMMARY....12

III. METHODOLOGY....13
- A. SPLIT MODEL AND FEDERATED LEARNING....13
- B. SYSTEM ARCHITECTURE....13
  1. Extract Intermediate Layer Outputs for Fine-Tuning Model....14
  2. Raspberry Pi v4 Preparation for Model Inference an SplitNet Fine-Tune Training....15
  3. Deploy Model from Raspberry Pi v4 for Edge TPU Inference....16
  4. Local Fine-tuning of Upper Layers....16
  5. Possibility of Federated Fine-tuning of Upper Layer....17
- C. DATASETS FOR PERFORMANCE EVALUATION....18
  1. CIFAR10 Dataset....18
  2. MNIST Dataset....19
- D. SUMMARY....20

IV. RESULTS AND ANALYSIS....21
- A. OVERVIEW....21
- B. MACHINE LEARNING MODEL ARCHITECTURES....21

1. CIFAR10 Initialized Convolutional Neural Network Model....21
2. MNIST Initialized Convolutional Neural Network Model....24
C. CORAL EDGE TENSOR PROCESSING UNIT AUGMENTED EDGE DEVICE PERFORMANCE....25
1. Use of CIFAR dataset....25
2. SplitNet Baseline Performance....25
3. Spit Net Performance on the Augmented Coral TPU Raspberry Pi v4....26
D. SUMMARY....29

V. CONCLUSIONS AND FUTURE WORK....31

LIST OF REFERENCES....33

INITIAL DISTRIBUTION LIST....35

LIST OF FIGURES

Figure 1. Raspberry Pi v4 with Coral TPU accelerator connected by USB 3.0. 7

Figure 2. SplitNet showing how a CNN graph's intermediate layer is split and fed into two differently classified CNNs running on separate GPUs. Source: [7]. 9

Figure 3. An example of a neural network multi-categorical (multi-head) and intermediate layer output. 10

Figure 4. Google's concept of federated learning. Source: [10]. 11

Figure 5. Security perimeters where a distributed edge sensor network would be deployed can range from foot patrol halts to permanent bases. 12

Figure 6. Flow diagram of SplitNet fine-tuning architecture 14

Figure 7. Basic CNN example of using Keras Functional API to split the output of the final categorical outputs and an intermediate model layer 15

Figure 8. Example of full Keras CNN model 17

Figure 9. Example of creating a Keras CNN model of the upper layers of the full model shown in Figure 8. 17

Figure 10. CIFAR10 Dataset Sample Images. Source: dataset extraction 19

Figure 11. MNIST Dataset Sample Images. Source: dataset extraction 20

Figure 12. Pre-trained Keras ten-head CNN classification model prediction scores for the CFAR10 test dataset on Google Colab 22

Figure 13. Confusion Matrix for Keras ten-head CNN classification model on Google Colab. 22

Figure 14. Pre-trained tflite CNN model prediction scores for the CFAR10 test dataset on Google Colab. 23

Figure 15. Confusion Matrix for tflite ten-head CNN classification model on Google Colaboratory. 24

Figure 16. Pre-trained Keras ten-head CNN classification model prediction scores for the MNIST test dataset on Google Colaboratory. 24

Figure 17. Confusion Matrix for MNIST ten-head CNN classification model on Google Colaboratory. 25

Figure 18. Pre-trained tflite CNN model prediction scores for the CFAR10 test dataset on Coral TPU augmented RPi. 26

Figure 19. Training Results of first test from Initial Model Training through 15 Fine-tune Training Sessions. 27

Figure 20. Training Results of second test from Initial Model Training through 15 Fine-tune Training Sessions. 28

Figure 21. Training Results of third test from Initial Model Training through an average of 7 Fine-tune Training Sessions. 29

LIST OF ACRONYMS AND ABBREVIATIONS

API	application programming interface
ARM	Advanced RISC Machine
CIFAR	Canadian Institute for Advanced Research
CNN	convolutional neural network
FedAvg	federated averaging
FL	federated learning
ML	machine learning
MNIST	modified National Institute of Science and Technology
RISC	reduced instruction set computer
RPI	Raspberry Pi v4
SplitNet	split network
tflite	TensorFlow Lite
TPU	Tensor Processing Unit

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The ability to deploy distributed networks of edge devices as a sensor network has become a reality. These sensor networks can be used in fixed locations to alert security forces to suspicious activities around fixed structures and may one day be used to temporarily deploy with military patrols in hostile environments to give commanders situational awareness of the battlefield. Deploying distributed networks of edge devices requires adapting to the environment they operate in.

Edge devices as Internet of Things have gained popularity as smart sensors [1] that can connect to powerful centralized resources with a large storage and machine learning (ML) models to process data collected by edge sensors. With the increasing capability of wireless networks, this would seem an insignificant issue. However, the large amounts of data necessary to train and fine-tune ML make this an unrealistic scenario for many edge applications due to the bandwidth and power requirements. Additionally, sharing data over wireless networks may compromise the privacy of data collected by edge devices unless strong security measures are taken.

High bandwidth network links are not necessarily required to deal with the bottleneck in performance from transmitting data between the cloud resource and edge device. Exposing data to the Internet opens data up to unauthorized collection and manipulation. Edge devices are capable of processing data and improving ML model performance at the edge without sharing collected data.

Fine-tuning ML models at the edge is limited by the processing power of the edge devices. Augmenting the edge device with a USB Tensor Processing Unit (TPU) allows for a dedicated processor to execute ML models with improved performance over the edge device's Advanced RISC Machine (ARM) [2] processor. The goal is to locally, on the edge device's ARM processor, improve ML models via fine-tuning. This would be conducted on a subset or the last few fully-connected layers of the ML model. It is key to realize that

a Coral TPU is a device that exclusively conducts inference, which means that it is not possible to conduct training on it.

To improve the ML model running on a network of distributed edge sensors, a single edge device would receive updates consisting of intermediate layer weights from all participating edge devices. The weights from these edge devices would then be averaged using a technique called federated averaging (FedAvg). The averaged weights are then sent back to participating edge devices for new inferencing. The result is a shared ML model performance improvement among all of the edge devices. This is known as centralized federated learning (FL) at the edge. Data collected at the edge is not exposed and it limits network overhead. This is done by not transmitting large amounts of data and keeping network traffic on the local edge device.

Previous work by M. Baxter [3] explored both fine-tuning convolutional ML models and conducting inference on the same computer processor. Local ML model's fine-tune training, inference, and inter-network communication for FL was executed on the Raspberry Pi v4 (RPI) ARM processor. The convolutional ML model used was pre-trained, then lower layers were frozen leaving upper classification layers as trainable to enable fine-tuning. Running inference and fine-tune training on the same processor creates significant limitations in the number of inferences per second as well as the number of layers within the convolutional ML model used for fine-tune training. This limited scaling of the number of layers was used for fine-tuning. A feasible improvement is to use a Coral TPU peripheral device for dedicated inferencing. This would result in freeing up the native ARM processor for fine-tuning trainable layers of the convolutional ML model.

B. RESEARCH FOCUS

The focus of this research is to create a shared ML environment among a distributed network of edge devices using the Coral TPU as the ML model engine and RPI computers as edge devices. The addition of the Coral TPU for dedicated inferencing to previous work by M. Baxter [3] frees up the native ARM processor of the RPI for fine-tuning larger, more realistic, convolutional ML models and a deeper starting point for upper convolutional layers. This approach would allow for a real-time inference with TPU and occasional fine-

tuning cycles on the ARM processor. The exploration of sharing ML model characteristics is conducted by splitting a convolutional neural network (CNN) architecture into a dual head output. Characteristics (weights and biases) of a fully-connected intermediate layer and the final classification layer results are output. The intermediate layer outputs are used as inputs to the remaining layers of the CNN when the classification outputs are outside of a determined threshold. Communication between edge devices is done via Wi-Fi (IEEE 802.11) and uses the Message Queuing Telemetry Transport messaging protocol to share the trainable ML model layer characteristics. One of the edge devices is designated for conducting FedAvg of all edge devices providing input. The FedAvg edge device then makes the new layer characteristics available for consumption by the participating edge devices. This results in a common ML environment where each edge device provides input to the shared learning of all participating edge devices.

Currently, the consumer market is being saturated by many solution providers in the embedded space with accelerators for ML models. An example is the USB Coral TPU, a dedicated processor designed to execute ML models at the edge. ML models are converted and compiled for execution on the TPU. To allow for transfer learning (fine-tuning), there are two methods designed for use with the Coral TPU. Weight imprinting and backpropagation [4] are both designed to allow the re-training of the last fully connected layer of a classification ML model. These techniques are designed to be used specifically on classification ML models, because the last fully connected layer is where classification occurs. Only the base layers of the ML model are compiled for inferencing on the TPU. Coral documentation recommends using a pre-trained ML classification model that is trained to classify similar items.

ML models are trained and fine-tuned using datasets. Once an edge device has conducted baseline training with large datasets, the fine-tuning can be done with locally obtained data. For a single edge device to fine-tune, there is no issue since that data is obtained by local sensors. In the case of using data to fine-tune a distributed network of edge devices to achieve shared ML, the data is exposed over the network. The goal is to achieve shared ML, so when an individual node learns to classify an object, it shares how it did this. This is done by sharing the last few fully-connected layer characteristics. The

reduction of network traffic and exposure of local data is mitigated using this technique. By only sharing a subset of the layer weights, exposure of the ML model characteristics is limited to the upper trainable layers. This is known as FL, which is the concept used for fine-tuning distributed edge devices running ML models that hold locally collected data. The FL approach allows groups of edge devices to build a common ML model across the edge network without sharing collected data. Input data collected at each edge device remains local.

C. RESEARCH QUESTIONS

- Is it possible to fine-tune ML models on an edge device while making real-time inferences on a Coral TPU?
- How can a CNN model be created as a split output architecture to facilitate fine-tuning upper layers of the model utilizing hybrid computations between an ARM processor and a TPU?
- How can a distributed network of edge devices using Coral TPU inference engines use FL to create a shared prediction model?
- What are the advantages of running inference using a Coral TPU over the edge device's general-purpose ARM processor?

D. CONTRIBUTIONS

Three contributions are made with this research by developing a split network (SplitNet) ML model to facilitate fine-tune training on the ARM processor of a RPI and model inferencing on a USB connect TPU.

1. Performance of fine-tuning a single edge device running a peripheral inference engine is demonstrated and quantified.
2. A path is proposed to a more secure FL technique on an edge device by splitting out an intermediate CNN layer for fine-tune training on an ARM processor allowing for centralized FL.

3. Overall, this thesis demonstrates a novel concept of utilizing an external inference engine in a SplitNet architecture with a general-purpose processor, where large CNN models can be used for both inferencing and fine-tuning in near real-time. This is, to our knowledge, the first SplitNet implementation residing fully on an edge device.

E. THESIS ORGANIZATION

Chapter II defines foundational concepts in ML, fine-tuning, FL, and describes the Coral TPU.

Chapter III describes creating a dual output CNN model that facilitates fine-tuning upper layers (trainable), the RPI edged device, the Coral TPU architecture, and the FL architecture research methodology.

Chapter IV discusses data analysis and evaluates research results of experiment.

Chapter V reviews architectural limitations and future enhancements.

THIS PAGE INTENTIONALLY LEFT BLANK

II. TECHNICAL BACKGROUND

A. EDGE DEVICE CAPABILITIES AND LIMITATIONS

Edge devices like the Raspberry Pi v4 (RPI) are generally small in size, allowing them to be easily deployed. The small form factor of these devices means, they have limited on-board processing resources and are typically battery power constrained. Power limitations can be overcome through alternate power supply configurations; however, two limitations that are difficult to overcome are the processing power and memory size. These limitations can be overcome by attaching additional processing units. Use of the Coral Edge Tensor Processing Unit (TPU) accelerator is explored here as a coprocessor to the RPI native Advance RISC Machine (ARM) processor.

B. CORAL EDGE TENSOR PROCESSING UNIT ACCELERATOR

The Coral TPU accelerator is a small ASIC processor that increases inference performance on resource constrained devices [5] (see Figure 1). It provides a coprocessor to the edge device's native ARM processor that allows edge devices to have a dedicated inference engine resulting in increased predication rates. This also frees up the native RPI ARM processor for deep convolutional layer fine-tuning.

Figure 1. Raspberry Pi v4 with Coral TPU accelerator connected by USB 3.0.

C. FINE-TUNING MACHINE LEARNING MODELS

Fine-tuning of a pre-trained machine learning (ML) model is where the upper, trainable, layers of the model are re-trained [6]. The lower, pre-trained, layers are used in combination with the upper layers of the model during fine-tune training. They are trained a number epochs to improve performance until a predetermined accuracy threshold is achieved. The result is new upper layer coefficients, reined in during the fine-tuning process, that allows the ML model to better predict what the input data represents.

The Coral TPU accelerator is capable of training classifiers on top of a convolutional neural network (CNN) [4]. There two forms of transfer learning on-device for the Coral TPU. First, weight imprinting which takes the output from lower layers of the model, adjusts the activation function, and then uses the resulting values to compute the new weights of the last layer. Second, backpropagation, which only updates the fully connected layer at the end of the model with new weights rather than updating all of the layers which requires a significant increase in the amount of data and epochs of training. In both cases, the available on-device transfer learning only allows for updating the final layer or the classifier on top of the model representations which does not give enough flexibility for real world applications.

For a ML model running on a Coral TPU to be able to conduct transfer learning beyond the last fully connected layer, there needs to be a way to extract and share model layer characteristics. The concept of split models or split network (SplitNet) [7] (see Figure 2) is used to expose intermediate layer outputs. This enables ML models running inference on a Coral TPU to output both the intermediate layer and the final classification layer outputs for fine-tuning of the upper layers on the RPI ARM processor.

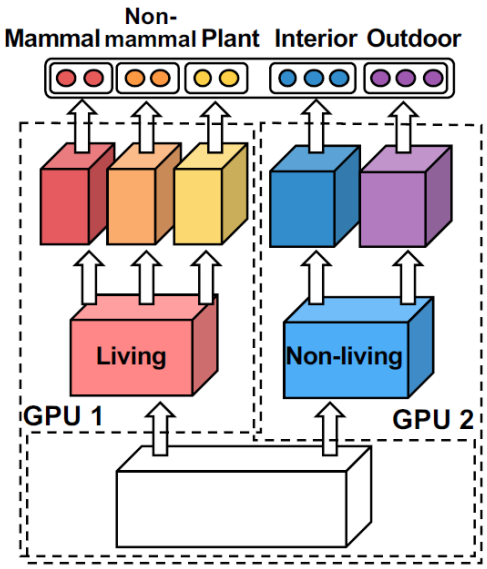


Figure 2. SplitNet showing how a CNN graph's intermediate layer is split and fed into two differently classified CNNs running on separate GPUs. Source: [7].

D. SPLIT MACHINE LEARNING MODELS

Splitting ML networks or models have been explored to reduce the number of parameters computed on a single processor through parallel processing and removing weak connections which also results in reducing the computation time requirement for the models [7]. CNN ML models contain a number of convolutional layers that learn image representations. The lower convolutional layers start with identifying primitive representations and subsequently upper layers which identify more detailed representations until the last results in a final prediction. Using the concept of SplitNet to fine-tune a convolutional ML model on a multi-processor setup, a single model inference can be continuously run on the Coral TPU while fine-tuning the model on the native ARM processor of the RPI.

Multi-classification output of a convolutional ML model means that the training data consists of multi-label data requiring the model to output multiple categories [8] (see the three outputs in Figure 3). Recreating the ML model layer by layer using Keras' Functional application programming interface (API) [9] allows for control over freezing layers and controlling outputs of each layer. Specifically, Keras Functional API is capable

of dealing with ML models with non-linear topologies, shared layers, and multiple inputs/ outputs (see Figure 3). The focus here is the capability to deal with multiple outputs. An intermediate layer output is used as input to fine-tune upper layers of the convolutional ML model.

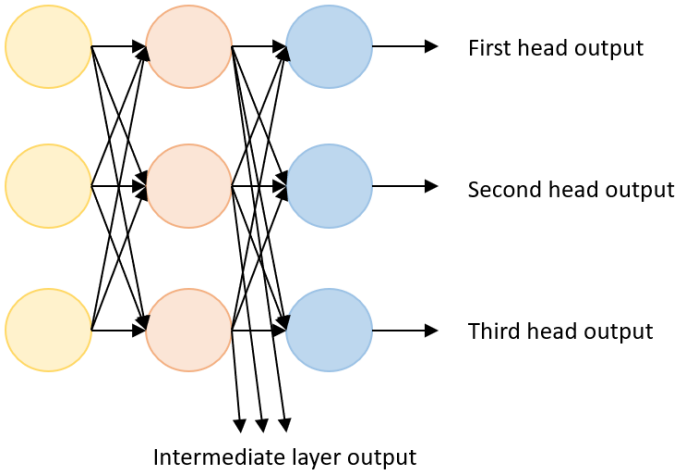
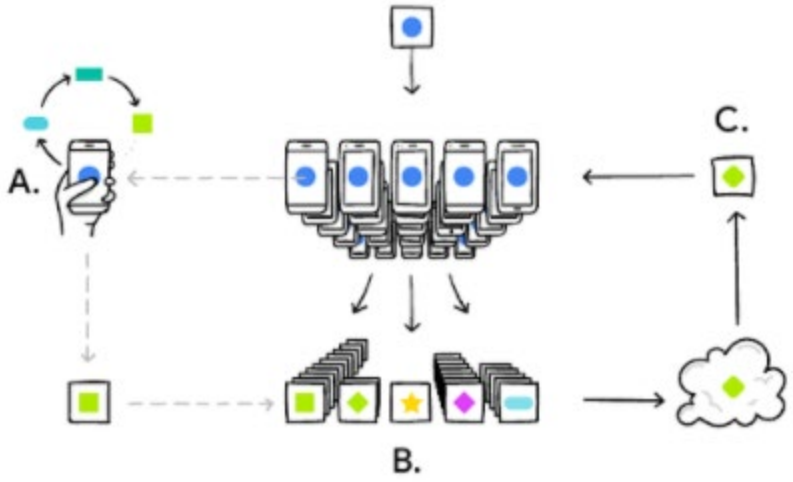


Figure 3. An example of a neural network multi-categorical (multi-head) and intermediate layer output.

E. FEDERATED LEARNING

Traditional training of ML models requires all of the data to be collected and available for training with a centralized infrastructure. This does not work well for a distributed network of edge devices. Two major issues are the network bandwidth requirement to push large amounts of data across a network from multiple nodes and exposing collected data when transmitting to the central processing machine. This is also an issue if one of the edge devices is the central processing unit for all the collected data. To overcome this issue of centralized training using distributed collected data, the concept of federated learn (FL) (see Figure 4) was introduced by Google to enable distributed edge devices to collaboratively learn a shared prediction model [10].



A: Local fine-tuning of ML model. B: Averaging of distributed edge devices local ML model's layer characteristics. C: Re-distribution of averaged ML model's layer characteristics.

Figure 4. Google's concept of federated learning. Source: [10].

For a distributed network of edge devices running ML models to benefit from a shared learning experience, they need to share something of themselves with each other. The edge device is deployed with a pre-trained ML model and improves their own model locally by fine-tuning it, using locally collected data. The changes in the ML model characteristics (weights and biases) are transmitted to a pre-determined edge device that conducts averaging operations on all ML model characteristics received from participating edge devices. The averaging operations are known as federated averaging (FedAvg) [11]. The collaboratively fine-tuned ML model characteristics are made available to participating edge devices for integration into a new inferencing model. This results in a shared ML model where all participating edge devices have benefitted from the others.

F. APPLICATIONS

Forward Operating Bases (FOB) are temporary austere bases pushed out to the edge of operationally controlled areas. Self-sustaining and self-learning distributed edge sensor networks would be beneficial to small units (see Figure 5). These networks would be able to learn to predict new threats as they evolve. A distributed edge network would be able to provide shared fine-tuning by any of the participating edge devices, thus not requiring

transmission to a centralized machine located at another geographical location. The central processing unit for FedAvg operations would be executed by any of the edge devices in a round-robin or low work load availability mode. The deployed sensor network would be able to evolve and sustain usefulness through shared learning among the edge sensors [12].

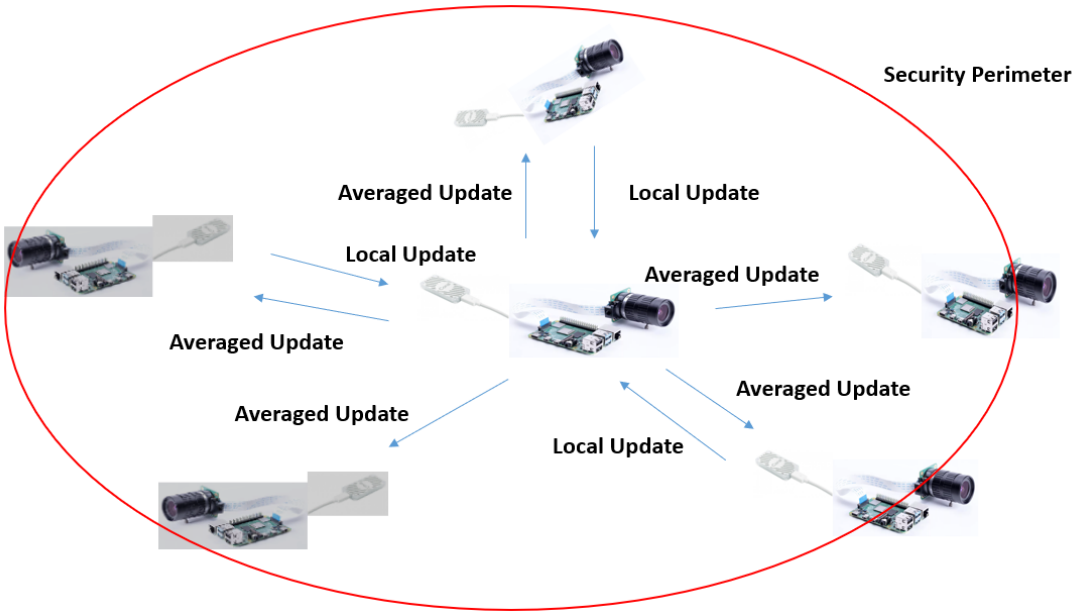


Figure 5. Security perimeters where a distributed edge sensor network would be deployed can range from foot patrol halts to permanent bases.

G. SUMMARY

The identified SplitNet [7] concept, led to the implementation of a convolutional ML model that outputs at both an intermediate layer as well as the categorical classification output. TensorFlow's Functional API is used to build the convolutional ML model, layer by layer, which gives great control over points of input and output. The methodology of how this was done is outlined as follows.

III. METHODOLOGY

A. SPLIT MODEL AND FEDERATED LEARNING

In order to access intermediate neural network (graph) layers, the definition of the model has to be flexible enough to provide outputs at various points. Convolutional neural network (CNN) outputs allow for multi-class classification, which is implemented in this research [8]. What is needed to support deeper layer fine-tuning of a convolutional machine learning (ML) model running inference on a Coral Tensor Processing Unit (TPU) is the ability to output the chosen intermediate layers. The split network (SplitNet) [7] implementation splits large CNN models at multiple intermediate layers for use with parallel processing [11] and is where the idea of splitting out the intermediate layers for this research was conceived.

For this research, the output of an intermediate layer of a CNN model is made available for fine-tuning the local model running on a Raspberry Pi v4 (RPI) Advance RISC Machine (ARM) processor. The result is the fine-tuned intermediate layer values that improve the model's prediction performance. This action could be mimicked by other participating edge devices which results in several sets of the same intermediate layer outputs. Since the fine-tuning can be done on separate edge devices, the values differ and require integration. These intermediate layer values could further be sent to a central edge device for averaging, also known as federated averaging (FedAvg). The result is a shared learning experience among all participating edge devices via sharing of the coefficients of the fine-tuned top layers. Although, the above-described schema is desirable, in this thesis only the single node fine-tuning was evaluated in a SplitNet configuration.

B. SYSTEM ARCHITECTURE

The overall architecture of how the convolutional ML model (see Figure 6) was developed using TensorFlow's Functional application programming interface (API) is discussed to show how the intermediate layer output of the model is accessed. Taking advantage of this intermediate output using an abbreviated convolutional ML model made

up of the upper layers of the full convolutional ML model are further investigated. Finally, the concept of deploying a SplitNet to hardware for inference and fine-tuning is provided.

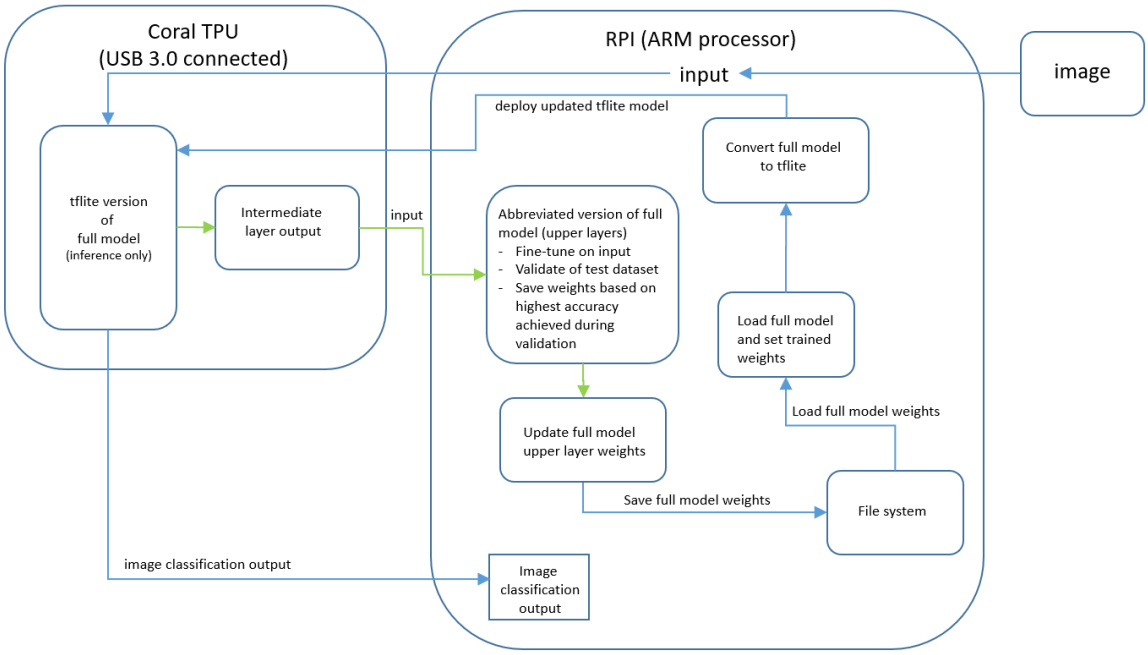


Figure 6. Flow diagram of SplitNet fine-tuning architecture

1. Extract Intermediate Layer Outputs for Fine-Tuning Model

The Keras Functional API [9] provides the flexibility to extract layer outputs at any point in a graph (see Figure 7). Using this API, the developer has access to each layer's input and output of a model. This enables the split model concept by exposing the intermediate layer outputs for fine-tuning and federated learn (FL).

Previous work in this area related to federated fine-tuning on edge devices using the RPI ARM processor for every aspect was done without a Coral TPU [3]. The RPI was used for model inferencing, fine-tuning, and network communications for FedAvg. This research focused on using the Coral TPU for inference and the RPI ARM processor for fine-tuning a local convolutional ML model. Splitting up the inferencing and model fine-tuning on separate processing units allowed for the TPU to continue conducting inference while the RPI ARM processor worked at updating the model. As the intermediate layer

values were updated by the centralized edge device, the local edge device redeployed the updated model intermediate layer characteristics. Splitting the processing allowed for the edge device to continue performing inferencing while also improving the performance of the model.

```
inputs = tf.keras.Input(shape=(32, 32, 3), name='layer_A0')
x = layers.Conv2D(32, (3, 3), activation='relu', name='layer_A1')(inputs)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Conv2D(64, (3, 3), activation='relu', name='layer_A2')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Conv2D(64, (3, 3), activation='relu', name='layer_A3')(x)
output_A = layers.Flatten(name='layer_A4')(x)
x = layers.Dense(64, activation='relu', name='layer_B1')(output_A)
output_B = layers.Dense(10, name='layer_B2', activation='softmax')(x)

model = keras.Model(
    inputs=inputs,
    outputs=[output_B, output_A],
)
```

Figure 7. Basic CNN example of using Keras Functional API to split the output of the final categorical outputs and an intermediate model layer.

2. Raspberry Pi v4 Preparation for Model Inference an SplitNet Fine-Tune Training

The ease of developing on the Google Colaboratory platform led to the initial development work being done. This led to some issues when deploying the model to the RPI platform. Google Colaboratory by default loads the latest TensorFlow version 2.4. The Debian and Ubuntu OS versions for the RPI loaded TensorFlow v1.15 by default which resulted in incompatibility between the TensorFlow v2.4 saved model and loading it using TensorFlow v1.15. Changing the Google Colaboratory development environment to use TensorFlow version to 1.15 was explored, but abandoned after realizing that TensorFlow Lite (tflite) conversion capabilities was no longer supported and lacked functionality. This led to the exploration of getting TensorFlow v2 operational on the Debian 32-bit OS for RPI working. Installing TensorFlow v2.0 and other required modules for running fine-tune training of a Keras CNN model on an ARM processor, converting it to tflite, and running inference on a Coral TPU were derived from reference [13].

3. Deploy Model from Raspberry Pi v4 for Edge TPU Inference

The outputs are made available on the RPI ARM processor by the PyCoral API [14]. When the categorical outputs fail to meet a pre-determined threshold, an abbreviated model of the full model is used to fine-tune the model. This is where the intermediate layer becomes the input layer weights for the fine-tuning of the abbreviated model. This results in a condensed model that is less computationally intensive and faster than the full model. Recall how the lower layers are frozen, so it is irrelevant that the lower layers are not present during fine-tuning. Once the necessary number of epochs are run to achieve the desired prediction reliability, the new upper layer characteristics are applied to the full model before it is converted and used to replace the current model running inference on the Coral TPU.

4. Local Fine-tuning of Upper Layers

To keep fine-tune training local to the edge device, the SplitNet concept can be applied to the RPI through the use of TensorFlow's Functional API techniques that allow for exposing intermediate layers of a model as output. Recall that a Coral TPU is a device that exclusively conducts inference, so it is not possible to conduct training on it. It should also be recognized that the TPU runs a quantized version of the full Keras CNN model where layer characteristics cannot currently be extracted for training. This is where creating a new Keras model based on the original full Keras model is used for fine-tuning upper layer characteristic weights. An example of a full Keras CNN model is shown in Figure 8 and an example of a model based on the upper layers of the full model are shown in Figure 9. The lower weights being frozen do not change, so they are not needed for fine-tuning. The updated upper layer characteristics are extracted and used to update the full model. Finally, the full Keras CNN model is converted to tflite and re-deployed for inference operations on the Coral TPU. The overall flow of the SplitNet implementation is shown in Figure 6.

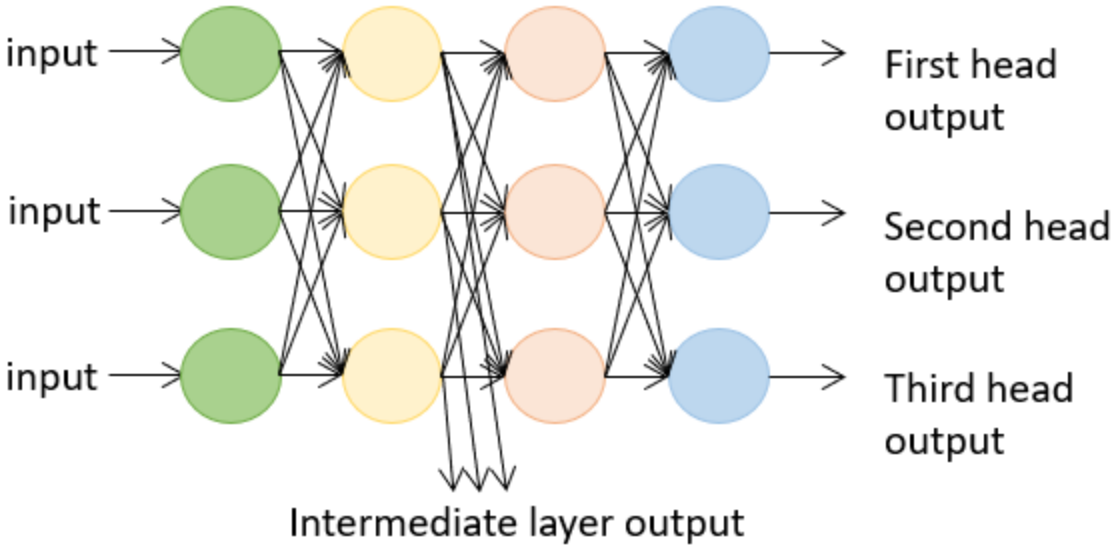


Figure 8. Example of full Keras CNN model

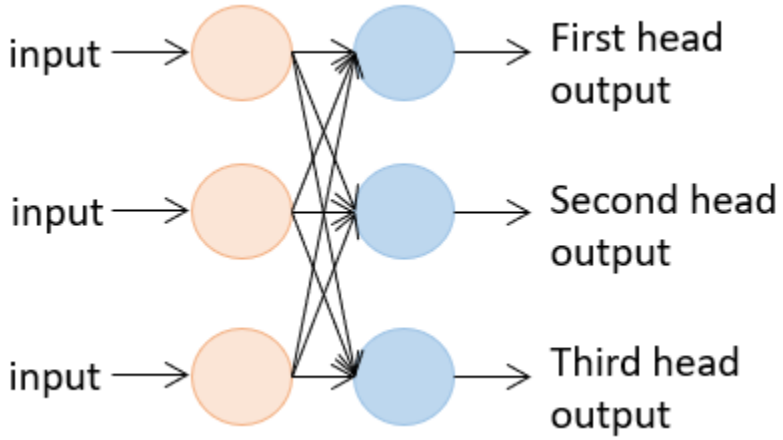


Figure 9. Example of creating a Keras CNN model of the upper layers of the full model shown in Figure 8.

5. Possibility of Federated Fine-tuning of Upper Layer

The CNN models are designed to expose the model categorical output as well as the output of the chosen intermediate layer. As the Coral TPU completes an inference of an image, the RPI assesses the categorical prediction output of the model. When the

model's prediction does not meet the threshold, the model is fine-tuned using a local dataset of an evenly distributed representation of classes of 2500 images. Once the threshold is met, the resulting layer characteristics of the intermediate layer can be sent to a central edge device for averaging, through FedAvg, with other participating edge devices that are providing intermediate layer updates. The newly averaged intermediate layer characteristics in such a system are sent back to all edge devices in the network where they are used to update the CNN model. The model is then converted to tflite and exposed to the Coral TPU for use during inferencing. This represents the logical flow of local fine-tuning and remote FedAvg updates of edge-based models.

C. DATASETS FOR PERFORMANCE EVALUATION

Four datasets were used to evaluate the performance of converting CNN models to tflite models and running inference on the Coral TPU.

1. CIFAR10 Dataset

The CIFAR10 dataset [15] consists of 6,000 images of 10 classes for a total of 60,000 images where each image is in color with dimensions of 32x32. The dataset is split 5:1 training and test images. As seen in Figure 10, the classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. This dataset is available at https://www.cs.toronto.edu/~kriz/cifar.html.

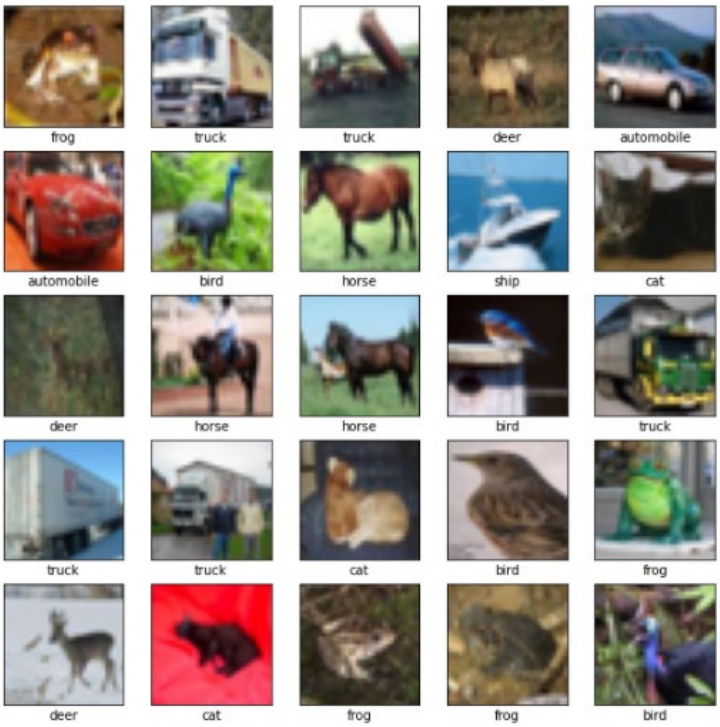


Figure 10. CIFAR10 Dataset Sample Images. Source: dataset extraction

2. MNIST Dataset

The MNIST dataset [16], as seen in Figure 11, consists of 10 classes of single handwritten character digit numbers from 0 to 9. The training set consists of 60,000 images while the testing set consists of 10,000 images where each image is grayscale with dimensions of 28x28. Examples are from approximately 250 writers. This dataset is available at: http://yann.lecun.com/exdb/mnist/.

Figure 11. MNIST Dataset Sample Images. Source: dataset extraction

D. SUMMARY

The SplitNet concept is the basis of this architecture and is implemented using the TensorFlow Functional API to make a chosen intermediate layer accessible for fine-tune training. This concept is further implemented on a Coral TPU augmented RPI to show that it is possible to run inference and fine-tune training on the same edge device. The CIFAR10 dataset was chosen because there are more than two classifications of images and the developed convolutional machine learning (ML) model was not able to quickly gain a dominate predication of the classes. This methodology allowed for successful fine-tune training of a SplitNet convolutional ML model while also conducting inference on a Coral TPU augmented RPI.

IV. RESULTS AND ANALYSIS

A. OVERVIEW

The objective of this research was the development of a multi-class classification convolutional neural network (CNN) that is capable of fine-tune training on a small-sized, low-power edge device augmented with a Coral Tensor Processing Unit (TPU) processor. This was achieved by using a Raspberry Pi v4 (RPI) as the edge device and TensorFlow v2 Functional application programming interface (API) for the model creation. The CIFAR10 and MNIST datasets were explored for demonstrating the ability of this architecture to perform on device fine-tune training.

B. MACHINE LEARNING MODEL ARCHITECTURES

1. CIFAR10 Initialized Convolutional Neural Network Model

A basic CNN was developed to test out the concept of developing a split model for updating the upper, trainable, layers of the CNN. This CNN was pre-trained using the CIFAR10 dataset with 6 epochs. The initially trained model on average, accurately predicts among the 10 class images 55% of the time. On average, each single class is predicted correctly 54% of the time. Since the dataset is evenly balanced, the F1 score which is the harmonic mean between precision and recall, is as expected, on average 54%. Figure 12 outlines each class's prediction scores. Note that the support column for each class is 1,000 which indicates that the test dataset is evenly balanced. It is also interesting to note that the training dataset of 50,000 images was also balanced yet resulted in varying prediction scores. The training dataset was split in two, resulting in 25,000 sets for training and 25,000 sets for fine-tuning.

	precision	recall	f1-score	support
airplane 0	0.62	0.44	0.52	1000
automobile 1	0.72	0.58	0.64	1000
bird 2	0.45	0.39	0.42	1000
cat 3	0.38	0.34	0.36	1000
deer 4	0.57	0.24	0.34	1000
dog 5	0.43	0.56	0.49	1000
frog 6	0.66	0.63	0.65	1000
horse 7	0.57	0.65	0.61	1000
ship 8	0.53	0.82	0.64	1000
truck 9	0.54	0.74	0.62	1000
accuracy			0.54	10000
macro avg	0.55	0.54	0.53	10000
weighted avg	0.55	0.54	0.53	10000

Accuracy: 0.539

Figure 12. Pre-trained Keras ten-head CNN classification model prediction scores for the CFAR10 test dataset on Google Colab

Figure 13 shows the confusion matrix of the test dataset used in this model. There are several classes that seem to be triggering false positive predictions. For instance, looking at the "cat" class, the model seems to be hitting false positive predictions on the "dog" and "frog" classes.

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	[[444	29	52	23	13	15	10	16	314	84]
automobile	[24	577	9	7	2	4	8	11	84	274]
bird	[78	9	391	80	52	144	62	74	64	46]
cat	[20	15	65	341	29	274	66	75	69	46]
deer	[37	11	176	76	238	139	113	153	40	17]
dog	[17	6	60	153	25	557	34	98	29	21]
frog	[9	19	73	105	26	36	629	29	27	47]
horse	[21	7	25	82	25	103	16	652	15	54]
ship	[49	44	11	7	3	9	3	8	822	44]
truck	[16	88	9	20	4	5	8	23	87	740]]

Figure 13. Confusion Matrix for Keras ten-head CNN classification model on Google Colab.

After the CNN model was converted to TensorFlow Lite (tflite), Figure 14 shows that the class predictions remain very nearly the same with variations of plus/minus 0–2% in the F1-scores. Figure 15 shows the confusion matrix for the tflite conversion of the CNN model with minor variations in predictions. Looking again at the "cat" class, the "dog" class is nearly the same as in the non-tflite version while the "frog" class seems not to affect the "cat" prediction as much. The conversion of the full CNN model to the tflite model has limited impact to model prediction.

	precision	recall	f1-score	support
airplane 0	0.62	0.64	0.63	1000
automobile 1	0.74	0.55	0.63	1000
bird 2	0.38	0.52	0.44	1000
cat 3	0.37	0.37	0.37	1000
deer 4	0.46	0.43	0.44	1000
dog 5	0.54	0.28	0.36	1000
frog 6	0.52	0.73	0.61	1000
horse 7	0.69	0.56	0.62	1000
ship 8	0.74	0.59	0.65	1000
truck 9	0.52	0.73	0.61	1000
accuracy			0.54	10000
macro avg	0.56	0.54	0.54	10000
weighted avg	0.56	0.54	0.54	10000

Accuracy: 0.539

Figure 14. Pre-trained tflite CNN model prediction scores for the CFAR10 test dataset on Google Colab.

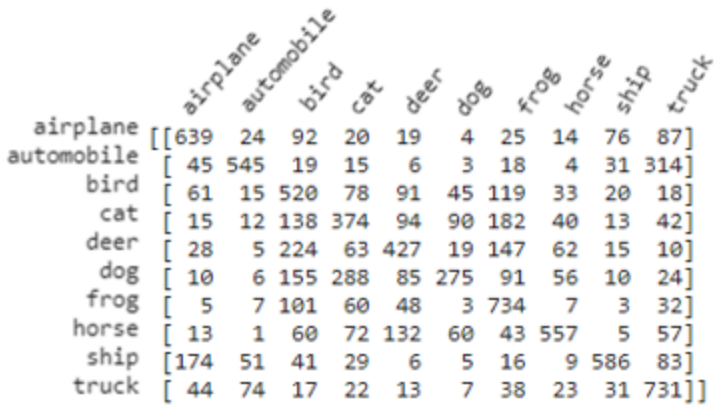


Figure 15. Confusion Matrix for tflite ten-head CNN classification model on Google Colaboratory.

2. MNIST Initialized Convolutional Neural Network Model

The same base CNN model was used for the MNIST dataset. It was initially trained on 6 epochs and brought down to 1 epoch. As seen in Figures 16 and 17, the prediction performance does not appear to impact the model based on the number of training intervals. Since the MNIST is an equally distributed dataset that results in an average over all classes and F1-score of 98%, it is not a good candidate for fine-tuning to improve prediction of the tested classes. The model was not converted to tflite for further use.

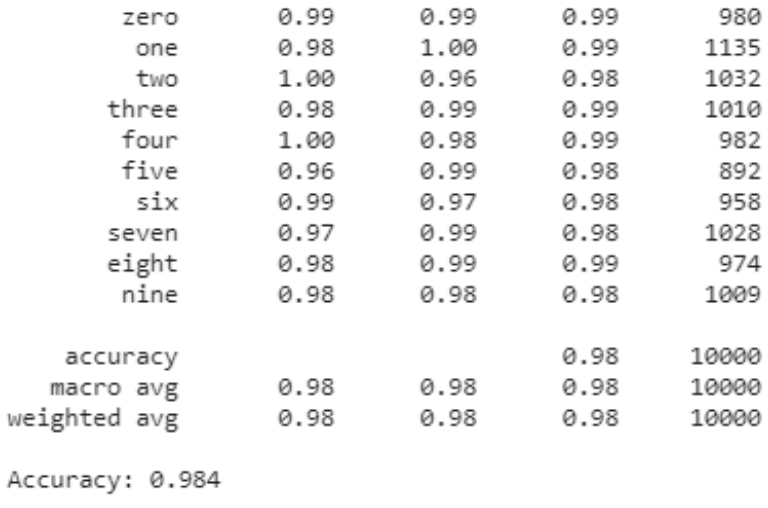


Figure 16. Pre-trained Keras ten-head CNN classification model prediction scores for the MNIST test dataset on Google Colaboratory.

	zero	one	two	three	four	five	six	seven	eight	nine
zero	[[975	0	0	0	0	2	2	1	0	0]
one	[0	1130	0	0	0	1	1	0	3	0]
two	[2	13	989	8	1	0	0	15	4	0]
three	[0	0	0	998	0	5	0	5	2	0]
four	[0	0	0	0	965	0	1	2	2	12]
five	[1	0	0	3	0	886	1	1	0	0]
six	[9	2	0	0	2	11	934	0	0	0]
seven	[0	3	1	2	0	1	0	1018	1	2]
eight	[2	0	0	1	0	3	1	1	963	3]
nine	[0	2	0	2	1	12	0	4	3	985]]

Figure 17. Confusion Matrix for MNIST ten-head CNN classification model on Google Colaboratory.

C. CORAL EDGE TENSOR PROCESSING UNIT AUGMENTED EDGE DEVICE PERFORMANCE

1. Use of CIFAR dataset

The CIFAR dataset was chosen over the MNIST dataset to demonstrate the capability of a split network (SplitNet) style classification CNN due to the increased number of epochs required to establish improved performance of the model. This was required to demonstrate the fine-tunability of the SplitNet architecture. The baseline model was trained on Google Colaboratory using 10k sets of images/labels for 5 epochs to achieve accuracy performance of 53.9% as shown in Figure 18. Validation of all training was done using a 10k sets of image/labels on which the model is never trained. All experiments of the SplitNet classification CNN model were conducted using the same baseline weights established using Google Colaboratory and TensorFlow v2.4.

2. SplitNet Baseline Performance

The full classification model based on TensorFlow v2.4 Functional API, Figure 12, has comparable performance when converted to tflite, as seen in Figure 14. It should be noted here that the conversion to tflite was done using TensorFlow v2.4 to show the comparable performance between the full model and the tflite version. The experiment on the RPI converted the full model from Google Colaboratory to tflite for inferencing on the Coral TPU. In this work, the most recent version of TensorFlow v2.4 was not available for

the RPI OS, so the fine-tune training on the RPI was demonstrated running TensorFlow v2.0. The performance of the Google Colaboratory tflite version and the RPI tflite version resulted in the same performance of 53.9% accuracy as show in Figures 18.

	precision	recall	f1-score	support
airplane 0	0.62	0.44	0.52	1000
automobile 1	0.72	0.58	0.64	1000
bird 2	0.45	0.39	0.42	1000
cat 3	0.38	0.34	0.36	1000
deer 4	0.58	0.24	0.34	1000
dog 5	0.43	0.56	0.49	1000
frog 6	0.66	0.63	0.65	1000
horse 7	0.57	0.65	0.61	1000
ship 8	0.53	0.82	0.64	1000
truck 9	0.54	0.74	0.62	1000
accuracy			0.54	10000
macro avg	0.55	0.54	0.53	10000
weighted avg	0.55	0.54	0.53	10000

Accuracy: 0.539

Figure 18. Pre-trained tflite CNN model prediction scores for the CFAR10 test dataset on Coral TPU augmented RPi.

3. Spit Net Performance on the Augmented Coral TPU Raspberry Pi v4

Three fine-tune training runs were conducted to validate model performance improvement. All three training runs on the RPI were done with 20 epochs in batches of 128 datasets. The fine-tune training sessions of each run consisted of 5k sets of data over a total of 40k different sets. Each training session was offset by 50% (i.e., 0–5000, 2500–7500, 5000–10000) to maximize the usage of data for the fine-tuning of the model. Keep in mind that each training session is being fine-tuned using varying sets of the 40k and the model is being evaluated during each epoch by the same 10k sets originally when training the baseline of the model. For the first two training sessions, a model check point callback was used to tell TensorFlow to only save the best model weight results. This was beneficial

in keeping the training sessions performance results trending up. Without the model checkpoint and validation datasets, the best training sessions are not saved.

The first two training results are shown in Figures 19 and 20. These two training runs were conducted exactly the same and were started at the initial training baseline of the model weights with overall accuracy performance of 53.9%. Notice how the first training run, depicted in Figure 19, shows cyclic model performance increases and decreases. From sessions 4, 5, 9, and 14 a trend of model performance decrease is observed. Now looking at the second training run, depicted in Figure 20, how the model consistently improves from session 2 to 10 and decreases two more times before ending. The different behavior depicted over the same dataset and data split, has slightly different results yet the final results are very similar. The first training run performance was 61.1% for an improvement of 7.2%. The second training run performance was 60.6% for an improvement of 6.7%.

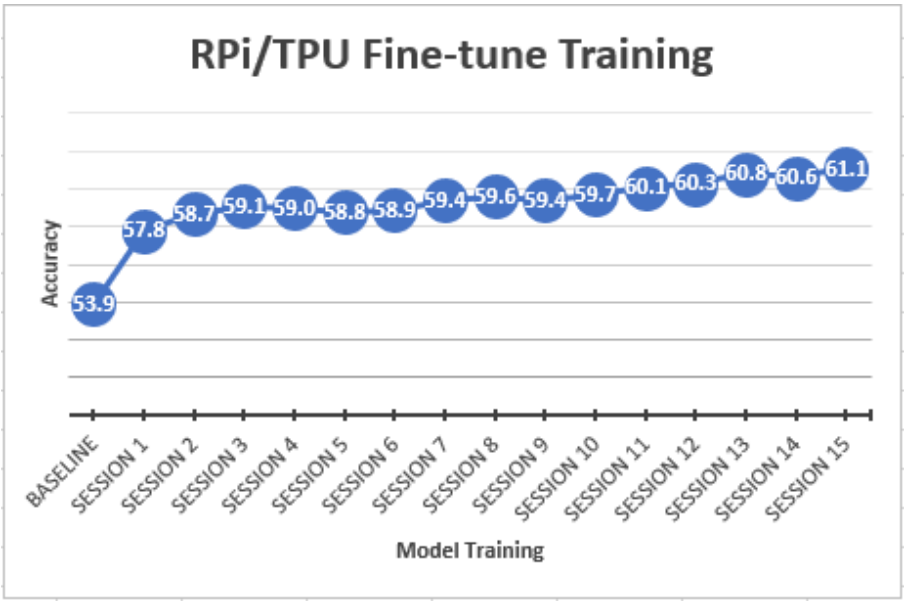


Figure 19. Training Results of first test from Initial Model Training through 15 Fine-tune Training Sessions.

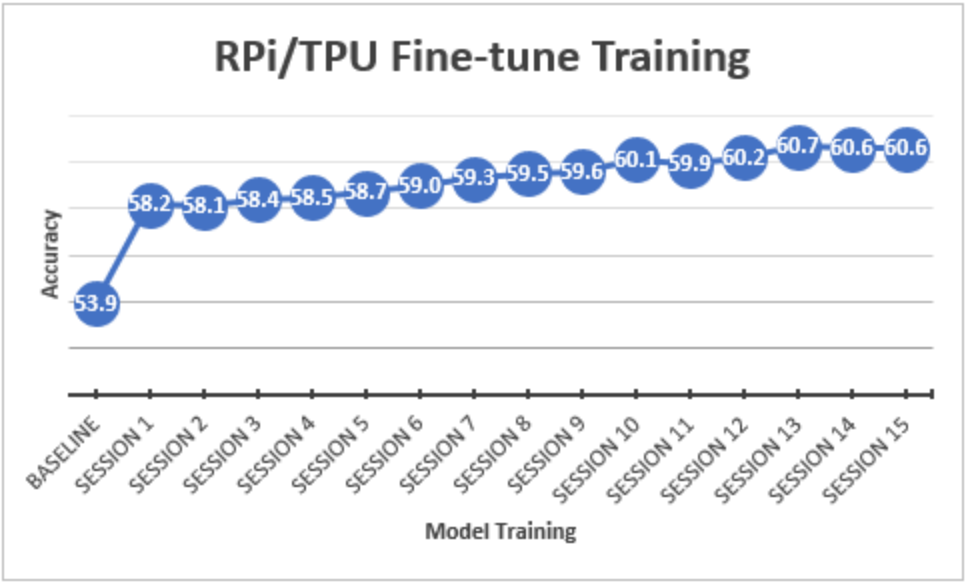


Figure 20. Training Results of second test from Initial Model Training through 15 Fine-tune Training Sessions.

For the third fine-tune training run, an early stop callback is added to each training session to stop training when the performance loss does not decrease by more than 0.001 for five epochs. Notice that the performance (see Figure 21) does not appear to be much different from when the model is run for the full 20 epochs. With the early exit callback, the training sessions regularly exited between 6–8 epochs. This greatly reduces the time to fine-tune the model and usage of the Advance RISC Machine (ARM) processor on the RPI. The model's performance improved from 53.9% to 60.6% for an improvement 6.7%.

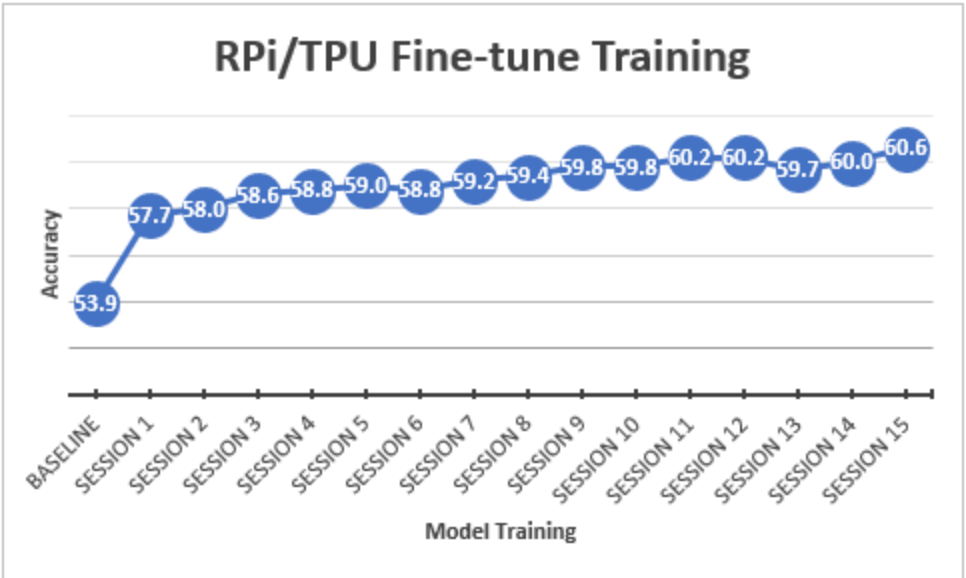


Figure 21. Training Results of third test from Initial Model Training through an average of 7 Fine-tune Training Sessions.

D. SUMMARY

The pre-trained results after converting the full convolutional machine learning model to tflite on the RPI and running inference on the Coral TPU were the same as on Google's Colaboratory. Note again that Google's Colaboratory was running TensorFlow v2.4 while the RPI was running TensorFlow v2.0 without degradation in model prediction. Fine-tune training on the RPI stayed consistent. The final test included an exit early callback when loss stopped minimizing which resulted in fewer epochs of training. Prediction results are consistent with training that did not exit early, while iterating over a smaller number of epochs. This technique helps to increase the speed of training.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND FUTURE WORK

This thesis explored fine-tune training of a Keras categorical classification convolutional neural network (CNN) model on a Raspberry Pi v4 (RPI) with a USB attached Coral Tensor Processing Unit (TPU) to show that a convolutional machine learning (ML) model can be implemented in such a way as to allow for fine-tune training on the native RPI Advance RISC Machine (ARM) processor and conduct inference on the attached Coral TPU. The main achievement is novel utilization of a hybrid ARM processor and TPU split training/inference architecture to overcome limitations of executing both training/inference on the same ARM processor. This is achieved by using a novel split network (SplitNet) model architecture that enables distributed computation across both ARM and TPU while not compromising real-time inferencing power of the TPU. Additional contributions involve significant embedded work in figuring out how to get the 32-bit RPI OS to work with TensorFlow v2.0. This was a significant piece of the work due to the incompatibility of a convolutional ML model implemented using TensorFlow v2.4 and the RPI OS running TensorFlow v1.15 when converting to TensorFlow Lite. The key to enabling fine-tune training of a Keras classification CNN model on a low-end edge device with limited resources is to split the model at a lower layer. This exposes an intermediate layer's output, which is then used as input to the upper layers of the original Keras classification CNN model. This was successfully demonstrated on a USB Coral TPU augmented RPI. The resulting improvement to the pre-trained model was realized as shown in Figures 19–21. Notice in Figure 19 where the accuracy starts to trend down at training sessions 4, 9, and 14. Each downward trend recovers and begins to increase with the introduction of new training data. This indicates that the model is capable of learning from real-world data collected by an edge device in the field which means this is a viable architecture for a network of sensors on the edge. On average, this SplitNet convolutional ML model was fine-tuned to 60.6% for an average performance improvement of about 7%. Opportunities exist to improve upon these results with better data splitting schema and more complex CNN models. Nevertheless, the results demonstrate the feasibility of SplitNet architecture and provides an initial quantification of such an approach.

The ability to conduct fine-tune training on a local edge device increases security by keeping all local data from being exposed through network transmissions to a centralized server that conducts fine-tune training. This also reduces unnecessary network traffic, allowing for lower bandwidth devices to be utilized in a network of sensors. Future work in this area of research is to continue the work towards implementing federated learn of a network of edge devices. This research would be the implementation of a network of TPU augmented edge devices that benefit from each device's local fine-tune training.

LIST OF REFERENCES

[1] Tech Briefs. “Smart Sensor Technology for IoT.” Accessed 14 September 2021 [Online]. Available: https://www.techbriefs.com/component/content/article/tb/pub/features/articles/33212

[2] arm, “Arm Architecture: A Foundation for Computing Everywhere.” Accessed 20 May 2021 [Online]. Available: https://www.arm.com/why-arm/architecture/cpu

[3] M. Baxter, “Centrally pretrained federated fine-tuning: enabling a secure and accurate military security application on embedded hardware,” M.S. thesis, Dept. of Comp. Sci., NPS, Monterey, CA, USA, 2020. [Online]. Available: http://hdl.handle.net/10945/66581

[4] Coral, “Transfer learning.” Accessed 6 October 2021. [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/#transfer-learning

[5] Coral, “Capability overview.” Accessed 15 December 2020 [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

[6] Tensor Flow, “Tensor Flow Core: Transfer learning and fine-tuning.” Accessed 6 October 2021. [Online]. Available: https://www.tensorflow.org/tutorials/images/transfer_learning

[7] J. Kim, Y. Park, G. Kim, S. Hwang, *SplitNet: Learning to semantically split deep networks for parameter reduction and model parallelization*. Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 2017

[8] Debugger Cafe, “Multi-head deep learning models for multi-label classification.” Accessed 6 October 2020. [Online]. Available: https://debuggercafe.com/multi-head-deep-learning-models-for-multi-label-classification/

[9] Tensor Flow, “Tensor flow core: The functional API.” Accessed 1 September 2020. [Online]. Available: https://www.tensorflow.org/guide/keras/functional

[10] B. McMahan and D. Ramage, “FL: Collaborative machine learning without centralized training data,” Google, April 6, 2017. [Online]. Available https://ai.googleblog.com/2017/04/federated-learning-collaborative.html

[11] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Aracs, “Communication-efficient learning of deep networks from decentralized data,” 2017. [Online]. Available: arXiv:1602.05629.

[12] mercury, “Enabling edge processing in military intelligent sensors .” Accessed 19 August 2021. [Online]. Available: https://www.mrcy.com/company/blogs/enabling-edge-processing-military-intelligent-sensors

[13] GitHub, “PINTO0309/Tensorflow-bin.” Accessed 1 May 2021 [Online]. Available: https://github.com/PINTO0309/Tensorflow-bin/

[14] Coral, " PyCoral API overview." Accessed 14 January 2021 [Online]. Available: https://coral.ai/docs/reference/py/

[15] A. Krizhevsky. "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings *of the IEEE*, 86(11):2278-2324, November 1998

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California