NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

CENTRALLY PRETRAINED FEDERATED FINE-TUNING: ENABLING A SECURE AND ACCURATE MILITARY SECURITY APPLICATION ON EMBEDDED HARDWARE

by

Matthew W. Baxter

December 2020

Thesis Advisor:	Marko Orescanin
Co-Advisor:	Gurminder Singh

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

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2020	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE CENTRALLY PRETRAINED FEDERATED FINE-TUNING: ENABLING A SECURE AND ACCURATE MILITARY SECURITY APPLICATION ON EMBEDDED HARDWARE	**5. FUNDING NUMBERS**
6. AUTHOR(S) Matthew W. Baxter	
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)

A persistent, precise, and adaptive security application is a requisite component to an effective force protection condition (FPCON) as U.S. military installations have become common targets for violent acts of terrorism and homicide. Current military security applications require a more automated approach as they rely heavily on limited manpower and limited resources. The current research developed an off-grid, deployed federated fine-tuning network composed of embedded hardware and evaluated embedded hardware system and model performance. Federated fine-tuning takes a centrally pretrained model and performs fine-tuning on a select number of model layers within a federated learning architecture. The federated fine-tuning models exhibited an average reduction in CPU load of 65.95% and an average reduction in current draw of 56.18%. The MobileNetV2 model transmitted 81.59% fewer global model parameters across the network. The centrally pretrained MNIST model began training with an initial accuracy improvement of 53.94% over the randomly initialized model. The centrally pretrained MobileNetV2 model demonstrated an initial average accuracy of 90.75% at training round 0 and experienced a 3.14% overall performance improvement after 75 federated training rounds. The results of the current research demonstrated that federated fine-tuning can improve system performance and model accuracy while providing stronger privacy and security against federated learning attacks.

14. SUBJECT TERMS machine learning, federated learning, TensorFlow, deep learning, force protection condition, FPCON	**15. NUMBER OF PAGES** 127
	16. PRICE CODE

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

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

THIS PAGE INTENTIONALLY LEFT BLANK

CENTRALLY PRETRAINED FEDERATED FINE-TUNING: ENABLING A SECURE AND ACCURATE MILITARY SECURITY APPLICATION ON EMBEDDED HARDWARE

Matthew W. Baxter
Lieutenant, United States Navy
BS, Northern Illinois University, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2020

Approved by: Marko Orescanin
Advisor

Gurminder Singh
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A persistent, precise, and adaptive security application is a requisite component to an effective force protection condition (FPCON) as U.S. military installations have become common targets for violent acts of terrorism and homicide. Current military security applications require a more automated approach as they rely heavily on limited manpower and limited resources. The current research developed an off-grid, deployed federated fine-tuning network composed of embedded hardware and evaluated embedded hardware system and model performance. Federated fine-tuning takes a centrally pretrained model and performs fine-tuning on a select number of model layers within a federated learning architecture. The federated fine-tuning models exhibited an average reduction in CPU load of 65.95% and an average reduction in current draw of 56.18%. The MobileNetV2 model transmitted 81.59% fewer global model parameters across the network. The centrally pretrained MNIST model began training with an initial accuracy improvement of 53.94% over the randomly initialized model. The centrally pretrained MobileNetV2 model demonstrated an initial average accuracy of 90.75% at training round 0 and experienced a 3.14% overall performance improvement after 75 federated training rounds. The results of the current research demonstrated that federated fine-tuning can improve system performance and model accuracy while providing stronger privacy and security against federated learning attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

1D	one dimension
ACL	access control list
AI	artificial intelligence
API	application programming interface
ARM	advanced RISC machine
AWS	Amazon web services
BERT	bidirectional encode representations from transformers
CA	certificate authority
CAC	common access card
CE-FedAvg	communication efficient federated averaging
CIFAR	Canadian Institute for Advanced Research
CNN	convolutional neural network
Conv2D	convolutional two dimension
COTS	commercial-off-the-shelf
CPU	central processing unit
CSV	comma separated values
CXR	chest x-ray
DL	deep learning
DNN	depthwise neural network
DoD	Department of Defense
EMNIST	extended modified National Institute of Science and Technology
FedAvg	federated averaging
FFT	federated fine-tuning
FL	federated learning
FLOPS	federated learning operations
FPCON	force protection condition
GAN	generative adversarial network
GPU	graphics processing unit
GroupNorm	group normalization
HTTP	hypertext transfer protocol

IC	integrated circuit
IoT	internet of things
KB	kilobyte
LoRaWAN	long range wide area network
LR	learning rate
LSTM	long short-term memory
MB	megabyte
ML	machine learning
MLP	multi-layer perceptron
MNIST	modified National Institute of Science and Technology
MQTT	message queued telemetry protocol
NLP	natural language processing
NWP	next word prediction
OEM	original equipment manufacturer
RAM	random access memory
ReLu	rectified linear unit
RPi	Raspberry Pi
SAR	system activity reporter
SGD	stochastic gradient descent
SoC	system-on-a-chip
SplitNN	split neural network
SVRG	stochastic variance reduced gradient
TLS	transport layer security

I. INTRODUCTION

A. MOTIVATION

On December 04, 2019, a U.S. Navy Sailor killed two Department of Defense civilians and wounded a third at Pearl Harbor Naval Shipyard, before taking his own life with a service pistol [1]. Over the past several years, military installations have become common targets for violent acts of terrorism and homicide. In order to counter potential threats against military installations an effective security posture is necessary. To ensure effective security on military installations, security applications must be persistent, accurate and adaptive to evolving threats. Current military security applications rely on limited manpower and physical resources and exhibit a need for automating persistence, accuracy and adaptiveness of the overall security system.

Artificial intelligence (AI) techniques are commonly used in image classification problems, such as video surveillance and traffic monitoring. However, once deployed these types of applications are static and not easily adapted to evolving classification problems without remote assistance. Emerging machine learning techniques, such as transfer learning and federated learning, make it possible for an image classification application to adapt and evolve to changing environmental conditions or change in the distribution of input features. It is proposed that machine learning can be integrated into a military security application in a way that supplements human tasks and improves the overall security posture of military installations.

B. PROBLEM DESCRIPTION

Base security systems capture and generate enormous amounts of private and sensitive data through base entry points and video security footage; for example, video footage that generates image data of vehicles and license plates, as well as common access card (CAC) readers that capture facial photos, DoD ID numbers, birthdays, etc. When any base security application is initially deployed, it must ensure that accurate information is provided to humans monitoring this data and that all data remains private and secure. This data is useful and could provide insights into identifying the pattern of life of base personnel, adversarial anomalies, and potential hostile acts. However, these types of security systems (i.e., closed circuit TV, CAC readers, human security guards) are typically stovepiped, and require extensive human intervention to be used to build a larger, more encompassing picture of the surrounding environment.

C. RESEARCH QUESTIONS

- What are the primary limitations and costs incurred in training a deep learning model on an edge device?
- How can these costs on embedded hardware be reduced through a federated learning architecture?
- How can federated learning be deployed on an end-to-end edge device network?
- How can a centrally pretrained state-of-the-art machine learning model be implemented on edge devices?
- What are the advantages to using a pretrained model over a randomly initialized model?
- How can on-device model fine-tuning be incorporated into federated learning on edge devices?

D. CONTRIBUTIONS

Three specific contributions are made in this work in developing a centrally pretrained federated fine-tuning model on edge device architecture in support of military installation security and insider threat detection:

1. Demonstrated and quantified the performance of federated learning in terms of edge device limitations (memory, computation, communication, and power).

2. Proposed, demonstrated and quantified performance of a more secure approach to federated learning through a pretrained MobileNetV2 model deployed on edge devices where only a few top layers are trained. In this manner, a reduced number of parameters are communicated making it difficult for an adversary to intercept wireless network traffic and reconstruct all of the model parameters, since most are never transmitted and remain hidden on the edge nodes.

3. Demonstrated an end-to-end deployment of federated learning on an edge device network, with all tasks performed by edge devices.

E. THESIS ORGANIZATION

Chapter II defines foundational machine learning concepts, transfer learning, and federated learning. It also includes federated learning-related work.

Chapter III discusses the federated learning architecture design and overall research methodology.

Chapter IV reviews the results and analysis of the experimentation, including metrics, findings, performance, and accuracy.

Chapter V covers system limitations and possible enhancements. It concludes with the contributions of the thesis work and lists possible future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. TECHNICAL BACKGROUND

This chapter explores several key technical concepts relevant to machine learning, deep learning and federated learning that were utilized throughout the thesis process. First, an overview of machine learning, deep learning and its applications on edge devices. Next, a discussion of machine learning models, frameworks and techniques used throughout the experimentation process. Finally, a discussion of military security and potential applications of federated learning of video surveillance within a secured military installation.

A. MACHINE LEARNING OVERVIEW

Machine learning is a subfield of artificial intelligence, which started in the 1950s by computer science pioneers that sought to understand if computers could automate intellectual tasks typically conducted by humans [2]. Artificial intelligence is a general field in computer science that encompasses both machine learning and deep learning. Machine learning can be described as computers finding patterns in data to create algorithmic models for prediction. Common applications of machine learning models include predictions of internet activity patterns, social networks, ecommerce, advertising and healthcare [2]. Deep learning can be described as allowing computers to learn from experience by building a hierarchy of concepts describing the world, where each concept is defined through its relation to a simpler concept [3] (see Figure 1). Building and gathering knowledge through experience means this approach does not require human input to specify all the knowledge that the computer needs.

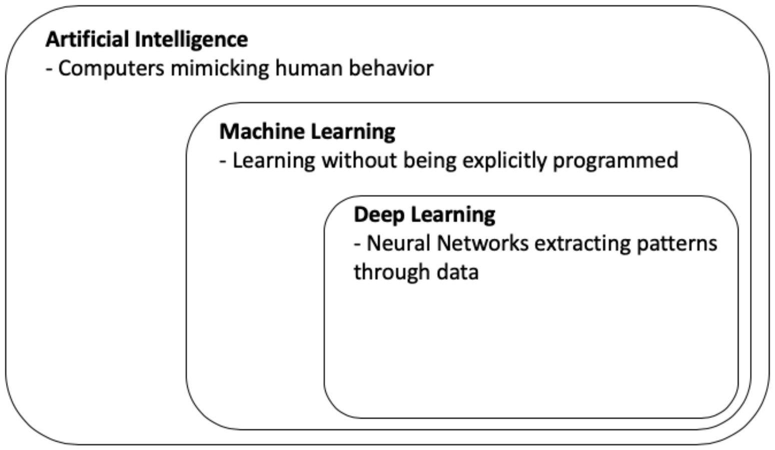

Artificial Intelligence
- Computers mimicking human behavior
Machine Learning
- Learning without being explicitly programmed
Deep Learning
- Neural Networks extracting patterns through data

Figure 1. Artificial Intelligence, Machine Learning, and Deep Learning.

1. Machine Learning

Traditional models within machine learning include supervised learning (algorithm has access to labeled data), unsupervised learning (algorithm has no access to labeled data), and semi-supervised models (some labeled data is available to the algorithm). In supervised learning models are trained with input labeled data and tasks solved are broadly regression and classification. In unsupervised learning the goal is to find patterns in a dataset and a common task is clustering of data to discover classes. In semi-supervised learning algorithms are developed on partially labeled data when unlabeled data is freely available and labeled data is expensive to obtain [4], [5].

Another classification of machine learning algorithms is based on whether the model must be trained using all of the data (batch learning) or if the model can incrementally learn on the fly through continuously fed data (online learning). Primary challenges of machine learning include insufficient training data (machine learning algorithms require extensive data) or poor-quality data (data with significant errors, outliers or noise) reflected in issues with overfitting and underfitting. Overfitting occurs during model training when a model is fit so closely to training data that the model fits poorly to new data. Underfitting occurs during model training when the model fails to capture the intricacy of the training data [4], [5].

2. Deep Learning

Deep learning is a subfield of machine learning that applies hidden layers between the input layer and output layer to extract features from data and transform the provided data into different representation levels [2]. It is commonly used for computer vision, next word prediction, and speech recognition applications. Deep Learning is an iterative machine learning process that typically involves four steps executed sequentially during model training—gather a batch of sample training data, perform a forward pass through the layers of the neural network, execute a loss function evaluation, and perform a backpropagation calculation of the parameter error with a weights (parameter) update [2].

At the start of the deep learning training process, weights are typically randomly initialized, which results in random transformations as the data passes through the network. Through each iteration of the training process, the weights are adjusted and the loss score decreases. Training stops when the loss, the difference between the predicted and target value, ceases decreasing [2] (see Figure 2).

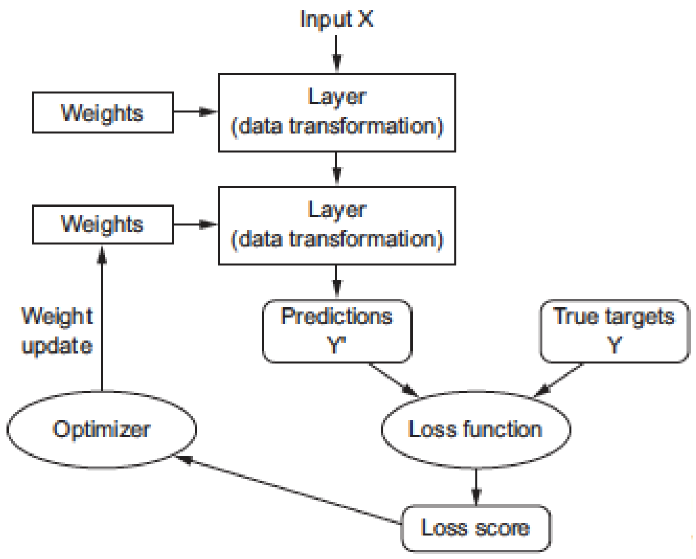


Figure 2. Deep Learning Training Cycle. Source: [2].

a. Cross Entropy Loss

Cross entropy loss is used throughout machine learning applications as a loss function for classification problems. The purpose of a loss function is to control the output of a neural network by measuring how far predicted output is from the actual or target output. The deep learning training loop seeks to identify weight values that minimize the loss function and produces outputs that are as close to the targets as possible. Common loss functions supported by TensorFlow/Keras include binary cross entropy, categorical cross entropy and sparse categorical cross entropy. Binary cross entropy is used when there are only two class labels (typically 0 and 1), with each example having a single floating-point value for each prediction. Categorical cross entropy is used when there are two or more label classes. Labels are provided as one-hot encoding, where a sparse vector has one target element set to 1 and all other elements set to 0. Sparse categorical cross entropy is used when there are two or more class labels and labels are provided as integers [2].

3. Deep Learning on Small, Low-Powered Edge Devices

In recent years, there has been a rise of interest in deployable machine learning technology, specifically deep learning, for internet of things (IoT) sensors and edge computing applications, such as image classification, image detection, anomaly detection, keyword spotting and next word prediction. However, IoT and edge devices generate large amounts of data that must be processed and often rely on central cloud servers to aggregate and process data. Concerns with implementing deep learning models on IoT devices include increased latency, decreased battery life of devices from high communication costs and privacy concerns if sensitive data is routinely transmitted.

a. Overview of Deep Learning Applications on Edge Devices

The prevalence of edge and mobile device sensors, such as cameras, has greatly increased the importance of image recognition. Deep learning techniques, such as convolutional neural networks (CNN), have been shown to identify people, handwriting and objects with high accuracy. Traditionally, data resided on a cloud server for processing, but edge devices have been used more and more to process images [6]. Multiple testbed image datasets are included with the TensorFlow API that are deployable for neural network implementation on edge devices with TensorFlow or TensorFlow Lite.

Real-time video is a critical sensor in IoT and edge devices that range from self-driving cars, to traffic safety, and surveillance. Until recently, accurately identifying objects from low-quality edge device video data had proven difficult. The computational capabilities of an edge device are a limiting factor in the edge devices ability to process camera images quickly. Qi and Liu used a quantized deep learning model with an integrated graphics processing unit (GPU) on a Nvidia Jetson TX2 and ARM processor to reach real-time video processing speed [7]. They reduced, quantized, the CNN parameters to 16-bit float and applied pruning techniques to improve deep learning model deployment on edge devices.

Image classification for medical imaging has yet to be proven accurate enough for automatic recognition in clinical use due to the variations in medical imaging—such as poor image quality, a variety of medical imaging protocols, and previously unseen variations in patients (i.e., zero-shot learning). However, it has proven useful for interactive recognition that incorporates human-in-the-loop approach. Wang et al. implemented a deep learning interactive segmentation framework in which the user selected a bounding box for images and scribbles. The bounding box allowed the user to select the image they wished to evaluate, and the scribbles were used for medical annotations. Their methodology was more robust than previous medical imaging applications and allowed for human intervention enabled fine-tuning of the model to improve accuracy [8].

Automatic speech recognition is rapidly developing due to smartphones and tablets that interact with technology through speech. However, interest is growing in the development of offline speech recognition systems with all training occurring on the device, with no reliance on cloud processing. This process involves limited-vocabulary speech recognition—one method is known as keyword spotting. The majority of devices stream audio to cloud servers for processing; however, activation of these devices typically relies on keyword spotting on-device, such as “Alexa” or “Hey Google.” Tucker et al. found that they could reduce false alarms and misses without increasing CPU usage by improving acoustic neural models with low-rank weight matrices and an ensemble of neural networks used during training [9].

b. Challenges in Deep Learning on Small Devices

Deep learning models rely on a large number of parameters, which incur a high computational cost and require a large amount of memory on the device. MobileNet is a relatively small model with 4,253,864 total parameters, while a much larger model like VGG16 has 138,357,544 total parameters [10]. Edge device sensors, such as video cameras and environmental sensors can generate enormous amounts of data, which has traditionally been transferred to the cloud for further processing. Deep learning is being utilized more and more to approach this problem of extracting edge device data in noisy and complex environments without the need for cloud processing [11].

Limiting factors of resource-constrained edge devices include memory, computational capability, communication costs and energy constraints. Random access memory (RAM) on edge devices can range from 512 MB to 8 GB with non-volatile memory commonly accessed via removable memory (e.g., micro SD card). Central processing units (CPU) can range from 160 MHz to 1.5 GHz with more advanced system-on-a-chip (SOC) boards including an integrated GPU (e.g., Nvidia Jetson Nano). IoT devices are restricted in their functionality due to memory and computational constraints and require communication with a central device to transfer data and to receive operating instructions. Commonly used IoT networking protocols include Bluetooth, Zigbee, LORAN and MQTT. Deployed edge devices must minimize computational and high communication costs in order to consume power efficiently and ensure maximum uptime with minimal interruption. While machine learning inference and minimal model training was demonstrated on, the primary limiting factors of edge devices are in general preventing training of deep learning models on these devices.

B. MACHINE LEARNING FRAMEWORKS/LIBRARIES

Several machine learning frameworks have board support packages for deployment on edge devices—Caffe/PyTorch, MXNET, TensorFlow, and TensorFlow Lite. A machine learning framework is a library that makes developing machine learning applications easier for users. Other common machine learning frameworks deployed on edge devices include—Theano, ML Kit (Google), and Core ML2 (Apple) [4]. There are numerous deep learning frameworks available, each with their own characteristic functionality and support for deployment on edge devices.

1. TensorFlow

TensorFlow is an open-source, large-scale, distributed machine learning platform for numerical computation on dataflow graphs. At TensorFlow's core is optimized C++ code executing a Python computational graph. To increase efficiency, Tensorflow can break up a graph into chunks to be run in parallel on multiple CPU's or GPU's. Distributed computing is supported such that multi-million parameters neural networks can be split and trained across multiple servers [3]. In conjunction with Keras (a high-level API supporting TensorFlow), TensorFlow allows for easy machine learning model building and training. TensorFlow allows for model deployment on-device, in a browser or in the cloud, regardless of programming language. TensorFlow provides excellent support for embedded devices and a defined, clear path to deployment on edge devices through TensorFlow Lite and the TensorFlow Edge TPU API. Multiple chip OEMs support TensorFlow Lite, such as the Qualcomm Snapdragon SoCs, Arduino Nano 33 BLE, SparkFun Edge, and Espressif ESP32-DevKit [12], [13].

2. TensorFlow Lite

TensorFlow Lite is designed to convert and run TensorFlow models on mobile, embedded and IoT devices. TensorFlow Lite workflow steps include: choice of a model, converting the model, running inference with the model, and optimizing the model for deployment to an edge device. A full TensorFlow model must be used for conversion into a TensorFlow Lite format, TensorFlow Lite cannot create or train a model. The TensorFlow Lite converter reduces the file size, provides optimization that does not affect accuracy, and increases speed of execution. The TensorFlow Lite interpreter is a library that executes operations on input data and provides access to the output from the TensorFlow Lite model. The TensorFlow Lite converter also support quantization by reducing TensorFlow 32-bit integers to 16- or 8-bit integers without significantly affecting accuracy [14].

C. MODERN MACHINE LEARNING MODELS

Deep learning models can include millions of parameters, which limits resource-constrained edge devices in their ability to train solely on the device itself. Much effort has been put into the development of small and efficient convolutional neural networks that are deployable to mobile and edge devices. These efforts typically involve model compression techniques, such as quantization, hashing or pruning. Another technique involves directly trained small networks, common "small" networks that have been developed are SqueezeNet, EfficientNet, MobileNetV1 and MobileNetV2 [15], [16].

Howard et al. developed MobileNet, a convolutional neural network architecture, that minimizes latency of smaller-scale networks to run on edge devices. MobileNet uses depth-wise separable convolutions to construct a streamlined, lightweight deep neural network. Depth-wise separable convolutions are more computationally efficient than standard convolutions by factorizing a 3D convolution into two separate convolutional operations. The use of depth-wise separable convolutions enables MobileNet to be 32 times smaller than a traditional model like VGG16 and 27 times less computationally intensive, while only reducing accuracy by 1% [17]. Nikouei et al. improved inference on MobileNet by developing a lightweight CNN that is capable of detecting pedestrians in a real-time human surveillance system on a Raspberry Pi 3 [18].

D. TRANSFER LEARNING

Transfer learning is a machine learning technique that decreases training time and computational costs by leveraging previously pre-trained models, such as the MobileNetV2 architecture on the ImageNet dataset and repurposes it for a task it was not originally trained for. When deep learning neural networks are trained on images the first few layers of the model always resemble the same low-level features (e.g., visual edges, colors, and textures), while the final layers in a neural network are specific to the dataset and the specific machine learning task (see Figure 3). In transfer learning the base layers serve as a foundation for a new machine learning model and the "general" features learned during the base layers are transferred and trained on a new "specific" machine learning model [19].

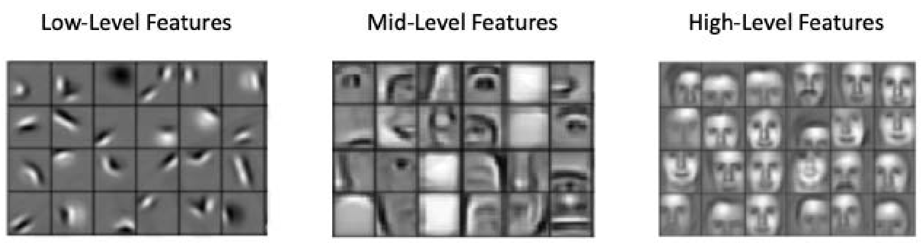


Figure 3. Image Features by Network Layer Depth. Source: [20].

1. Feature Extraction

There are two primary methods of transfer learning from a pretrained network: feature extraction and fine-tuning. Feature extraction takes the representations learned by a previously trained network to extract useful features from new samples by taking the convolutional base and running new data through it to train a new classifier on top of the base. The lower layers of the convolutional bases are likely to learn general generic feature maps of an image (such as visual edges, colors and textures). This allows the early layers to be easily repurposed, while the final fully connected layers can be specific to the new task of the classifier [2] (see Figure 4).

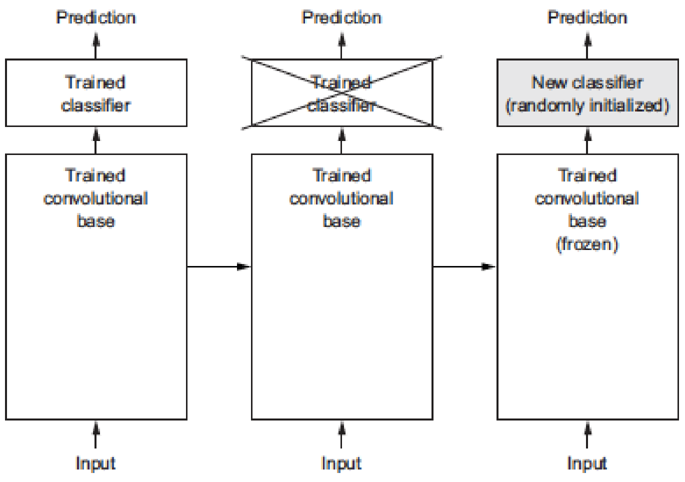

Prediction
Prediction
Prediction
Trained classifier
~~Trained classifier~~
New classifier (randomly initialized)
Trained convolutional base
Trained convolutional base
Trained convolutional base (frozen)
Input
Input
Input

Figure 4. Feature Extraction with a New Classifier Trained on Top of the Convolutional Base. Source: [2].

2. Fine-Tuning

Fine-tuning improves performance further by releasing some of the model parameters in the layers of the base model for training (known as “unfreezing”) and jointly trains the base layer and the classifier that has been added to the convolutional base. Fine-tuning can slightly refine the more abstract representations of the convolutional base, in order to make it more specific to the new task. The general steps to fine-tune a network involve 1) adding a new classifier on top of a pretrained network, 2) setting the convolutional base to non-trainable, 3) training the new classifier, 4) making some of the layers in the convolutional base trainable, and 5) training the entire network to include the added classifier [2] (see Figure 5). Feature extraction and fine-tuning are powerful techniques that allow for training accurate models with small training datasets, otherwise impossible task if one was to train from randomly initialized model.

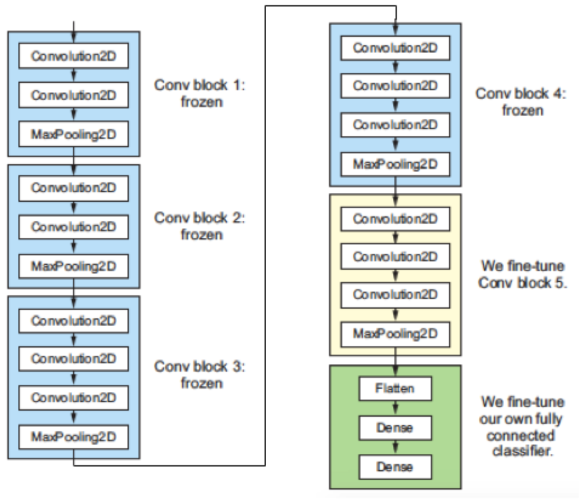
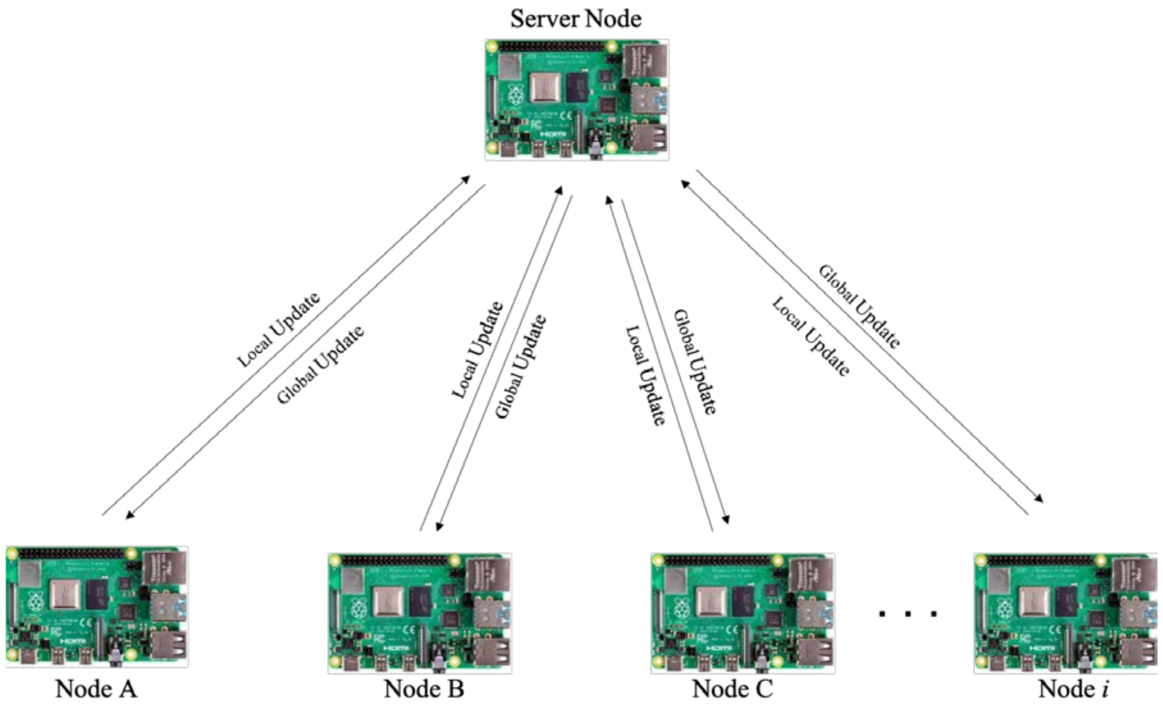


Figure 5. Fine-Tuning with the Last Convolutional Block of VGG16.
Source: [2].

E. FEDERATED LEARNING

Federated learning is a distributed approach to machine learning in which private client data residing on edge devices is completely decoupled from the training of the machine learning model and never transmitted off the edge device. In federated learning, clients use private local data to train a global model and send the updated parameters to a central server. The central server aggregates and averages the parameters to generate an updated global model that is sent to clients. Once global parameters are aggregated, averaged and sent to edge devices, the central server discards the previously aggregated weights [21] (see Figure 6). Commercial approaches to federated learning commence model training with a randomly initialized model that improves through many successive rounds of training. Utilizing this methodology requires a large amount of training rounds to achieve a model with acceptable accuracy.



Federated learning steps: A) Edge device trains model locally with private, local data, B) Edge device local updates are sent to the server, C) local updates are aggregated to form a new global update, D) Global update is sent to edge devices and the process repeats.

Figure 6. Federated Learning Overview.

Advantages of federated learning approaches compared to a conventional distributed cloud-centered machine learning framework include: efficient use of bandwidth, data privacy since labeled training data is never transmitted to the server, and low latency resulting from model training occurring on the edge devices. In a federated learning architecture, less information is required to be transmitted to a central server resulting in a reduction of communication costs. Participating nodes only need to send updated parameters for aggregation rather than raw data, which significantly reduces communication costs. Assuming that participating nodes are non-malicious, user data is kept private as it resides locally on the end device and is never sent across the network. A federated learning scheme improves latency as inference can occur directly on the device as opposed to a remote cloud server. Traditional approaches depend on cloud services to process data and make inferences, while end nodes in a federated learning network can perform real-time execution on device [21], [22].

1. Federated Learning Related Work

Konecny et al. developed the federated SVRG (stochastic variance reduced gradient) algorithm as a practical alternative to traditional approaches to the federated optimization problem [23]. The federated optimization problem arises due the fact that as data rapidly increases, a single node cannot store an entire dataset. This requires a distributed computational framework, in which the training data is distributed across a cluster of nodes. During each round of federated learning the federated SVRG algorithm performs a full gradient computation on the server node, all clients downloading the new global model, followed by several distributed stochastic gradient descent (SGD) updates by each client, and SGD client updates shared with the server to be aggregated to form an updated global model. Konecny et al. determined that federated SVRG is computationally expensive and therefore most applicable for sparse convex problems and not neural networks since they yield non-convex functions [23].

McMahan et al. developed the FedAvg algorithm as a practical solution for federated learning that is based on iterative model averaging [22]. Their federated learning scheme starts with the server deploying a randomly initialized model and distributing hyperparameters (number of epochs per round, batch size, learning rate and learning rate decay) to a fraction of the clients. The clients train the global model received from the server with their local data and send the updated weights back to the server. The server averages all received weights and repeats the process with a new fraction of clients.

McMahan et al. used two different neural network architectures for experimentation—an MNIST 2NN with two hidden layers and MNIST CNN with two 5x5 convolutional layers. Their work indicated that FedAvg can train high quality models within a relatively small number of federated learning training rounds [22].

Caldas et al. expanded on the work of McMahan et al. by developing LEAF, a benchmark for federated learning settings. LEAF is a modular benchmarking framework that includes a suite of publicly available federated datasets, an evaluation algorithm, and a set of reference implementations focused on identifying federated learning obstacles [24]. They currently include the following open-source datasets for benchmarking—EMNIST (image classification), Shakespeare (next character prediction), Twitter (sentiment analysis), CelebA (image classification), Synthetic Dataset (classification), and Reddit (language modeling). Within their framework, the client nodes are simulated and not intended for embedded deployment. Their evaluation metrics within LEAF included number of FLOPS (federated learning operations), the number of bytes downloaded/uploaded, and weighted accuracy across devices (e.g., determining if each device is equally important in the network). They demonstrated that their open-source datasets were modular and able to be incorporated into additional simulated experimental pipelines [24].

Hard et al. successfully trained a recurrent neural language model that used federated averaging for next-word prediction on the Google Gboard [25]. They found that their randomly initialized next-word prediction federated learning model outperformed an identical server-trained next-word prediction model. Yang et al. used federated averaging in a commercial, global-scale setting to train, evaluate and deploy a federated learning GBoard keyboard search suggestion model without directly accessing local user data [26]. The model setup included two stages—a server-side baseline model to generate keyboard query suggestions and a federated learning triggering model that removed low quality queries suggested by the server baseline model. Their work was one of the first successful end-to-end examples of federated learning deployed in the real-world [26].

Nilsson et al. benchmarked three federated learning algorithms (federated SVRG, FedAvg, and CO-OP) and compared their performance against a traditional centralized approach to distributed machine learning frameworks that rely on a central server for data storage [27]. Using McMahan's MNIST 2NN model as a baseline, they identified that FedAvg performed the best with comparable results to the traditional approach. However, they identified that FedAvg did not perform as well with non-i.i.d. (independent and identically distributed) data [27].

Bonawitz et al. identified several challenges and solutions to building a scalable system for federated learning [28]. Federated learning converges slower than traditional ML designs and increased parallelism of clients would decrease the convergence time of a federated learning model. Another limitation of federated learning is that clients may not have new data to train on and when called upon by the server and they will be training on previously seen data, which requires device scheduling to ensure that only new data is used for training. They determined that even though federated learning does not require user data to be communicated, uploading local model updates still requires a significant communication cost and compression techniques will be important to bring federated learning to production [28].

A large volume of research on federated learning utilizes random initialization of the models to begin the federated learning process. However, this paradigm requires a large number of rounds to reach convergence. Starting with a pre-trained model and using transfer learning to improve the model would reduce the number of rounds for convergence. Stremmel and Singh found that a pretrained word embedding model converged faster than a randomly initialized word embedding model across 1,500 rounds of training [29]. Their LSTM neural network consisted of four layers, nearly eight million trainable parameters, and 31.3MB in size. They did not find that using a pretrained model exceeded performance of the randomly initialized federated averaging approach; however, they did demonstrate that pretraining provides an initial boost in accuracy over random initialization [29].

Gao et al. investigated federated learning and SplitNN (split neural network) on edge devices to compare learning performance and device overhead [30]. SplitNN is a federated learning method in which a neural network is split into two sections vertically. The first few layers are on the IoT device and the remaining layers reside with the server (e.g., cloud). The client and server cooperatively train the entire network. Their dataset consisted of sequential time-series data and the model architecture had four 1D CNN layers and two dense layers. The first two 1D CNN layers were trained on the Raspberry Pi 3B and the remainder of the model trained on the server (laptop). They determined that FL was a more practical recommendation for an IoT architecture and state of the art models could not be trained on resource-constrained edge devices [30].

Liu et. al investigated recognition of COVID-19 pneumonia CXR images and compared four machine learning models within a federated learning framework [31]. One of the models they utilized was a MobileNetV2 model. All of their experimentation was simulated with all virtual clients trained on one machine using an NVIDIA GPU. They determined that ResNeXt (similar to ResNet18) achieved the highest performance in classification of COVID -19 chest x-ray images [31].

Liu and Miller demonstrated that a bidirectional encode representations from transformers (BERT) model could be pretrained and fine-tuned in a federated manner [32]. BERT has been developed for natural language processing (NLP); however, their research shows it is possible to pretrain and fine tune within a federated setting.

Hsu, Qi and Brown analyzed two large-scale real-world datasets (species and landmark classification) for real-world problems in a federated setting [33]. They applied a virtual client scheme with 10 clients selected every federated round. A MobileNetV2 model with a GroupNorm layer and softmax classifier was pretrained on ImageNet. Their experimentation demonstrated that large-scale visual classifiers can be trained using a federated approach. Through their research, they determined that federated learning with pretraining required fewer communication rounds than training from random state to achieve a high accuracy [33].

Executing multiple rounds of training with various hyperparameters on resource constrained edge devices is cost prohibitive. Federated learning adds additional hyperparameters to the tuning process, such as training rounds, number of clients per training round, global model update algorithm rules, etc. Kairouz et al. identified hyperparameter tuning as an open problem in federated learning [34]. Khodak et al. were one of the first to analyze hyperparameter tuning within federated learning and developed FedEx as a method to enable federated learning hyperparameter tuning for a variety of federated learning algorithms [35].

Mills, Hu and Min adapted the FedAvg algorithm with an adam optimizer and compression to produce communication-efficient federated averaging (CE-FedAvg), which reduced the total data uploaded to the server and reduced the number of training rounds when compared to similarly compressed FedAvg [36]. They demonstrated that they could reach a target accuracy in up to 6x fewer rounds than FedAvg. Additionally, they implemented their experiments on 10 RPi with a desktop computer acting as a server over a wireless network. They determined that the server work was small and had a minimal impact on training time, with the RPi requiring a majority of the training time. Their edge device network was able to reach a target accuracy in up to 1.7x less time than FedAvg [36].

Das and Brunschwiler demonstrated the feasibility to train deep neural networks on Raspberry Pi as edge devices. They trained a CNN, LSTM, and MLP on the MNIST dataset [37]. They determined that the CNN could achieve 85% accuracy within two minutes of training, while exchanging less than 10MB of data per edge device. Their CNN consisted of two Conv2D layers, one max pooling layer and one fully connected layer with 47,000 total parameters. Their MLP was comprised of three Fully Connected Layers and had 1,700,000 parameters. Their network consisted of five Raspberry Pi and a MacBook Pro as the central server. Their research also indicated that 95% accuracy could be achieved within six federated training rounds with additional epochs per training round on each device [37].

a. Federated Learning Attacks and Security Vulnerabilities

Multiple adversarial attacks against federated learning have been identified, including data poisoning, model update poisoning, and model evasion attacks [34]. Federated learning has introduced new attack surfaces within adversarial machine learning since the datasets and model training are distributed across a network. Data poisoning occurs when an attacker cannot directly corrupt the server node, so they manipulate client data to corrupt the global model [34]. Model update poisoning typically occurs when an attacker can directly alter the output of the clients to bias the local model update towards their objective. Common methods to protect against adversarial attacks on federated learning schemes include encryption, accuracy checking, and weight update statistics [34].

Another security concern with federated learning is the ability to reconstruct valuable model data from the parameters shared between the clients and server node. Shokri et al. demonstrated that they could determine if an output was a member of the model's training set by only using information leaked by the machine learning model [38]. Hitaj, Ateniese and Perez-Cruz developed a generative adversarial network (GAN) that was able to exploit federated learning models and generate prototypical samples of the target's private dataset [39]. A requirement of their approach to attack the federated learning model relies on local federated learning nodes improving accuracy over time. They also demonstrated that their GAN attack is successful against common security techniques, such as differential privacy or other common obfuscation methods. However, they acknowledge that a model only releasing a portion of the global parameters provides stronger privacy and thwarts their attack [39].

2. Federated Fine-Tuning on Edge Devices

Previous research was identified that implemented various federated fine-tuning techniques; however, all of the identified research was simulated and not actually deployed to edge devices. Federated fine-tuning is a machine learning technique that takes a centrally pretrained global model with desired accuracy and then deploys the pretrained model to edge devices to be trained on the device's private local data incrementally through iterative fine-tuning training rounds. This scheme has the potential to reduce the limiting factors of edge devices (memory, computation, communication, and energy costs), while enabling a network of edge devices to train a complex deep learning model that was traditionally outside the scope of edge device capabilities. Federated fine-tuning may reduce:

- Memory limitations by distributing the dataset across multiple nodes and minimizing the RAM necessary to support training a deep learning model.

- Computational limitations by reducing on device CPU load through minimal training rounds.
- Communication costs by starting with a pretrained trained model that requires a limited number of training rounds to achieve high accuracy and only requiring a portion of the global model to be shared.
- Energy costs by minimizing the memory, computational and communication costs necessary to conduct on device training of a deep learning model.

F. POTENTIAL MILITARY INSTALLATION APPLICATIONS AND IMPROVED SECURITY

Deep learning technology has facilitated the automation of surveillance and insider threat networks that were traditionally operated by humans, with high accuracy in identification and anomaly detection in real time [40]. However, these systems are not typically designed to evolve after deployment and require a central cloud server for large datasets or additional model training. Federated learning technology provides a framework for machine learning models to evolve and adapt after deployment and allows for large datasets to be distributed across multiple nodes.

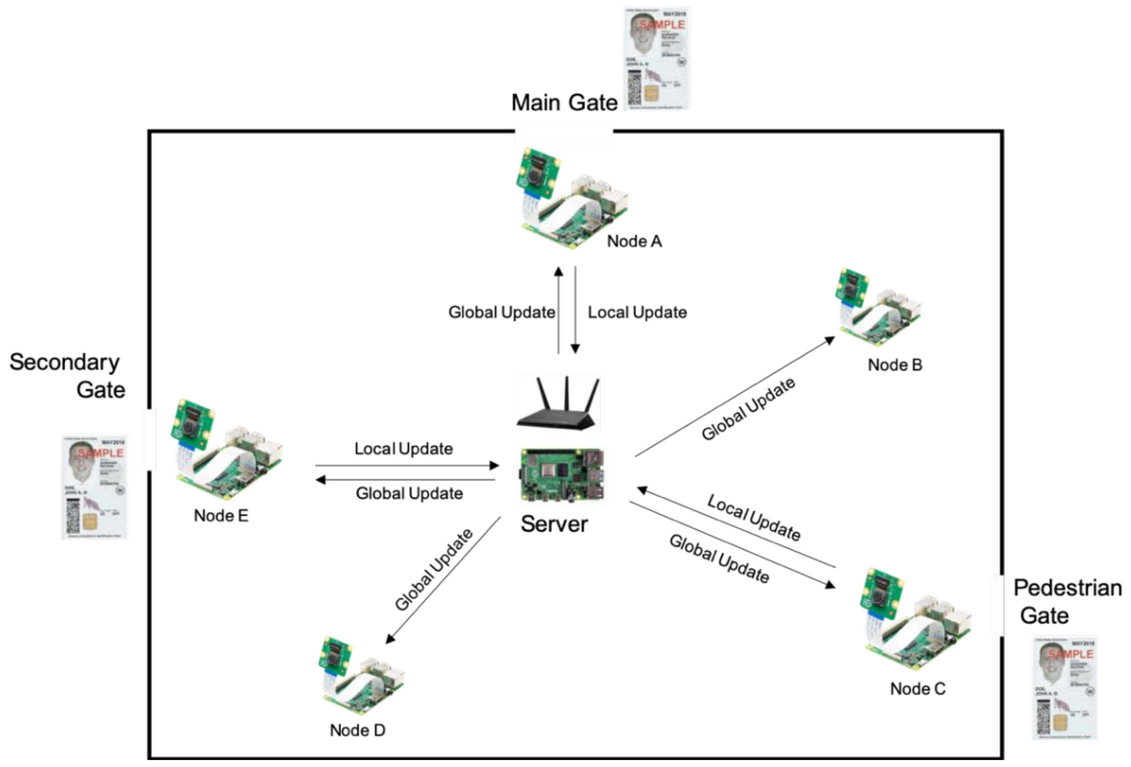
Many federated learning approaches within the commercial setting utilize randomly initialized machine learning models that improve over a large number of iterative training rounds. In contrast, DoD security applications must be accurate, adaptive upon initial deployment of the architecture, and protect sensitive data collected on military installations. A centrally pretrained federated learning architecture places an emphasis on model performance at the time of deployment, security of the global model parameters, and optimization of edge device performance. It accomplishes this through distribution of the dataset, distribution of computational costs, and a minimization of edge device limitations.

Although private data is not transmitted in a federated learning framework, it is still possible for adversaries to reconstruct the raw data from the global parameters that are transmitted. Federated learning can expose training results, such as parameter updates from an SGD algorithm, and leak private information when combined with a data structure (e.g., image pixels). Given these risks, federated learning needs to safeguard the full global model during communication with the central server and ensure communication occurs as few times as possible.

Transfer learning is able to leverage a previously pre-trained model with high accuracy to support a new task it was not trained for. In conjunction with transfer learning, federated learning allows a distributed network architecture to incrementally improve while ensuring that sensitive data remains on the device and is never transmitted across the network. Combining transfer learning and federated learning can support military security and insider threat systems in deployment of a highly accurate model that will continue to improve throughout its lifetime.

Federated fine-tuning addresses security risks of the full global model since only a small number of parameters are shared. In traditional federated learning parameters of the global model are shared, but in federated fine-tuning only a portion of the parameters of the global model are shared. Employing federated fine-tuning addresses security risks on the global model since only a select number of global parameters are shared with a majority of the global parameters remaining hidden on device. Thus, making it difficult for an adversary to intercept the shared parameters when transmitted and reconstruct the full global model [39].

There is a need for more complex models and networks designed for vision tasks to be deployed in support of military installation security. Military installation security applications have the advantage of leveraging persistent security footage and CAC information to identify an individual or vehicle. This information can be used to label previously unseen data on the fly to improve the accuracy of the security system. It is feasible that future applications of military installation security implement a centrally pretrained federated fine-tuning model to ensure persistence, accuracy and adaptability. In this model, some of the nodes within the architecture would serve as primary client nodes and perform federated fine-tuning in conjunction with inference on the data stream. Secondary client nodes would support image inference, anomaly detection, and support additional tasks as demanded. The secondary client nodes would not have access to CAC data or perform federated learning, but would still monitor for security anomalies through the shared global model and send alerts requesting human analysis and follow-on training. The server node would provide local model aggregation and global model distribution. If the primary server node is compromised or the network experiences degradation, a minimally tasked secondary client node could undertake the role of the server. The workload of the server is minimal enough to be supported by the secondary client nodes (see Figure 7).



Nodes A, C and E are primary client nodes, Nodes B and D are secondary client nodes, and the server node coordinates local model aggregation and global model distribution. Primary client nodes can use labeled data, such as CAC information, vehicle license plates, etc., for federated training.

Figure 7. Notional Federated Learning Base Security Architecture.

G. SUMMARY

Over the last few years, deep learning has become an important implementation in edge devices in support of real-time video, image classification, medical and smart home advancements. With deep learning applications expanding, they are likely to proliferate in military applications as well. Deep learning models are well suited to process the large amounts of data generated by edge devices and sensors. However, the primary limitations of commercial-off-the-shelf (COTS) edge devices—memory, computational, communication and power costs—have been unable to support the high costs of training an accurate deep learning model on device. It is proposed that a distributed network of edge devices can maintain an accurate deep learning model while addressing global model security risks through a centrally pretrained federated fine-tuned deep learning model.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTAL DESIGN AND SYSTEM SET-UP

This chapter describes the six experiments conducted to support the findings in this thesis, the datasets used, the machine learning models developed for the experimentation, the federated averaging algorithm utilized, networking protocol used, hardware setup for experimentation, and performance tests employed during experimentation.

A. THESIS EXPERIMENTS

Six experiments were designed to evaluate deployment of federated learning utilizing TensorFlow on a COTS edge device architecture and to analyze how performance is impacted as an edge device federated learning architecture increases in complexity (see Table 1).

1. Single Node Centrally Trained

Utilizes a randomly initialized MNIST CNN and serves as the baseline for training a deep learning model on an edge device (see Table 2). All training and evaluation occurred on one edge device. Experiment I is not a test of federated learning, rather it is a baseline to compare performance costs and potential gains when implementing a federated learning scheme on edge devices.

Table 1. Overview of Experiments I through VI.

Experiment	Objective	Dataset	Parameters Shared
I. Single Node Centrally Trained	Baseline DL model trained on one edge device to identify edge device costs and limitations incurred when training a DL model.	MNIST	0%
II. Randomly Initialized Federated Learning	Multi-node federated learning CNN architecture to identify how edge device costs and limitations are reduced in a multi-node edge device architecture.	MNIST	100%
III. Centrally Pretrained Federated Fine-Tuned	Multi-node federated learning architecture where weights are pretrained on a central server with only a select number of parameters shared for federating averaging in order to reduce edge device costs/limitations and improve security.	MNIST	40.26%
IV. Extended Class Centrally Pretrained Federated Fine-Tuned	Multi-node pretrained federated learning architecture with a more complex classification problem over MNIST. EMNIST is TensorFlow's recommended federated learning testbed dataset.	EMNIST	59.48%
V. MobileNetV2 Centrally Pretrained Federated Fine-Tuned	Multi-node pretrained federated learning architecture utilizing a state-of-the-art model. This experiment analyzed the impacts of local dataset size, type of centrally pretrained model and model layer depth from which to conduct federated fine-tuning.	CELEBA	18.42%–39.61%
VI End-to-End FedAvg Edge Device Network	End-to-end multi-node pretrained federated learning architecture utilizing a state-of-the art model. All hardware is composed of battery-powered edge devices. Includes a secondary client node for predictions and anomaly alerts for accuracies below specified threshold.	CELEBA	18.45%

Experimentation begins with a baseline model that trains a deep learning model on one edge device and concludes with a multi-node edge device network training a MobileNetV2 model in a federated learning architecture.

Table 2. MNIST CNN Model Architecture for Experiments I-III. Adapted from [47].

Layer	Shape	Total Parameters
Conv2D	(3, 3, 32, 64)	320
Max Pooling	(64,)	0
Conv2D	(3, 3, 64, 64)	18,496
Max Pooling	(64,)	0
Conv2D	(576, 64)	36,928
Flatten	(64,)	0
Dense	(64, 10)	36,928
Dense	(10,)	650

This architecture is equivalent to the validation architecture used by McMahan et al. in validating the FederatedAveraging algorithm.

2. Randomly Initialized Federated Averaging

Utilizes a randomly initialized MNIST CNN and serves as the federated averaging baseline for federated learning (see Table 2). All model training occurs on the client edge devices and federated averaging occurs on the server edge device. Experiment II is focused on determining the viability of performing federated learning solely on edge devices and how the distribution of data and computation on multiple edge device nodes improves performance over a single node training a CNN.

3. Centrally Pretrained Federated Fine-Tuning

Utilizes the centrally pretrained MNIST CNN parameters (see Table 2) and serves as a minimal implementation of a centrally pretrained federated fine-tuning architecture (Experiment III). Experiment III performs federated fine-tuning on the final two dense layers (37,578 trainable parameters) of the model. The focus is on determining if a pretrained model can decrease computational, communication and power costs on the edge devices. Federated fine-tuning also provides stronger security since a reduced number of the total global parameters shared. This will decrease the probability of an adversary reconstructing the global model from the transmitted parameters as only two layers are shared, and the base layers remain fully hidden on the edge devices [39].

4. Extended Class Centrally Pretrained Federated Fine-Tuning

Utilizes the centrally pretrained EMNIST CNN weights (see Table 3) to test the performance of an extended class federated fine-tuning architectures on edge devices. The EMNIST CNN trains on the final two dense layers (81,854 trainable parameters) of the EMNIST CNN model. The focus is on presenting the edge devices with a more complex classification problem and the ability to achieve suitable accuracy with a minimal number of training rounds. This design will decrease the ability for an adversary to reconstruct the global model, since only 59.5% of the parameters are sent to the server and the remaining parameters remain hidden on the edge devices [39].

Table 3. Experiment IV EMNIST CNN Model Architecture. Adapted from [47].

Layer	Shape	Total Parameters
Conv2D	(3, 3, 32, 64)	320
Max Pooling	(64,)	0
Conv2D	(3, 3, 64, 64)	18,496
Max Pooling	(64,)	0
Conv2D	(576, 128)	36,928
Flatten	(128,)	0
Dense	(128, 62)	73,856
Dense	(62,)	7,998

This architecture is roughly equivalent to the validation architecture used by McMahan et al. to validate the FederatedAveraging algorithm with the final dense layer expanded to 62 classes vice 10 classes for the MNIST and CIFAR10 datasets used by McMahan et al.

5. MobileNetV2 Centrally Pretrained Federated Fine-Tuning

Utilizes the centrally pretrained MobileNetV2 parameters (see Table 4 and 5) to test the performance and viability of a state-of-the-art federated fine-tuning architecture achieving high accuracy. The MobileNetV2 Model fine tunes a select number of MobileNetV2 layers and the classification head. This design only shares 18.42%-39.61% of the global parameters, depending on the MobileNetV2 layers fine-tuned, with the remaining parameters remaining hidden on the edge devices.

Table 4. Experiment V MobileNetV2 Model Architecture. Adapted from [48].

Layer	Shape	Total Parameters
MobileNetV2	(1, 1, 320, 1280)	2,257,984
Global Avg Pooling	(1280,)	0
Dropout	(1280, 1)	0
Dense	(1,)	1,281

The MobileNetV2 and classification head have 2,259,265 total parameters.

Table 5. Experiment V MobileNetV2 Block 16 and Classification Head Architecture. Adapted from [48].

Layer	Shape	Total Parameters
Expand Conv2D	(1,1,160,960)	153,600
BatchNorm	(960,)	3,840
ReLU	(960,)	0
Depthwise Conv2D	(3,3,960,1)	8,640
BatchNorm	(960,)	3,840
ReLU	(960,)	0
Project Conv2D	(1,1,960,320)	307,200
BatchNorm	(320,)	1,280
Conv2D	(1,1,320,1280)	409,600
BatchNorm	(1280,)	5,120
ReLU	(1280,)	0
Global Avg Pooling	(1280,)	0
Dropout	(1280, 1)	0
Dense	(1,)	1,281

6. End-to-End FedAvg Edge Device Network

This final experiment utilizes the centrally pretrained MobileNetV2 model in Experiment V (see Table 5 and 6). A secondary client node is added to make predictions from a camera triggered by movement in the vicinity. The entire architecture utilizes RPi 4B that run off battery power in an off-grid network. This network includes anomaly alert detection for predictions below the specified threshold for follow on human directed analysis. This experiment serves as a proof of concept that a COTS FedAvg network can function fully off-grid on battery power.

Table 6. Experiment VI MobileNetV2 Model Architecture. Adapted from [48].

Layer	Shape	Total Parameters
MobileNetV2	(1, 1, 320, 1280)	2,257,984
Global Avg Pooling	(1280,)	0
Dropout	(1280, 2)	0
Dense	(2,)	2,562

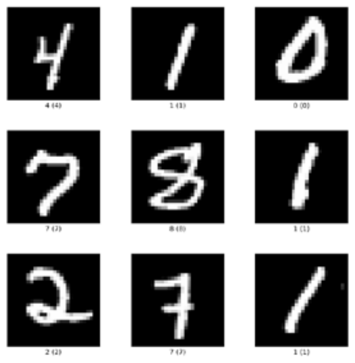
The MobileNetV2 and classification head have 2,260,546 total parameters.

B. DATASETS

Three datasets were chosen to evaluate deep learning performance on edge devices ranging from a standard machine learning benchmark dataset, to a federated learning testbed dataset, to a large-scale face attribute dataset.

1. MNIST Dataset

The MNIST dataset is the standard benchmark for machine learning, classification and computer vision research. MNIST is a relatively small database of handwritten digits (see Figure 8). The dataset consists of 10 classes of 28x28 pixel images. There are 60,000 training examples and 10,000 test examples [41]. Experiments in the current research randomly partitioned the data into 750 training (600 train, 150 validation) and 100 testing examples, matching the data sample sizes used by McMahan et al. and other benchmark federated averaging research [22]. Since the focus of the current research is federated learning edge device performance, data was assumed independent and identically distributed (IID) and not divided by class for non-IID client partitions.

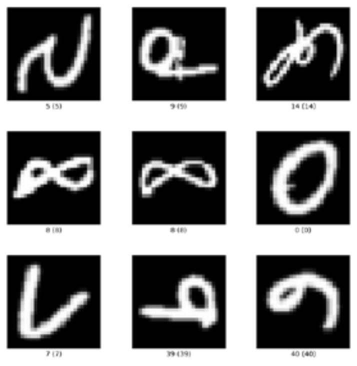


All MNIST images are 28x28 pixel greyscale and evenly divided into 10 classes.

Figure 8. MNIST Dataset Sample Images. Source: [42].

2. EMNIST Dataset

The extended MNIST (EMNIST) dataset is a dataset of handwritten characters derived from the NIST Special Database 19, that has been converted to 28x28 pixel images with a structure that directly matches the MNIST dataset. There are 62 classes with 697,932 training examples and 116,323 test examples (see Figure 9). EMNIST is TensorFlow's recommended small testbed for federated learning research, as it has a natural user-level partitioning [43]. Experimentation in the current research used a separate partitioned sample of 45,000 EMNIST images from the full EMNIST dataset on each edge device.

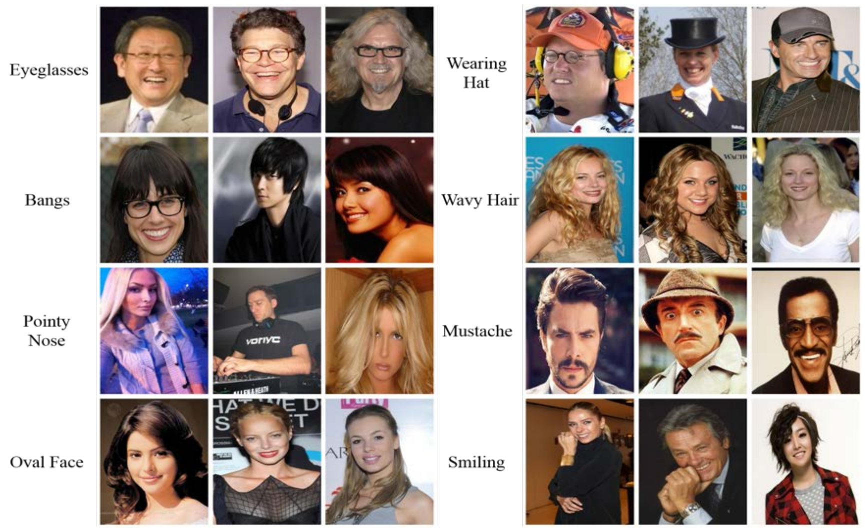


All EMNIST images are 28x28 pixel greyscale and divided into 62 classes.

Figure 9. EMNIST Dataset Sample Images. Source: [44].

3. CelebA Dataset

The CelebFaces attributes dataset (CelebA) is a large-scale dataset of facial attributes with 202,599 facial images, each with 40 binary attributes annotated. The dataset covers background clutter and large pose variations (see Figure 10). CelebA is able to be employed as a training and test set for multiple computer vision tasks—face attribute recognition, face detection, face landmark localization and face synthesis [45]. Experiments in the current research saved a random sample of resized (96 ,96 ,3) CelebA images on each edge device with the edge device data partitioned into test, validation and train datasets. The dataset was resized to conserve memory on the edge devices and this is also the minimum input shape for MobileNetV2.



CelebA images were resized to (96, 96, 3) in order to conserve memory on the edge devices.

Figure 10. CelebA Sample Images. Source: [46].

C. DEEP LEARNING MODEL ARCHITECTURES

Four deep learning model architectures were developed to evaluate federated learning performance on edge devices ranging from a TensorFlow convolutional neural network image classification model [47] to a state-of-the-art MobileNetV2 model [48].

1. MNIST and EMNIST CNN Models

Three Convolutional Neural Network (CNN) models were developed to perform and evaluate federated learning on the MNIST and EMNIST datasets. Each model developed for the current research utilized the Keras API, and were equivalent to the validation architecture used by McMahan et al. in validating the FederatedAveraging algorithm. These models are not state-of-the-art models, but are sufficient to show the relative performance of federated learning on an architecture of edge devices. The model architecture is a TensorFlow CNN [47] with three 3x3 convolution layers—the first with 32 channels and the second and third with 64 channels. Each of the first two convolutional layers is followed by a 2x2 max pooling, the third convolutional layer is followed by a flatten layer, and two fully connected layers. The MNIST CNN's have a total of 93,332 parameters and the EMNIST CNN has 137,598 total parameters (see Table 2).

2. Randomly Initialized MNIST CNN

The MNIST model for Experiment I and II was designed to begin model training with random initialization of the weights, as is the standard in federated learning (see Table 2 and 3). A majority of academic research utilizes random initialization of the weights for federated learning research. This federated learning methodology of random initialization assumes that the server has no access to client data and seeks to ensure privacy.

3. Centrally Pretrained MNIST CNN

For Experiment III, an MNIST CNN was centrally pretrained on Google CoLab with 750 MNIST image samples (600 train samples, 150 validation samples) and designed with a callback for early stopping to cease training when the model stopped showing improvement. This model followed the same model architecture as the randomly initialized models (see Table 2). Validation accuracy was monitored for a minimum change of less than 1e-2 for five epochs. This was done so that the model did not excessively overfit and could still benefit from federated learning. The model early stopped after nine epochs with a validation loss of 0.4540 and validation accuracy of 0.8810. All weights were saved in .h5 format and transferred to the edge devices for central pretrained federated fine-tuning.

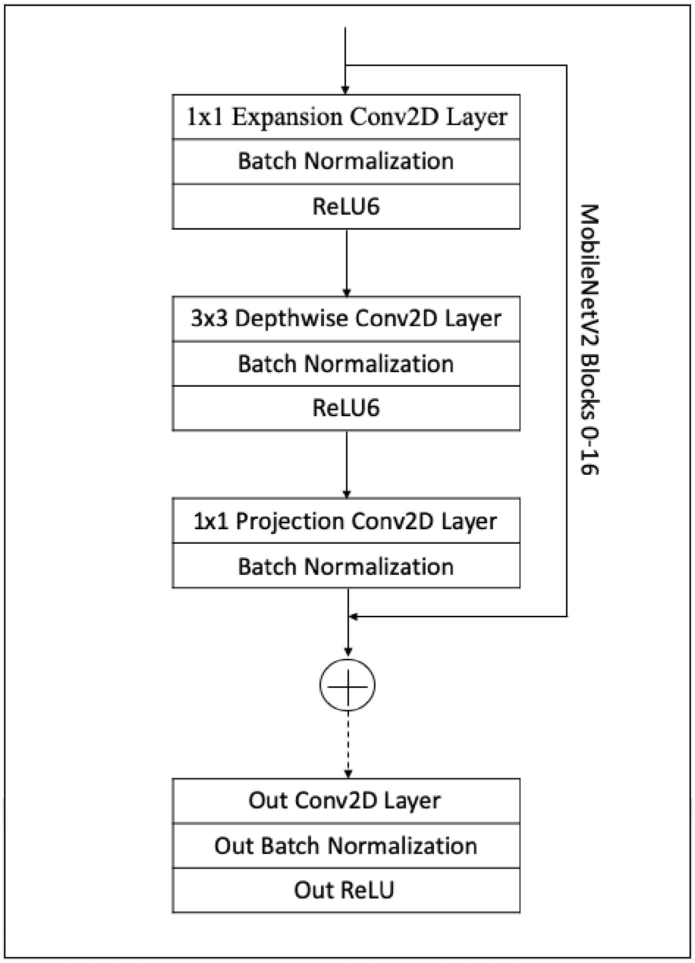
4. Centrally Pretrained EMNIST CNN

For Experiment IV, an EMNIST CNN was pretrained on a MacBook laptop with 350,000 EMNIST train images and 60,000 test images and designed with the same early stopping metrics as the MNIST CNN (see Table 3). The model early stopped after 14 epochs with a validation loss of 0.4223 and a validation accuracy of 0.8006. All weights were saved in .h5 format and transferred to the edge devices for central pretrained federated fine-tuning.

5. MobileNetV2 Federated Fine-Tuning Model

For Experiments V and VI, the pretrained MobileNetV2 model utilized the built in MobileNetV2 base architecture included with the Keras API, using the ImageNet weights with classification head removed [48], [49]. A global average pooling 2D layer, dropout layer and fully connected layer were added as a classification head (see Table 4, 5, and 6). The CelebA dataset was used with all images resized (96, 96, 3) for memory optimization on the Raspberry Pi. Three separate MobileNetV2 models were designed and centrally pretrained on a MacBook laptop in order to evaluate the ideal parameters for a centrally pretrained MobileNetV2 model on edge devices. Each of the three models were set up for binary classification on gender. It was designed with the same early stopping metrics as the MNIST and EMNIST CNN models. Weights were saved in .h5 format and transferred to the edge devices for central pretrained federated fine-tuning.

Blocks 0 thru 16 of the MobileNetV2 model included with the Keras API follow the same structure as block 16 (see Figure 11).



All MobileNetV2 Blocks included with the Keras API follow the same structure throughout. The above diagram includes the base architecture of 17 blocks and an out Conv2D Layer. A classification head is added to complete a MobileNetV2 model.

Figure 11. MobileNetV2 Model Block Structure. Adapted from [48].

D. FEDERATED AVERAGING ALGORITHM

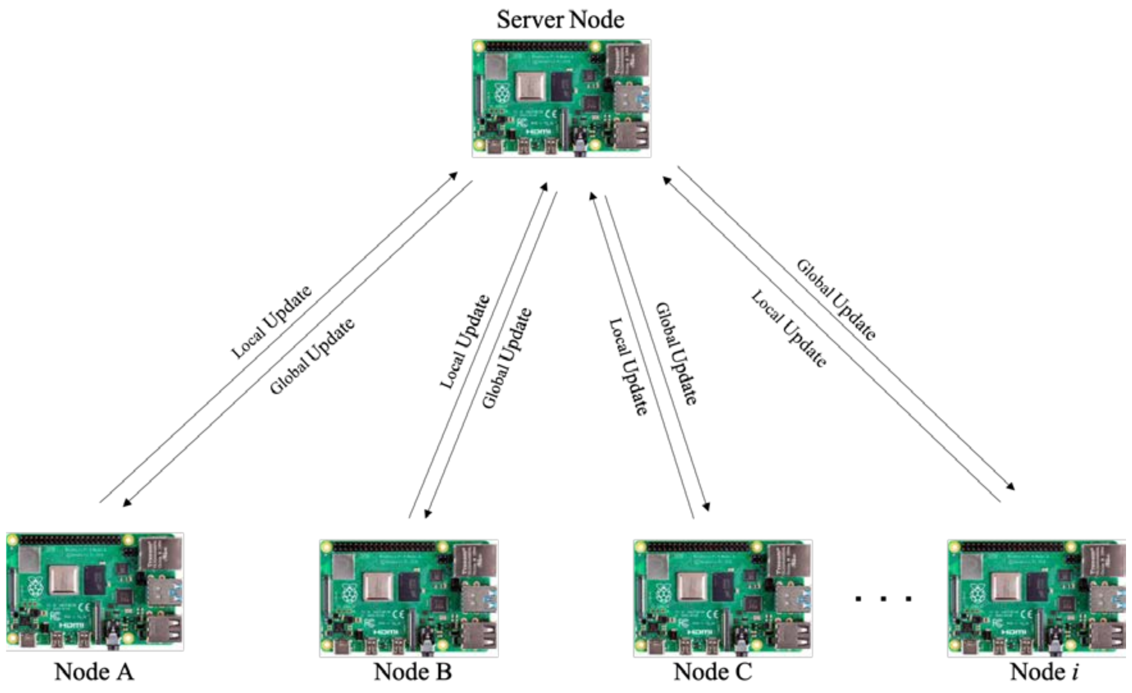
The federated averaging (FedAvg) algorithm, developed by McMahan et al., coordinates training through a central server that maintains the global model w_t, where *t* signifies the communication round. Model optimization occurs on the edge device using stochastic gradient descent (SGD). The FedAvg algorithm used in the current research had four primary hyperparameters: batch size B, number of local epochs E, learning rate η, and number of training rounds TR. Additional hyperparameters for Experiment V and VI include: pretrained model to use for federated learning, MobilenetV2 layer to fine tune from, number of training samples per training round, number of test samples for evaluation, and number of validation samples for validation (see Figure 12). One communication round of FedAvg consists of:

1. Server node selects hyperparameters and distributes the current global model to edge devices

2. Client nodes train an updated local model on data residing locally on the edge device

3. Client nodes send the updated local parameters to the server node

4. The server node aggregates and averages the client node local parameters and generates a new global model to be retransmitted to client nodes

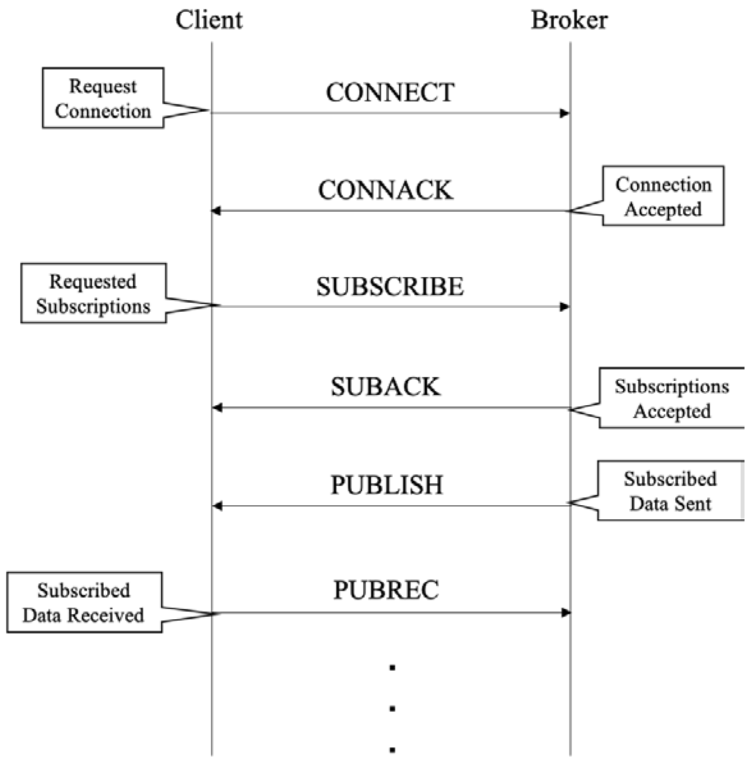


1) Primary client nodes receive hyperparameters and global model from server, 2) primary client nodes train model on local data, 3) primary client nodes send local model update to server node, and 4) server aggregates local models and distributes new global model to primary client nodes.

Figure 12. Edge Device Federated Learning Architecture Overview.

E. NETWORKING PROTOCOL

All networking communication between edge devices was executed with message queued telemetry transport (MQTT) protocol. MQTT is an open source IoT networking protocol that is lightweight and suitable for use on low power single board computers. It uses the TCP/IP stack and follows a publisher/subscriber model (see Figure 13), which makes it suitable for edge device computing on lower power sensors and embedded hardware [50]. MQTT is an asynchronous protocol making it very useful in federated learning scenarios, whereas HTTP is a synchronous protocol that lacks scalability and relies on a request/response pattern of communication. MQTT is a widely accepted IoT protocol that is supported and utilized by major applications such as IBM, Amazon AWS IoT, and Facebook Messenger.



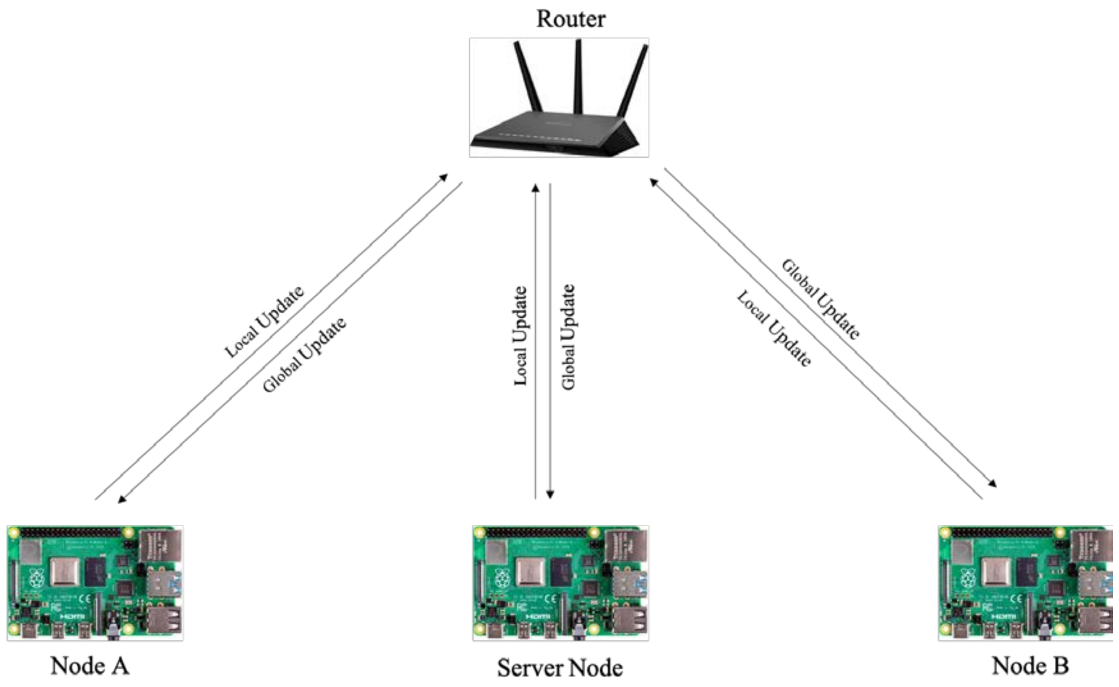
In the current research the server node acted as the MQTT broker and the client nodes acted as the MQTT clients.

Figure 13. MQTT Protocol Communication Flow. Adapted from [50].

Throughout the current research, the server node was utilized as the MQTT broker and coordinated local updates and global model transmissions. All parameters were sent as binary strings and reshaped by the edge devices once received. MQTT has several options to improve security, including TLS with CA, server keys, and certificates. For additional security, the MQTT broker can also establish restricted topics and implement an access control list (ACL). The maximum packet size allowed by MQTT is 250MB; however, the largest parameter transmitted in the current research was 1.63 MB [50].

F. FEDERATED FINE-TUNING HARDWARE SETUP

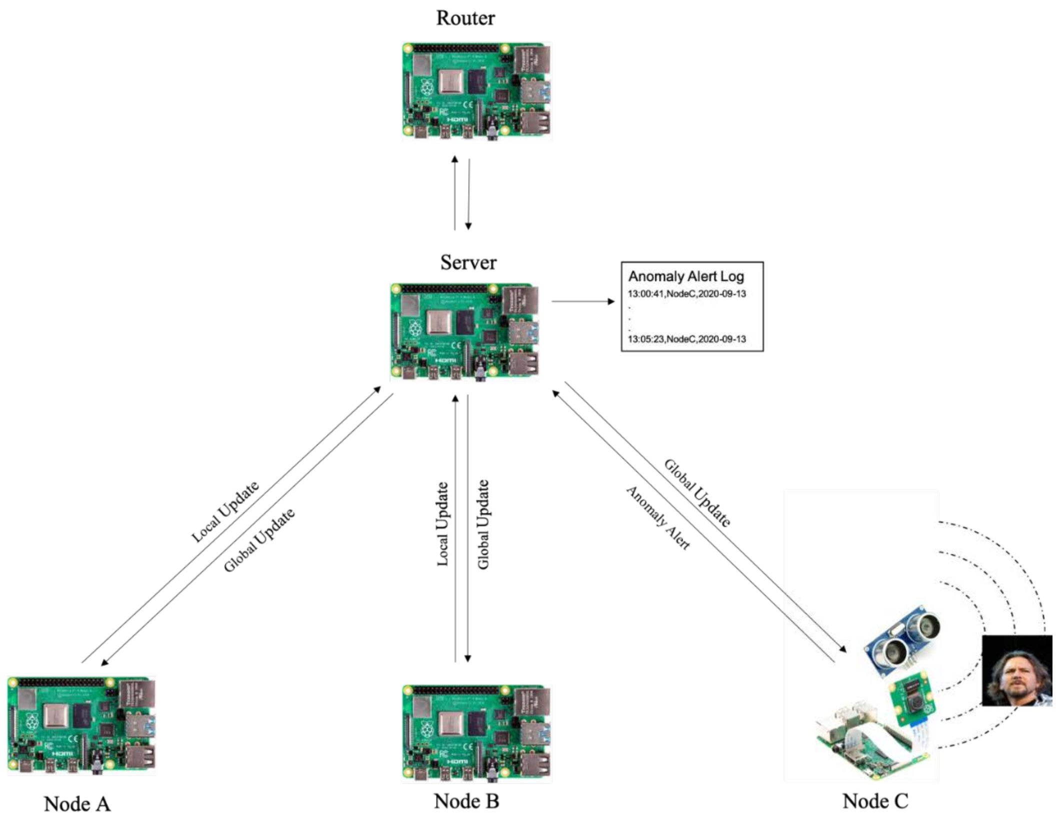
The hardware setup for Experiments I through V consisted of three Raspberry Pi 4B's—two primary client nodes and one server node (see Figure 14). The server node conducted federated averaging as well as functioning as the MQTT broker for the network. The primary client nodes performed model training on local data that was randomly chosen from the client dataset during each training round in order to simulate multiple clients. The network router used was a Netgear Nighthawk AC1900. It is believed that this research is the first to have an edge device perform the role of the server node. This architecture makes it possible for federated learning architecture to be deployable and non-reliant on a remote cloud server or GPU enabled server node.



Hardware setup includes 2 primary client nodes, 1 server node, and 1 router. 1) NodeA and NodeB perform federated averaging on local dataset with global model, 2) NodeA and NodeB send local updates to server node, 3) server node aggregates local updates and publishes a new global update to all nodes.

Figure 14. Experiment I-V Hardware Setup.

The architecture for Experiment VI consisted of five Raspberry Pi 4B—two primary client nodes, one secondary client node, one server node and one router (see Figure 15). The router was a Raspberry Pi 4B with the hostapd access point software package installed. This network was an isolated off-grid network with no internet access, ensuring the system was completely deployable. It is believed that this research is the first end-to-end edge device federated learning architecture with all edge device hardware components. This testbed architecture demonstrates a federated learning architecture can be tactically deployed to remote areas without dedicated power or internet access.



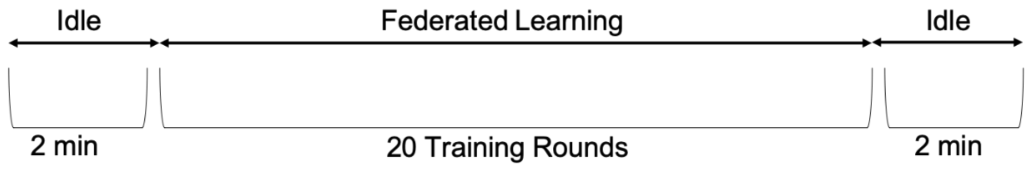
Hardware setup includes 2 primary client nodes, 1 secondary client node, 1 server node, and 1 router. Primary client NodeA and NodeB perform federated averaging on local dataset, 2) NodeA and NodeB send local updates to server node, 3) server node aggregates local updates and publishes a new global update to all nodes, 4) secondary client NodeC predicts on local data using the most up to date global model, 5) NodeC sends an anomaly alert for any predictions below specified threshold, and 6) server node logs anomaly alert for follow on human analysis and additional model training.

Figure 15. Experiment VI Hardware Setup.

During all testing of experiments I through VI, edge devices were powered with 10,000 mA power banks to simulate an end-to-end deployment of a COTS edge device architecture. For software, the RPi network used TensorFlow 2.2 and Python 3.6 with MQTT as the networking protocol.

G. EDGE DEVICE PERFORMANCE TESTS ON MEMORY, COMPUTATION, COMMUNICATION AND POWER

Performance tests were designed to evaluate and compare memory, computation, communication and power performance of federated learning on edge devices. Each performance test was designed to compare performance when executing federating learning and performance when the edge devices were idle. A two-minute idle period was evaluated prior to the start of the federated training rounds and a two-minute idle period was recorded after the final federated training rounds were completed (see Figure 16). In between the idle periods, each performance test included 20 federated training rounds for evaluation of federated learning.



Each performance test began with a 2-minute idle period, followed by 20 federated learning training rounds and concluded with a 2-minute idle period.

Figure 16. Edge Device Performance Test Overview.

The following system metrics were captured and analyzed to monitor edge device performance on memory, computation, communication and power:

1. Memory

Metrics were captured by running Linux SysStat SAR commands and averaging recorded memory statistics (see Table 7). SAR is part of the SysStat package, which is composed of utilities designed to monitor system performance and usage activity.

Table 7. Edge Device Limitation Performance Metrics.

Edge Device Limitation	Performance Metric
Memory	RAM total, RAM free, RAM buffered, swap space total, swap space free, memory read/write speeds, context switches
Computation	CPU load, CPU temp, seconds per machine learning epoch, seconds per machine learning step
Communication	Bytes received per second (BRS), bytes transmitted per second (BTS)
Power	Current (mA), power (mW), supply voltage (V)

Metrics were recorded using Linux SysStat SAR commands, Raspberry Pi vcgencmd commands, Keras API stats, and the INA219 current shunt and power monitor IC.

2. Computation

Metrics were captured by running Linux SysStat SAR commands and averaging computation statistics (see Table 7), as well as averaging training metrics from the Keras API fit method.

3. Communication

Metrics were captured by running Linux SysStat SAR commands and averaging recorded communication statistics (see Table 7) as well as averaging training time metrics from the Keras API fit method.

4. Power

Performance was monitored through a RPi 3B and a Texas Instruments INA219 current shunt and power monitor IC (see Table 7). The INA219 is able to monitor both shunt voltage drop and bus supply voltage, with programmable conversion times and filtering with accuracy within 0.5% [51]. A python script was written to capture bus voltage (V), bus current (mA), power (mW), shunt voltage (mV), and supply voltage (V) to a CSV file for analysis. Bus voltage reads the voltage between GND and V, and is the total voltage seen by the circuit under test (supply voltage—shunt voltage) [51]. Shunt voltage reads the voltage drop across the INA219 shunt resistor. Bus current is derived by Ohms Law from the measured shunt voltage.

IV. RESULTS AND ANALYSIS

A. OVERVIEW

Significant memory, computational, and power costs are incurred when training a deep learning model on a single edge device, such as the Raspberry Pi. A solution to reduce these costs and improve performance is a multi-node federated learning architecture composed of edge devices. The current research demonstrated that a federated learning architecture can be successfully deployed on edge devices with TensorFlow Version 2.2. TensorFlow Federated, TensorFlow's federated learning API, is currently available for simulation only and based on an exhaustive investigation in current research it was not identified that federated learning had been implemented solely on an edge device network. Research was identified with federated learning on edge devices that used TensorFlow, but with a more powerful device (e.g., workstation with GPU or laptop) used to support and coordinate the architecture, not a full IoT system.

The primary findings in the current research include:

- A multi-node network of edge devices executing federated learning can improve edge device system performance over a traditional deep learning model trained on a single edge device.

- Centrally pretrained models can achieve high accuracy in a minimal number of federated training rounds, whereas a randomly initialized model requires a large number of federated training rounds to achieve high accuracy.

- A state-of-the-art machine learning model (MobileNetV2) can be centrally pretrained and deployed on a network of edge devices for federated fine-tuning and improve memory, computation, communication and power costs on embedded hardware.

- A centrally pretrained model shares a minimal percentage of the global model, which improves the security of the model from a federated learning attack on the transmitted parameters. When the whole global model is transmitted in a federated learning network it is susceptible to an adversarial federated learning attack.

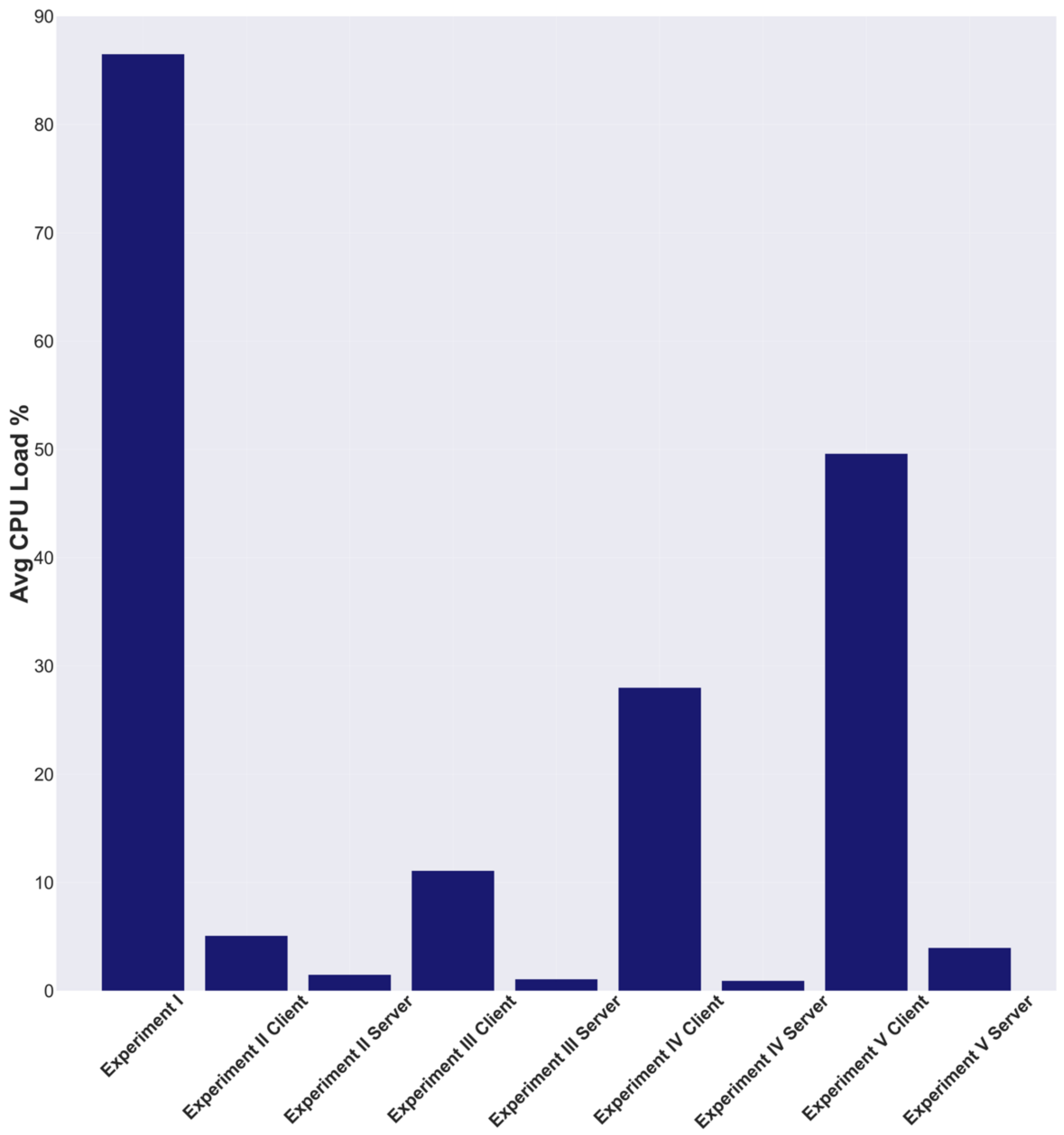
- A federated learning architecture can be composed completely of battery-powered COTS edge devices, thus making it fully deployable and off-grid for tactical scenarios.
- A true IoT networking protocol (MQTT) can be used to support deep learning and federated learning applications. This makes it possible for severely resource constrained embedded hardware and sensors to be directly involved in expanded applications involving real-time federated learning.
- An off-grid battery-powered COTS embedded hardware federated learning architecture was developed as a prototype to analyze and quantify the capabilities and limitations of federated learning on edge devices, which can be used for follow-on research.

B. EDGE DEVICE PERFORMANCE TESTS

Memory, computation, communication and power performance tests were conducted in conjunction with Experiments I-V to determine the impacts of multi-node federated learning networks. Experiment VI is an extension of Experiment V and it is not included in this particular section. To analyze edge device performance, a two-minute idle period was evaluated prior to the start of the federated training rounds and a two-minute idle period was recorded after the final federated training rounds were completed. In between the idle periods, each edge device performance test included 20 federated training rounds for evaluation of federated learning.

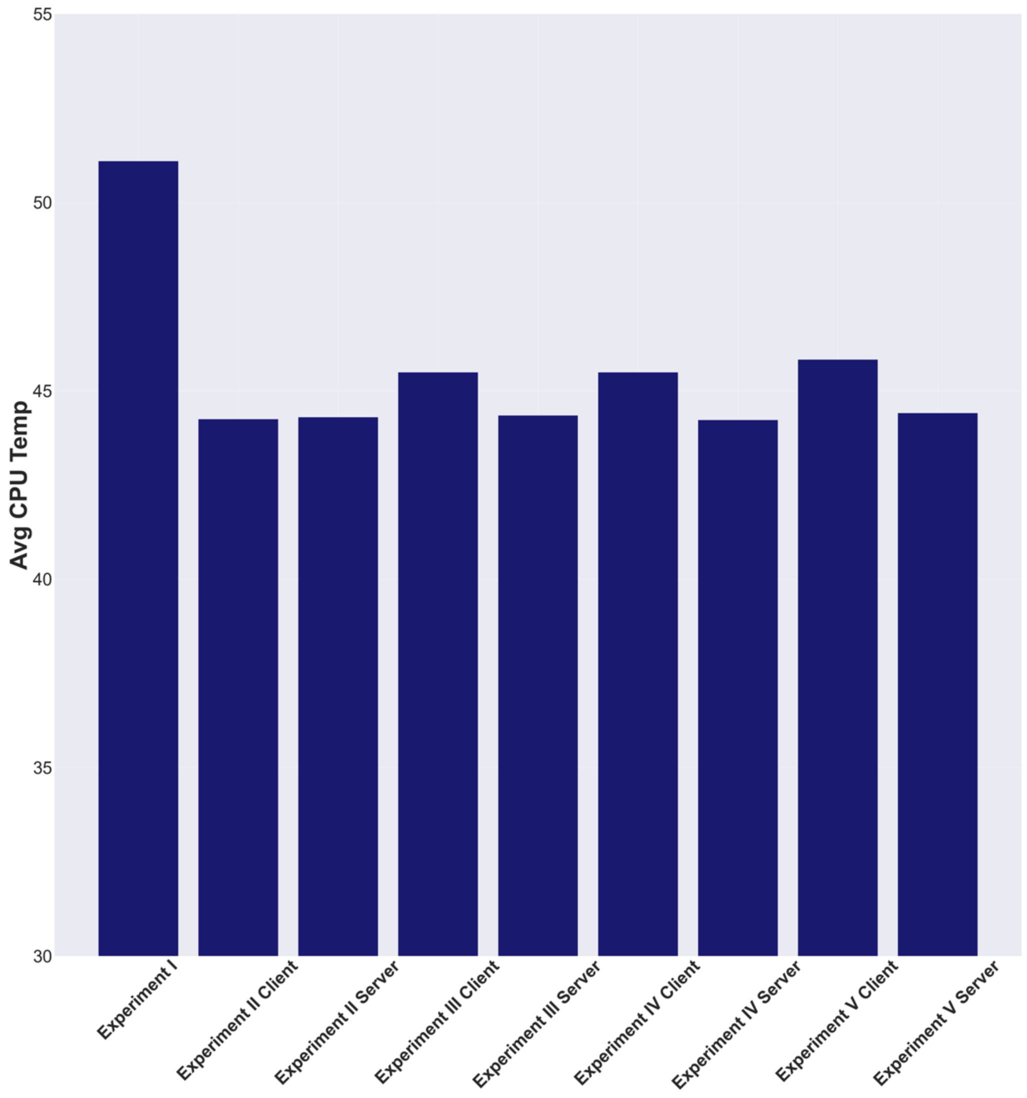
1. Computation Costs

Computation costs were calculated through Linux SysStat SAR commands and RPi vcgencmd commands. Average CPU load percentage is the CPU used for processes owned by normal users and system processes [48]. Average CPU temperature is the core temperature of the BCM2835 RPi Broadcom SoC. Seconds per epoch is the time it takes to make one full cycle through the training data for the specified federated training round. Milliseconds per step is the time it takes to process one batch of examples to perform one gradient update. The single node centrally trained model (Experiment I) experienced an average CPU load of 86.5% across all four cores and an average CPU temperature of 51.1 Celsius while training the MNIST CNN model (see Figure 17 and 18). The same RPi edge device had an average CPU load of .25% and an average CPU temperature of 43.0 Celsius while at idle (see Table 8).



Average CPU load percentage is measured across all 4 cores of the RPi and is the CPU load used for processes owned by normal users and system processes. Primary client nodes in Experiments II-V saw an average 72.99% reduction in CPU load over Experiment I.

Figure 17. RPi 4B Average CPU Load for Experiments I-V.



Average CPU temperature is measured from the core temperature of the BCM2835 SoC. Experiments II-V saw a 11.61% reduction in CPU temperature over Experiment I.

Figure 18. RPi 4B Average CPU Temperature for Experiments I-V.

Table 8. RPi Computational Costs for Experiments I-V.

	Experiment	Average CPU Load %	Average CPU Temperature	Seconds per Epoch	Milliseconds per Step
I	**Single Node Centrally Trained**	86.5%	51.1	403	400
II	**Randomly Initialized (Primary Client Node)**	5.09%	44.25	3	105
II	**Randomly Initialized (Server Node)**	1.5%	44.3	n/a	n/a
III	**Centrally Pretrained (Primary Client Node)**	11.1%	45.49	3	105
III	**Centrally Pretrained (Server Node)**	1.07%	44.35	n/a	n/a
IV	**Extended Class Centrally Pretrained (Primary Client Node)**	28%	45.49	24	157
IV	**Extended Class Centrally Pretrained (Server Node)**	.93%	44.23	n/a	n/a
V	**MobileNetV2 Centrally Pretrained (Primary Client Node)**	49.26%	45.83	5	1000
V	**MobileNetV2 Centrally Pretrained (Server Node)**	3.93%	44.41	n/a	n/a

Average CPU load is measured across all four RPi CPU cores. Average CPU temp is the core temperature of the RPi BCM2835 SoC. The Keras API fit method records the seconds per epoch and milliseconds per epoch each epoch. Experiment I experienced an average CPU load of 86.5%, while all other experiments experienced greatly reduced CPU loads.

All primary client nodes and server nodes for Experiments II-V experienced significantly reduced average CPU loads and CPU temperatures over the centrally trained single node (see Table 8). The MobileNetV2 primary client nodes experienced the highest CPU load (49.26%) for Experiments II-V, but trained on over 23 times as many parameters as Experiment I. The server nodes experienced the least impact on CPU load, which indicates that the server nodes could be tasked with additional responsibilities and tasks as required.

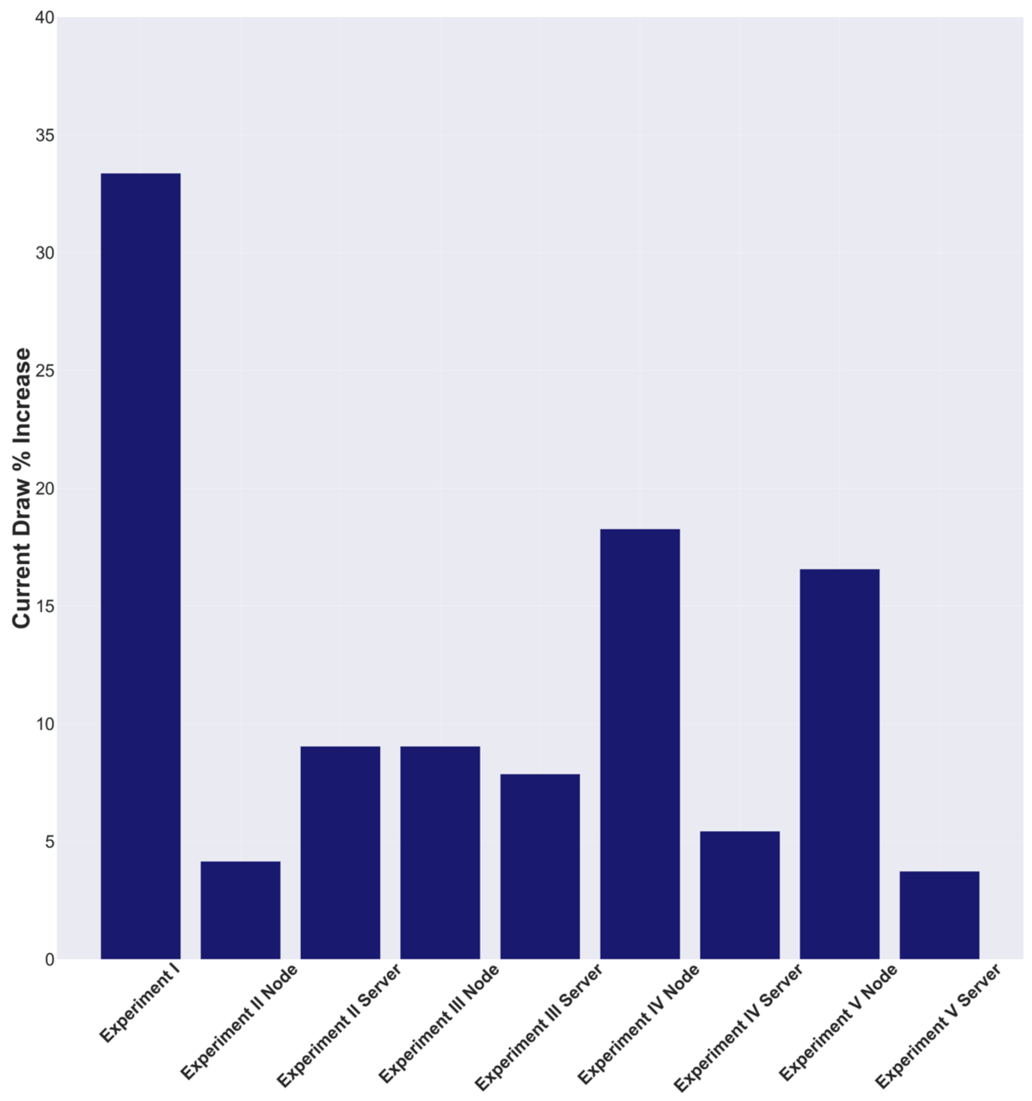
Additionally, the single node model (Experiment I) experienced much longer machine learning training times (403 seconds per epoch) as it is responsible for training a full dataset on one device. The federated client nodes have a smaller local dataset than a centrally trained model (29,625 train images and 4,950 test images each), which results in reduced training time. However, this does not limit the federated nodes ability to learn. They are able to leverage the other edge device's local data through the distributed global model.

2. Power Costs

Power measurements were based on the percentage increase from when the RPi was operating at idle versus when it was performing federated learning training rounds. The server nodes were serving as the MQTT broker during testing and the primary client nodes were serving as MQTT subscribers during testing. Battery life was based on a nominal measurement from a 10,000 mA external battery pack. Performance was monitored through a RPi 3B and a Texas Instruments INA219 current shunt and power monitor IC. A python script was developed to capture bus voltage (V), bus current (mA), power (mW), shunt voltage (mV), and supply voltage (V) and write the results to a .CSV file for analysis.

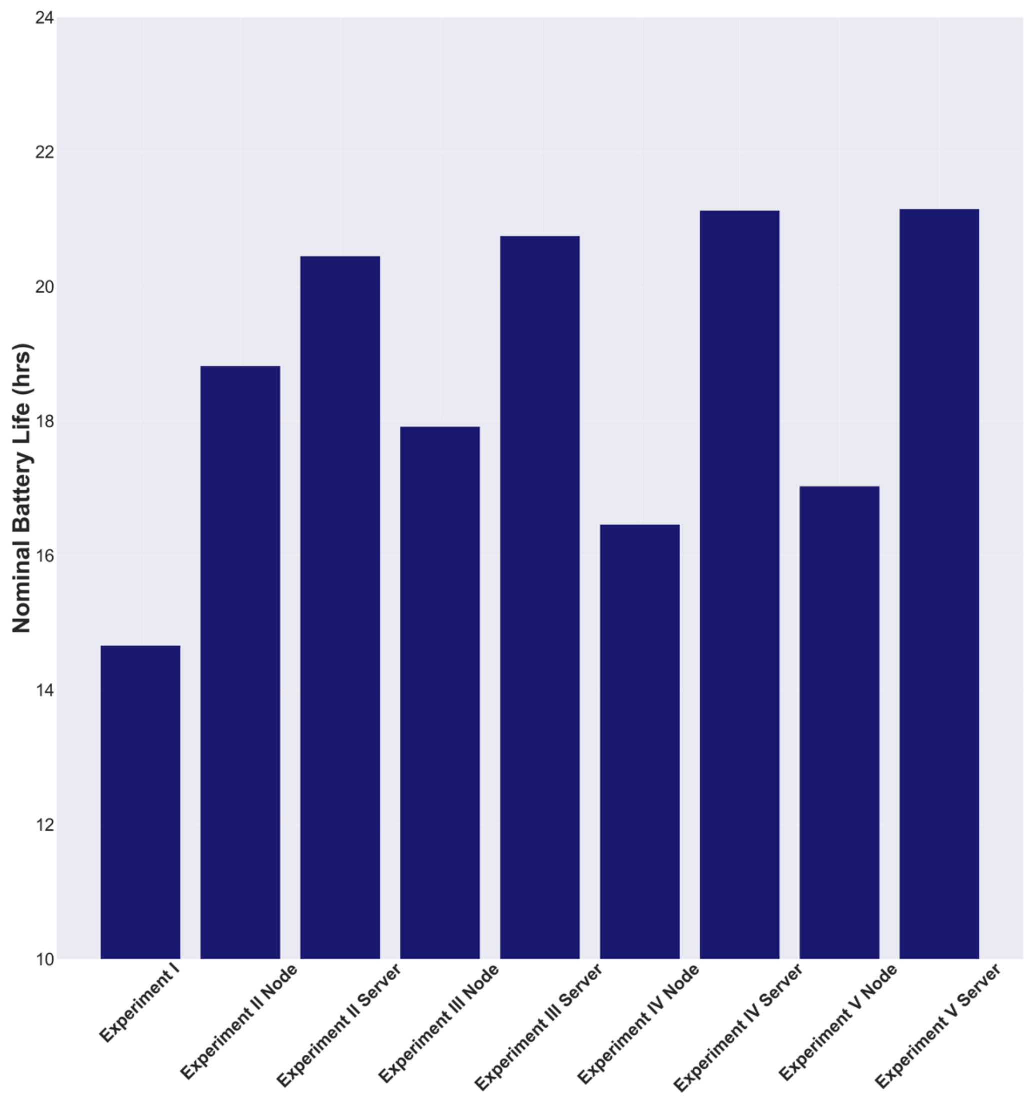
The single node centrally trained model for Experiment I experienced a 33.37% increase in current draw when training the deep learning model, which resulted in a nominal expected battery life of 14 hours and 28 minutes (see Figure 19 and 20). All primary client nodes in Experiments II-V drew less current than the single node centrally trained model (see Table 9), which would result in a longer battery life before recharging is necessary. The primary client nodes train on a much smaller dataset, since the dataset is distributed across multiple nodes, which impacts current draw from model training.

The server nodes saw a very minimal increase in current draw over idle, which would allow server nodes to handle additional tasks as required (see Figure 21-25). The server nodes were tasked with two responsibilities during federated training—global model aggregation/distribution and MQTT broker of the network. This minimal impact on current draw of the server nodes indicates that in a degraded environment the role of the server node could be passed to a node not as heavily tasked (i.e., the secondary client nodes of Experiment VI).



The primary client nodes in Experiments II-V drew 64.00% less current than the single node setup in Experiment I.

Figure 19. RPi 4B Current Consumption for Experiments I-V.



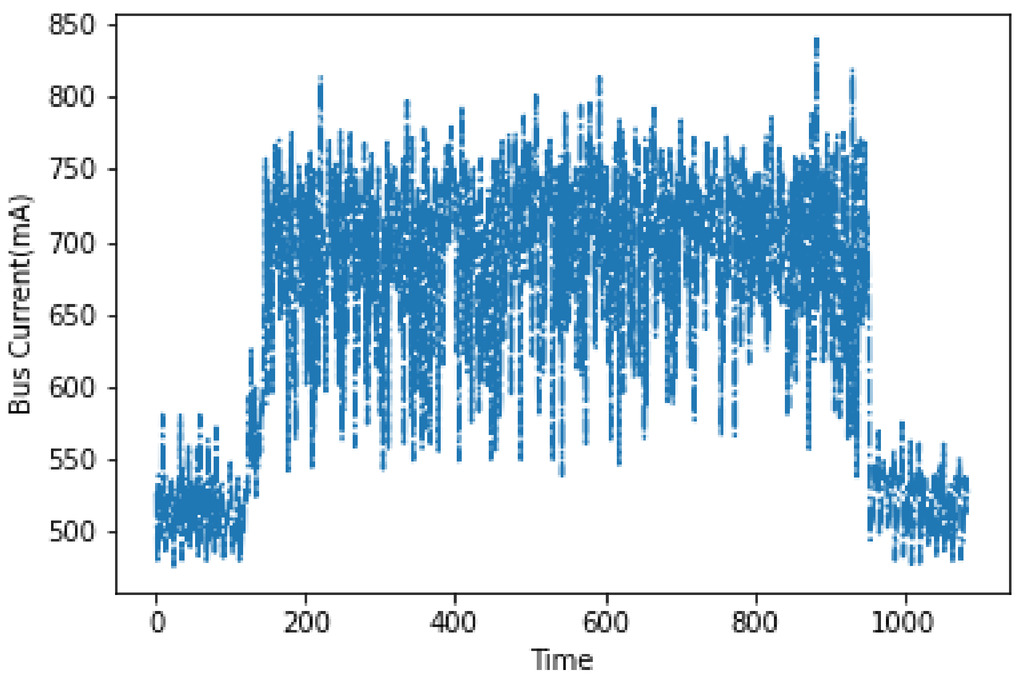
The primary client nodes in Experiment II-V showed a 21.37% improvement in nominal battery life over the single node setup in Experiment I.

Figure 20. RPi 4B Nominal Battery Life for Experiments I-V.

Table 9. RPi Power Costs for Experiments I-V.

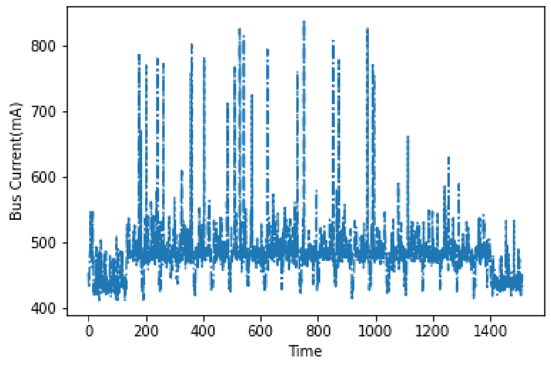
	Experiment	Bus Current (mA)	Power (mW)	Shunt Voltage (V)	Nominal Battery Life
I	Single Node Centrally Trained	+33.37%	+31.64%	+33.45%	14h28m
II	Randomly Initialized (Primary Client Node)	+4.16%	+3.88%	+4.24%	18h49m
II	Randomly Initialized (Server Node)	+9.04%	+8.56%	+8.86%	20h27m
III	Centrally Pretrained (Primary Client Node)	+9.04%	+6.84%	+7.18%	17h55m
III	Centrally Pretrained (Server Node)	+7.86%	+7.50%	+7.96%	20h45m
IV	Extended Class Centrally Pretrained (Primary Client Node)	+18.27%	+17.65%	+19.18%	16h28m
IV	Extended Class Centrally Pretrained (Server Node)	+5.44%	+5.09%	+5.66%	21h08m
V	MobileNetV2 Centrally Pretrained (Primary Client Node)	+16.57%	+15.07%	+16.19%	17h02m
V	MobileNetV2 Centrally Pretrained (Server Node)	+8.93%	+8.76%	+9.66%	21h09m

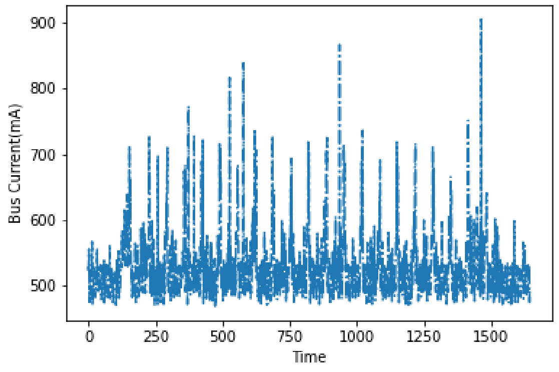
Bus current, power, and shunt voltage are the average percentage increase over idle when executing federated training rounds. Battery life is the nominal battery life of a 10,000 mAH rechargeable battery pack based off bus current.



Current draw was measured by the INA219 every second during the duration of the test. The power performance test for Experiment I began with a two-minute idle period, then 10 model training epochs, followed by a two-minute idle period. The single node centrally trained model in Experiment I exhibited a 33.37% increase in current draw when performing model training.

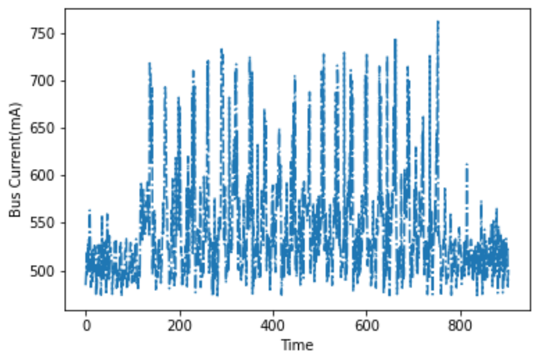
Figure 21. RPi Current Consumption for Experiment I.

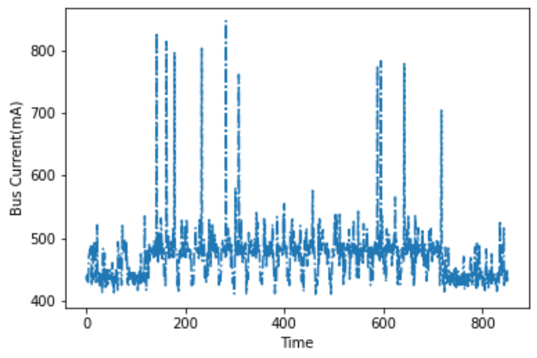




Primary client node (top) and server node (bottom). The power performance test for Experiment II began with a two-minute idle period, then 20 rounds of federated learning training rounds, followed by a two-minute idle period. The primary client node exhibited a 4.16% increase in current draw over idle and the server node exhibited a 9.04% increase in current draw over idle. The spikes in current draw are MQTT transmissions across the network.

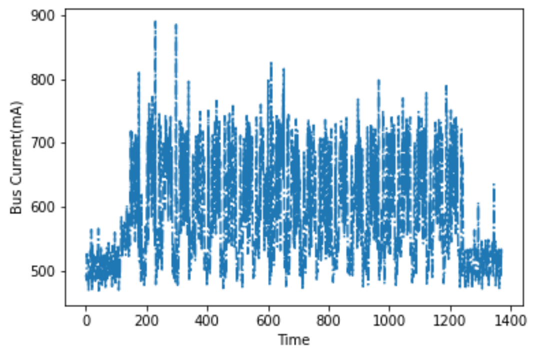
Figure 22. RPi Current Consumption for Experiment II.

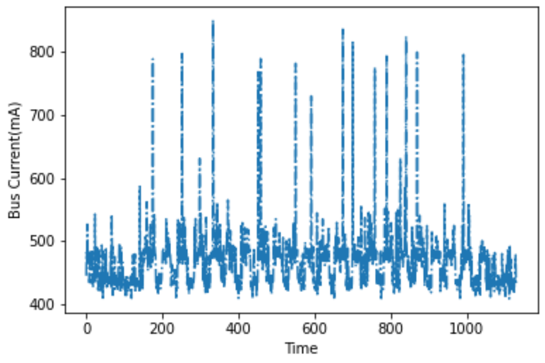




Primary client node (top) and server node (bottom). The power performance test for Experiment III began with a two-minute idle period, then 20 rounds of federated learning training rounds, followed by a two-minute idle period. The primary client node exhibited a 9.04% increase in current draw over idle and the server node exhibited a 7.86% increase in current draw over idle. The spikes in current draw are MQTT transmissions across the network.

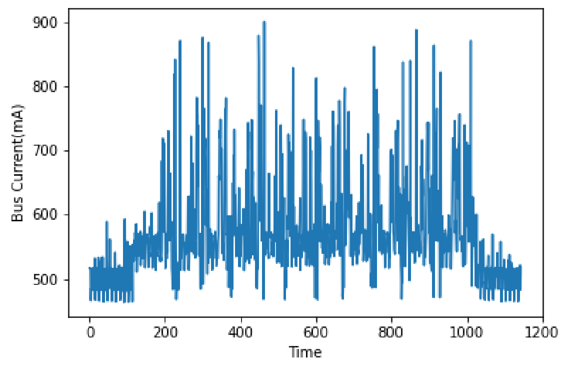
Figure 23. RPi Current Consumption for Experiment III.

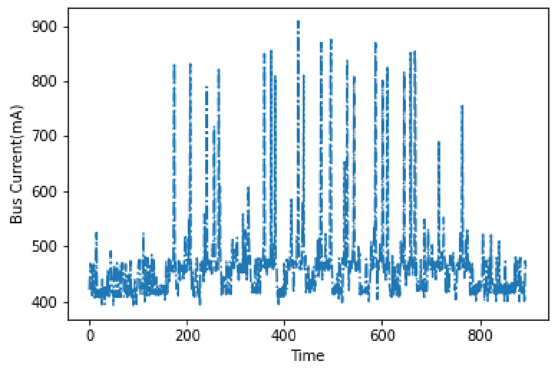




Primary client node (top) and server node (bottom). The power performance test for Experiment IV began with a two-minute idle period, then 20 rounds of federated learning training rounds, followed by a two-minute idle period. The primary client node exhibited a 18.27% increase in current draw over idle and the server node exhibited a 5.44% increase in current draw over idle. The spikes in current draw are MQTT transmissions across the network.

Figure 24. RPi Current Consumption for Experiment IV.





RPi primary client node (top) and server node (bottom). The power performance test for Experiment V began with a two-minute idle period, then 20 rounds of federated learning training rounds, followed by a two-minute idle period. The primary client node exhibited a 16.57% increase in current draw over idle and the server node exhibited a 8.93% increase in current draw over idle. The spikes in current draw are MQTT transmissions across the network.

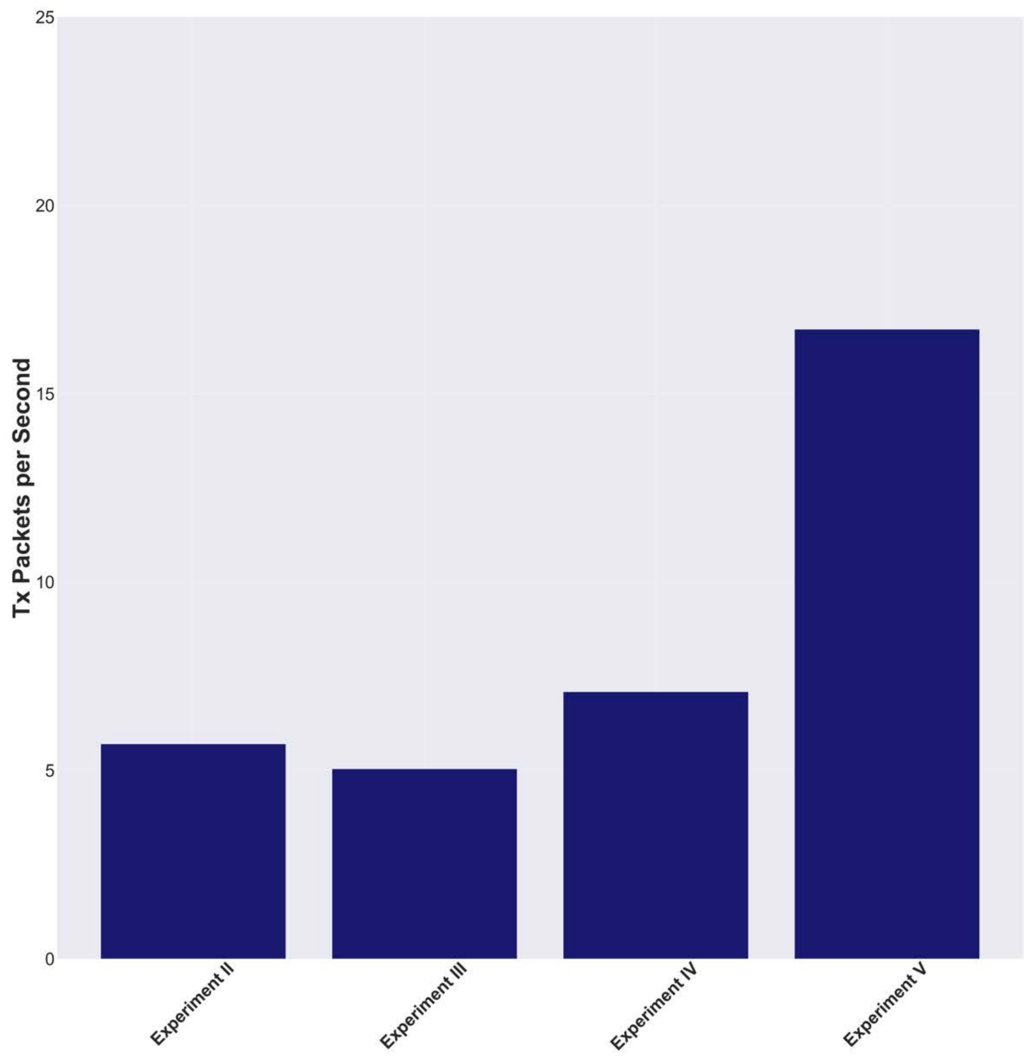
Figure 25. RPi Current Consumption for Experiment V.

3. Communication

Communication costs evaluated the percentage of parameters shared, packets transmitted/received per second, and kB transmitted/received per second (see Figure 26 and 27). The randomly initialized federated learning model (Experiment II) sent 100% of the global model parameters. Since it was randomly initialized when federated training began all weights must be transmitted so that it can improve through iterative federated learning training rounds. Each of the centrally pretrained models (Experiments III–V) only send a fraction of the model parameters, since the parameters have been trained prior to deployment on the edge devices. Only sending a percentage of the global parameters allows a significant portion of the model to remain hidden on the client nodes (see Table 10). As identified by Hitaj, Ateniese and Perez-Cruz, this structure of minimal parameter makes it incredibly difficult for an adversary to reconstruct the global model from the transmitted parameters [39].

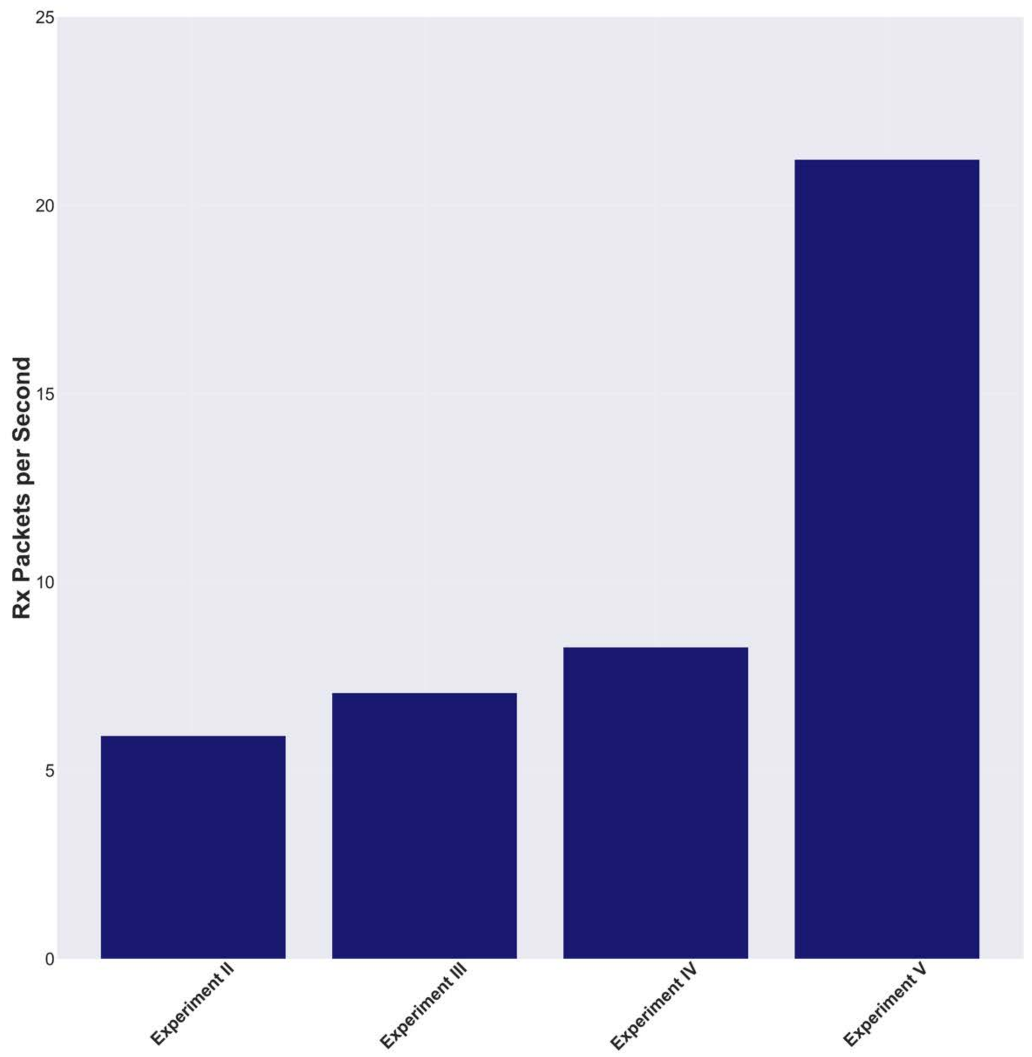
Additionally, the centrally pretrained models achieve high accuracy in fewer training rounds than a randomly initialized model. The centrally pretrained MNIST model (Experiment III) primary client nodes had an average accuracy of 96.50% after the first federated training round. The centrally pretrained EMNIST model (Experiment IV) primary client nodes began with an average accuracy of 77.80% after the first federated training round. The centrally pretrained MobileNetV2 model (Experiment V) primary client nodes began with an average accuracy of 91.75% after the first federated training round. This initial boost in model accuracy over a randomly initialized model enhances security since a minimal number of communication rounds are required with the server node to establish suitable accuracies. Hitaj, Ateniese and Perez-Cruz determined that an adversarial attack that attempt to reconstruct the global model by intercepting the transmitted parameters requires iterative training rounds with increasing accuracy [39].

The MobileNetV2 model (Experiment V) transmits 64.43% more packets per second than the average of the other three models. It also receives 66.62% more packets per second than the average of the other three models. The MobileNetV2 model transmits 416,001 parameters every federated training round, which is 18.41% of the total model parameters. While the MobileNetV2 sends the most model parameters of the models in the current research, it sends the smallest percentage of respective global parameters. This results in a higher communication cost than the other models, but provides the highest security of all the federated learning models in the current research.



The MobileNetV2 model (Experiment V) transmits 64.43% more packets per second than the overall average of the other three models.

Figure 26. RPi 4B Transmitted Packets Per Second for Experiments II-V.



The MobileNetV2 model (Experiment V) receives 66.62% more packets per second than the overall average of the other three models.

Figure 27. RPi 4B Received Packets Per Second for Experiments II-V.

Table 10. RPi Communication Costs for Experiments II-V.

	Experiment	Parameters Shared	Tx Packet per second	Rx Packet per second	Rx Kb per second	Tx Kb per second
II	Randomly Initialized (Primary Client Node)	100.00%	5.70	5.92	5.68	6.42
III	Centrally Pretrained (Primary Client Node)	40.26%	5.04	7.06	5.42	5.79
IV	Extended Class Centrally Pretrained (Primary Client Node)	59.49%	7.09	8.27	6.93	7.87
V	MobileNetV2 Centrally Pretrained (Primary Client Node)	18.41%	16.71	21.22	20.01	24.82

Parameters shared is the fraction of global parameters shared across the network for federated learning. Packets Tx/Rx per second and Kb Tx/Rx per second are captured through Linux SysStat SAR commands and averaged across 20 federated training rounds.

4. Memory

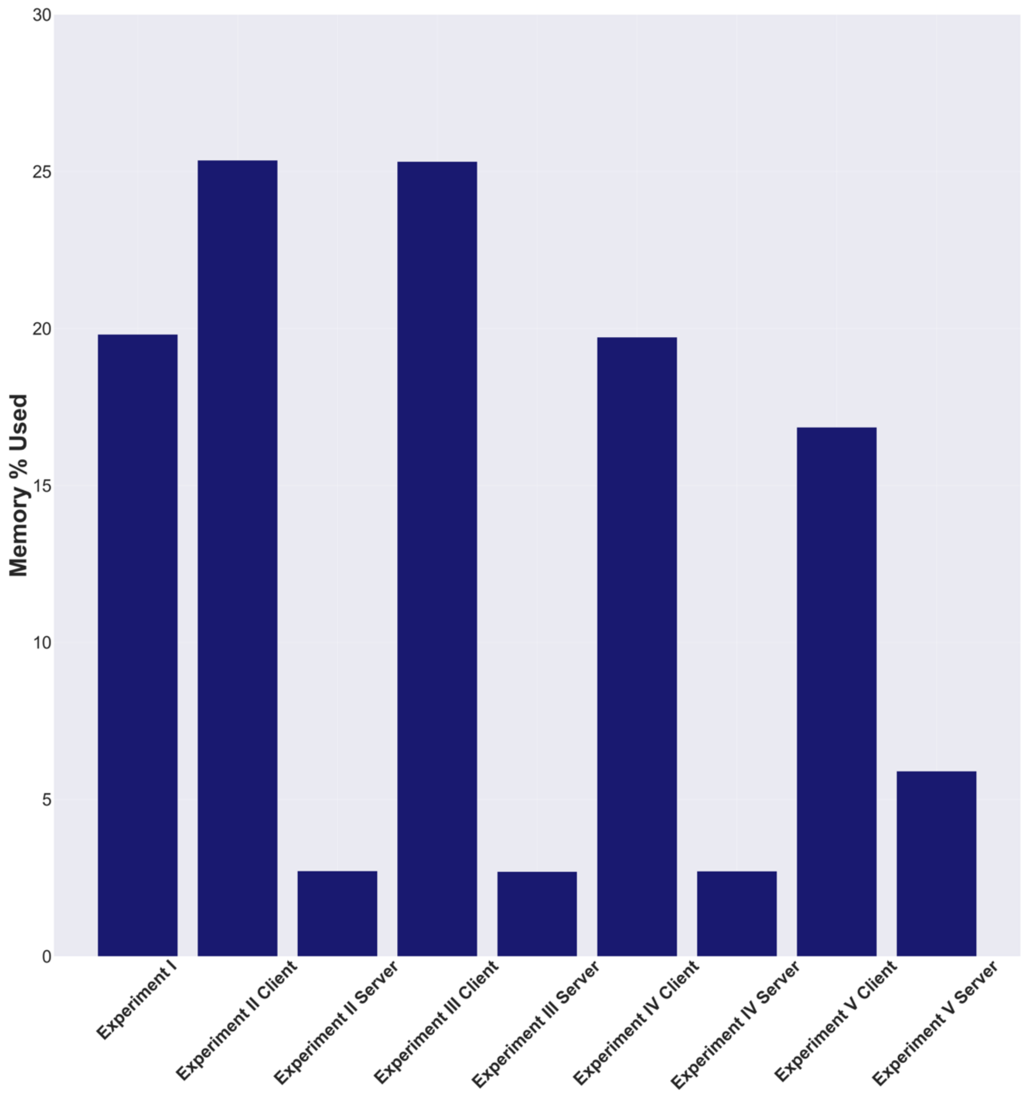
Memory costs evaluated the percentage of memory (RAM and swap) used, percentage of memory needed for current workload in relation to the amount of total memory (RAM and swap), number of kb paged in by the system per second, number of kb paged out by the system per second, number of page faults (major and minor) made by the system per second, and the total number of context switches per second (see Table 11) [48].

The primary client nodes in Experiments II-V demonstrated comparable impacts on memory (except the number of page faults) to Experiment I; however, the server nodes demonstrated very minimal memory utilization for the workload placed on them (see Figure 28 and 29). The server nodes sent over 400,000 parameters per training round in Experiment V with minimal impacts on memory. This minimal impact on memory opens the possibility for server nodes to be tasked with a larger workload or additional tasks as demanded.

Table 11. RPi Memory Costs for Experiments I-V.

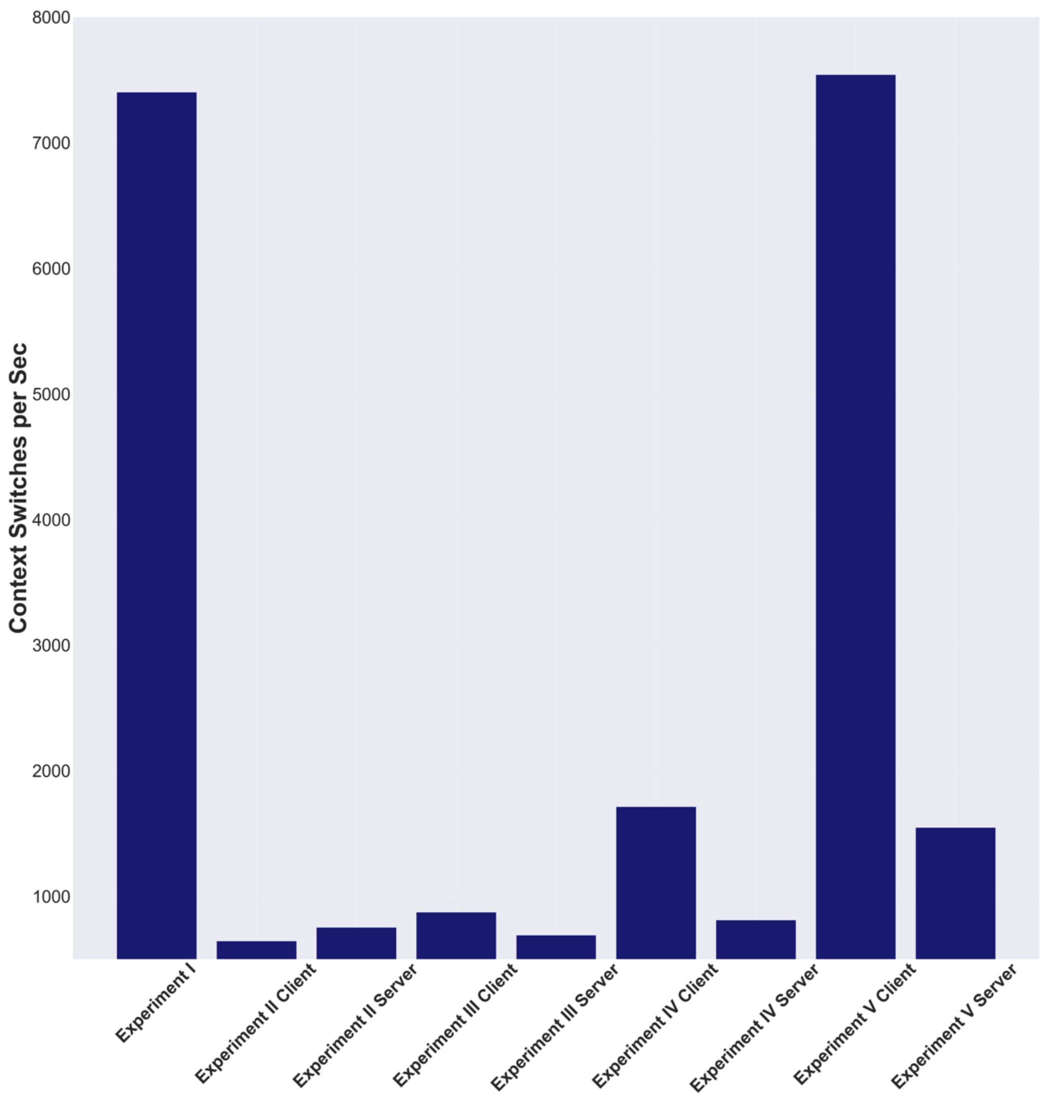
	Experiment	Memory Used	% Commit	Pages in per sec	Pages out per second	Faults per second	Context Switches per sec
I	**Non-Distributed Centrally Trained**	19.81	20.46	0.23	33.08	29311.94	7401.11
II	**Randomly Initialized (Primary Client Node)**	25.36	24.55	0.01	8.95	4098.62	645.08
II	**Randomly Initialized (Server Node)**	2.72	3.89	0.01	18.02	41.92	753.49
III	**Centrally Pretrained (Primary Client Node)**	25.23	24.58	2.56	9.33	9155.24	874.20
III	**Centrally Pretrained (Server Node)**	2.70	3.95	0.00	15.72	41.94	692.21
IV	**Extended Class Centrally Pretrained (Primary Client Node)**	19.72	20.09	527.78	9.10	21689.50	1714.72
IV	**Extended Class Centrally Pretrained (Server Node)**	2.71	3.89	0.00	9.17	93.15	812.05
V	**MobileNetV2 Centrally Pretrained (Primary Client Node)**	25.91	26.13	1.95	1.95	6367.87	592.66
V	**MobileNetV2 Centrally Pretrained (Server Node)**	2.58	3.13	3.92	3.92	2.71	532.84

Memory costs evaluated the percentage of memory (RAM and swap) used, percentage of memory needed for current workload in relation to the amount of total memory (RAM and swap), number of kb paged in by the system per second, number of kb paged out by the system per second, number of page faults (major and minor) made by the system per second, and total context switches [48].



Memory used is the percentage of memory (RAM and swap) used averaged over 20 federated training rounds. The primary client nodes did not show a reduction in memory costs; however, the server nodes showed minimal impacts on memory used indicating that the server nodes could be tasked with additional requirements.

Figure 28. RPi 4B Memory Percentage Used for Experiments I-V.



Context switches are averaged over 20 federated training rounds. The MobileNetV2 primary client node demonstrated an equivalent number of context switches, which is likely do to the large CelebA datasets on each primary client node that are required for federated learning.

Figure 29. RPi 4B Context Switches Per Second for Experiments I-V.

C. MODEL PERFORMANCE

Model performance for Experiments II-V was performed using the Keras API evaluate method on test dataset partitions that were disjoint from train datasets. Experiments II-III utilized well-performing hyperparameters (E=1, B=10, η = .15) from McMahan et al. research. Experiments II and III used the same dataset and model architecture as McMahan et al., so it was determined to use the same hyperparameters. Experiment IV's dataset is an expanded version of Experiment II and III's dataset and used the same model architecture but composed of 52 additional classes (62 total classes). A reduced learning rate and increase in local edge device epochs, similar to Nillson et al., was utilized in order to achieve improved performance (E=5, B=20, η = .088).

Experiment V included a validation dataset to assist in determining effective hyperparameters since this was a much more complex model architecture. Nilsson et al. performed extensive research determining the optimal hyperparameters for the FedAvg algorithm and determined that the optimal hyperparameters were E=10, B=20, and η = .088. The current research used these hyperparameters as a starting point and ran multiple federated training rounds on similar hyperparameters (E =1, 5, and 10; B=1, 10, 20; and η=.01, .05, .15), using validation accuracy as a benchmark for hyperparameter choice. Ultimately it was determined that Nilsson et al.'s optimal hyperparameters performed best on the tested hyperparameters; however, E=5 was chosen to conserve battery life as multiple epochs increase current draw during federated training rounds.

In Experiments II-V, one federated training round consisted of:

1. Primary client nodes perform model training on a random partition of their local training dataset (see Figure 30)

2. Primary client nodes evaluate model performance utilizing local test dataset

3. Primary client nodes extract updated local model weights

4. Primary client nodes send the updated local model to the server node

5. Server node aggregates and averages the local model weights

6. Server node sends the updated global model to the primary client nodes

7. Primary client nodes update their local model weights with the newly updated global model

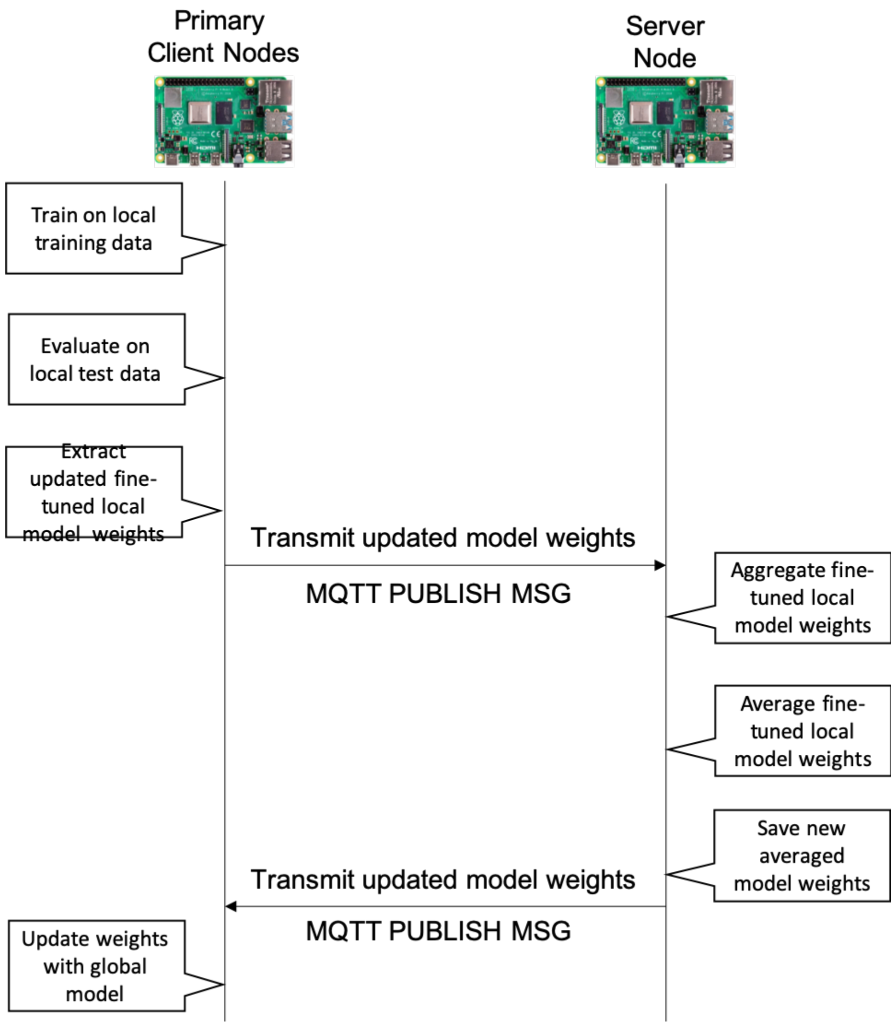
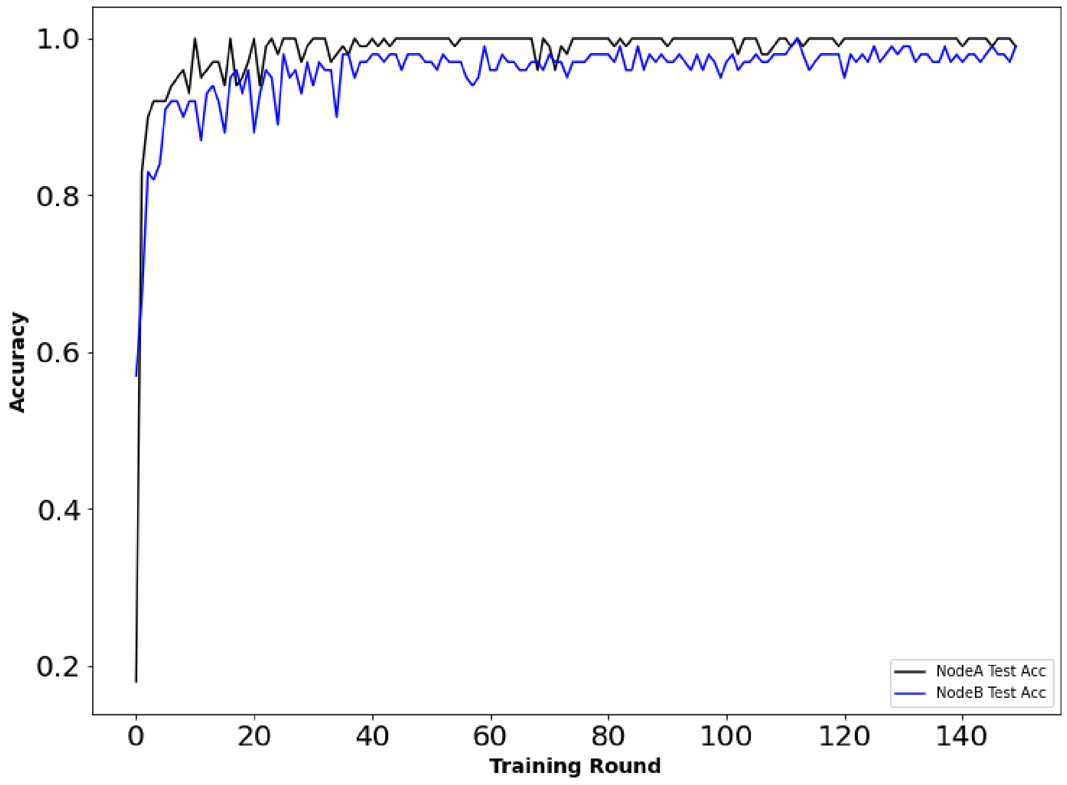

Primary Client Nodes
Server Node
Train on local training data
Evaluate on local test data
Extract updated fine-tuned local model weights
Transmit updated model weights
MQTT PUBLISH MSG
Aggregate fine-tuned local model weights
Average fine-tuned local model weights
Save new averaged model weights
Transmit updated model weights
MQTT PUBLISH MSG
Update weights with global model

Figure 30. Federated Training Round Cycle for Experiments II-V.

Each federated learning experiment experienced improvements in accuracy through federated averaging. The pretrained models did not experience large improvements, but did show improvements from the initial training round. This is similar to the findings of Stremmel and Singh in their research on a pretrained federated fine-tuned long short-term memory (LSTM) model on next word prediction (NWP) using a Stack Overflow dataset [29]. Although accuracy improved, it was observed that there was increased variability in accuracy and test loss across training rounds for the pretrained models. This variability is likely due to the fact that the base layers are frozen and the weights in these layers do not update during federated training rounds. These frozen weights are reliant on the previous training they experienced during centralized pretraining. If pretraining was not optimally performed or not enough data was used to pretrain, these weights are likely not fully optimized. While this federated learning structure may be limited in its ability to improve, it provides high accuracy from early training rounds and improves security since only a select number of parameters are shared.

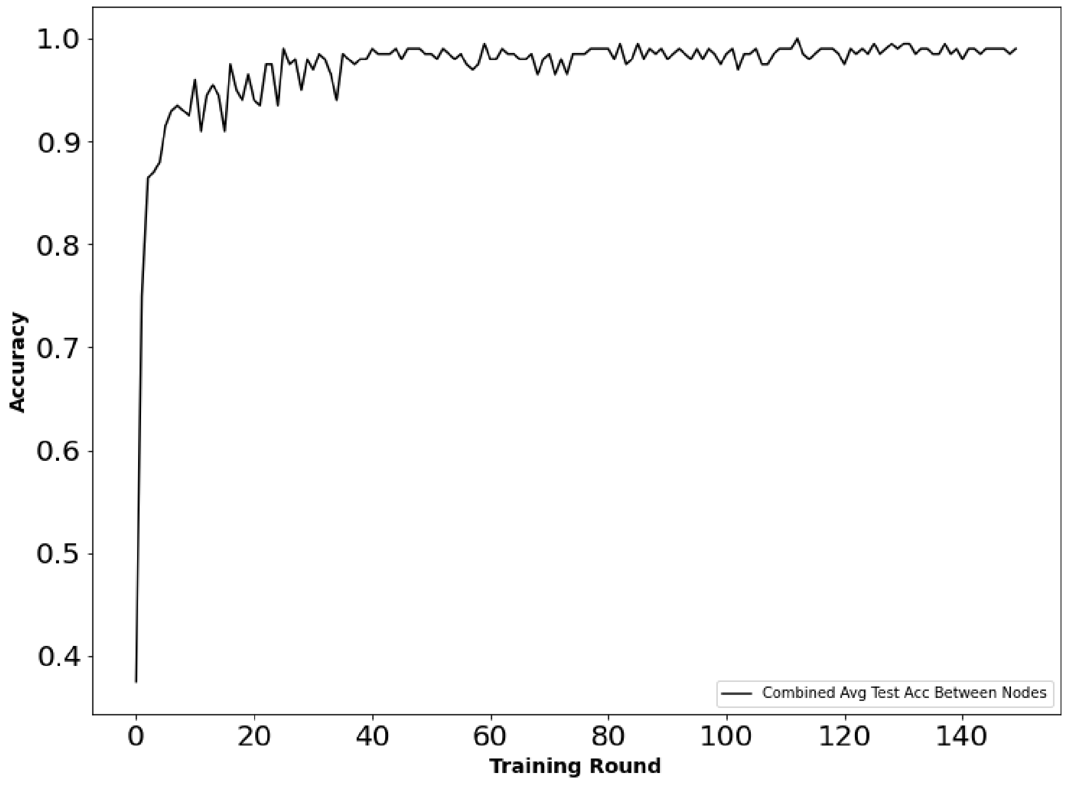
1. Randomly Initialized MNIST (Experiment II)

The randomly initialized MNIST model (Experiment II) utilized train accuracy, train loss, test accuracy and test loss to analyze model performance. Primary client node performance was evaluated individually and average performance between nodes was also captured. NodeA had an initial test accuracy of 18.00% after federated training round 0, while NodeB had an initial test accuracy of 57.00%. After 150 federated training rounds, NodeA had an overall average test accuracy of 98.37% and NodeB had an overall average test accuracy of 95.75% (see Figure 31). The average accuracy across the primary client nodes was 97.06% (see Figure 32). Average training loss for the primary client nodes at federated training round 0 was 1.978 and after 150 federated training rounds average training loss dropped to 0.1026 (see Figure 33).



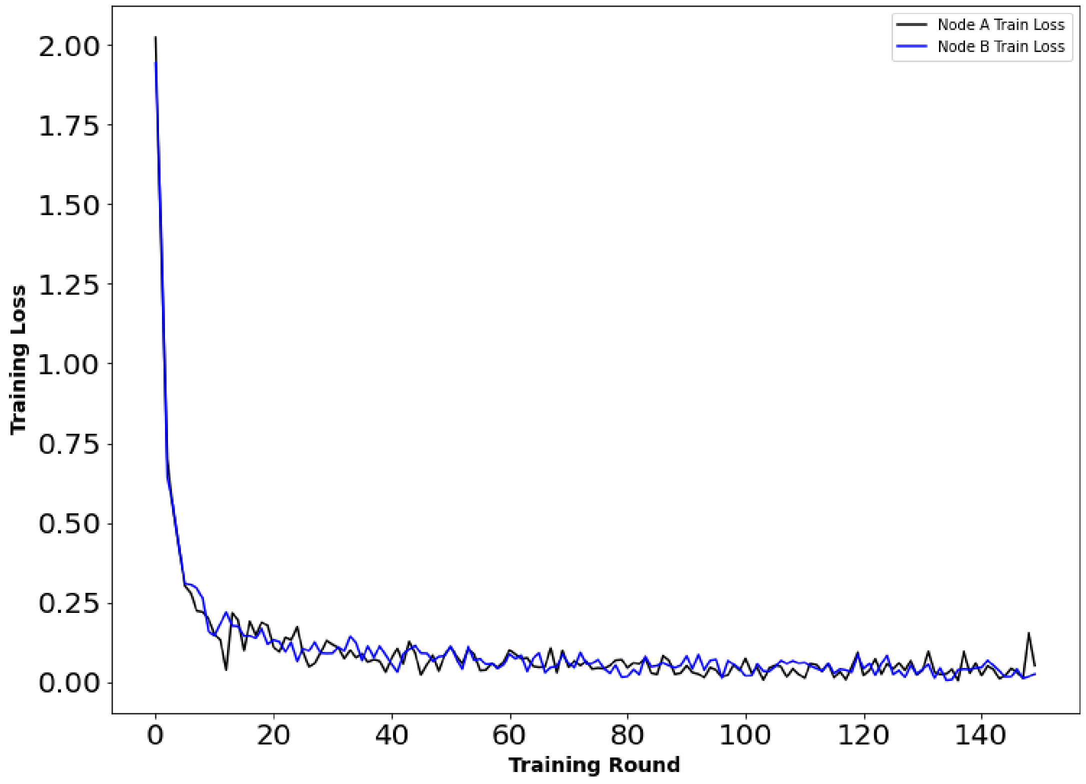
NodeA had an initial test accuracy of 18% after federated training round 0, while NodeB had an initial test accuracy of 57%. After 150 federated training rounds, NodeA had an overall average test accuracy of 98.37% and NodeB had an overall average test accuracy of 95.75%.

Figure 31. Randomly Initialized MNIST CNN (Experiment II) Individual Node Test Accuracy.



The average accuracy across the primary client nodes was 97.06%.

Figure 32. Randomly Initialized MNIST CNN (Experiment II) Average Accuracy for NodeA and NodeB.



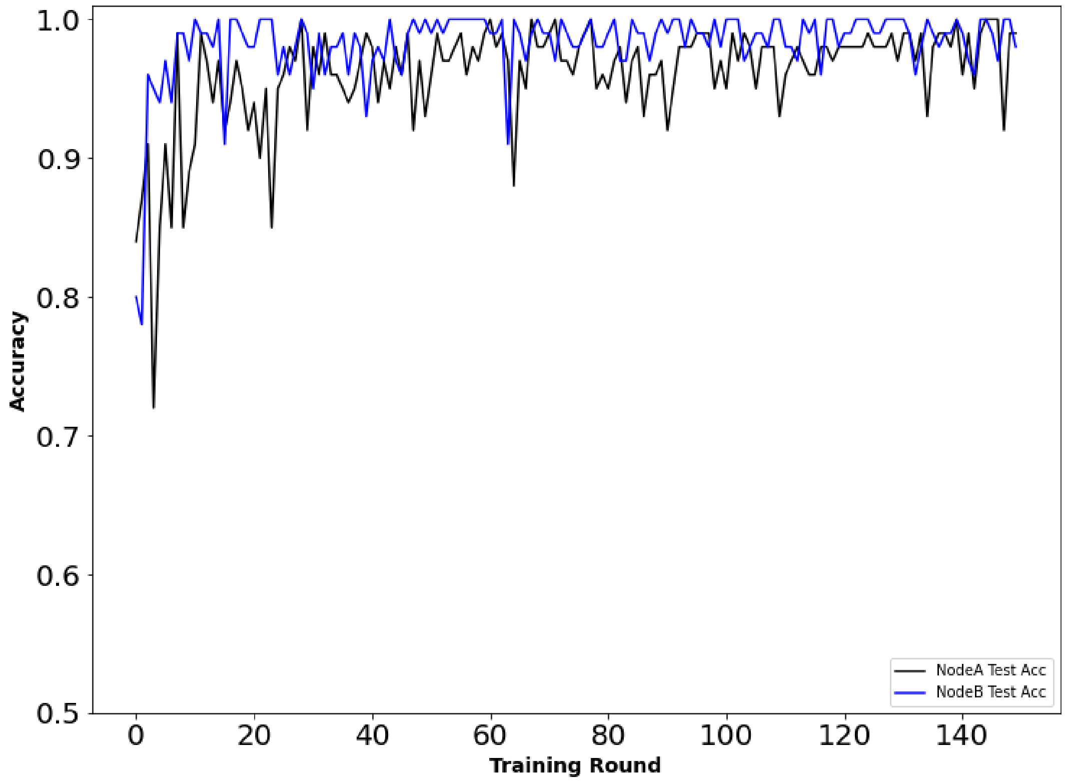
Average training loss for the primary client nodes at federated training round 0 was 1.635 and after 150 federated training rounds average loss dropped to 0.1069.

Figure 33. Randomly Initialized MNIST CNN (Experiment II) Training Loss for NodeA and NodeB.

These results demonstrate that with two primary client nodes executing federated learning can achieve high accuracy with a relatively small training dataset used per federated training round. The dataset partitions were the same sizes utilized by McMahan et al. The current research utilized two primary Client nodes per federated training round with a new random training data partition per round, while McMahan et al. utilized 10 client nodes per round from a pool of 100 client nodes. The deviation in the current research was made in order to move federated learning from a simulated environment, where a large number of client nodes can be utilized, to a deployed federated learning environment that includes a limited hardware setup.

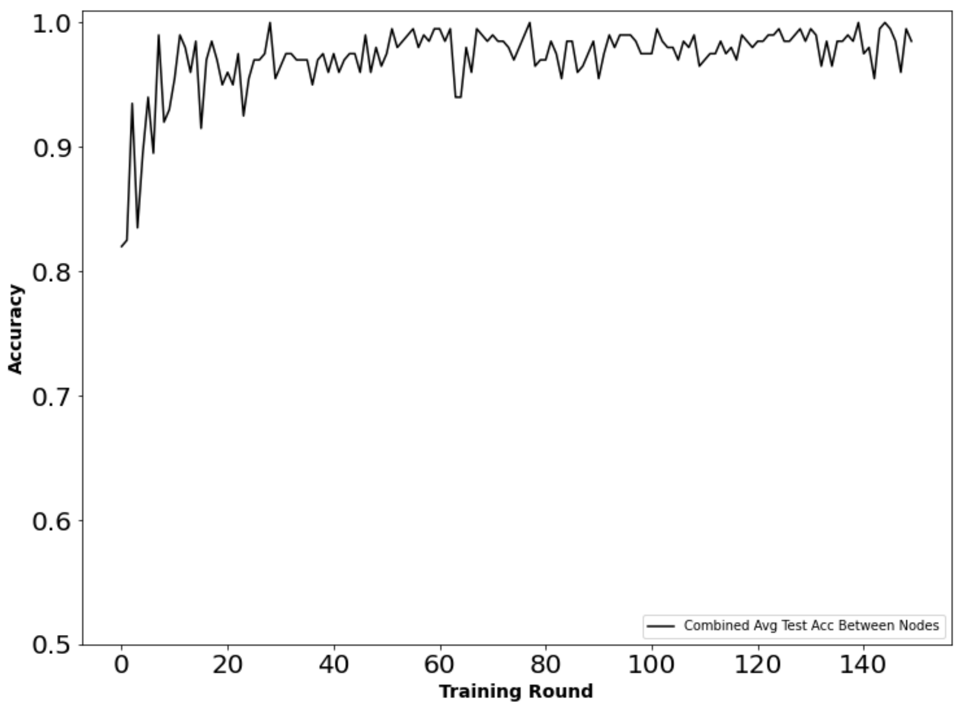
2. Centrally Pretrained MNIST (Experiment III)

The centrally pretrained MNIST model (Experiment III) utilized train accuracy, train loss, test accuracy and test loss to analyze model performance. Primary client node performance was evaluated individually and average performance between nodes was also captured. NodeA had an initial test accuracy of 84.00% after federated training round 0, while NodeB had an initial test accuracy of 80.00%. After 150 federated training rounds, NodeA had an overall average test accuracy of 96.12% and NodeB had an overall average test accuracy of 98.29% (see Figure 34). The average accuracy across the primary client nodes was 97.20% (see Figure 35). Average training loss for the primary client nodes at federated training round 0 was 0.7923 and after 150 federated training rounds average training loss dropped to 0.0867 (see Figure 36).



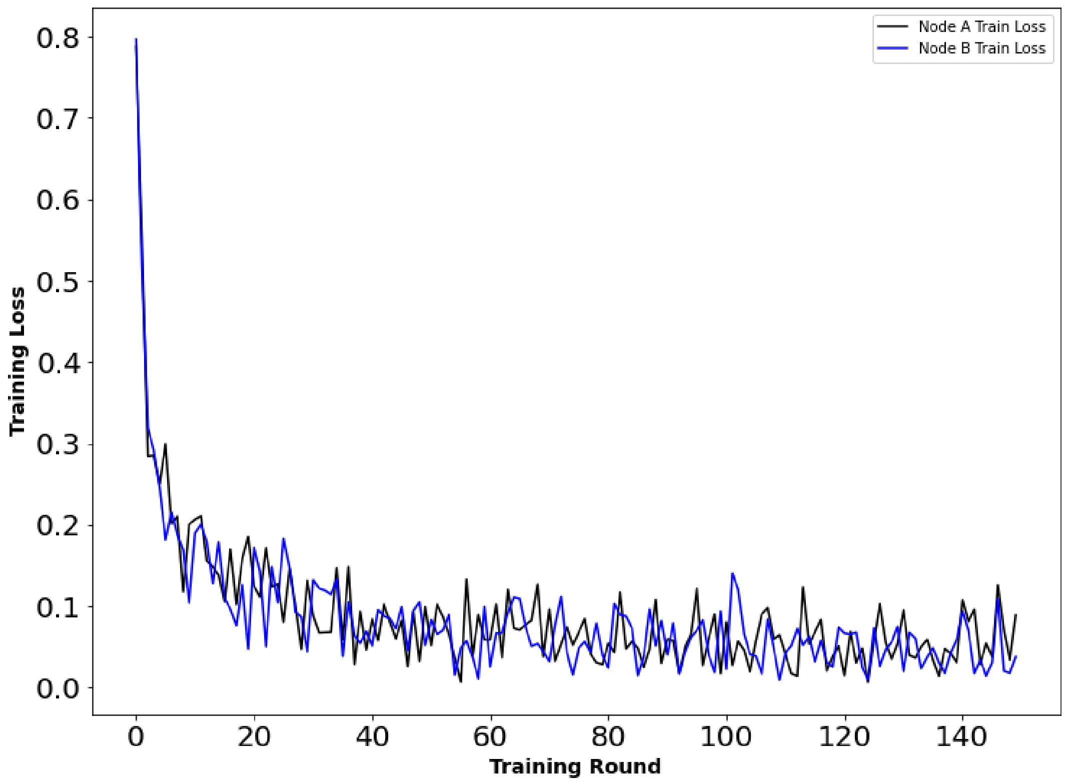
NodeA had an initial test accuracy of 84.00% after federated training round 0, while NodeB had an initial test accuracy of 80.00%. After 150 federated training rounds, NodeA had an overall average test accuracy of 96.12% and NodeB had an overall average test accuracy of 98.29%.

Figure 34. Centrally Pretrained MNIST CNN (Experiment III) Individual Node Test Accuracy.



The average accuracy across the primary client nodes was 97.20%.

Figure 35. Centrally Pretrained MNIST CNN (Experiment III) Average Accuracy for NodeA and NodeB.



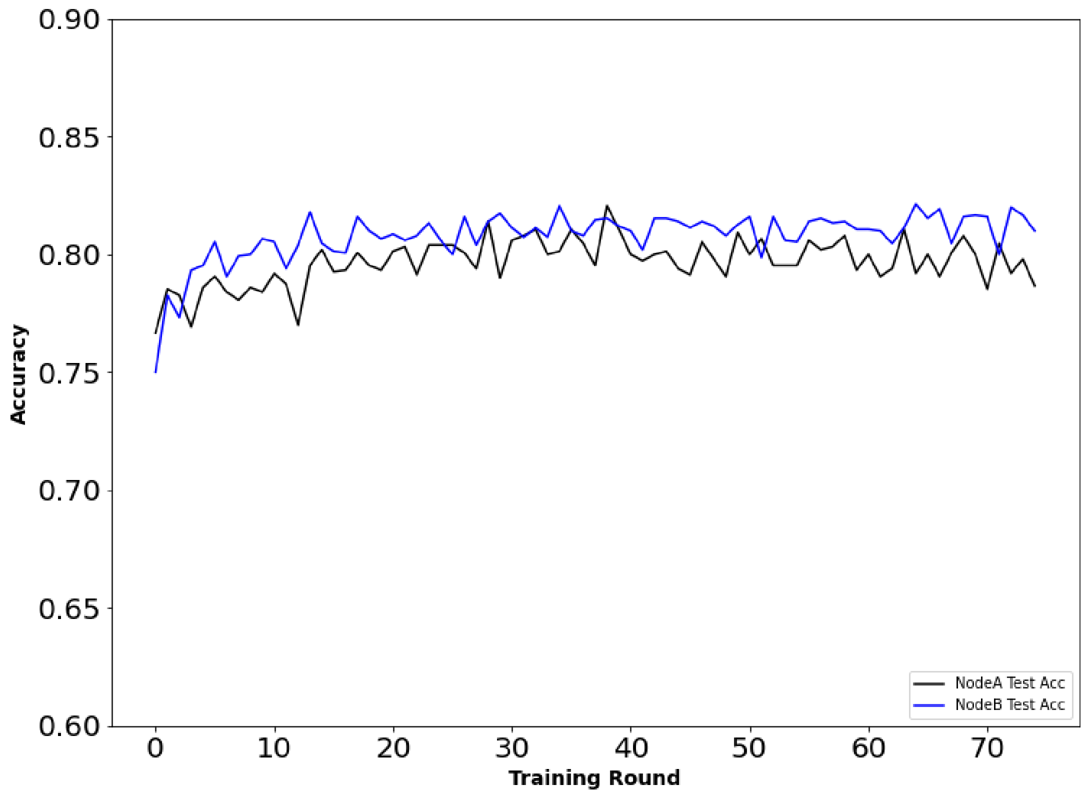
Average training loss for the primary client nodes at federated training round 0 was 0.7923 and after 150 federated training rounds average loss dropped to 0.0867.

Figure 36. Centrally Pretrained MNIST CNN (Experiment III) Training Loss for NodeA and NodeB.

The centrally pretrained model conducted federated averaging on the final two dense layers of the pretrained model which resulted in a higher accuracy at training round 0 (Experiment II had an initial average accuracy of 37.50%, while Experiment III had an initial average accuracy of 82.00%). The centrally pretrained MNIST model did not show any noticeable improvements after federated training round 6. During training rounds 0 thru 6, the average accuracy between NodeA and NodeB was 87.78%, and after training round 7 the average accuracy increased to 97.66%. This may indicate that a centrally pretrained model quickly reaches its maximum performance within a minimal number of training rounds, and does not noticeably improve after that point. Whereas a randomly initialized model will attain similar performance, but over many more training rounds.

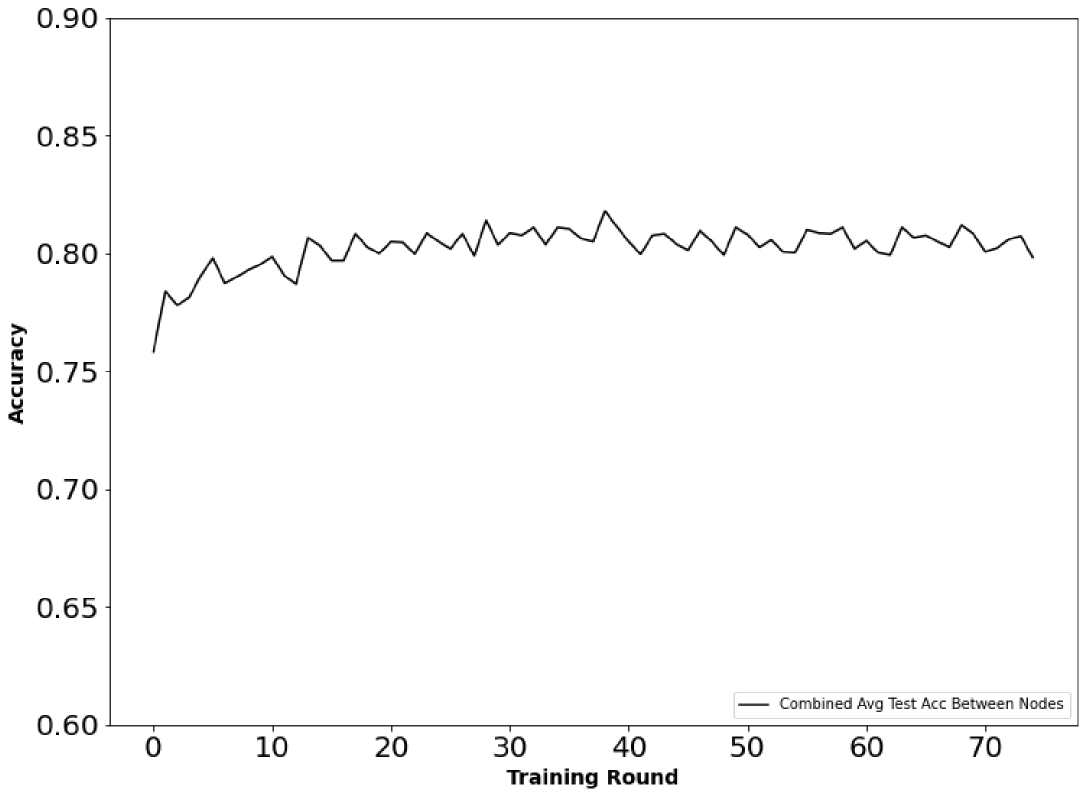
3. Extended Class Centrally Pretrained EMNIST (Experiment IV)

The extended class centrally pretrained EMNIST model (Experiment IV) utilized train accuracy, train loss, test accuracy and test loss to analyze model performance. Primary client node performance was evaluated individually and average performance between nodes was also captured. NodeA had an initial test accuracy of 76.66% after federated training round 0, while NodeB had an initial test accuracy of 75.00%. After 75 federated training rounds, NodeA had an overall average test accuracy of 79.67% and NodeB had an overall average test accuracy of 83.16% (see Figure 37. The average accuracy across the primary client nodes was 80.78% (see Figure 38). Average training loss for the primary client nodes at federated training round 0 was 0.3268 and after 75 federated training rounds average train loss dropped to 0.1019 (see Figure 39).



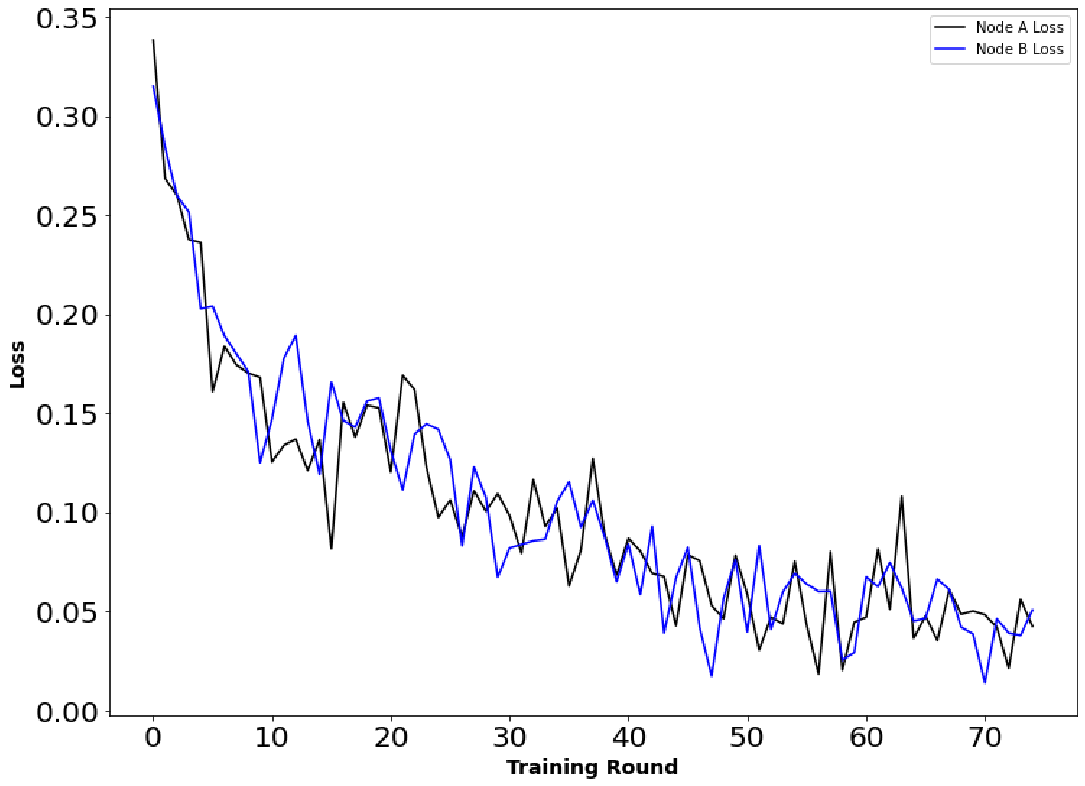
NodeA had an initial test accuracy of 76.66% after federated training Round 0, while NodeB had an initial test accuracy of 75.00%. After 75 federated training rounds, NodeA had an overall average test accuracy of 79.67% and NodeB had an overall average test accuracy of 80.78%.

Figure 37. Extended Class Centrally Pretrained EMNIST CNN (Experiment IV) Individual Node Test Accuracy.



The average accuracy across the primary client nodes was 80.22%.

Figure 38. Extended Class Centrally Pretrained EMNIST CNN (Experiment IV) Average Accuracy for NodeA and NodeB.



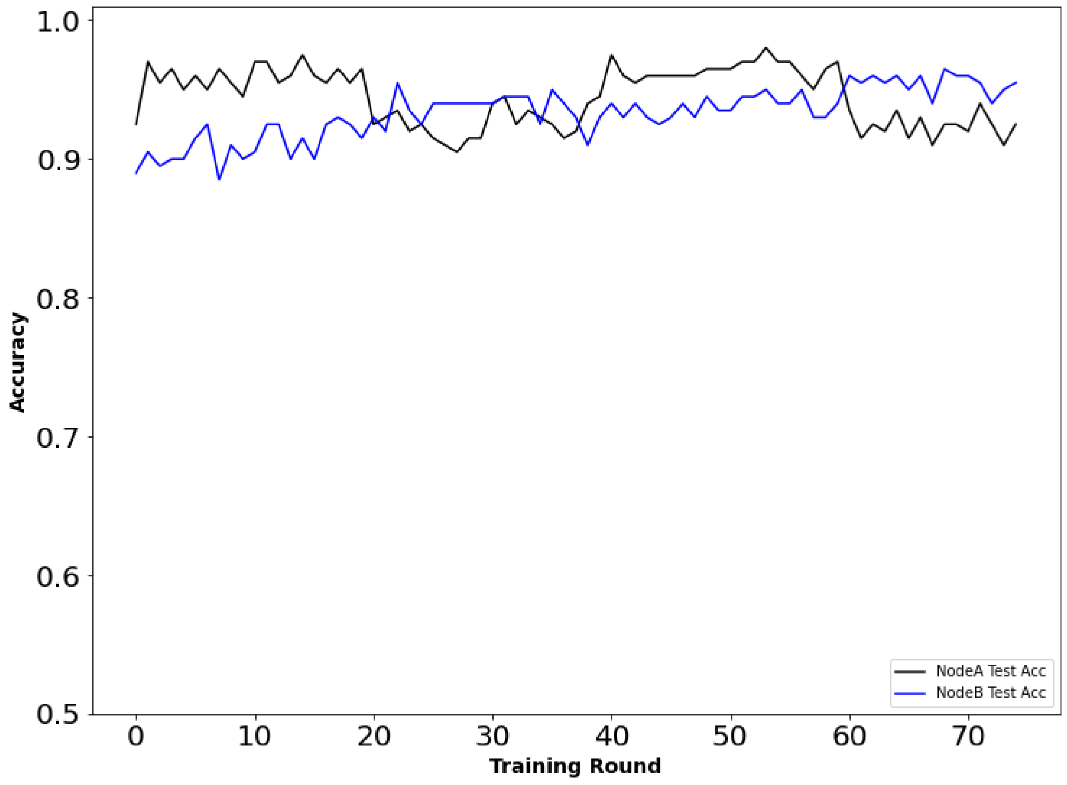
Average loss for the primary client nodes at federated training round 0 was 0.3268 and after 75 federated training rounds average train loss dropped to 0.1019.

Figure 39. Extended Class Centrally Pretrained EMNIST CNN (Experiment IV) Training Loss for NodeA and NodeB.

The centrally pretrained EMNIST model did not show any noticeable improvements after training round 12. During training rounds 0 thru 12, the average accuracy between NodeA and NodeB was 75.83%, and after training round 12 the average accuracy increased to 80.22%. This follows the findings of Experiment III that a centrally pretrained model quickly reaches its maximum performance within a minimal number of training rounds, and does not noticeably improve after that point. Whereas a randomly initialized model will attain similar performance, but over many more training rounds.

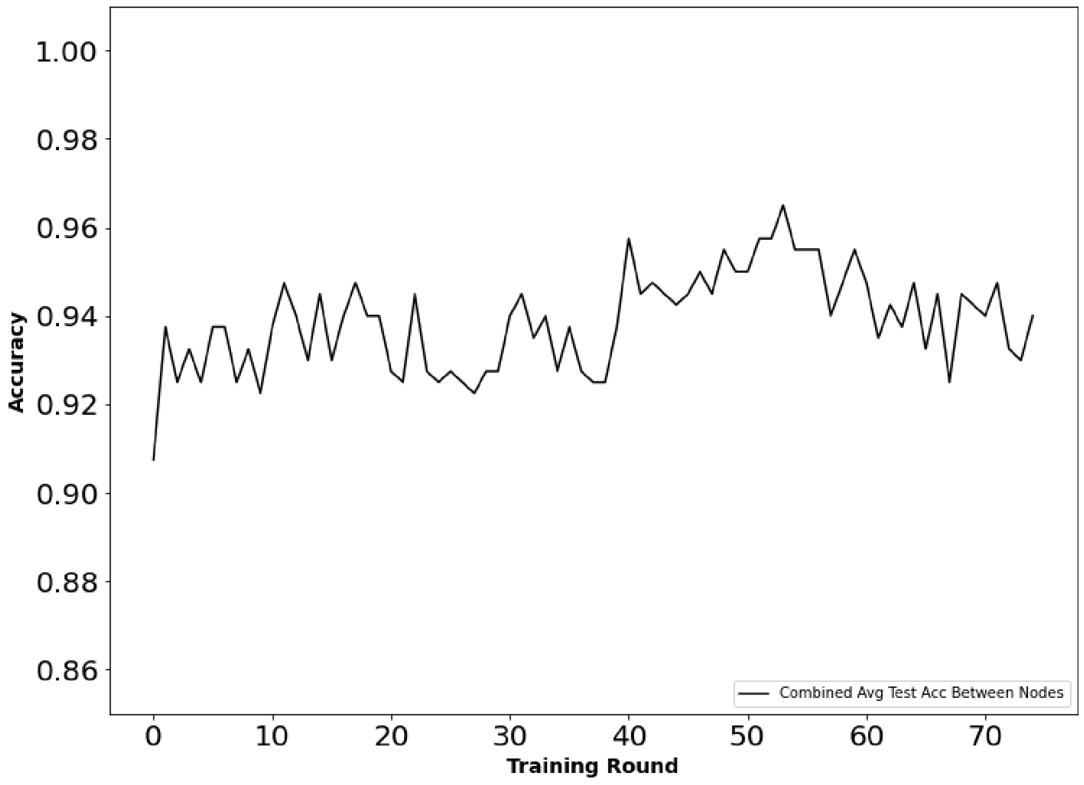
4. MobileNetV2 Centrally Pretrained CelebA (Experiment V)

The MobileNetV2 centrally pretrained CelebA model (Experiment V) utilized train accuracy, train loss, validation accuracy, validation loss, test accuracy and test loss to analyze model performance. Primary client node performance was evaluated individually and average performance between nodes was also captured. NodeA had an initial test accuracy of 92.50% after federated training Round 0, while NodeB had an initial test accuracy of 89.00%. After 75 federated training rounds, NodeA had an overall average test accuracy of 94.28% and NodeB had an overall average test accuracy of 93.49% (see Figure 40). The average accuracy across the primary client nodes was 93.89% (see Figure 41). Average train loss for the primary client nodes at federated training round 0 was 0.00876 and after 75 federated training rounds average train loss dropped to 0.00456 (see Figure 42).



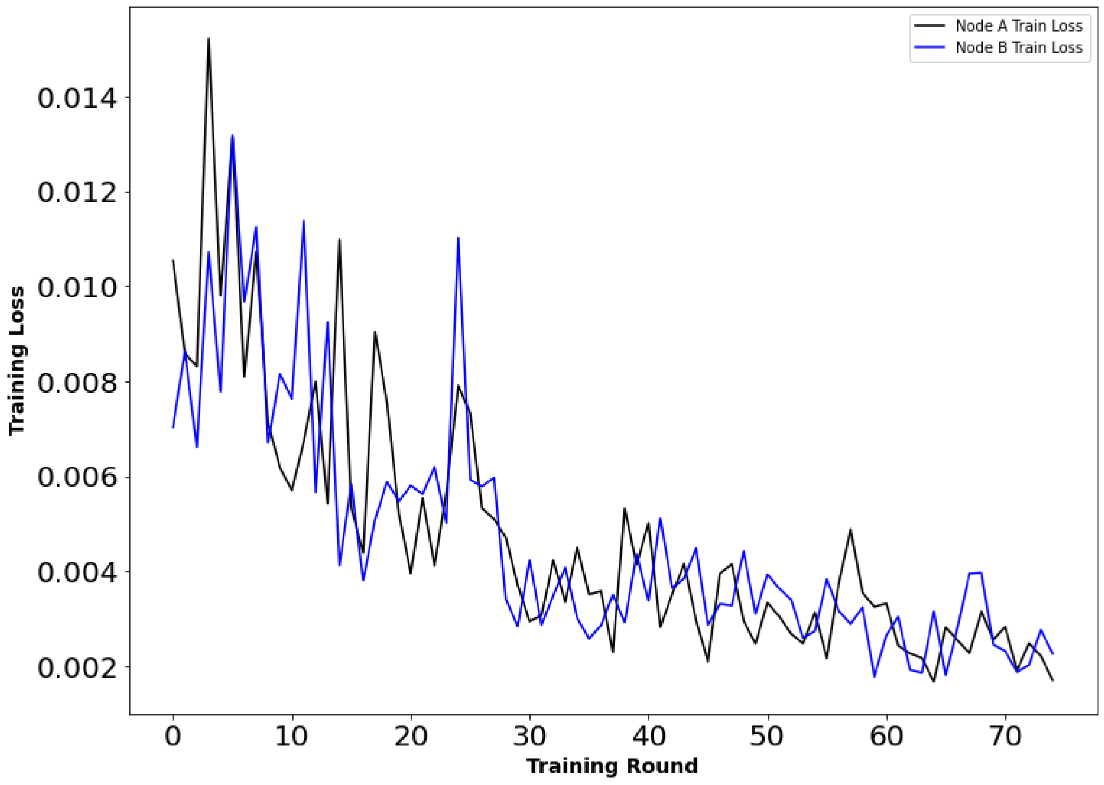
NodeA had an initial test accuracy of 92.50% after federated training round 0, while NodeB had an initial test accuracy of 89.00%. After 75 federated training rounds, NodeA had an overall average test accuracy of 94.28% and NodeB had an overall average test accuracy of 93.49%.

Figure 40. MobileNetV2 Centrally Pretrained Model (Experiment V) Individual Node Test Accuracy.



Average accuracy for NodeA and NodeB. The average accuracy across the primary client nodes was 93.89%.

Figure 41. MobileNetV2 Centrally Pretrained Model (Experiment V).



Average train loss for the primary client nodes at federated training round 0 was 0.00876 and after 75 federated training rounds average train loss dropped to 0.00456.

Figure 42. MobileNetV2 Centrally Pretrained Model (Experiment V) Training Loss for NodeA and NodeB.

The centrally pretrained MobileNetV2 model did not show any noticeable improvements after training round 10. During training rounds 0 thru 10, the average accuracy between NodeA and NodeB was 92.82%, and after training round 12 the average accuracy increased to 94.04%. This follows the findings of Experiments II and III that a centrally pretrained model quickly reaches its maximum performance within a minimal number of training rounds, and does not noticeably improve after that point. Whereas a randomly initialized model will attain similar performance, but over many more training rounds.

In developing the final hyperparameters and model architecture for Experiment V, local edge device dataset size, type of centrally pretrained model, and model layer from which to perform federated fine-tuning was analyzed on the edge device architecture.

1. **Centrally pretrained model**: Evaluated how performance is impacted by the type of transfer learning conducted on the pretrained model prior to deployment to an edge device network.

- Feature extraction with a minimal dataset centrally pretrained
- Feature extraction with a larger dataset centrally pretrained
- Fine-tuning on the feature extracted larger dataset

2. **Local dataset size**: Evaluated how the local dataset size on an edge device impacts performance. While dedicated workstations can support large local datasets, embedded hardware is limited in the amount of data it can process to train a deep learning model. An understanding of how much data can be utilized for training on embedded hardware, while still allowing for improvements in performance is valuable.

3. **Fine-tuning layer**: Within a federated learning scheme, it is important to understand what is the ideal layer from which to fine-tune a state-of-the-art machine learning model, like MobileNetV2.

a. MobileNetV2 Centrally Pretrained Models

Three centrally pretrained MobileNetV2 models were developed and evaluated for federated fine-tuning performance. A feature extracted model using a minimal size dataset (minimal feature extraction model), a feature extraction model with a significantly larger dataset (large feature extraction model), and a fine-tuned model (fine-tuned centrally pretrained model) that implemented fine-tuning on the feature extracted larger dataset model (see Table 12). The fine-tuned model was fine-tuned up through block 16 of the MobileNetV2 base architecture (Layer 144) which included 886,080 total trainable parameters. All models were trained with an early stopping callback for a validation loss minimum delta of 0.001 and patience of five epochs. These models were each pretrained centrally on a MacBook laptop prior to deployment to edge devices.

Table 12. Experiment V MobileNetV2 Centrally Pretrained Models.

	Model	Transfer Learning Type	Dataset	Hyperparameters	Server Validation Metrics
I	**Minimal Feature Extraction Model**	Feature Extraction	CelebA Train: 10,000 Test: 2,000	Epochs: 24 (early stopped) LR: .0001 Batch Size: 20 Val Split: 0.20	Loss: 0.3244 Acc: 85.50%
II	**Large Feature Extraction Model**	Feature Extraction	CelebA Train: 48,000 Test: 12,000	Epochs: 19 (early stopped) LR: .0001 Batch Size: 20 Val Split: 0.20	Loss: 0.2957 Acc: 88.59%
III	**Fine-Tuned Centrally Pretrained Model**	Fine-Tuning	CelebA Train: 48,000 Test: 12,000	Epochs: 29 (early stopped) LR: .00001 Batch Size: 20 Val Split: 0.20	Loss: 0.118 Acc: 93.40%

Three pretrained models were developed to analyze federated fine-tuning performance based on the pretrained model.

b. MobileNetV2 Performance by Local Dataset Size

Each of the three pretrained models were tested with three various sizes of local datasets for training and testing on the edge devices. The small dataset (Dataset I) included 100 random CelebA train images, 50 random CelebA validation images, 50 random

CelebA test images per training round (see Table 13). The medium dataset (Dataset II) included 250 random CelebA train images, 100 random CelebA validation images, and 100 random CelebA test images per training round. The large dataset (Dataset III) included 500 random CelebA train images, 200 random CelebA validation images, and 200 random CelebA test images. The training images were randomly selected each successive federated training round. The overall dataset sizes were 3,000 images per node for the small dataset, 6,000 images per node for the medium dataset and 12,000 images per node for the large dataset.

Table 13. Experiment V MobileNetV2 Pretrained Model CelebA Dataset Partitions

	Dataset	Image Shape	Train Images Per Round	Test Images Per Round	Validation Images per Node	Total Images per Node
I	Small	(96, 96, 3)	100	50	50	3,000
II	Medium	(96, 96, 3)	250	100	100	6,000
III	Large	(96, 96, 3)	500	200	200	12,000

CelebA images were originally (178, 218, 3), but were resized to (96, 96, 3) in order to conserve memory on RPi's.

Results indicate that a federated learning edge device network will exhibit the best performance utilizing a centrally pretrained fine-tuned model with the largest local dataset an edge device can support (see Table 14). The largest average performance gains were exhibited with a smaller local dataset (max exhibited was a 6.76% improvement with the large dataset model); however, the smaller local dataset did not achieve an overall average accuracy as high as the fine-tuned model utilizing the largest local dataset (overall average accuracy was 93.89%).

Table 14. Experiment V MobileNetV2 Pretrained Model Accuracy on Edge Device.

Centrally Pretrained Model	Local Dataset Size per Training Round	Training Round 0 Avg Accuracy	Overall Avg Node Accuracy	Avg Node Improvement
Minimal Feature Extraction Model	100/50/50	83.00%	89.11%	+6.11%
Minimal Feature Extraction Model	250/100/100	88.00%	90.13%	+2.13%
Minimal Feature Extraction Model	500/200/200	86.50%	89.66%	+3.16%
Large Feature Extraction Model	100/50/50	77.00%	83.76%	+6.76%
Large Feature Extraction Model	250/100/100	80.00%	85.34%	+5.34%
Large Feature Extraction Model	500/200/200	88.00%	89.96%	+1.96%
Fine-Tuned Model	100/50/50	91.00%	91.66%	+0.66%
Fine-Tuned Model	250/100/100	89.00%	89.65%	+0.65%
Fine-Tuned Model	500/200/200	90.75%	93.89%	+3.14%

On device hyperparameters were E = 5, B = 20, η = .01, TR = 25. Each pretrained model (minimal feature extraction, large feature extraction, fine-tuned model) was tested with the three different local dataset sizes. It can be observed that the smaller the dataset the larger the overall improvement, but the best overall performance resulted from the largest dataset partitions.

c. *MobileNetV2 Pretrained Model Fine-Tuning Layer Performance*

Within a federated learning scheme, it is important to understand what is the ideal layer from which to fine tune a state-of-the-art machine learning model, like MobileNetV2. The current research was designed to implement on-device fine-tuning at select layers within the MobileNetV2 architecture (see Table 15). The final MobileNetV2 block (Block 16) was analyzed for on-device fine-tuning. The fine-tuning occurred at the block 16 convolutional layers and the final out convolutional layer (layer 144, layer 147, layer 150 and layer 152, respectively).

Additionally, the fine-tuned pretrained model was also trained up through block 16 on the server prior to edge device deployment. Given that the current research is focused on developing a more secure AI based military installation surveillance system, the minimum number of parameters are selected for federated learning and cross network communication. Research without a security focus could fine-tune more layers within a model, but this would increase the risk for an adversary to rebuild a model based on intercepted parameters.

Table 15. Experiment V MoblieNetV2 Layers Fine-Tuned on Edge Device

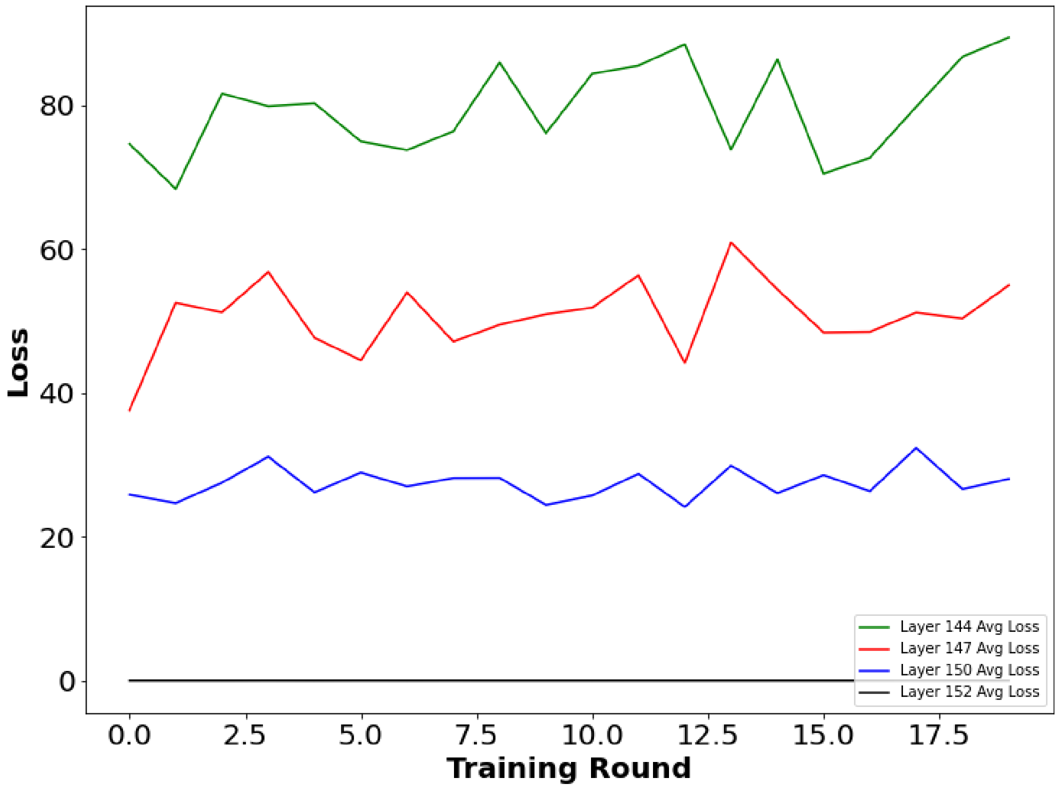
MobileNetV2 Layer Index Number	**MobileNetV2 Layer Name**	**Trainable Parameters**
144	Block 16 Expand Conv2D	894,409
147	Block 16 Depthwise Conv2D	736,961
150	Block 16 Project Conv2D	724,481
152	Out Conv2D	406,001

It was determined that the ideal layer from which to perform federated fine-tuning on the MobileNetV2 model (block 16 and out Conv2D) under the current federated learning architecture is the out convolutional layer (MobileNetV2 layer index 152) (see Table 16). The convolutional layers of block 16 each exhibited average accuracies around 50%, which indicates no learning occurred in a binary classification problem (see Figure 43). It was also observed that the loss increased with each added layer fine-tuned (see Figure 44). The out convolutional layer had an average loss of 0.0080, while the block 16 expand convolutional layer had an average loss of 79.4773.

Table 16. Table 16: Experiment V MobileNetV2 Layers Fine-Tuned on Edge Device Results.

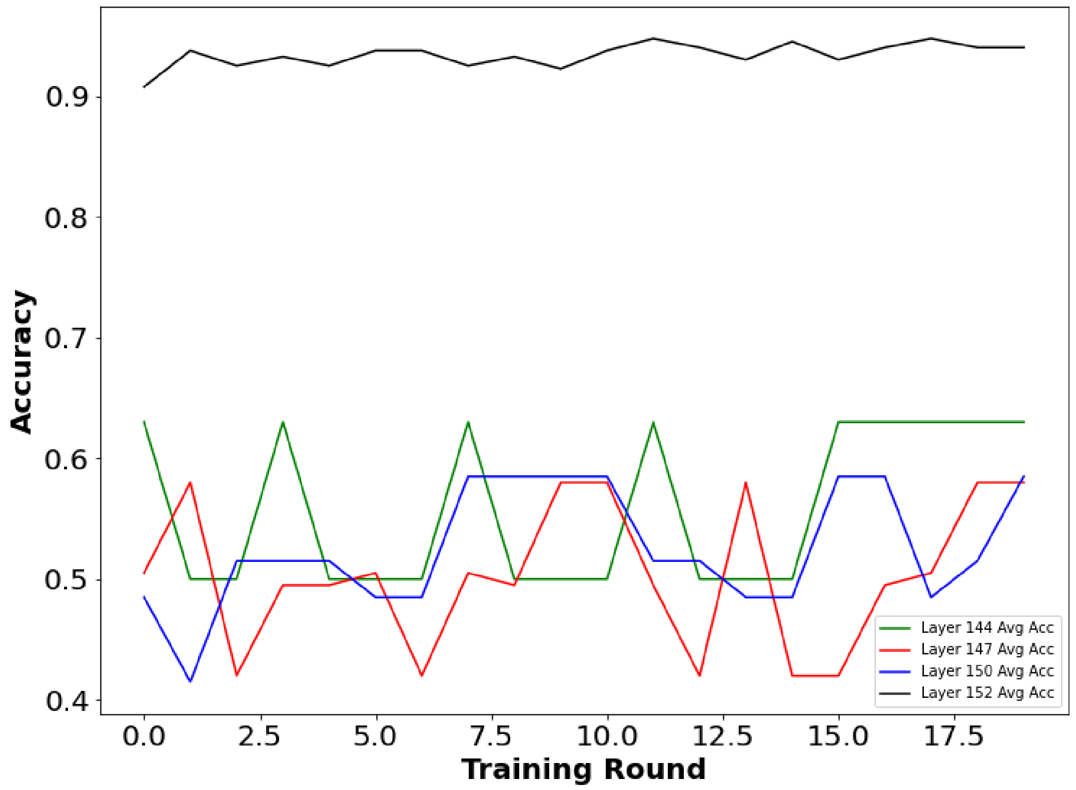
MobileNetV2 Layer Index	**MobileNetV2 Layer Name**	**Average Loss**	**Average Accuracy**
144	Block 16 Expand Conv2D	79.477	55.85%
147	Block 16 Depthwise Conv2D	50.655	50.65%
150	Block 16 Project Conv2D	27.443	52.55%
152	Out Conv2D	0.0080	93.40%

On-device hyperparameters were E = 5, B = 20, η = .088, TR = 20, train sample size = 500, validation sample size = 200, test sample size = 200.



Layers fine-tuned were block 16 expand Conv2D (Layer 144), block 16 depthwise Conv2D (layer 147), block 16 project Conv2D (Layer 150) and out Conv2D (layer 152). On device hyperparameters were E = 5, B = 20, η = .088, TR = 20, train sample size = 500, validation sample size = 200, test sample size = 200.

Figure 43. On Device Average Training Loss for NodeA and NodeB Based on Layer Federated Fine-Tuned.



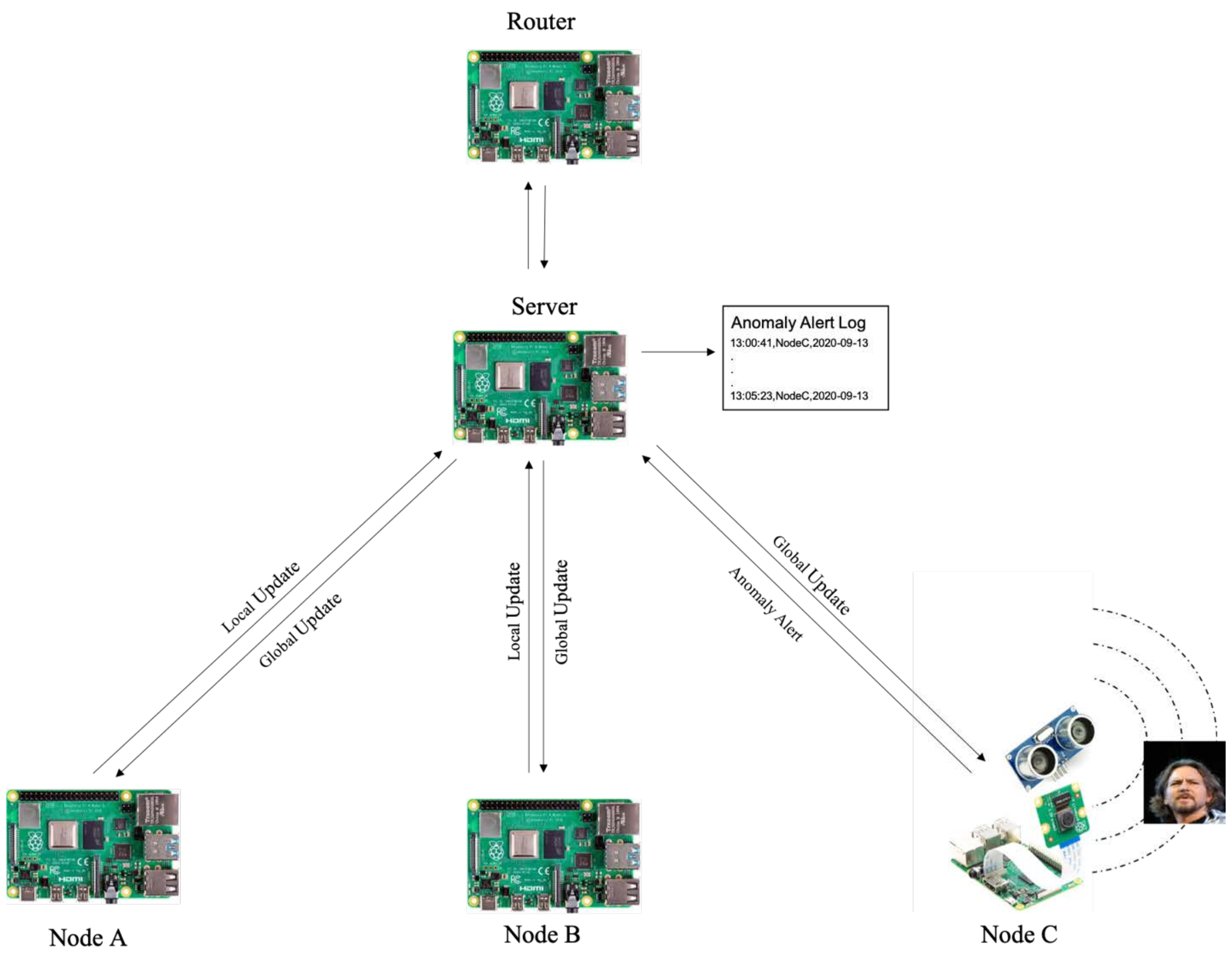
Layers fine-tuned were block 16 expand Conv2D (layer 144), block 16 depthwise Conv2D (layer 147), block 16 project Conv2D (layer 150) and out Conv2D (layer 152). On device hyperparameters were E = 5, B = 20, η = .088, TR = 20, train sample size = 500, validation sample size = 200, test sample size = 200.

Figure 44. On Device Average Test Accuracy for NodeA and NodeB Based on Layer Federated Fine-Tuned.

It is not fully known why the performance drops significantly when performing federated fine-tuning when adding in additional layers past the out convolutional layer. In a traditional fine-tuning scenario with a centralized dataset the performance should improve with each additional layer fine-tuned. It is assessed that this reduced performance is occurring since a very small local dataset (500 images) is changing the weights developed by a much larger dataset (48,000). Overall, these results indicate that it is best to perform federated fine-tuning from the final convolutional layer in the model architecture.

d. End to End FedAvg Edge Device Network

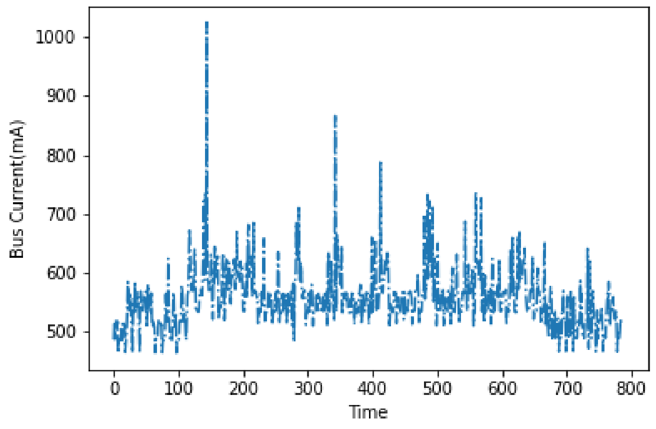
The final experiment was designed as a proof of concept that an end-to-end edge device network could execute federated fine-tuning on battery power. This experiment builds off of Experiment V, which used a state-of-the-art machine learning model to perform federated fine-tuning. The Netgear Nighthawk router was replaced with an RPi 4B as a router using hostapd software. The primary client nodes had the same setup as in Experiment V and a secondary client node was added with an RPi camera that was triggered by a HC SR04 ultrasonic sensor (see Figure 45. The ultrasonic sensor was chosen, due to the fact that ultrasonic sensors draw less current than a live camera feed or a proximity IR sensor. The RPi camera draws 280 mA on average, while the HC SR04 draws only 15 mA on average. Using an RPi camera as a sensor would decrease the nominal battery life of a secondary client node by 5 hours and 20 minutes, while the HC SR04 would only decrease the nominal battery life by 25 minutes.



Hardware included 2 primary client nodes, 1 secondary client node, 1 server node, and 1 edge router all composed of Raspberry Pi's.

Figure 45. Experiment VI Architecture.

The secondary client node included anomaly detection to alert the server node of any predictions below 95%. These anomalous predictions were sent via MQTT and recorded in a CSV file on the server node for additional human directed analysis. The secondary client node had a 9.66% increase in current draw while executing predictions once per second from the updated global model received from the server node (see Figure 46). The nominal battery life of a 10,000 mAH battery is 18 hours and 0 minutes. The average CPU load on the secondary client node was 22.64%, average RAM memory used was 6.99%, and 0.00% swap space was utilized. These results indicate that a secondary client node can be supported in a deployed network and could accept additional tasks as needed.



The power performance test for Experiment VI secondary client note began with a two-minute idle period, then 20 rounds of federated learning training rounds, followed by a two-minute idle period. The secondary client node exhibited a 9.66% increase in current draw over idle. The spikes in current draw are MQTT transmissions across the network.

Figure 46. End-to-End Edge Device Network Current Consumption for Secondary Client Node.

The RPi router transmitted an average of 24.32 packets per second and received an average of 23.49 packets per second. The average CPU load on the router was only 0.66%, average RAM memory used was 1.07%, and 0.00% swap space was utilized. These results indicate that a battery powered edge device router can easily support a deployed federated learning network.

Experiment VI did not rely on a wireless internet connection to conduct federated fine-tuning and could continuously operate for at least 18 hours with 10,000 mAH battery packs. This experiment demonstrates that federated learning can be deployed on edge devices without a cloud server, no internet connection, and completely reliant on battery power. An end-to-end edge device federated learning network of this sort could be utilized in multiple real world forward deployed military applications, such as a forward operating base security application.

V. CONCLUSIONS AND FUTURE WORK

A. SUMMARY

The goals of the current research were to evaluate the system performance of federated learning on edge devices and to analyze the model performance of a centrally pretrained state-of-the-art model conducting federated fine-tuning on an edge device network. The experiments conducted throughout the thesis process demonstrated that a multi-node architecture distributes the computational, memory, communication, and power requirements to a sufficient level in order to support federated learning on edge devices. In computational costs the federated fine tuning models (Experiments III-V) exhibited a 65.95% average reduction in CPU load over the baseline model (Experiment I) and a 10.75% average reduction in CPU temp over the baseline model. In power costs the federated fine tuning models exhibited a 56.16% average reduction in current draw and an average improvement in nominal battery life of 18.47% over the baseline model. For communication costs the baseline federated learning model (Experiment II) shared 100% of the model parameters, while the MobileNetV2 model shared 18.41% of model parameters. For memory costs the federated fine tuning servers utilized 86.55% less memory than the baseline model, which allows server nodes to be tasked with additional requirements.

Additionally, it was demonstrated that a multi-node federated fine-tuning architecture begins federated learning training with a higher accuracy over a randomly initialized model and incrementally improves over iterative federated training rounds. The centrally pretrained federated fine-tuning MNIST model (Experiment III) began training with an initial accuracy improvement of 53.94% over the randomly initialized federated learning MNIST model (Experiment II) and achieved an average accuracy of 97.66% within seven federated training rounds. The centrally pretrained EMNIST model demonstrated a final performance improvement of 4.39% over the initial federated training round. Finally, the MobileNetV2 model demonstrated a final performance improvement of 3.14% over the initial federated training round.

B. BENEFITS

The current research exhibited that a federated fine-tuning COTS edge device network supports a secure and accurate application that can be used in a military security application framework. Four primary benefits of federated fine-tuning were identified—initial accuracy, efficiency, security, and feasibility of deployment. It was demonstrated that a centrally pretrained model can initiate training at federated training round 0 with high accuracy, whereas a randomly initialized federated averaging model would take many rounds to achieve acceptable accuracy. This initial boost in performance of a federated learning network is vital in military applications that depend on high accuracy from initial deployment. In the proposed scenario of a military surveillance application, the network cannot rely on multiple training rounds for accuracy and must be highly accurate, persistent and adaptable upon deployment, while enabling the model to improve over time.

It was validated that a multi-node federated learning edge device network sufficiently distributes the computational load across the network and supports state-of-the-art model training, which had not been previously demonstrated in academic research. The reduction in computational load also results in lowered current draw on devices and in turn longer edge device battery life. A distributed network also benefits from a distributed dataset, as no single node has the full dataset. This improves efficiency as an edge device is only required to process a small local dataset versus the entire dataset of the network and yet can leverage full network dataset through the shared global model.

Additionally, an off-grid edge device network improves the overall security of the network. Experiments II-VI did not rely on a remote cloud server and were isolated from any outside networks, which ensures that no data transmits outside the local area network. This guarantees that an adversary can only intercept transmissions if they are physically located within Wi-Fi range of the network. Experiments III-VI employed centrally pretrained models that utilized only a select number of model layers for training and local model updates. Hitaj, Ateniese and Perez-Cruz identified that federated learning architectures releasing only a portion of global parameters provide stronger security and are more resistant to federated learning attacks than models that transmit the full global model [39].

Finally, the federated learning network presented in the current research is a proof-of-concept that demonstrates the possibility of a deployable federated learning network that can be utilized in forward deployed tactical scenarios. No previous research was identified that has evaluated and tested the feasibility of a battery-powered federated learning network. Commercial entities employing a federated learning architecture would typically rely on dedicated power for the network; however, many military applications would benefit from a federated learning network with no external power requirements. The networking protocol utilized in the current research is scalable, widely accepted and asynchronous, which also ensures the network is deployable. The architecture was composed of all COTS edge devices, which ensures that the network is reproducible and easily acquired for research and deployment.

C. LIMITATIONS

While the research questions proposed for the current research were supported by the analysis and results, there are draw-backs and limitations that should be addressed for any follow-on work. The MNIST and EMNIST datasets are recommended testbed datasets for machine learning and federated learning research; however, they are not real-world datasets and often result in high accuracies with little data pre-processing. The CelebA dataset contains more female photos than male photos and has been shown to demonstrate a bias toward the female category [52]. These datasets are limited in their scope and not suited for real-world deployment.

The experiments performed did not conduct federated learning training rounds until batteries were fully discharged. Therefore, the expected battery life is a nominal measurement based off the battery packs used and current draw. The architecture also utilized a limited number of Raspberry Pi’s as edge devices for federated learning. McMahan et al.’s original research design utilized 100 clients with 10 percent randomly chosen per federated training round. Finally, the edge devices used only had 4 GB RAM and did not contain a GPU. There are more advanced edge devices available with additional RAM and GPU capability to improve system performance.

D. FUTURE WORK

It is recommended that future work utilizes a dataset specifically geared towards real world video surveillance detection and classification. VIRAT is a large-scale surveillance video dataset that includes videos from stationary ground cameras and moving aerial vehicles [53]. The Live Videos (LV) Dataset contains a large collection of video surveillance sequences detecting dangerous events, such as car accidents, robberies, kidnappings and other abnormal situations [54]. The current research focused on determining if an edge device network can even support federated learning with a state-of-the-art architecture, so it was determined that the utilized datasets be standard testbeds for machine learning and federated learning.

Future work would also benefit from benchmarking additional COTS edge device architectures to compare performance. Nvidia provides several options for embedded hardware, including the Jetson Nano Developer Kit, Jetson TX2, and the Jetson AGX Xavier. The Nvidia Jetson products include GPU capabilities and up to 32 terabit operations per second (TOPS) on the AGX Xavier. The Google Coral provides a system-on-module (SoM) development board with Edge TPU and compatible modules for prototyping.

Additional research into security applications would be valuable to continued federated fine-tuning research. Much of the research surrounding computer vision involves image detection and classification; however, real-world security applications cannot waste valuable resources on classifying each object in frame. Outlier detection utilizing autoencoders would help resource constrained networks by only utilizing valuable computation on anomalous activity. Additional sensors and data sources (e.g., audio, network IDS, firewall logs) utilized in federated training could provide a more robust understanding of insider threat in military security applications.

E. CONCLUSIONS

The results of the current research demonstrate that a distributed network of edge devices can support an accurate deep learning model while addressing global model security risks through a centrally pretrained federated fine-tuned model. Deep learning applications are continually expanding and it is likely that deep learning will be implemented more and more in DoD applications. Utilizing COTS hardware and open-source software for federated fine-tuning allows the DoD to quickly develop and deploy security applications that adapt to evolving threats, while preserving the security of the application itself. In summary, this thesis has validated that federated fine-tuning supports high accuracy from initial deployment, improves efficiency through a distributed network of edge devices, provides stronger privacy through minimal parameter sharing, and has the potential for off-grid deployment in tactical scenarios.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Navy Times, “Sailor who killed two in Pearl Harbor shooting spree identified,” Dec. 6, 2019. [Online]. Available: https://www.navytimes.com/news/your-military/2019/12/06/sailor-who-killed-two-in-pearl-harbor-shooting-spree-identified/

[2] F. Chollet, *Deep Learning with Python.* Shelter Island, NY, USA: Manning Publications, 2016.

[3] A. Geron, *Hands-on Machine Learning with Scikit-Learn, Keras and Tensorflow: Concepts, Tools and Techniques to Build Intelligent Systems.* Sebastopol, CA, USA: O’Reilley Media, 2019.

[4] J. Grus, *Data Science from Scratch*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2019.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* Cambridge, MA, USA: MIT Press, 2016.

[6] Google, “Machine learning glossary.” Accessed May 13, 2020. [Online] Available:https://developers.google.com/machine-learning/glossary

[7] X. Qi and C. Liu, “Enabling deep learning on IoT edge: Approaches and evaluation.” *Proc. of the Int. Conf. on Comput. -Aided Des.*, article no. 135, pp. 367–372, Nov. 2018. [Online]. doi: https://doi-org.libproxy.nps.edu/10.1145/3240765.3243473

[8] G. Wang et al, “Interactive medical image segmentation using deep learning with image-specific fine tuning.” *IEEE Trans. on Med. Imaging*, vol. 37, no. 7, pp. 1562–1573, July 2018. [Online]. doi: 10.1109/TMI.2018.2791721

[9] G. Tucker, M. Wu, M. Sun, S. Panchapagesan, G. Fu, and S. Vitaladevuni, “Model compression applied to small-footprint keyword spotting.” InterSpeech 2016, pp 1878–1882, Sept. 8–12, 2016. [Online]. doi: http://dx.doi.org/10.21437/Interspeech.2016-1393

[10] Keras, “Keras applications.” Accessed: Oct. 6, 2020. [Online]. Available: https://keras.io/api/applications/

[11] H. Li, K. Ota, and M. Dong, “Learning IoT in edge: Deep learning for the internet of things with edge computing.” *IEEE Network*, vol. 32, no. 1, pp. 96–101, Jan. 2018. [Online]. doi: 10.1109/MNET.2018.1700202

[12] TensorFlow, “Accelerating TensorFlow Lite on Qualcomm Hexagon DSPs.” Accessed: Oct. 6, 2020. [Online]. Available: https://blog.tensorflow.org/2019/12/accelerating-tensorflow-lite-on-qualcomm.html

[13] TensorFlow, “TensorFlow Lite for microcontrollers.” Accessed: Oct. 6, 2020. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers

[14] TensorFlow, “Getting started with TensorFlow Lite.” Accessed May 20, 2020. [Online]. Available: https://www.tensorflow.org/lite/guide/get_started

[15] F. Iandola, S. Han, M. Moskewcz, K. Ashraf, W. Dally, and K. Keutzer, “SqueezeNet: AlexNet-Level accuracy with 50x fewer parameters and <0.5MB model size,” 2016. [Online]. Available: arXiv:1602.07360.

[16] T. Tan and Q. Lu, “EfficentNet: Rethinking model scaling for convolutional neural networks,” 2019. [Online]. Available: arXiv:1905.11946.

[17] A. Howard et al., “MobileNets: Efficient convolutional neural networks for mobile vision applications,” 2017. [Online]. Available: arXiv:1704:04861.

[18] S. Nikouei, Y. Chen, S. Song, R. Xu, B. Choi, and T. Faughnan, “Smart surveillance as an edge network service: From harr-cascade, SVM to a lightweight CNN,” 2018. [Online]. Available: arXiv:1805.00331.

[19] TensorFlow, “Transfer learning and fine-tuning.” Accessed Aug. 13, 2020. [Online]. Available: https://www.tensorflow.org/tutorials/images/transfer_learning

[20] Stanford University, “Convolutional neural networks. Stanford University CS231n convolutional neural networks for visual recognition.” Accessed: May 14, 2020. [Online]. Available: https://cs231n.github.io/convolutional-networks/

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

[22] 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.

[23] J. Konecny, B. McMahan, D. Ramage, and P. Richtarik., “Federated optimization: Distributed machine learning for on-device intelligence,” 2016. [Online]. Available: arXiv:1610.02527.

[24] S. Caldas et al., “LEAF: A benchmark for federated settings,” 2017. [Online]. Available: arXiv:1812.01097

[25] A. Hard et al., “Federated learning for mobile keyboard prediction,” 2018. [Online]. Available: arXiv:1811.03604.

[26] T. Yang et al., “Applied federated learning: Improving Google keyboard query suggestions,” 2018. [Online]. Available: arXiv:1812.02903.

[27] A. Nilsson, S. Smith, G. Ulm, E. Gustavsson, and M. Jirstrand, “A performance evaluation of federated learning algorithms.” *DIDL ‘18*, pp. 1–8, Dec. 10–11, 2018, Rennes, France. [Online]. doi:https://doi.org/10.1145/3286490.3286559

[28] H. Bonawitz et al., “Towards federated learning at scale: System design,” 2019. [Online]. Available: arXiv:1902.010146.

[29] J. Stremmel and A. Singh, “Pretraining federated text models for next word prediction,” 2020. [Online] Available: arXiv:2005.04828.

[30] Y. Gao et al., “End-to-end evaluation of federated learning and split learning for internet of things, “ 2020. [Online]. Available: arXiv:2003.13376.

[31] B. Liu, B. Yan, Y. Zhou, Y. Yang, and Y. Zhang, “Experiments of federated learning for COVID-19 chest x-ray images,” 2020. [Online]. Available: arXiv:2007.05592.

[32] D. Liu and T. Miller, “Federated pretraining and fine-tuning of BERT using clinical notes from multiple silos,” 2020. [Online]. Available: arXiv: 2002.08562.

[33] T. Hsu, H. Qi, and M. Brown, “Federated visual classification with real-world data distribution,” 2020. [Online]. Available: arXiv:2003.08082.

[34] P. Kairouz et al., “Advances and open problems in federated learning,” 2019. [Online]. Available: arXiv:1912.04977.

[35] M. Khodak, T. Li, L. Li, M. Balcan, V. Smith, and A. Talwalkar, “Weight sharing for hyperparameter optimization in federated learning.” *Int. Workshop on Federated Learning for User Privacy and Data Confidentiality in Conjunction with ICML 2020*. [Online]. Available: http://www.cs.cmu.edu/~mkhodak/docs/FL2020Workshop.pdf

[36] J. Mills, J. Hu, and G. Min, “Communication-Efficient Federated Learning for wireless edge intelligence in IoT.” *IEEE Internet of Things J.*, vol. 7, no. 7, pp. 5986–5994, July 2020. [Online]. doi:10.1109/JIOT.2019.2956615

[37] A. Das and R. Brunschwiler, “Privacy is what we care about: Experimental investigation of federated learning on edge devices.” *AIChallengeIoT’19*, Nov. 10–13, 2019, New York, NY. [Online]. doi:https:/doi.org/10.1145/3363347.3363365

[38] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, “Membership inference attacks against machine learning models.” *2017 IEEE Symp. on Security and Privacy,* pp. 3–18, San Jose, CA, 2017. [Online]. doi:10.1109/SP.2017.41

[39] B. Hitaj, G. Ateniese, and F. Perez-Cruz, “Deep models under the GAN: Information leakage from collaborative deep learning,” 2017. [Online]. Available: arXiv: 1702.07464.

[40] F. Turchini, L. Seidenari, T. Uricchio, and A. Del Bimbo, “Deep learning based surveillance system for open critical areas.” *Inventions*, vol. 4, no. 69., 2018. [Online]. doi:10.3390/inventions/3040069

[41] Y. LeCun, C. Cortes, and C Burges, “The MNIST database of handwritten digits.” Accessed Aug. 14, 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[42] TensorFlow, “MNIST.” Accessed Aug. 16, 2020. [Online]. Available: https://www.tensorflow.org/datasets/catalog/mnist

[43] G. Cohen, S. Afshar, J. Tapson, and A. Schaik, “EMNIST: An extension of MNIST to handwritten letters,” 2017. [Online]. Available: arXiv:1702.05373.

[44] TensorFlow, “EMNIST.” Accessed Aug. 16, 2020. [Online]. Available: https://www.tensorflow.org/datasets/catalog/emnist

[45] Z. Liu, P. Luo, X. Wang, and X. Tang, “Deep learning face attributes in the wild.” *2015 IEEE Int. Conf. on Comput. Vision (ICCV),* Santiago, CL, 2015, pp. 3730–3738. doi: 10.1109/ICCV.2015.425

[46] Z. Liu, P. Luo, X. Wang, and X. Tang, “Large scale CelebFaces attributes (CelebA) dataset.” Accessed Aug 17, 2020. [Online]. Available: http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[47] TensorFlow, “Convolutional neural network: TensorFlow core tutorials.” Accessed: Aug. 13, 2020. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn

[48] TensorFlow, “Transfer learning and fine-tuning: TensorFlow core tutorials.” Accessed: Aug. 13, 2020. [Online]. Available: https://www.tensorflow.org/tutorials/images/transfer_learning

[49] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, “MobileNetV2: inverted residuals and linear bottlenecks,” 2018. [Online]. Available: arXiv:1801.04381.

[50] G. Hillar, *Hands-On MQTT Programming with Python.* Birmingham, AL, USA: Packt Publications, 2018.

[51] Texas Instruments, “INA219 zero-drift, bidirectional current/power monitor with I2C interface.” Accessed: Sep. 23, 2020. [Online]. Available: https://www.ti.com/lit/ds/symlink/ina219.pdf?ts=1601005067674&ref_url=https%253A%252F%252Fwww.google.com%252F

[52] K. Karkkainen and J. Joo, “Fair face: attribute dataset for balanced race, gender and age,” 2019. [Online]. Available: arXiv: 1908.04913.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California