POLITECNICO DI TORINO

Master's Degree in ICT FOR SMART SOCIETIES

Master's Degree Thesis

IoT: Wi-Fi Sensing and Diagnostic for Customer Premises Equipment using Transfer Learning and USP

Supervisors

Prof. Albertengo GUIDO

Domenico LOTITO

Candidate

Lin HUANG

July 2024

Summary

This paper presents a lightweight deep transfer learning based human activity detection and diagnostic recognition approach using WiFi sensing. In our method, the amplitude matrix of each WiFi Channel State Information stream is reorganized as an image. Therefore, WiFi based human activity recognition is transformed into an image classification task. Leveraging the high potential of Convolutional Neural Networks in image processing, a CNN-based transfer learning model is employed to reduce the need for extensive network training and to extract features more suited to the Channel State Information matrix. The proposed methods are trained and tested on a public Channel State Information dataset, demonstrating an accuracy of approximately 94% to 99% across six activities. This performance outperforms the state-of-the-art in Human Activity Recognition for Customer Premises Equipment.

We integrate the transfer learning model that demonstrated the best performance into Customer Premises Equipment and deploy it on a Raspberry Pi 4 for local detection applications. The User Services Platform serves as the standard for remote manipulation of connected Customer Premises Equipment. Utilizing the User Services Platform protocol, end-users can independently manage and monitor their Customer Premises Equipment through one or more Controllers.

Table of Contents

List of Tables VI

List of Figures VII

Acronyms X

1 Introduction 1
- 1.1 Overview 1
- 1.2 General Pipeline 2
- 1.3 Thesis Structure 5

2 Related Works 7

3 Implemented Technologies 8
- 3.1 Channel State Information 8
- 3.2 Deep Learning 9
  - 3.2.1 Convolutional Neural Networks 9
  - 3.2.2 Transfer Learning 10
- 3.3 Customer Premises Equipment 10
- 3.4 User Services Platform 11

4 Experiment 14
- 4.1 Human Activity Recognition Datasets 14
- 4.2 Data Pre-process 15
  - 4.2.1 Merge Classes 16
  - 4.2.2 Convert CSI Matrices to Images 16
  - 4.2.3 Image Augmentation 16

5 Proposed Methodologies & Evaluation 18
- 5.1 Proposed Classification Models 18
  - 5.1.1 Model 1: VGG16 based Transfer Learning 18
  - 5.1.2 Model 2 & 3: MobileNetV3-Large based Transfer Learning . 26

5.2 Model Evaluation 31

6 Deployment of Machine Learning Models in Local Environments 34

6.1 Environment Configuration 34

6.1.1 User Services Platform Agent-CPE 34

6.1.2 User Services Platform Controller-ACS 38

6.2 Model Deployment 39

6.2.1 Enable Model Execution in OB-USP-Agent 40

6.2.2 Preprocessing of Input Images 41

6.2.3 Model Benchmark Information Collection 41

6.3 Model Testing 41

7 Conclusion & Future Work 47

Bibliography 49

List of Tables

5.1 Hyperparameters used for the proposed model I. 25
5.2 Hyperparameters used for the proposed model II. 30
5.3 Model Evaluation. Accuracy and Loss are refer to validation accuracy and validation loss. Bach size are the same, result in same amount of data in one epoch. 32

List of Figures

1.1 Model Framework. 4
1.2 CSI images of each class. Classes **a** and **b** are merged to class **Down**, classes **c** and **d** are merged to class **Fall**, classes **e** and **f** are merged to class **Move** 5

3.1 CSI Matrices. Scource: [10] 9
3.2 CPE WAN Management Protocol Architecture. Source: [11] 11
3.3 User Services Platform Architecture. 12
3.4 USP Protocol Stack. 12
3.5 USP Agent and Controller Architecture. Source: [13] 13

4.1 Experimental Environment. Source: [4] 15

5.1 VGG16 Architecture. 19
5.2 VGG16 Architecture (D) vs VGG19 Architecture (E). Source: [15] . 20
5.3 Global Average Pooling (GAP) vs Flattening. 22
5.4 ReLU Activation Function. 23
5.5 Dropout Technology: Left is the network after applying dropout. . . 24
5.6 Softmax Activation Function. 24
5.7 Comparison of the original VGG16 and VGG16-based transfer learning model. 25
5.8 MobileNetV3-Small (left) and MobileNetV3-Large (right) specifications. Source: [16] 26
5.9 MobileNetV2 Block vs MobileNetV3 Block. Source: [16] 27
5.10 Sigmoid and swish nonlinearities and their 'hard' counterparts. Source: [16] 27
5.11 Comparison of original last stage and efficient last stage. Source: [16] 28
5.12 MobileNetV3-Large Architecture. 28
5.13 Comparison of the original MobileNetV3-Large and MobileNetV3-Large based transfer learning model. 29
5.14 Early Stopping Technology. 31

5.15 VGG16 Model Confusion Matrix. 33
5.16 MobileNetV3L Model Confusion Matrix. 33
5.17 MobileNetV3L Model with FT Confusion Matrix. 33

6.1 Raspberry Pi 4 Model B. Source: [17] 35
6.2 User Services Platform Notification Mechanism. Source: [12] 39
6.3 Operate Message Flow for Synchronous Operations. Source: [13] . . 39
6.4 USP over MQTT Framework. 40

Acronyms

IoT

Internet of Things

HAR

Human Activity Recognition

RF

Radio Frequency

RSSI

Received Signal Strength Indication

CSI

Channel State Information

DL

Deep Learning

RNN

Recurrent Neural Network

LSTM

Long Short-Term Memory

CNN

Convolutional Neural Network

CPE

Customer Premises Equipment

USP	User Services Platform
OFDM	Orthogonal frequency-division multiplexing
SVM	Support Vector Machine
GAP	Global Average Pooling
CWMP	Customer Premises Equipment WAN Management Protocol
ACS	Auto-Configuration Server
AP	Access Point
PIL	Python Imaging Library
VGG16	Visual Graphics Group-16
FC	Fully-Connected
ReLU	Rectified Linear Unit
OB-USP-Agent	Open Broadband-User Services Platform-Agent
MTP	Configure Message transport Protocol

STOMP	Streaming Text Oriented Messaging Protocol
MQTT	Message Queuing Telemetry Transport
CoAP	Constrained Application Protocol
TLS	Transport Layer Security
ONNX	Open Neural Network Exchange
FT	Fine-Tuning

Chapter 1

Introduction

1.1 Overview

The Internet of Things (IoT) has become a vibrant research field, due to the recent advancements in communication systems and wireless technology over the last decade. Things or objects are connected to the internet and exchange data or information with each other over the network. As one of the most important IoT applications, smart houses bring a lot benefits to daily life. Allow people to monitor the situation in the house for healthcare of elderly adults, disabled, children or pets. These tasks could be done by using Human Activity Recognition (HAR) techniques, which has emerged as one of the most prominent and influential research topics in several fields, including fall detection, elderly monitoring, gesture recognition, and gender estimation. Among the HAR techniques, WiFi-based methods (WiFi sensing) are becoming most popular in the present internet world, ascribe the ubiquitous WiFi signals that permeate our surroundings, especially indoor environments. Additionally, compared with other methods like camera-based and RF-based, WiFi devices are easier and less expensive to implement, free from light restrictions, and have less invasion of privacy.

The two signals mainly used for WiFi sensing are Received Signal Strength Indication (RSSI) and Channel State Information (CSI). RSSI estimates the power of the received signals, which could be useful for research on WiFi impairments or indoor localization. However, compared to CSI signals, RSSI is less sensitive to capturing small-scale changes in signals produced by activities between WiFi nodes. Therefore, CSI is more used in HAR tasks. CSI mainly contains fine-grained information about how signals propagate from transmitter to receiver and illustrates the effects of power attenuation, scattering, reflection, and refraction. Human's body shapes and activities, as well as the presence of obstacles will impact the propagation of the signal. That means a fall activity and a running activity will have different wireless signal reflections, resulting in different Channel State Information, making it easy to classify human activity. The CSI data can be collected by using some specific extraction tools and platforms. For instance, Nexmon CSI tool [1], Atheros CSI tool [2] or the CSI extraction method based on Intel 5300 NIC released by Halperin [3]. Many researches are done by using these methods.

The retrieved CSI data can be used as inputs of Deep Learning (DL) models for classification. As a time-series data with temporal dependency, Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) are often considered suitable for training based on prior experiences. However, LSTMs process sequential data in a unidirectional manner, considering only past CSI data, which limits their ability to differentiate between activities with similar starting positions but distinct final positions, such as lying down and sitting down. To address this limitation, Convolutional Neural Networks (CNN) are employed, as they automatically extract features from input image pixels and perform final classification, eliminating the need for manual feature extraction. In our study, we convert the CSI data into images by plotting the CSI matrices using a Python library.

In this paper, we focus on the recognition of six gestures: bend, fall, run, walk, lie down, and sit down. We further merge similar activities into one class, the six labels then become three: fall, down, and move. This makes the classification more meaningful, because activities like falling are more crucial for us in healthcare. Due to limited training samples, transfer learning is proposed, which involves reusing previously acquired knowledge from similar tasks. Large-scale and lightweight CNN-based pre-trained models are both included in experiments and compared. We aim at human activities identification using a CNN-based lightweight model that can be deployed in Customer Premises Equipment (CPE).

To deploy the model in CPE, we choose Raspberry Pi as the local device, which is mostly used in IoT. To meet the desire on the most of end-users to have sophisticated control over their CPEs, we use the User Services Platform (USP) protocol released by Broadband Forum. The best lightweight model among experiments will be integrated in the USP Agent on a Raspberry Pi to perform HAR tasks in local environment.

1.2 General Pipeline

Figure 1.1 shows the overall framework of transfer learning model building. The first step is to collect the raw CSI data of different human activities from routers. In this study, a public CSI amplitude dataset [4] is used. To convert CSI data into grayscale images, the amplitude values in CSI matrices are first normalized to a range between 0 and 255 for all activities. These matrices are then transformed into arrays, from which images are generated and subsequently saved. Each matrix element is linearly mapped to a grayscale colormap. Images are also augmented and then reshaped to a certain size before training to meet the input constraints of specific CNN-based transfer learning models. Some of these images for each activity class are depicted in Figure 1.2. Since the images are not noisy, we do not apply pre-processing techniques like denoising filters, which may cause information lost.

Transfer learning is applied to define the classification machine learning model. The layers of the pre-trained CNN model are imported, excluding the fully connected layers of the model. We use bootstrap extractor for feature extraction, which is to write our custom fully connected layers and integrate them with the pre-trained layers. These fully connected layers will be initialized with random weights, which will be updated via back propagation during training. Furthermore, another transfer learning approach, fine-tuning, is applied to the best model among the experiments to make the model better fit our data.

The optimal model is subsequently integrated into the USP Agent on a Raspberry Pi for human activity classification. Real-time CSI values from a WiFi environment are collected and used as input for the model deployed within the USP Agent. The prediction results will be automatically transmitted via User Services Platform Notify messages to one or more connected USP Controllers.

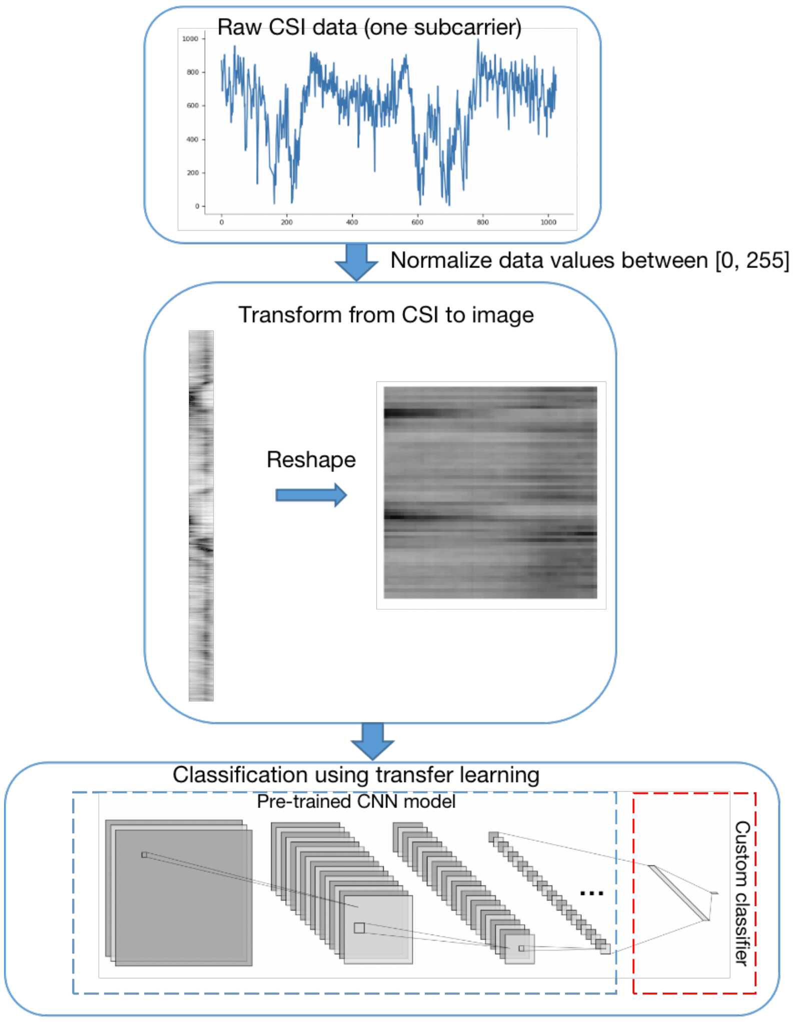

Raw CSI data (one subcarrier)
1000
800
600
400
200
0
0
200
400
600
800
1000
Normalize data values between [0, 255]
Transform from CSI to image
Reshape
Classification using transfer learning
Pre-trained CNN model
...
Custom classifier

Figure 1.1: Model Framework.

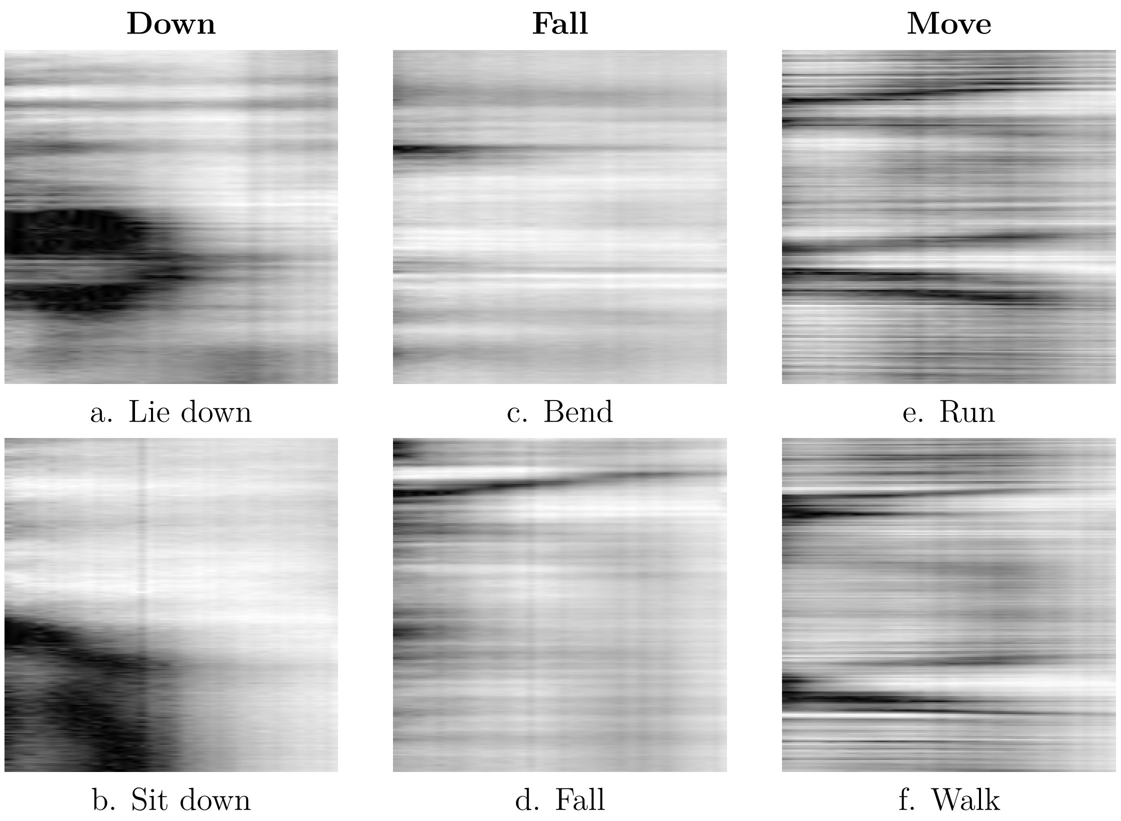


Figure 1.2: CSI images of each class. Classes **a** and **b** are merged to class **Down**, classes **c** and **d** are merged to class **Fall**, classes **e** and **f** are merged to class **Move**

1.3 Thesis Structure

The rest of this paper is organized as follows:

- Chapter 2: Integrate state-of-art pre-trained CNN models with transfer learning for wifi sensing use cases.
- Chapter 3: Introduction to the implemented technologies in the thesis work, including dataset used, deep learning algorithms, Customer Premises Equipment, and User Services Platform protocol.
- Chapter 4: Description of Human Activity Recognition dateset selection and data pre-processing procedures.
- Chapter 5: Introduction to transfer learning models proposed in the work. Including large-scale models based on VGG, as well as smaller-scale models based on MobileNet. And evaluation of the proposed models across various metrics.

- Chapter 6: Description of developed User Services Platform Agent and User Services Platform Controller, followed by the deployment and testing of a predictive algorithm based on the best-performing model from experiments, integrated into Customer Premises Equipment.
- Chapter 7: Summary of the conducted work and analysis of potential future developments.

Chapter 2

Related Works

Since Halperin [3] published a measuring method of CSI based on Intel 5300 NIC, and the release of Nexmon CSI Tool [1], researchers started to study human activity recognition based on commercial WiFi devices, using CSI signals.

[5] listed various existing deep learning approaches for WiFi sensing. It released a comprehensive benchmark with an open-source library for deep-learning-based WiFi human sensing with comparison among different famous public CSI dataset as well as different deep learning architectures. Article [4] indicated that, in CSI-based HAR tasks, 2D-CNN model has best performance among most of the other deep learning methods such as LSTM and 1D-CNN. [6] used VGG16 and VGG19 for feature extraction, support vector machines(SVM) for classification on WiFi based gesture recognition. Additionally, fine-tuning technique was used and obtained better experimental results on the dataset compare to not use. The deep and complex architecture of VGG required extraordinarily long execution time. Therefore, [7] improved performance by downsized the VGG architecture. Moreover, P. Sruthi and Siba K. Udgata [8] proposed a two-phase deep learning model for WiFi sensing based person identification and activity recognition. They focused on both recognizing a participant and classifying the activity performed by the participants. Additionally, they tried to reduce the false negatives of a key person and associated critical activity by using a multistage model. An approach presented by [9] used MobileNetV3 transfer learning on device-based real-time tree species identification shows a big advantage of MobileNetV3 model in handling tasks on local devices.

Inspired by these paper and the development of CNN in vision field, we propose to use CNN to achieve human activity recognition. In order to let CNN achieve good result on limited dataset, we transfer the pre-trained CNN of vision areas to the WiFi based activity recognition filed.

Chapter 3

Implemented Technologies

3.1 Channel State Information

Channel State Information (CSI) can be utilized to characterize the signal transmission process from the transmitter to the receiver, where the signal undergoes deflection, reflection, and scattering upon encountering obstacles or objects. Multiple sub-carriers may be present within the physical link between each pair of transmitter and receiver antennas. Since each sub-carrier can support multiple data streams, the CSI associated with each sub-carrier will be distinct. A CSI data point can be represented by a complex-valued channel matrix of dimensions $m{\times}t{\times}r$, where m denotes the number of Orthogonal Frequency-Division Multiplexing (OFDM) sub-carriers and t, r represent the number of transmitter and receiver antennas. Each CSI complex value consists of amplitude and phase. By stacking N consecutive CSI data points along the temporal dimension, we can form a 4D tensor with dimensions $N{\times}m{\times}t{\times}r$, like Figure 3.1. This data structure facilitates the retrieval of CSI at specific time instances and allows for the analysis of CSI evolution for individual antennas.

The number of available sub-carriers varies based on the hardware configuration and channel bandwidth utilized. The dataset we used in the study used a Raspberry Pi 4 and a TP-Link Archer C20 operating over a 5 GHz frequency band with a 20 MHz bandwidth, which allows access to 52 data sub-carriers.

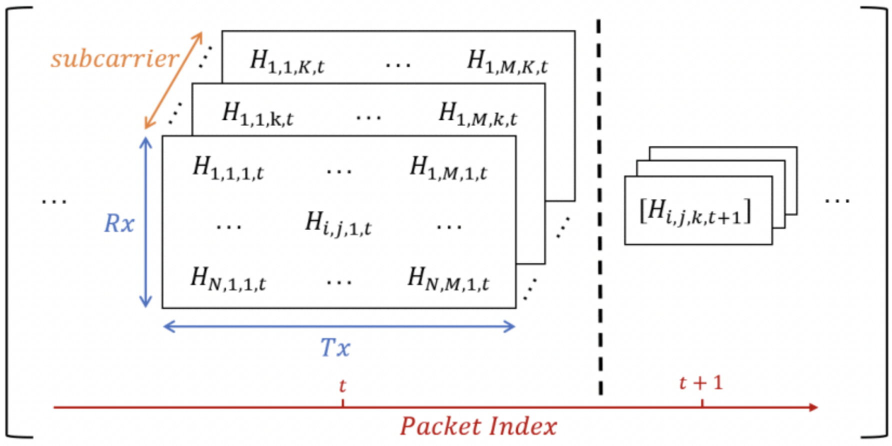

subcarrier
$H_{1,1,K,t}$ … $H_{1,M,K,t}$
$H_{1,1,k,t}$ … $H_{1,M,k,t}$
$H_{1,1,1,t}$ … $H_{1,M,1,t}$
… $H_{i,j,1,t}$ …
$H_{N,1,1,t}$ … $H_{N,M,1,t}$
Rx
Tx
$[H_{i,j,k,t+1}]$
…
t
$t+1$
Packet Index

Figure 3.1: CSI Matrices. Scource: [10]

3.2 Deep Learning

In recent years, deep learning algorithms have been extensively employed for classification tasks. As a specialized branch of machine learning, deep learning leverages multi-layered neural networks, referred to as deep neural networks. Enhancing the number of hidden layers and nodes within these networks can lead to improved accuracy. However, this increase in layers and nodes also necessitates a greater number of parameters and computing resources. Compared to conventional neural networks, deep learning enables computers to tackle more intricate problems autonomously.

3.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are feed-forward neural networks proficient in extracting features from data via convolutional operations, making them exceptionally suitable for image classification tasks. CNNs comprise several layers, including Convolution, Pooling, Flatten, Global Average Pooling (GAP), and Dense layers. The convolutional layers are responsible for extracting image features, while the pooling layers reduce the dimensionality of these features. With multiple convolutional layers in the architecture, CNNs can capture features ranging from low-level to high-level, thus enhancing recognition accuracy. The fully connected (dense) layers then integrate the extracted features into a comprehensive feature vector. In this study, CNNs are employed to learn the characteristics from the effects of activities on Channel State Information data.

A significant advantage of CNNs is their reduced need for extensive data preprocessing compared to other classification techniques. Furthermore, CNNs can automatically learn features from the data without user intervention, thereby Significantly simplifies the feature extraction process.

3.2.2 Transfer Learning

Transfer learning, as a technique of deep learning, is wildly used in machine learning studies. When data used to train deep learning networks is not enough, transfer learning is applied to acquire the knowledge learned in previous settings. By integrating pre-trained layers excluding the final classification layer, two primary transfer learning approaches can be employed: feature extraction and fine-tuning. Feature extraction itself can be categorized into two types: stand-alone extractor and bootstrap extractor.

1. **Feature Extraction Approaches**:
   - **Stand-Alone Extractor**: In this scenario, pre-trained layers are utilized to extract image features a single time. These extracted features subsequently form a new dataset, which is then employed for further task purposes.
   - **Bootstrap Extractor**: Develop custom fully connected layers and integrate them with the pre-trained layers. Initialize these fully connected layers with random weights, which will update through backpropagation during the training process.
2. **Fine-Tuning Approach**: The initial pre-trained layers of the model are fixed, while a subset of the layers, typically the last few, are set to be trainable. The trainable layers will learn the characteristics of the new dataset.

This way, the deep learning model is trained for custom tasks, with the weights being updated according to the new dataset. Transfer learning speeds up the training process compared to building a new network from scratch, as it does not require estimating all parameters anew. Studies on transfer learning for Human Activity Recognition have demonstrated that CNN-based transfer learning models achieve superior performance.

3.3 Customer Premises Equipment

Customer Premises Equipment (CPE) refers to devices and equipment located at the end user's premises, which are used to access telecommunications services.

These devices include routers, modems, gateways, set-top boxes, and other networking hardware that connect to service providers' networks. CPE serves as the critical interface between the user's local network and the broader internet or telecommunications infrastructure.

The Customer Premises Equipment WAN Management Protocol (CWMP), standardized as TR-069 by the Broadband Forum, is a communication protocol designed to manage and control network devices such as modems, routers, gateways, and other CPEs in a broadband network. CWMP facilitates remote management by allowing service providers to configure, monitor, and update CPEs without the need for user intervention. The protocol ensures secure and reliable communication between the CPE and the Auto-Configuration Server (ACS), Figure 3.2 shows the CWMP architecture. This paper explores the deployment of machine learning algorithms within CPE to enable local, real-time data processing, thereby extending the functionality and intelligence of these devices.

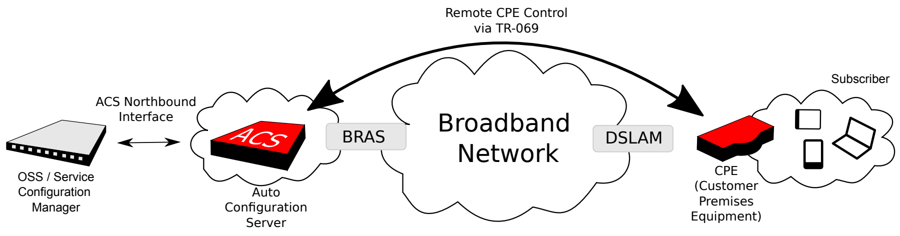


Figure 3.2: CPE WAN Management Protocol Architecture. Source: [11]

3.4 User Services Platform

The User Services Platform protocol, standardized by the Broadband Forum [12], is an advanced protocol designed to manage, monitor, update, and control connected devices within a network of Controllers and Agents. An Agent is an endpoint that exposes service elements to one or more Controllers. A Controller is an endpoint that allows end-users to manipulates Agents' functions. User Services Platform, also known as TR-369, extends the capabilities of the earlier TR-069 protocol, providing enhanced performance, security, and scalability. A major improvement made by User Services Platform is it allows secure communication between various USP Controllers and a connected USP Agent embedded in network devices, such as Customer Premises Equipment, as depicted in Figure 3.3. Moreover, User Services Platform supports different message-oriented approach and modern communication mechanisms like Streaming Text Oriented Messaging Protocol, Message Queuing

Telemetry Transport, WebSockets and Constrained Application Protocol for User Services Platform messages exchange through networks, see Figure 3.4.

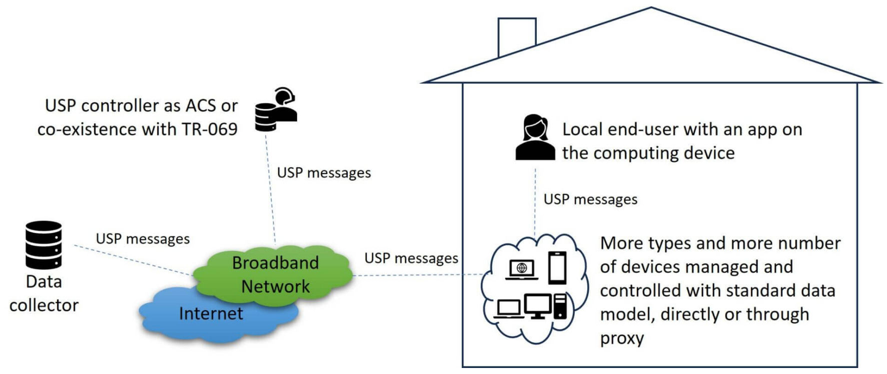

USP controller as ACS or co-existence with TR-069
USP messages
USP messages
Data collector
Broadband Network
Internet
USP messages
Local end-user with an app on the computing device
USP messages
More types and more number of devices managed and controlled with standard data model, directly or through proxy

Figure 3.3: User Services Platform Architecture.

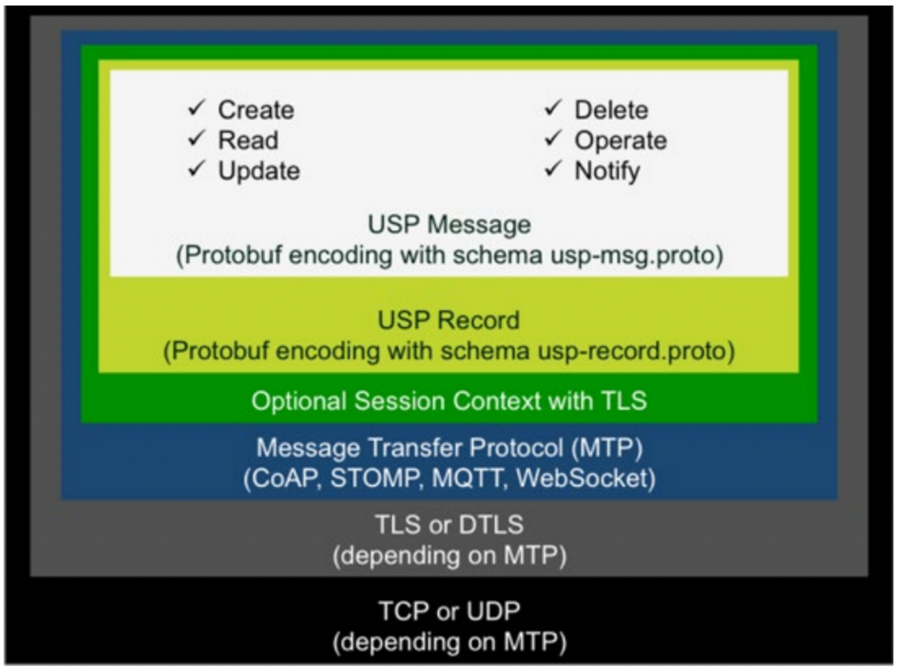

✓ Create
✓ Read
✓ Update
✓ Delete
✓ Operate
✓ Notify
USP Message
(Protobuf encoding with schema usp-msg.proto)
USP Record
(Protobuf encoding with schema usp-record.proto)
Optional Session Context with TLS
Message Transfer Protocol (MTP)
(CoAP, STOMP, MQTT, WebSocket)
TLS or DTLS
(depending on MTP)
TCP or UDP
(depending on MTP)

Figure 3.4: USP Protocol Stack.

Figure 3.5 shows a more detailed Agent and Controller architecture for User Services Platform.

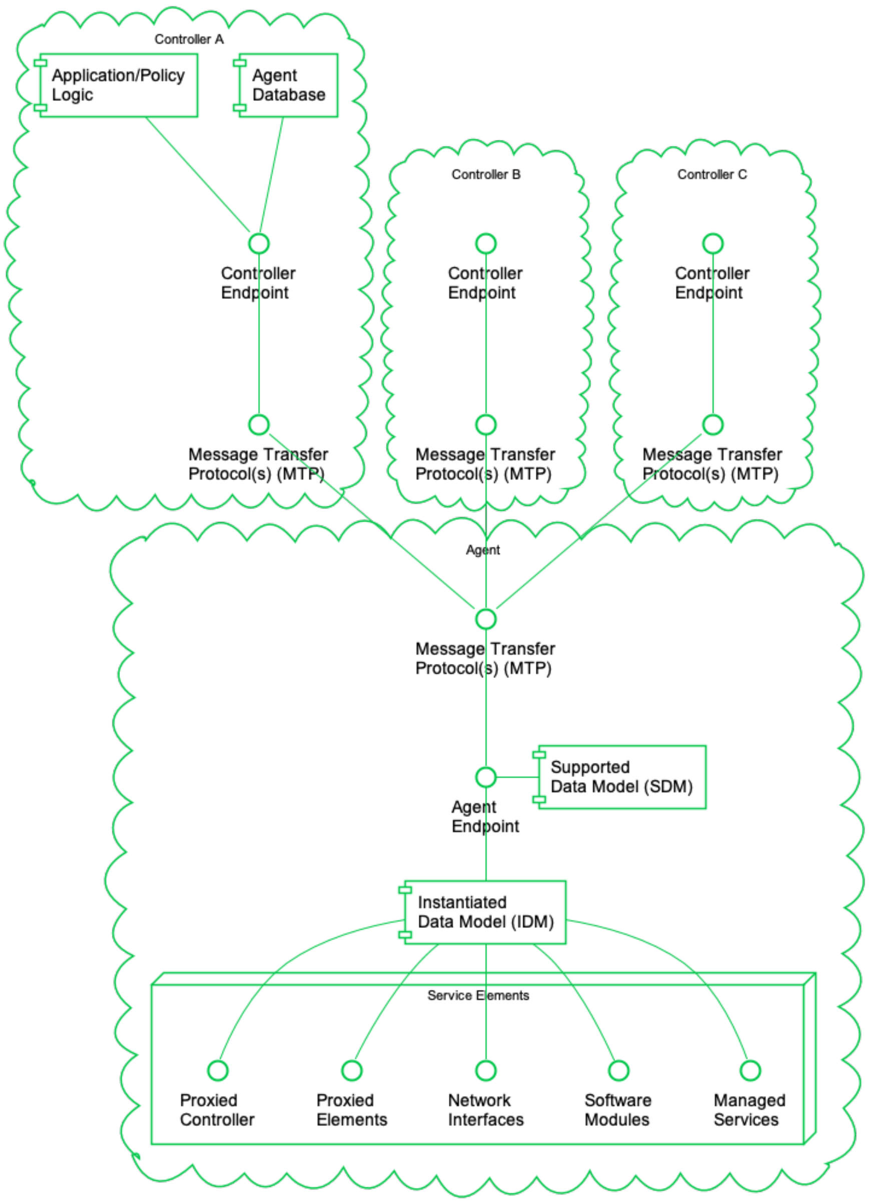


Figure 3.5: USP Agent and Controller Architecture. Source: [13]

Chapter 4

Experiment

4.1 Human Activity Recognition Datasets

The quantity of data required to train a neural network for HAR tasks is variable, depending on the task complexity and the selected algorithm. A relevant study [4] in WiFi-based human activity recognition employs a 2D-CNN model similar to our approach. Therefore, we utilize the public CSI dataset provided by this study for our research. CSI is a complex value consisting of amplitude and phase. However, CSI amplitude data tends to be more stable and less affected by noise or hardware imperfections. Also, many human activities produce distinct patterns in the amplitude data that are sufficiently informative for HAR tasks. Therefore, in this work, we focus on the CSI amplitude.

The authors employ Nexmon CSI Tool [1] and collect CSI data for seven daily human activities, including walk, run, fall, lie down, sit down, stand up, and bend. 3 volunteers of different ages perform 7 different activities 20 times, resulting in 420 samples in total. They use Raspberry Pi 4 and a Tp-link archer c20 as an Access Point (AP) in 20 MHz bandwidth on channel 36 in IEEE 802.11ac standard. The transmitter and receiver are at a distance of three meters and are both one meter above the ground to ensure an unobstructed signal path. The experimental environment is depicted in Figure 4.1. The packets are collected for around 20 s, giving 4000 packets of data. The period of time taken for the activities varies slightly, around 3 to 6 s, which are around 600 to 1100 rows of 4000 total rows in the matrix. CSI complex numbers are extracted, and after removing null and pilot sub-carriers, they export activity rows according to the video of each activity and stopwatch.

Due to reflections induced by human activity, each sub-carrier for any given link experiences a variation. Each sub-carrier includes critical information that will increase recognition accuracy. A higher proportion of sub-carriers boosts precise feature detection since it provides additional information and boosts identification of challenging features to analyze a subset of sub-carriers.

The CSI amplitude matrices of the benchmark dataset have the form *No. of transmitter antennas* × *No. of receiver antennas* × *No. of sub-carriers* × *No. of frames*. In this experiment, a single transmit/receive antenna pair is used, with 52 available data sub-carriers, and 600 to 1100 frames, varying according to the duration of each activity. Consequently, the resulting CSI amplitude matrix is structured into *52 columns* × *600 to 1100 rows*. The dataset is available on GitHub `https://github.com/parisafm/CSI-HAR-Dataset`.

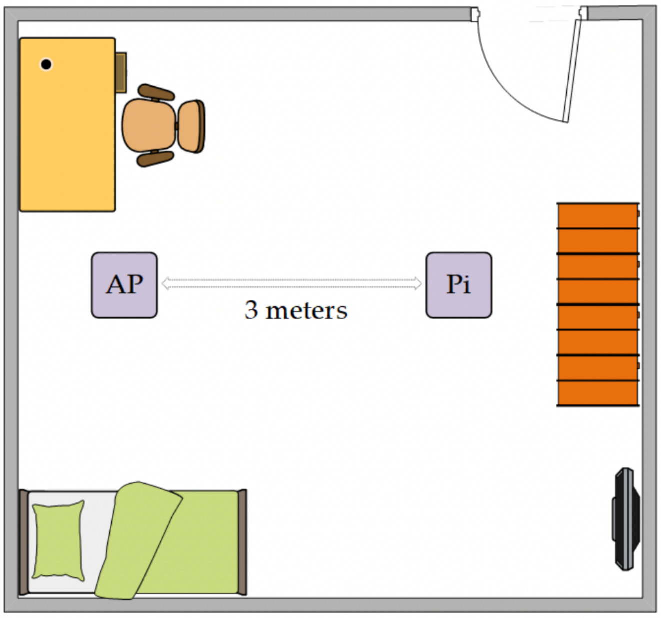


Figure 4.1: Experimental Environment. Source: [4]

4.2 Data Pre-process

Data preprocessing mainly has 3 procedures: Merge classes, convert CSI matrices to images, and image augmentation. We aim to have as less data preprocessing before putting data into models as possible, and include some data preprocessing method inside the models. In order to make models more portable and less error prone. This also makes models more applicable to universal data.

4.2.1 Merge Classes

To enhance the relevance of the data to our target application, we consolidate the classes as follows: bend and fall data are merged into a single class, fall; sit down and lie down data are combined into the class down; and run and walk data are merged into the class move. The stand up data is excluded. Classes are merged together based on their similarity. As illustrated in Figure 1.2, the CSI images for activities within the same column exhibit significant similarity. This reclassification is particularly meaningful for healthcare, as activities resembling falls are critical to monitor. Additionally, consolidating data into fewer categories improves the model's ability to learn the characteristics of each category during training. Consequently, the dataset now comprises three classes: fall, down, and move. And the dataset is balanced in the terms of total images in each class.

4.2.2 Convert CSI Matrices to Images

Given the high potential of Convolutional Neural Networks in image processing, we convert the CSI amplitude matrices into images and develop models based on CNN architecture. Images can be either color or grayscale. However, since CSI data are single-channel and RGB images require three channels, creating RGB images would necessitate generating a pseudocolor plot from the matrices. This addition of artificial color information could adversely affect classification accuracy. Meanwhile, the additional information for RGB images increases complexity, resulting in longer model execution times, which is not we expect. Therefore, we choose to convert the CSI amplitude matrices into grayscale images, which only require a single channel.

Initially, the values in the CSI amplitude matrices are normalized to a range of 0 to 255 across all activities. We then generate images from these matrices using the Image module from the Python Imaging Library (PIL) and save them. Each matrix element is linearly mapped to the grayscale colormap. Figure 1.2 illustrates some of these CSI images, showing distinct texture and structural differences between activity classes. This visual differentiation supports the transformation of WiFi sensing-based human activity recognition into an image classification problem, allowing us to use image-based techniques for feature extraction and classification.

The converted imaged are split into training and test sets with proportions of 75% and 25%, respectively, and stored in corresponding directories.

4.2.3 Image Augmentation

Image augmentation involves applying various transformations to original images, producing multiple altered versions of the same image. This technique not only increases the dataset size but also introduces variability, enhancing the model's ability to generalize to unseen data and reducing the risk of overfitting. Additionally, training on these modified images increases the model's robustness to new, slightly different inputs.

The Keras *ImageDataGenerator* class provides a convenient and efficient method for image augmentation, encompassing techniques such as standardization, rotation, shifts, flips, brightness adjustment, rescaling, and more. Its primary advantage is its capability for real-time data augmentation, generating altered images on the fly during the training process. This ensures the model receives new image variations at each epoch during the model training process. The *ImageDataGenerator* class only returns transformed images without adding them to the original dataset. Unlike loading all images simultaneously, using *ImageDataGenerator* allows images to be loaded in batches, significantly conserving memory.

We employ several augmentation techniques using the Keras *ImageDataGenerator* class, specifically brightness adjustment, width and height shift, rescale, input preprocessing. Given the nature of time series images, augmentation methods like rotation and zoom are not suitable for our dataset. Additionally, each Keras model application expects a specific kind of input preprocessing. However, input preprocessing function and rescaling are not applied to the input data before training the MobileNetV3-Large based transfer learning model, as MobileNetV3-Large already includes an integrated Rescaling layer to transfer inputs to be float tensors of pixels with values in the [-1, 1] range.

Images are augmented and reshaped to a desired size (224, 224) before training to meet the input constraints of specific CNN models (all propsed CNN-based transfer learning models in this paper require the input image shape to be (224, 224, 3)). Since the images are not noisy, we do not apply pre-processing techniques like denoising filters, which may cause information lost.

Chapter 5

Proposed Methodologies & Evaluation

5.1 Proposed Classification Models

The primary objective of the paper is to identify human activities utilizing a CNN-based transfer learning approach. To achieve this, we have developed three distinct models, leveraging different pre-trained CNNs as their foundations. There are various type of CNN models, including DenseNet, ResNet, VGG, AlexNet, MobileNet and so on. This paper focuses on transfer learning with large-scale CNN architecture, VGG16, and lightweight architecture, MobileNetV3-Large with and without fine-tuning. By forming performance metrics comparison among proposed models of varying scales, we aim to elucidate the factors influencing model size, assess the impact of different architectures on overall model performance, and determine the suitability of each model for deployment on local IoT devices.

The models are trained and tested on Google Colab, using the GPU resources available to Colab Pro subscribers. The implementation is carried out in Python using the Keras API from TensorFlow.

5.1.1 Model 1: VGG16 based Transfer Learning

Visual Graphics Group-16 (VGG16)

VGG is one of the most common CNN models pre-trained on ImageNet dataset [14], specifically the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 dataset. This extensive dataset includes images from 1,000 classes, with 1.3 million images for training, 50,000 images for validation, and 100,000 images for testing.

As introduced in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" [15]. The VGG16 architecture achieves 92.7% top-5 test accuracy in ILSVRC classification. It consists of 19 weight layers, of which 3 are fully connected layers, 13 are convolution layers. These convolution layers are organized into 5 convolution blocks. Figure 5.1 depicts the VGG16 architecture. Additionally, another VGG architecture VGG19 is slightly deeper than VGG16 and has 16 convolution layers and 3 fully connected layers. More detailed layer information and architecture comparison between VGG16 and VGG19 are presented in Figure 5.2 . After experiments, we found that transfer learning models with VGG16 and VGG19 have similar performance, but VGG16 has fewer layers and parameters result in a smaller model size. With the aim of deploying lightweight machine learning model in local environment, we will focus on VGG16 in this study.

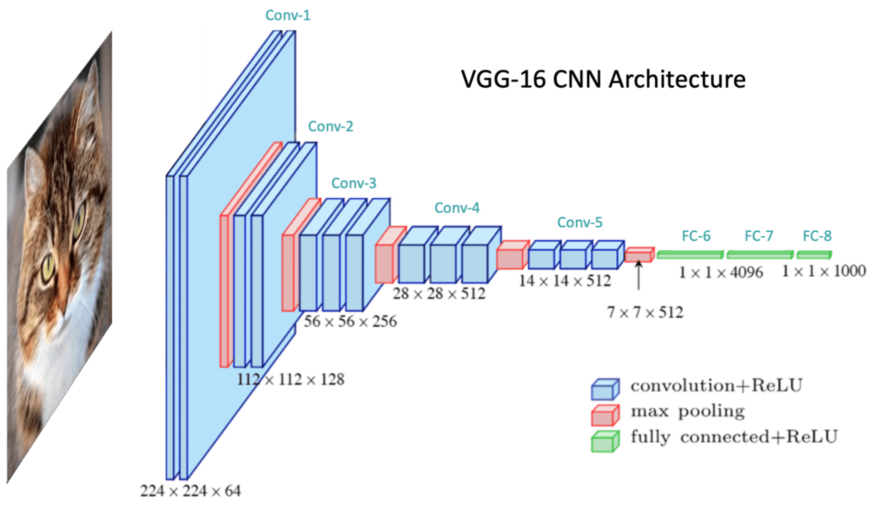
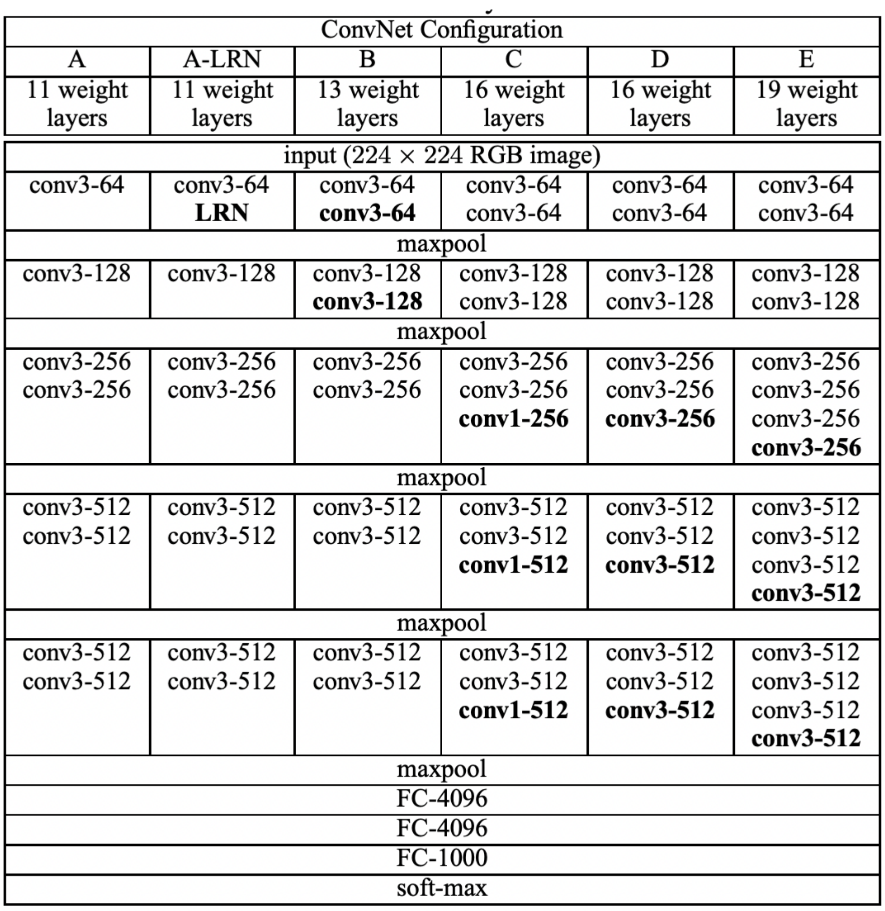


Figure 5.1: VGG16 Architecture.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 5.2: VGG16 Architecture (D) vs VGG19 Architecture (E). Source: [15]

Feature Extraction

After choosing the model for transfer learning, the primary thing is to pick which layer of VGG16 to use for feature extraction. The very last classification layers (also known as fully connected layers) are considered not very useful. The classification layers are mainly built based on the using dataset, one example is the number of artificial neurons of last fully connected layer is always equal to the number of target classes in classification tasks. However, our dataset and target classes are not the same as ImageNet. So the classification layers are not suitable for feature extraction in custom tasks due their lack of generality.

On the other hand, since VGG16 has 5 convolution blocks, we let the feature extraction layer be the MaxPooling layer after the first convolution block to the one after the fifth convolution block respectively, and compare their performance.

The reason for choosing the MaxPooling layer as output layer is, it extracts important features from the last convolution layer while simultaneously reducing the dimensions of the data. Furthermore, it helps reduce the image size, making it more manageable for the convolutional neural network, which speeds up training and requires less memory. So it is a better choice to use the MaxPooling layer as the output layer instead of the convolution layer.

We choose to depend on the MaxPooling layer positioned after the fifth convolution block and before the classifier. This layer, also referred to as the *bottleneck layer*, is frequently selected for feature extraction in transfer learning by numerous studies follow the common practice.

To build feature extractor, we firstly instantiate a VGG16 model pre-loaded with weights trained on ImageNet. By setting the *include_top* argument to *False*, the model is loaded without the classification layers at the top, which is what we expected for feature extraction:

```
base_vgg16 = tf.keras.applications.vgg16(include_top=False,
                                         weights='imagenet',
                                         input_shape=(224, 224, 3))
```

Another important step in feature extraction is to freeze the created convolutional base, to use it as a feature extractor. Freezing prevents the pre-trained weights in a given layer from being updated during training. To do that, we set the trainable flag of base model to *False*:

```
base_vgg16.trainable = False.
```

Later, we will add custom classification layers on top of it and only train the top-level classification layers.

Custom Fully Connected Layers

Following feature extraction, we define the custom fully connected layers. The proposed classifier comprises the following layers:

- **Global average pooling layer**: This layer is used to prepare data for dense layers. To mitigate the risk of overfitting and reduce the computational load associated with a large number of parameters, a Global Average Pooling (GAP) layer will be used instead of a Flatten layer. Both GAP and Flatten layers serve to collapse the spatial dimensions of the input features into the channel dimension. However, in Keras, the Flatten layer reshapes the tensor into a 1D vector with a length equal to the total number of elements in the tensor, preserving all nodes from each feature map. In contrast, the GAP layer outputs the mean value of each feature map by doing average pooling operation, resulting in an 1D output vector whose length corresponds to the number of feature maps. This output will subsequently serve as the input to the dense layers. Figure 5.3 clearly demonstrates the difference between GAP layer and Flatten layer. The GAP layer reduces the output shape and parameter count by nearly 50-fold. This reduction significantly accelerates the running time while maintaining high accuracy in model performance.

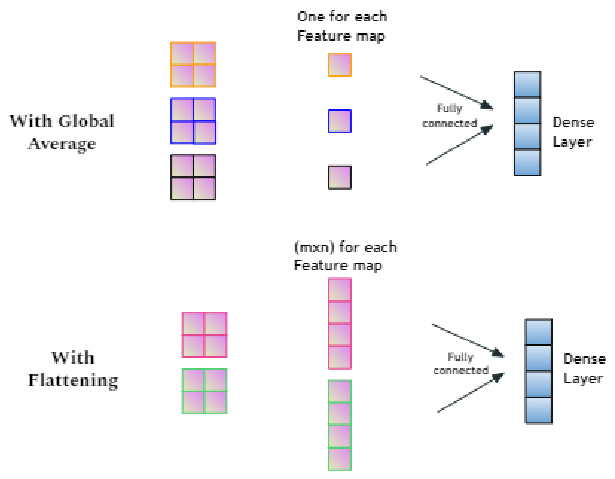


Figure 5.3: Global Average Pooling (GAP) vs Flattening.

- **Dense layer**: Dense layers are often used at the end of the network to make predictions based on the features learned during the feature extraction stage. Unlike convolutional layers, dense layers are fully connected, so they are also known as Fully-Connected (FC) layers. The dimensionality reduction achieved by the global average pooling layer eliminates the need for multiple FC layers at the top of the CNN, significantly decreasing the number of parameters and limiting the risk of overfitting. Although additional FC layers can enhance network robustness, they also substantially increase the number of parameters at the same time. Therefore, our model utilizes two dense layers, as opposed to the three in the standard VGG16 architecture. The first dense layer contains 128 units and employs the non-linear Rectified Linear Unit (ReLU) activation function, Figure 5.4 shows the plot of ReLU. Unlike traditional activation functions such as sigmoid or tanh, ReLU mitigates the vanishing gradient problem by maintaining stable gradients during backpropagation, promoting efficient learning and convergence. Additionally, ReLU induces sparse activation, where only a subset of neurons are active at any time, enhancing network efficiency and reducing overfitting. The other dense layer is the softmax layer, which will be discussed later.

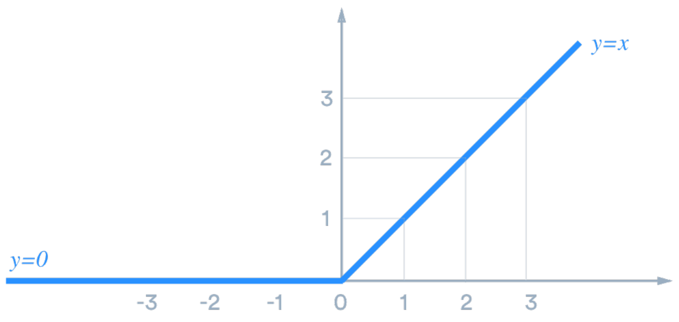


Figure 5.4: ReLU Activation Function.

- **Dropout layer**: The above dense layer will be followed with a dropout layer. A dropout layer randomly deactivates a proportion of neurons in the network, the proportion known as the dropout rate. When neurons are deactivated, their incoming and outgoing connections are also switched off, as illustrated in Figure 5.5. This technique reduces architectural complexity, decreases overfitting, while improve the generalization capability of network. In this study, the dropout rate is set to 0.25, signifies that during training, 25% of the neurons in this dense layer are randomly deactivated each iteration. Higher dropout rates like 0.5 or higher might excessively reduce the network's capacity to learn useful features, while lower rates that lower than 0.1 might not provide sufficient regularization. For moderately complex CNN architectures and standard-sized or small-sized datasets similar to our task, a dropout rate around 0.25 often strikes a good balance.

- **Softmax layer**: The final layer is the softmax layer, where we get our predicted results. It is the second dense layer with 3 channels, one for each target class. For muti-classification taks the activation function is often set to softmax, Figure 5.6. It is able to transform raw output scores into a probability distribution across all target classes. This transformation is crucial as it converts the model's outputs into probabilities that sum to one, providing a clear and interpretable prediction for each class.

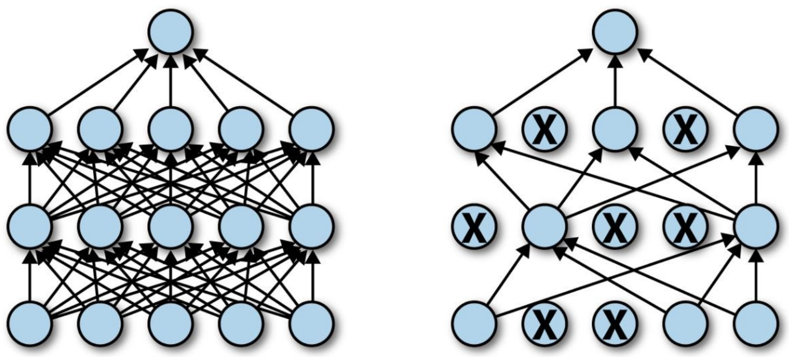


Figure 5.5: Dropout Technology: Left is the network after applying dropout.

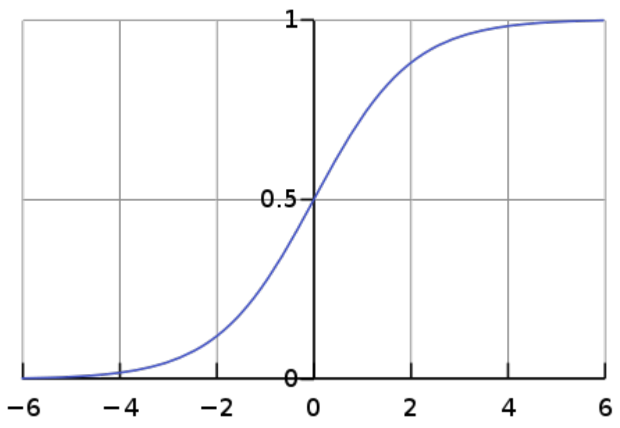


Figure 5.6: Softmax Activation Function.

The configuration of the fully connected layers is consistent across all three transfer learning model proposals. So the detailed custom classifier definition will not be discussed further in the subsequent sections on the other two transfer learning models.

Model Structure

The block diagram of proposed model structure is depicted in Figure 5.7. It presents a comparative overview of the flattened architecture of VGG16 alongside our transfer learning model based on VGG16. For brevity, MaxPooling layers following each convolutional block are omitted. The section highlighted in the red box represents our custom classifier.

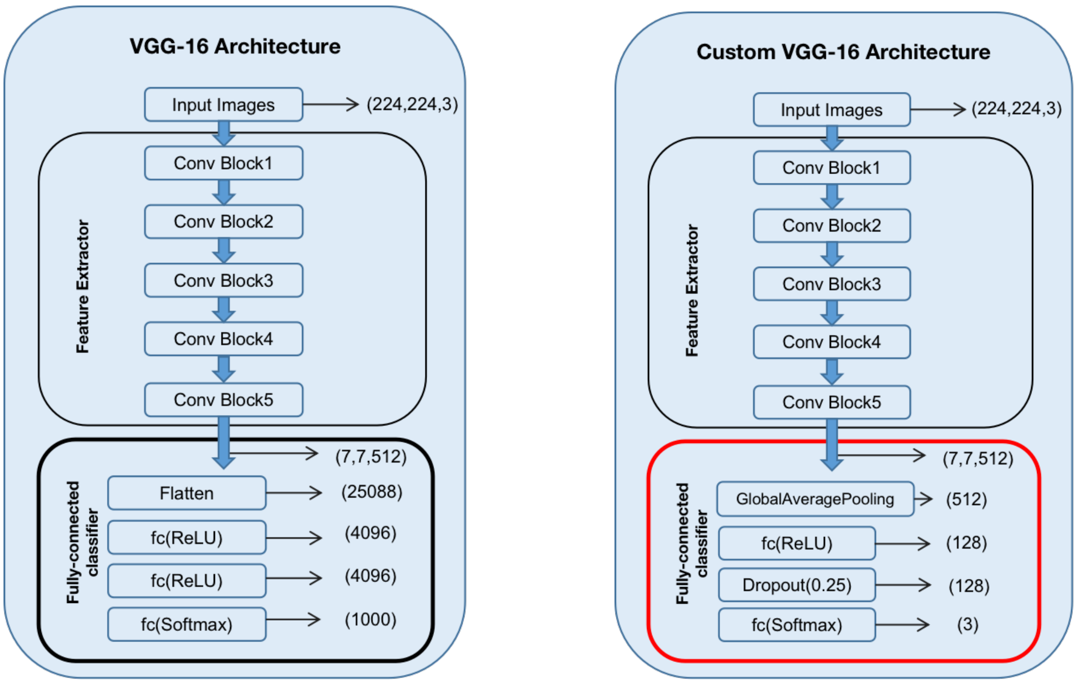


Figure 5.7: Comparison of the original VGG16 and VGG16-based transfer learning model.

Hyperparameters

Some important hyperparameters are tabulated in Table 5.1

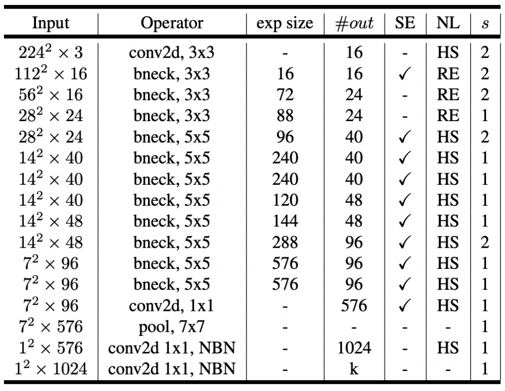
Table 5.1: Hyperparameters used for the proposed model I.

Hyperparameters	**Value**
Bach size	32
Optimizer	Adam, learning rate=0.001
Loss function	Cross Entropy
Epochs	50

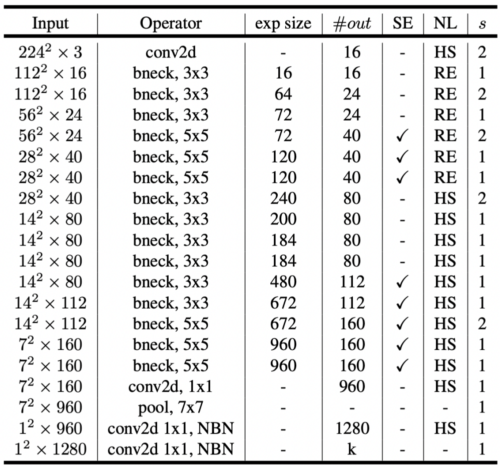
5.1.2 Model 2&3: MobileNetV3-Large based Transfer Learning

MobileNetV3-Large

The second proposed transfer learning model is based on another existing CNN model MobileNetV3-Large, which belongs to the MobileNet family. MobileNet is a CNN model developed by Google and also pre-trained on ImageNet. Unlike VGG or ResNet series, it offers lightweight architecture, reduced memory usage, and improved computation speed, making it suitable for real-time detection and monitor in applications on embedded devices. MobileNet has many versions and releases. The two releases of MobileNetV3: MobileNetV3-Small and MobileNetV3-Large are targeted for high and low resource use cases. Comparing to the older version MobileNetV2, MobileNetV3-Large is 3.2% more accurate on ImageNet classification while reducing latency by 20% compared to MobileNetV2. MobileNetV3-Small is 6.6% more accurate compared to a MobileNetV2 model with comparable latency [16]. Figure 5.8 indicates the specification for MobileNetV3-Large and MobileNetV3-Small.



Input	Operator	exp size	*#out*	SE	NL	*s*
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1024	-	HS	1
$1^2 \times 1024$	conv2d 1x1, NBN	-	k	-	-	1



Input	Operator	exp size	*#out*	SE	NL	*s*
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Figure 5.8: MobileNetV3-Small (left) and MobileNetV3-Large (right) specifications. Source: [16]

The efficiency, lightweight architecture, and high accuracy of MobileNetV3 are attributed to several key innovations: the incorporation of Squeeze-and-Excite modules within the residual layers (Figure 5.9), the introduction of the h-swish nonlinearity (Figure 5.10), and a more efficient final stage (Figure 5.11). After conducting experiments, MobileNetV3-Large was selected for its superior performance on our dataset. Figure 5.12 depicts the MobileNetV3-Large architecture.

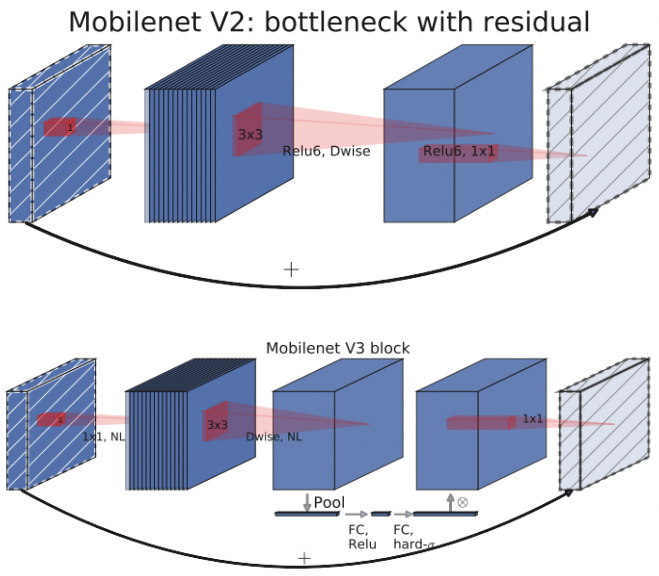


Figure 5.9: MobileNetV2 Block vs MobileNetV3 Block. Source: [16]

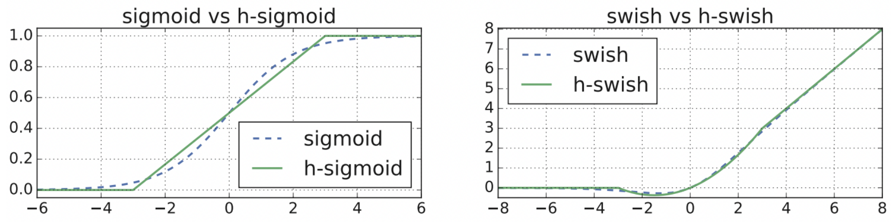


Figure 5.10: Sigmoid and swish nonlinearities and their 'hard' counterparts. Source: [16]

Feature Extraction and Custom Fully Connected Layers

Based on the discussion in the previous Section 5.1.1. The feature extraction layer selected is the *bottleneck layer* preceding the Average Pooling layer. To construct the feature extractor, we begin by instantiating a MobileNetV3-Large model pre-trained weights. Setting the *include_top* argument to *False* ensures that the model

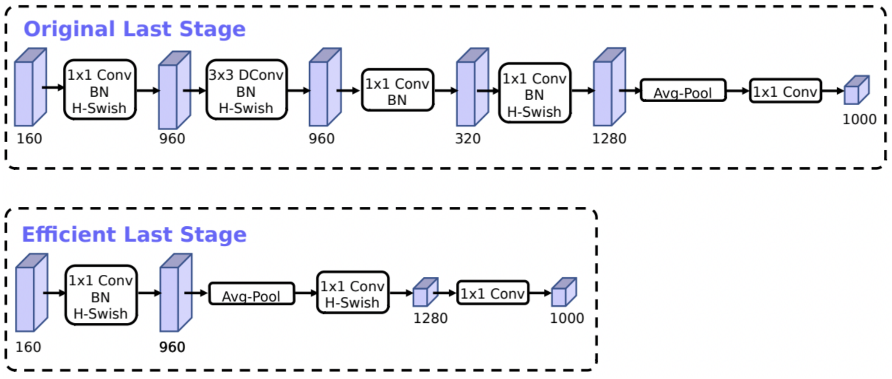


Figure 5.11: Comparison of original last stage and efficient last stage. Source: [16]

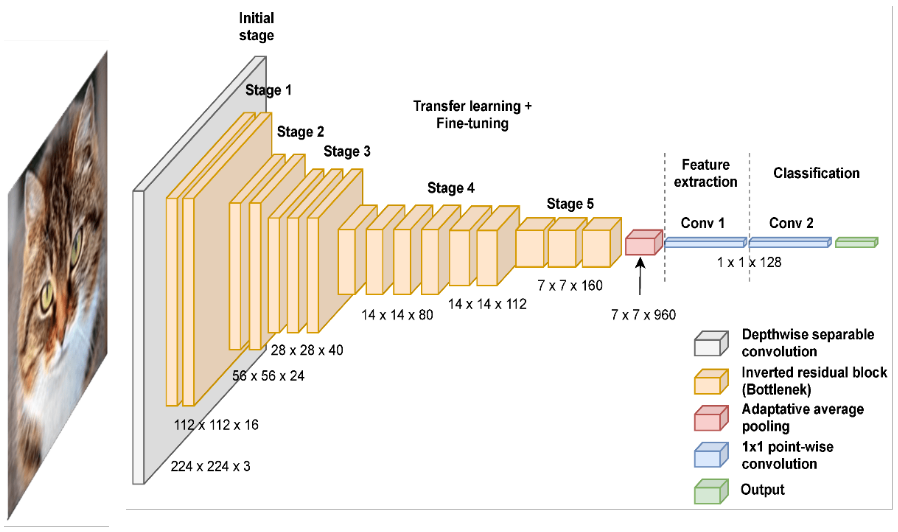


Figure 5.12: MobileNetV3-Large Architecture.

is loaded without its classification layers, aligning with our intention for feature extraction:

```
base_MobileNetV3L = tf.keras.applications.MobileNetV3Large(
                                        include_top=False,
                                        weights='imagenet',
                                        input_shape=(224, 224, 3))
```

To prevents the pre-trained weights in a given layer from being updated during training, we set the trainable flag of base model to *False*:

```
base_MobileNetV3L.trainable = False.
```

A custom classifier is introduced on top of the feature extractor. During model training, only the weights of the custom classifier are updated. The custom classifier comprises the following components: a GlobalAveragePooling layer, a fully connected layer with 128 nodes utilizing the ReLU activation function, a dropout layer with a dropout rate of 0.25, and a final fully connected layer with 3 nodes employing the Softmax activation function. Further details are provided in Section 5.1.1 earlier.

Model Structure

Figure 5.13 depicts the proposed model structure. Whereas The section highlighted in the red box is the custom classifier.

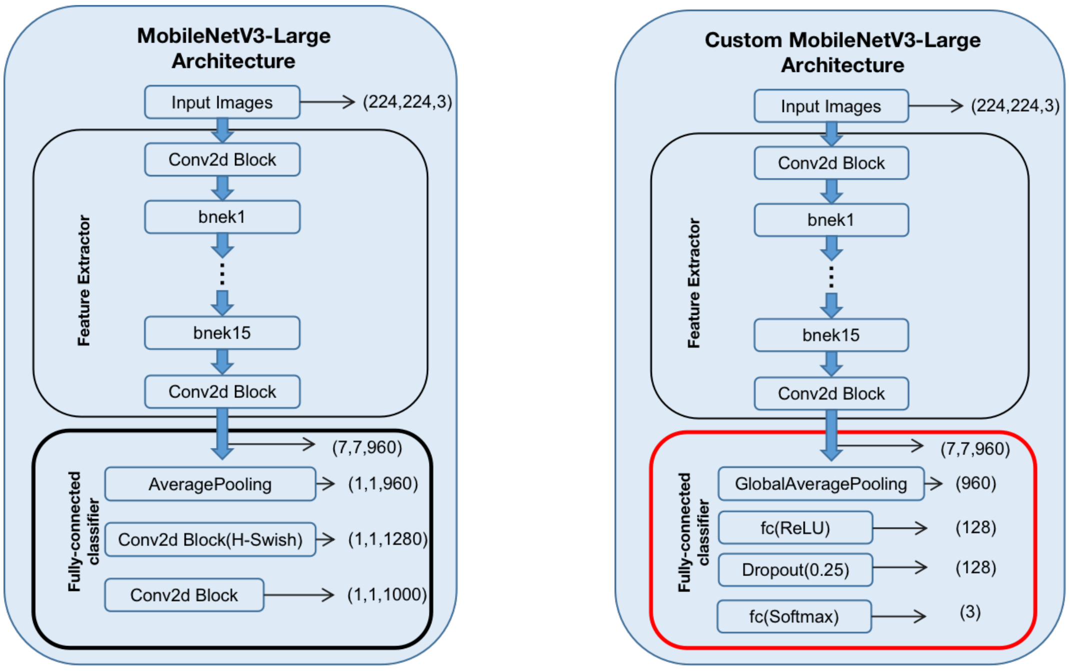


Figure 5.13: Comparison of the original MobileNetV3-Large and MobileNetV3-Large based transfer learning model.

Fine-Tuning Approach

Although the method of directly extracting features using existing CNN models has achieved good performance, there remains potential for improvement. As we know, deep CNN gradually abstracts the image information from the bottom layer to the top layer when performing feature extraction. The CNN models VGG16 and

MobileNetV3-Large are pre-trained on ImageNet, that is very different from our CSI data. Thus these features extracted for natural image classification are not well fit for the CSI classification. To better align with our specific task requirements, we employ fine-tuning techniques to retrain the existing model. The fine-tuning process is to initialize the network with pre-trained parameters from ImageNet and subsequently adjusting these parameters using our dataset. During this process, due to the good versatility of the underlying image features of the training CNN model, we particularly focus on retraining the weights of the top layers of MobileNetV3-Large and the remaining fully connected layers to enhance their adaptability to CSI classification. To do that, we unfreeze the *base_MobileNetV3L* model and set layers of last convolution block of MobileNetV3-Large to be trainable:

```
base_MobileNetV3L.trainable = True
fine_tuning = 256
# Freeze all the layers before fine_tuning
for layer in MobileNetV3L.layers[:fine_tuning]:
    layer.trainable = False
```

Except that, the proposed model architecture is the same as the MobileNetV3-Large based transfer learning model shows in Figure 5.13.

Additionally, the parameters of the unfrozen last convolutional layers are updated based on pre-trained parameters from ImageNet rather than random initialization. This necessitates very small adjustments to avoid missing optimal convergence. Therefore, the learning rate of the optimizer is set to a significantly lower value of 0.0005 (see Table 5.2), compared to the learning rate of 0.001 used without fine-tuning. Meanwhile, more epochs are needed because of a lower learning rate.

Hyperparameters

The hyperparameters of model 2, MobileNetV3-Large based transfer learning without fine-tuning, are same as model 1, as shown in Table 5.1. The hyperparameters of model 3, which use fine-tuning approach are tabulated in Table 5.2.

Table 5.2: Hyperparameters used for the proposed model II.

Hyperparameters	**Value**
Bach size	32
Optimizer	Adam, learning rate=0.0005
Loss function	Cross Entropy
Epochs	100

5.2 Model Evaluation

In our training process, we incorporate both Early Stopping and Model Checkpoint techniques to enhance the model performance and prevent overfitting. Early Stopping is a widely used regularization technique, it is employed to halt the training when the performance on the validation set ceases to improve for a predetermined number of epochs, known as the patience parameter. This approach ensures that training does not continue unnecessarily, thus avoiding overfitting, as shown in Figure 5.14. Concurrently, Model Checkpoint is utilized to monitor the model performance at each epoch, saving the model's weights whenever an improvement in the validation metric is observed. This technique ensures that the best-performing model is preserved, allowing us to restore the model weights from the optimal point in the training process. By combining these techniques, we maintain model generalizability and efficiency, ensuring robust performance on unseen data.

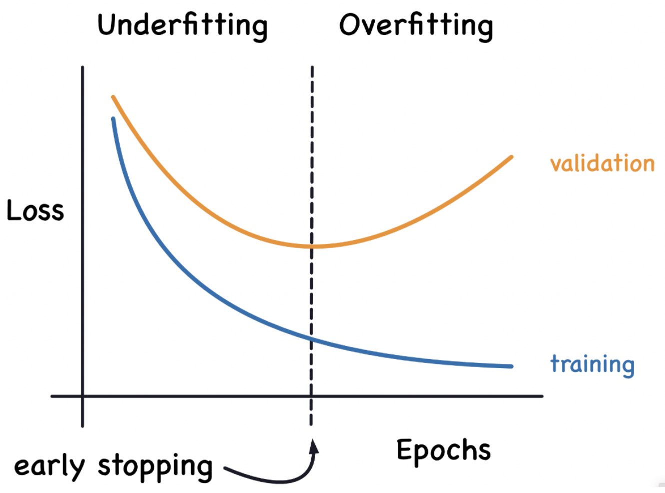


Figure 5.14: Early Stopping Technology.

The raw CSI amplitude data, represented as a 52-dimensional vector, is converted into grayscale images and subsequently input into VGG16 based and MobileNetV3-Large based transfer learning models. The architectures of these networks are illustrated in Figures 5.7 and 5.13.

In Table 5.3 we compare the experimental results of the VGG16 based transfer learning model, MobileNetV3-Large based transfer learning model, and MobileNetV3-Large based transfer learning with fine-tuning. The performance are also compared with other transfer learning approach for WiFi sensing tasks from different research, namely VGG16 with SVM [6] and MobileNetV2-based transfer learning [9], as shown in the first two lines in the following table.

The proposed models are validated with test dataset. The model based on VGG16 shows the training accuracy up to 96% and validation accuracy 94.44%. While the model based on MobileNetV3-Large achieves training accuracy of 98.89% and validation accuracy 97.78% at 26 epochs. After applying fine-tuning, the validation accuracy of MobileNetV3-Large based transfer learning model improves to 98.89% with 84 epochs. It confirms that fine-tuning helps enhance the applicability of pre-trained models for custom classification tasks comes with much more iterations required.

The MobileNetV3-Large model with and without fine-tuning take almost same training time, which is 60% less than the time that VGG16-based model uses for training. This huge difference also appears when comparing the model size. This due to the structure and the number of parameters of pre-trained models. VGG16 model has 14.8 million parameters, while MobileNetV3-Large model contains 3.1 million parameters and is constructed in a more efficient way. Drawing from the evaluation results, choosing our MobileNetV3-Large based model with fine-tuning, achieves higher accuracy with time-efficiency and relative smaller model size, making it well-suited for deployment in Customer Premises Equipment and IoT devices for localized HAR tasks.

Table 5.3: Model Evaluation. Accuracy and Loss are refer to validation accuracy and validation loss. Bach size are the same, result in same amount of data in one epoch.

Model	Accuracy (%)	Loss (%)	Training Time (s/epoch)	Model Size
VGG16-SVM [6]	97.75	-	-	56.13 MB
MobileNetV2 [9]	86	0.4	-	9.86 MB
VGG16	94.44	0.17	55-60	56.38 MB
MobileNetV3-Large	97.78	0.04-0.05	**15-20**	11.9 MB
MobileNetV3-Large with Fine-Tuning	**98.89**	**0.02-0.03**	18-22	11.9 MB

Analyzing the confusion matrices in 5.15, 5.16 and 5.17 for the three proposed models reveals distinct performance differences. The VGG16 based model struggles with accurately identifying the 'move' activity, often misclassifying it as 'fall' or 'down.' In contrast, the MobileNetV3-Large based models demonstrate higher overall accuracy but occasionally confuse 'fall' and 'down' activities, likely due to their inherent similarities. Understanding these specific limitations is crucial for targeted model improvements and for making informed assessments of model outputs in practical applications.

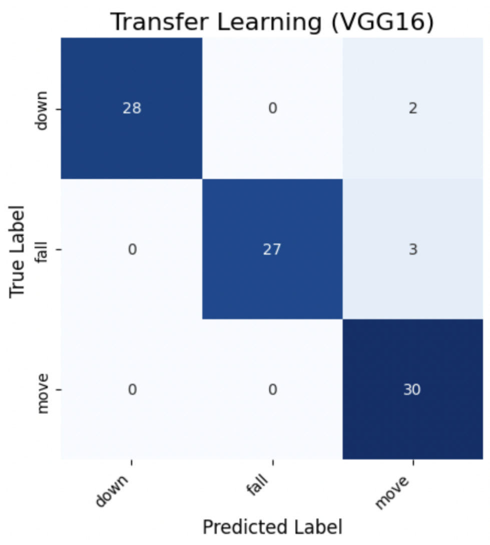
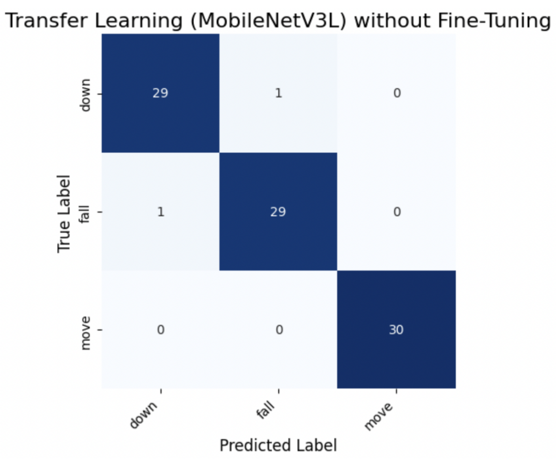


Figure 5.15: VGG16 Model Confusion Matrix.



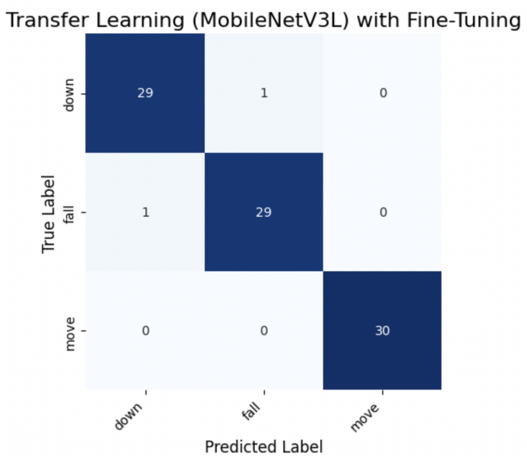


Figure 5.16: MobileNetV3L Model Confusion Matrix.

Figure 5.17: MobileNetV3L Model with FT Confusion Matrix.

Chapter 6

Deployment of Machine Learning Models in Local Environments

6.1 Environment Configuration

This section describes the User Services Platform Agent and User Services Platform Controller built for this study in terms of selected hardware and developed software.

The USP Agent is implemented in C language and hosted on a Raspberry Pi 4 Model B Single-Board Computer with 4GB of RAM. The USP Controller is implemented in Python.

6.1.1 User Services Platform Agent-CPE

Hardware Selection

Given that the User Services Platform Agent serves as a representation of potential Customer Premises Equipment at the physical location of end-users, a Single-Board Computer is an ideal choice due to its relevance in home automation. Among the various Single-Board Computers available on the market, we selected the Raspberry Pi 4 Model B, see Figure 6.1, for the following reasons:

- Has an operating system, Raspberry Pi OS, based on a Debian Linux distribution, which is essential for installing the User Services Platform Agent.
- Equipped with an integrated Broadcom WiFi Chip (bcm43455c0), which requires running a 4.9, 4.14, or 4.19 version of the Raspbian kernel to operate. To extract Channel State Information (CSI) from OFDM-modulated WiFi

frames, the Nexmon tool is utilized. Nexmon consists of a series of firmware patches specifically designed for the Broadcom chip used by the Raspberry Pi for WiFi connectivity.

- Offers a low price compared to other Single-Board Computer with similar features.

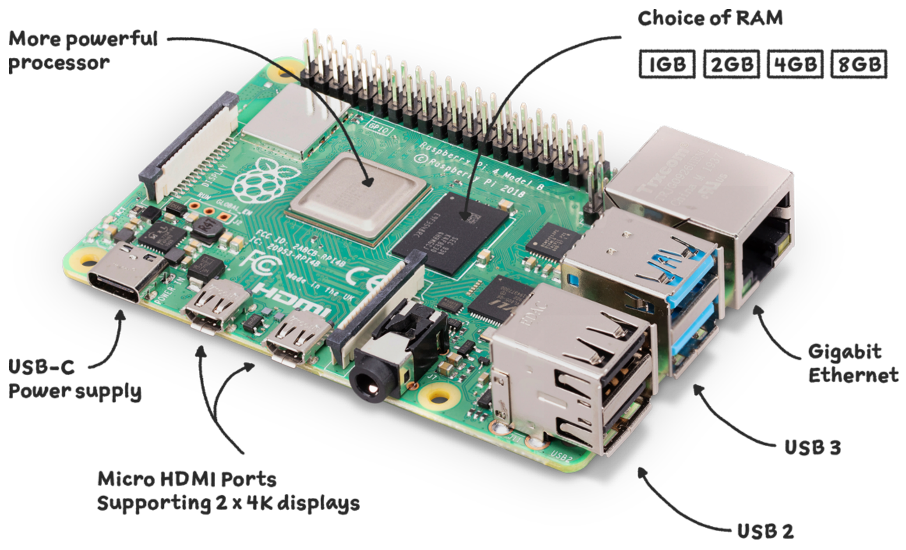


Figure 6.1: Raspberry Pi 4 Model B. Source: [17]

The Raspberry Pi 4 Model B and a laptop are connected to the same hotpots of a mobile phone. We connect to Raspberry Pi 4 Model B from the laptop with SSH by running following command in terminal:

```
ssh [Pi's username]@[the Pi's IP Address]
```

Software Development

Open Broadband-User Services Platform-Agent (OB-USP-Agent) creates a reference implementation of the User Services Platform specification from an 'Agent' perspective. For the development of the software that implement USP Agent, we need to refer to implementation made available by Broadband Forum on GitHub [18]. Following the quick start guide, to run OB-USP-Agent on Raspberry Pi it is necessary to:

- Download and install the latest stable OB-USP-Agent version from the official site (latest stable release is 8.0.0 when working on this part);

- Install necessary dependencies;
- Configure Message transport Protocol (MTP) to be Message Queuing Telemetry Transport (MQTT) and disable other MTPs;
- Create MQTT Broker;
- Run the USP Agent with a specific command in terminal.

OB-USP-Agent currently supports four Message Transfer Protocol (MTP), they are Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), and WebSockets. We choose to use MQTT MTP for it's simplicity, low power consumption, and real-time capabilities, which is ideal protocol for IoT applications, in particular those dealing with resource-constrained devices and real-time data collection and monitoring like our HAR task.

We established a MQTT Broker on HiveMQ Cloud, a proven enterprise MQTT platform. Utilizing HiveMQ's cloud service, we deployed a serverless MQTT broker, which supports basic authentication using username and password, and operates over Transport Layer Security (TLS). To securely connect the User Services Platform Agent to the MQTT Broker via TLS transport protocol, we configured the parameter *Device.MQTT.Client.1.TransportProtocol* in the database to *TLS*.

Before running OB-USP-Agent for the first time, it needs a database containing the settings of the USP Controller to contact. This is known as the factory reset database. A file *mqtt_factory_reset_example.txt* is contained in the installed OB-USP-Agent. To specify the data model parameters and values used to create the factory reset database, modify this file. To run OB-USP-Agent connecting to a MQTT server from network interface eth0, launch the following command:

```
obuspa -p -v 4 -r mqtt_factory_reset_example.txt -t hivemq.cloud.pem
-i eth0
```

This will also create a database if the factory reset database is not exist before. The factory reset parameters specified in *mqtt_factory_reset_example.txt* file are as follows:

```
## Agent Endpoint ID
Device.LocalAgent.EndpointID "proto::rx_usp_agent_mqtt"

## Adding Boot Parameters
Device.LocalAgent.Controller.1.BootParameter.1.Enable true
Device.LocalAgent.Controller.1.BootParameter.1.ParameterName
    "Device.LocalAgent.EndpointID"
Device.LocalAgent.Subscription.1.Alias cpe-1
Device.LocalAgent.Subscription.1.Enable true
Device.LocalAgent.Subscription.1.ID default-boot-event-ACS
Device.LocalAgent.Subscription.1.Recipient
    Device.LocalAgent.Controller.1
Device.LocalAgent.Subscription.1.NotifType Event
Device.LocalAgent.Subscription.1.ReferenceList Device.Boot!
Device.LocalAgent.Subscription.1.Persistent true

## MQTT Setup
Device.LocalAgent.MTP.1.MQTT.ResponseTopicConfigured "/usp/agent"
Device.LocalAgent.MTP.1.MQTT.Reference "Device.MQTT.Client.1"
Device.MQTT.Client.1.BrokerAddress "OMITTED"
Device.MQTT.Client.1.ProtocolVersion "5.0"
Device.MQTT.Client.1.BrokerPort "8883"
Device.MQTT.Client.1.TransportProtocol "TLS"
Device.MQTT.Client.1.Username "OMITTED"
Device.MQTT.Client.1.Password "OMITTED"
Device.MQTT.Client.1.Alias "cpe-1"
Device.MQTT.Client.1.Enable true
Device.MQTT.Client.1.ClientID ""
Device.MQTT.Client.1.KeepAliveTime "60"
Device.MQTT.Client.1.ConnectRetryTime "5"
Device.MQTT.Client.1.ConnectRetryIntervalMultiplier "2000"
Device.MQTT.Client.1.ConnectRetryMaxInterval "60"

## Default Controller Setup
Device.LocalAgent.Controller.1.Alias "cpe-1"
Device.LocalAgent.Controller.1.Enable true
Device.LocalAgent.Controller.1.PeriodicNotifInterval "86400"
Device.LocalAgent.Controller.1.PeriodicNotifTime
    "0001-01-01T00:00:00Z"
Device.LocalAgent.Controller.1.ControllerCode ""
Device.LocalAgent.Controller.1.MTP.1.Alias "cpe-1"
Device.LocalAgent.Controller.1.MTP.1.Enable true
Device.LocalAgent.Controller.1.MTP.1.Protocol "MQTT"
Device.LocalAgent.Controller.1.EndpointID "usp-controller-default"
Device.LocalAgent.Controller.1.MTP.1.MQTT.Reference
    "Device.MQTT.Client.1"
Device.LocalAgent.Controller.1.MTP.1.MQTT.Topic
    "/usp/controller/default"
```

```
Device.LocalAgent.MTP.1.Alias "cpe-1"
Device.LocalAgent.MTP.1.Enable true
Device.LocalAgent.MTP.1.Protocol "MQTT"
```

6.1.2 User Services Platform Controller-ACS

As one of the main advantages of User Services Platform protocol, it is possible to define different types of Controller for the same Agent. Types of Controller could be data warehouse server for data collection, third party managed service provider, or remote control from end-user by using web page or mobile applications. In the thesis work, we build a Python script executable from terminal, which runs like an application as a User Services Platform Controller. It is able to:

- Onboard to the USP Agent through Message Queuing Telemetry Transport;
- Subscribe or unsubscribe to receive notifications messages of desire information;
- View the real-time WiFi sensing based human activity prediction result with benchmark information;
- Perform diagnostics of the Agent and Controller.

MQTT Onboard

The initial operation involves establishing a connection to an available Agent. This connection is facilitated through the Message Queuing Telemetry Transport protocol. The primary requirement is that both the Agent and Controller can connect to the MQTT broker hosted in our cloud infrastructure, as detailed in Section 6.1.1. This connection to the MQTT broker is secured with Transport Layer Security and requires authentication.

USP ValueChange Notification

Real-time communication between the USP Agent and the USP Controller is facilitated through USP messages. By subscribing to an object in the Agent's supported data model with a notification type set to *ValueChange*, the Controller automatically receives notifications whenever the value of the subscribed object changes, see Figure 6.2. The Operate Response can be disabled or enabled by modify *send_resp* flag, based on the user intention. Figure 6.3 the Operate Message flow.

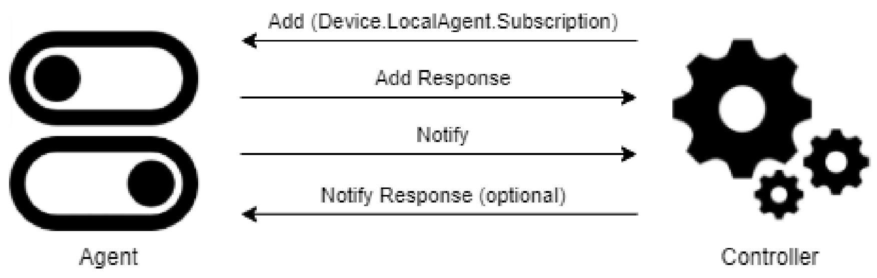

Add (Device.LocalAgent.Subscription)
Add Response
Notify
Notify Response (optional)
Agent
Controller

Figure 6.2: User Services Platform Notification Mechanism. Source: [12]

In this study, the USP Controller receives notify messages each time new CSI data is collected, enabling end-users to view real-time WiFi-sensing-based human activity predictions and benchmark information (see Section 6.2.3 for benchmark information details).

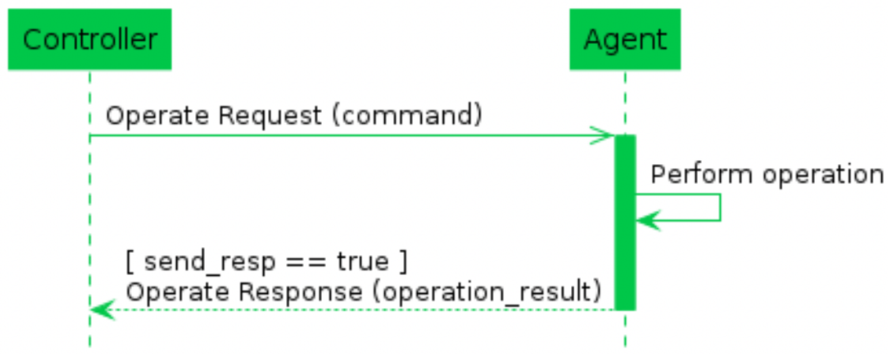

Controller
Agent
Operate Request (command)
Perform operation
[send_resp == true]
Operate Response (operation_result)

Figure 6.3: Operate Message Flow for Synchronous Operations. Source: [13]

6.2 Model Deployment

The transfer learning model demonstrating the best performance, specifically the MobileNetV3-Large with fine-tuning (refer to Section 5.2), is integrated as a function within the User Services Platform Agent to enable real-time WiFi sensing-based human activity classification. The overall architecture is illustrated in Figure 6.4.

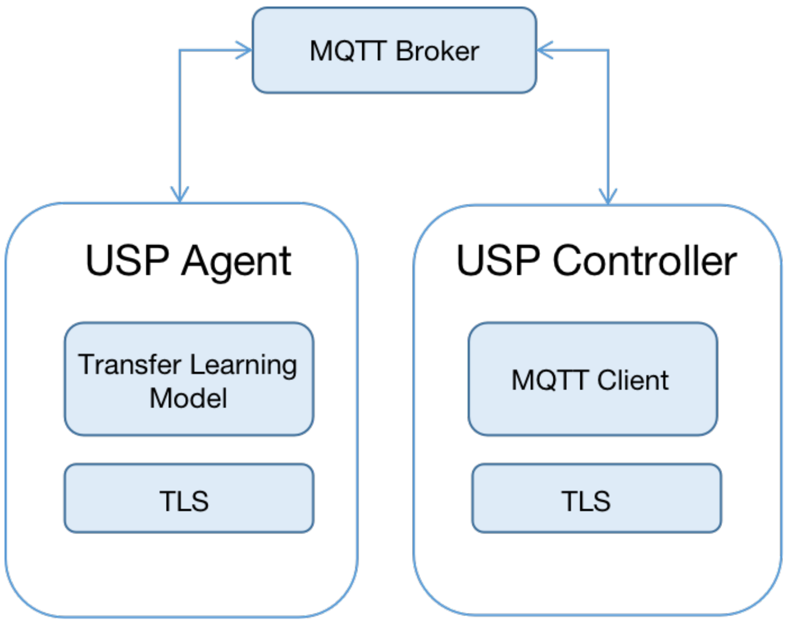


Figure 6.4: USP over MQTT Framework.

6.2.1 Enable Model Execution in OB-USP-Agent

Considering the OB-USP-Agent is coded in C language, while the transfer learning model is completely implemented in Python, we need to find a way to run the inference in the OB-USP-Agent. There are possibly two ways to do that:

- TensorFlow for C [19]. It provides a C API that can be used to build bindings for other languages;
- Onnx2c method [20]. Onnx2c is an Open Neural Network Exchange (ONNX) to C compiler. It will read an ONNX file, and generate C code to be included in the project.

We are doing model deployment in a local device, so the model size would be an important reference standard. By saving the model in TensorFlow and ONNX format respectively, we find that the saved model size of TensorFlow Keras standard format .h5 is 14.3 MB, while the model saved as .onnx format has size 12.4 MB, which is slightly smaller than the TensorFlow one.

The interoperability of ONNX allows models to be trained in one framework and deployed in another, facilitating seamless transitions between frameworks such as PyTorch, TensorFlow, and others. This flexibility accelerates the development process and enhances the efficiency of deploying machine learning models across various platforms and environments. Therefore, deploying the ONNX format model to the Agent offers significant advantages for future model conversions or export.

Furthermore, from the perspective of implement difficulty, if use onnx2c method, we need to:

1. Install ProtocolBuffers libraries;
2. Get the sources from Github [20];
3. Run a standard CMake build which creates onnx2c binary;
4. Run command: *./onnx2c [your ONNX model file] > model.c*. This command generates a file named model.c. At the end of model.c, there is a function named *void entry(...)*, which is the function we can invoke from the OB-USP-Agent main program to execute the model.

If use TensorFlow for C method, the model need to be fully rewritten using the API defined in *c_api.h* [21], which involves significant effort. In comparison, we have selected to utilize the onnx2c compiler due to its streamlined approach.

6.2.2 Preprocessing of Input Images

Deployed transfer learning algorithm expects input image size to be (224, 224). To achieve that constraint, we use FreeImage library for C to resize input Channel State Information images (we assume that the collected real-time Channel State Information data is already converted to images).

6.2.3 Model Benchmark Information Collection

To evaluate the performance of integrated model, benchmark information is gathered, focusing specifically on the model execution time. This is accomplished by including the time.h header in the OB-USP-Agent main program, which provides functions for measuring elapsed time with high precision. By integrating these functions into the model's code, we can accurately record the start and end times of the model's execution. This data is critical for assessing the efficiency and practicality of the model, particularly when deployed in resource-constrained environments like Customer Premises Equipment. Detailed analysis of the execution time helps in understanding the computational requirements and optimizing the model for better performance.

6.3 Model Testing

This section details the execution of the deployed model within the MQTT communication framework of the USP Agent and the USP Controller. The CSI data used for testing remains consistent with that used for model training [4].

Message Queuing Telemetry Transport Communication

Firstly, the USP Agent must be execute before the USP Controller by running the following command in the Raspberry Pi terminal:

```
obuspa -p -v 4 -r mqtt_factory_reset_example.txt -t hivemq.cloud.pem
-i eth0
```

We will see the following trace, meaning that OB-USP-Agent is successfully connected to the MQTT server:

```
USP_CONNECT_RECORD sending at time 2024-07-14T16:58:02Z,
to host over MQTT
version: "1.3"
to_id: "usp-controller-default"
from_id: "proto::rx_usp_agent_mqtt"
payload_security: PLAINTEXT
mac_signature[0]
sender_cert[0]
mqtt_connect {
  version: V5
  subscribed_topic: "/usp/agent"
}

NOTIFY sending at time 2024-07-14T16:58:02Z, to host over MQTT
version: "1.3"
to_id: "usp-controller-default"
from_id: "proto::rx_usp_agent_mqtt"
payload_security: PLAINTEXT
mac_signature[0]
sender_cert[0]
no_session_context {
  payload[225]
}

header {
  msg_id: "Event-2024-07-14T16:58:02Z-1"
  msg_type: NOTIFY
}
body {
  request {
    notify {
      subscription_id: "default-boot-event-ACS"
      send_resp: false
      event {
        obj_path: "Device."
        event_name: "Boot!"
        params {
          key: "CommandKey"
          value: ""
        }
        params {
          key: "Cause"
          value: "LocalReboot"
        }
        params {
          key: "FirmwareUpdated"
          value: "false"
        }
        params {
          key: "ParameterMap"
          value: "{"Device.LocalAgent.EndpointID":
          "proto::rx_usp_agent_mqtt"}"
        }
      }
    }
  }
}
```

After execute the USP Controller Python script, communication between USP Agent and USP Controller will take place through the exchange of User Services Platform messages. Controller will automatically send a User Services Platform *ADD* message to the Agent which it is connected. With this message it requests the human activity prediction result from the Agent. This message will look like this:

```
header {
  msg_id: "7a991ae0-4204-11ef-b11a-1e00d90a7531"
  msg_type: ADD
}
body {
  request {
    add {
      allow_partial: false
      create_objs {
        obj_path: "Device.LocalAgent.Subscription."
        param_settings {
          param: "Enable"
          value: "true"
          required: true
        }
        param_settings {
          param: "ID"
          value: "WiFiSensing"
          required: true
        }
        param_settings {
          param: "NotifType"
          value: "ValueChange"
          required: true
        }
        param_settings {
          param: "ReferenceList"
          value: "Device.Services.WiFiSensing.CSI.Value"
          required: true
        }
      }
    }
  }
}
```

The Agent response is like this:

```
header {
  msg_id: "7a991ae0-4204-11ef-b11a-1e00d90a7531"
  msg_type: ADD_RESP
}
body {
  response {
    add_resp {
      created_obj_results {
        requested_path: "Device.LocalAgent.Subscription."
        oper_status {
          oper_success {
            instantiated_path: "Device.LocalAgent.Subscription.2."
            unique_keys {
              key: "Alias"
              value: "cpe-2"
            }
            unique_keys {
              key: "ID"
              value: "WiFiSensing"
            }
            unique_keys {
              key: "Recipient"
              value: "Device.LocalAgent.Controller.1"
            }
          }
        }
      }
    }
  }
}
```

Once a response has been received from the Agent, the communication between Agent and Controller is established successfully. Controller will keep listening on the topic associated to it to receive *ValueChange* type notifications if the prediction result changes in value. These notifications reach Controller as User Services Platform Notify messages like this:

```
header {
  msg_id: "ValueChange-2024-07-14T17:14:44Z-2"
  msg_type: NOTIFY
}
body {
  request {
    notify {
      subscription_id: "WiFiSensing"
      send_resp: false
      value_change {
        param_path: "Device.Services.WiFiSensing.CSI.Value"
        param_value: "{"moving":"0.949602","cpu_time_used":"0.611123"}"
      }
    }
  }
}
```

Field *param_path* specifies the path name of the changed parameter, while field *param_value* contains the corresponding value of the parameter identified in *param_path*. The model prediction result outcome from an input CSI image is encapsulated as a JSON string within the *param_value* field. The JSON string shows in the above Notify message specifies that the analyzed CSI data has a 95% probability of belonging to the 'moving' class, and the time taken to process this CSI image is 0.611123 seconds.

The model prediction result is extracted and print in a separate line, for a user-understandable consideration:

```
{"action": "move","consumed time":"0.611123"}
```

This message informs the Controller a person is moving within the detection range, with the model execution time being 0.611123 seconds.

Chapter 7

Conclusion & Future Work

This paper evaluates CNN-based transfer learning models for WiFi sensing and compares their performance for human activity recognition using Customer Premises Equipment (CPE). The models are trained and tested on a public Channel State Information (CSI) amplitude dataset. We assess the performance of proposed models based on accuracy, loss, and training time. Model size is also considered a crucial metric since our aim is to deploy these models on CPE in a local environment. The MobileNetV3-Large based transfer learning model with fine-tuning achieves a best validation accuracy of 98.89% at 84 epochs, while the accuracy without fine-tuning is 97.78%, and VGG16 reaches 94.44%. A portion of pre-trained layers are retrained through fine-tuning, where the parameters are updated using our dataset. This adaptation enhances the model suitability for CSI based human activity classification and results in improved experimental performance on our dataset. Moreover, the MobileNetV3-Large methods require two-thirds less training time and have 80% less model size compared to the VGG16 method. Consequently, it can be preliminarily concluded that the deep transfer learning approach utilizing MobileNetV3-Large is optimal for deployment in CPE and IoT devices due to its lightweight architecture, cost-effectiveness, and time efficiency in executing local deep learning tasks.

In this study, the CSI data used to test the deployed transfer learning algorithm within the USP framework is sourced from a public dataset, simulating real-time data collection. However, in practical application scenarios, the model will undergo retraining and updating based on newly collected data. This iterative process aims to enhance the model's adaptability and improve its performance over time.

Future Possibilities

Another machine learning approach, federated learning, enables decentralized model training across multiple devices while preserving data privacy. Integrating CNN-based transfer learning models in this context allows individual devices, such as Customer Premises Equipment, to locally train on user data and collaboratively improve a global model without sharing raw data. Therefore, this approach addresses privacy concerns and enhances model generalization across diverse environments. The potential benefits include more robust human activity recognition, reduced communication costs, and scalability in real-world applications, paving the way for more personalized and efficient IoT solutions in smart homes and healthcare monitoring.

Regarding model deployment in Customer Premises Equipment, our current approach involves using a Python script executable via terminal as the User Services Platform Controller. However, within the User Services Platform protocol, there is the capability to define various other types of Controllers. A prospective avenue for future development of User Services Platform Controller is the creation of a mobile application for handheld devices. This direction would offer end-users greater flexibility in remotely managing their connected Customer Premises Equipment.

Bibliography

[1] Daniel Wegemer Matthias Schulz and Matthias Hollick. *Nexmon: The C-based Firmware Patching Framework*. 2017. URL: `https://github.com/seemoo-lab/nexmon` (cit. on pp. 2, 7, 14).

[2] Yaxiong Xie, Zhenjiang Li, and Mo Li. «Precise Power Delay Profiling with Commodity WiFi». In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. MobiCom '15. Paris, France: ACM, 2015, pp. 53–64. ISBN: 978-1-4503-3619-2. DOI: `10.1145/2789168.2790124`. URL: `http://doi.acm.org/10.1145/2789168.2790124` (cit. on p. 2).

[3] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. «Tool Release: Gathering 802.11n Traces with Channel State Information». In: *ACM SIGCOMM CCR* 41.1 (Jan. 2011), p. 53. URL: `https://dhalperi.github.io/linux-80211n-csitool/` (cit. on pp. 2, 7).

[4] Parisa Fard Moshiri, Reza Shahbazian, Mohammad Nabati, and Seyed Ali Ghorashi. «A CSI-Based Human Activity Recognition Using Deep Learning». In: *Sensors* 21.21 (2021). ISSN: 1424-8220. DOI: `10.3390/s21217225`. URL: `https://www.mdpi.com/1424-8220/21/21/7225` (cit. on pp. 2, 7, 14, 15, 41).

[5] Jianfei Yang, Xinyan Chen, Han Zou, Chris Xiaoxuan Lu, Dazhuo Wang, Sumei Sun, and Lihua Xie. «SenseFi: A library and benchmark on deep-learning-empowered WiFi human sensing». In: *Patterns* 4.3 (2023), p. 100703. ISSN: 2666-3899. DOI: `https://doi.org/10.1016/j.patter.2023.100703`. URL: `https://www.sciencedirect.com/science/article/pii/S2666389923000405` (cit. on p. 7).

[6] Qirong Bu, Gang Yang, Jun Feng, and Xingxia Ming. «Wi-Fi Based Gesture Recognition Using Deep Transfer Learning». In: *2018 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. 2018, pp. 590–595. DOI: `10.1109/SmartWorld.2018.00122` (cit. on pp. 7, 32).

[7] Nurzarinah Zakaria and Yana Mazwin Mohmad Hassim. «Improved VGG Architecture in CNNs for Image Classification». In: *2022 IEEE International Conference on Artificial Intelligence in Engineering and Technology (IICAIET)*. 2022, pp. 1–4. DOI: `10.1109/IICAIET55139.2022.9936735` (cit. on p. 7).

[8] Sruthi P. and Siba K. Udgata. «Wi-Fi sensing based person identification and activity recognition using two-phase deep learning model». In: *Engineering Applications of Artificial Intelligence* 132 (2024), p. 107904. ISSN: 0952-1976. DOI: `https://doi.org/10.1016/j.engappai.2024.107904`. URL: `https://www.sciencedirect.com/science/article/pii/S0952197624000629` (cit. on p. 7).

[9] Ambreen Hussain, Bidushi Barua, Ahmed Osman, Raouf Abozariba, and A. Taufiq Asyhari. «Performance of MobileNetV3 Transfer Learning on Handheld Device-based Real-Time Tree Species Identification». In: *2021 26th International Conference on Automation and Computing (ICAC)*. 2021, pp. 1–6. DOI: `10.23919/ICAC50006.2021.9594222` (cit. on pp. 7, 32).

[10] Yao Ge, Ahmad Taha, Syed Aziz Shah, Kia Dashtipour, Shuyuan Zhu, Jonathan Cooper, Qammer H. Abbasi, and Muhammad Ali Imran. «Contactless WiFi Sensing and Monitoring for Future Healthcare - Emerging Trends, Challenges, and Opportunities». In: *IEEE Reviews in Biomedical Engineering* 16 (2023), pp. 171–191. DOI: `10.1109/RBME.2022.3156810` (cit. on p. 9).

[11] Wikipedia. *TR-069*. `https://en.wikipedia.org/wiki/TR-069` (cit. on p. 11).

[12] Broadband Forum. `https://usp.technology/` (cit. on pp. 11, 39).

[13] Broadband Forum. *TR-369 – The User Services Platform*. `https://usp.technology/specification/index.pdf` (cit. on pp. 13, 39).

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «ImageNet: A large-scale hierarchical image database». In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848` (cit. on p. 18).

[15] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: `1409.1556 [cs.CV]`. URL: `https://arxiv.org/abs/1409.1556` (cit. on pp. 19, 20).

[16] Andrew Howard et al. *Searching for MobileNetV3*. 2019. arXiv: `1905.02244 [cs.CV]`. URL: `https://arxiv.org/abs/1905.02244` (cit. on pp. 26–28).

[17] Raspberry Pi. *Raspberry Pi 4 Model B*. `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/` (cit. on p. 35).

[18] Broadband Forum. *OB-USP-Agent*. `https://github.com/BroadbandForum/obuspa/blob/master/QUICK_START_GUIDE.md` (cit. on p. 35).

[19] TensorFlow. *TensorFlow for C*. `https://www.tensorflow.org/install/lang_c` (cit. on p. 40).

[20] *Onnx2c compiler*. `https://github.com/kraiskil/onnx2c` (cit. on pp. 40, 41).

[21] TensorFlow. *$c_api.h$*. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h` (cit. on p. 41).

[22] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: `1905.11946 [cs.LG]`. URL: `https://arxiv.org/abs/1905.11946`.

[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. 2019. arXiv: `1801.04381 [cs.CV]`. URL: `https://arxiv.org/abs/1801.04381`.

[24] Mahnaz Chahoushi, Mohammad Nabati, Reza Asvadi, and Seyed Ali Ghorashi. «CSI-Based Human Activity Recognition Using Multi-Input Multi-Output Autoencoder and Fine-Tuning». In: *Sensors* 23.7 (2023). ISSN: 1424-8220. DOI: `10.3390/s23073591`. URL: `https://www.mdpi.com/1424-8220/23/7/3591`.

[25] M. Humayun Kabir, M. Hafizur Rahman, and Wonjae Shin. «CSI-IANet: An Inception Attention Network for Human-Human Interaction Recognition Based on CSI Signal». In: *IEEE Access* 9 (2021), pp. 166624–166638. DOI: `10.1109/ACCESS.2021.3134794`.

[26] Juan Augusto Campos-Leal, Arturo Yee-Rendón, and Inés Fernando Vega-López. «Simplifying VGG-16 for Plant Species Identification». In: *IEEE Latin America Transactions* 20.11 (Aug. 2022), pp. 2330–2338. URL: `https://latamt.ieeer9.org/index.php/transactions/article/view/6728`.

[27] Sheldon Mascarenhas and Mukul Agarwal. «A comparison between VGG16, VGG19 and ResNet50 architecture frameworks for Image Classification». In: *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*. Vol. 1. 2021, pp. 96–99. DOI: `10.1109/CENTCON52345.2021.9687944`.