UNIVERSITY OF LIVERPOOL

DEPARTMENT OF ELECTRICAL ENGINEERING & ELECTRONICS

Final report for project 'NFC Device Identification Using Deep Learning and Radio Frequency Fingerprint'

Author: Tianyou Li (201676614)

Project Supervisor: Junqing Zhang

Project Assessor: Valerio Selis

Declaration of academic integrity

By submitting this work I confirm that I have read and understood the University's Academic Integrity Policy.
By submitting this work I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.
By submitting this work I confirm that the work I am submitting is my own. I have not commissioned production of the work from a third party or used artificial intelligence software in an unacceptable manner to generate the work*. I have not copied material from another person or source, nor committed plagiarism, nor fabricated, falsified or embellished data when completing the attached piece of work. I have not colluded with any other student in the preparation and/ or production of this work.
*Software applications include but are not limited to, ChatGPT, Bing Chat, DALL.E, Bard

SIGNATURE: Tianyou Li DATE: April 15, 2024

Abstract

Tag clone technology is threatening NFC tag security. With the help of a smart card application, anyone can steal information and then clone it into a 2nd gen tag or use the cell phone to emulate the genuine tag. Due to the feature of this approach, using tag ID and cryptographic mechanisms may fail to detect the forge tag. To solve tag abuse, RF fingerprinting technologies employ the physical layer signal, which contains unique distortion due to imperfections during manufacturing. However, due to the maturity of NFC tag manufacture the previous works have to employ high-end but expensive signal acquisition devices to capture the extremely small distortion caused by NFC tags. This project aims to further explore the practicability of the NFC Radio Frequency Fingerprinting Identification (RFFI) system in low-end conditions without the modification of the interrogation signal between the reader and tag. Our project employs Triplet loss and SoftTriple loss to effectively authenticate and identify the unknown packets acquired by low-end SDR devices. Additionally, the instability of the RF fingerprint is successfully addressed by designing a special card slot, achieving significant results.

Original Specifications

Specifications	**Results**
Accuracy over 95%	98.438%
FPR (False positive rate) below 5%	Achieved
Generalization ability	Achieved
Time cost below 0.5s	3.379s

Content

1 Introduction

This project explores the innovative application of Radio Frequency (RF) fingerprinting to enhance Near Field Communication (NFC) security, a technology pivotal in entry control and contactless payment systems. With the growing need for robust security solutions in wireless communications, especially following increased digital transactions and data exchanges, this research contributes to the field of wireless communication security.

The scope of this project covers the development and evaluation of a deep learning-based approach to extract device-specific features, known as RF fingerprints, which are unique and distinctive due to manufacturing imperfections. This method leverages the advanced ability of deep learning to significantly improve the reliability and accuracy of NFC security systems without exhausting manual feature extraction.

The relevance of this study to the current research area lies in its potential to significantly reduce security breaches in NFC operations, providing a more secure scheme for industries relying heavily on contactless technologies. The research findings could pave the way for wider real-world applications in various high-security environments, aligning with the technological trends towards more integrated and intelligent systems.

The remainder of the paper is organized as follows: Section 2 demonstrates the industrial relevance of this project. Section 3 provides a comprehensive literature review of recent relevant research. Section 4 introduces the basic background of the topics involved in this project, providing essential context and foundational knowledge. Section 5 focuses on the main design of this project. Section 6 presents the experimental results and analysis, offering insights into the effectiveness and implications of the research findings. And, Section 7 discusses the main problems encountered during this project and the future improvements. Then, Section 8 provides some reflections on what I learned by finishing this project. Finally, Section 9 concludes the whole report.

2 Industrial Relevance

Near-field communication (NFC) technology is commonly utilized in access control and payment systems, playing an increasingly significant role in people's daily lives [2]. Following the COVID-19 epidemic, the demand for contactless communication has surged [6]. The global NFC market was estimated to be $15 billion in 2019 and is expected to witness constant growth, reaching over $54 billion by the end of 2028 [7], [8].

One of the key features that brought NFC technology such prosperity is that NFC technology is compatible with mobile phone applications and therefore accelerated by the rapid development of mobile phones [19]. This feature enabled mobile devices to modify the data stored in the NFC tags and to emulate the NFC tags to be used for NFC readers. According to [19], the NFC application consists of three different modes: Card Emulation, Reader/Writer, and P2P. The application distribution is shown in Table 2:

Table 2: NFC Application Operation Mode [19]

Operation Mode	Count	Percentage
Card Emulation	6	16.22
Reader/Writer	29	78.38
P2P	2	5.40
Total	35	100

After analyzing the application distribution identified in previous research, it can be concluded that card emulation mode is the most important. This is because emulating devices, such as mobile phones, are commonly used to perform the same tasks as NFC tags for NFC readers. By storing card information on mobile devices, users can leverage these devices for various applications, such as payment applications or electronic key applications. For instance, in the realm of payment applications, users can make payments using their emulating mobile phone, like with Apple Pay, without the need to carry physical credit cards. However, despite this application's dominant place in the NFC market, this project does not focus on such applications. This is because complex encryption algorithms have been applied to secure NFC payments and powerful point-to-sale (POS) devices provide the required power to support the complex algorithms.

But for the second case, users are capable of opening the hotel room with their mobile devices or entering entry-restrict facilities by emulating the hotel room card or staff ID cards. Even though the majority of hotels set a valid time for the room cards, the cardholder or attacker can still use the emulated card to enter the room, which is a significant security problem.

The previously proposed RFID system for NFC technologies lacks real-world applicability, due to the incompatibility caused by modification of communication signals or the high cost of the signal acquisition device required. As to the system proposed in [2], their system needs a designed signal to initiate the construction of chains used as unique identification, which

might cause unnecessary worries in users since this might violate the information rights of the users. As to the system proposed in [4], the applied signal acquisition devices are USRP X300 and UBX160 daughter board. According to the official information provided by Ettus Research, the prices of them are £6,940 and £1,940. About £9,000 total price makes it impossible for users to implement this system even for a high-security required-entry system.

However, this project is focused on a low-end signal acquisition device and the cost of the whole system is listed below, the cost of the NFC reader is not included:

Table 3: System Implementation Cost

Device	**Name**	**Price (£)**
Antenna	MIKORE Regular NFC 40x30 Antenna	10.00
Cable	UFL to SMA Male Bulkhead	6.92
SDR	Nooelec RTL-SDR v5 SDR	33.95
Total	/	50.87

The total cost of about £50 significantly improves the real-world applicability of this system. Additionally, the system offers flexibility in scaling up the number of devices. By expanding the embedding space to generate distinguishable fingerprints [12], users can easily increase the number of legitimate devices. By analyzing experimental results, it is indicated that our system becomes operational immediately after adding labeled samples of a new device, facilitated by few-shot learning techniques. The pre-trained networks are designed to perform a distinguishing task based on features extracted from each legitimate device, rather than a classification task. As a result, the pre-trained network is capable of distinguishing both new devices and those previously registered. Further details are included in other sections.

3 Literature Review

RF fingerprinting has been employed for identifying specific wireless devices by investigating the signal features due to device-specific hardware imperfections during the manufacturing or the drift of electronic components.

Wang et al. [5] successfully used the device-specific features extracted by a deep convolutional neural network (CNN) to detect NFC relay attacks with an SDR-based testbed. By analyzing the features contained in the magnitude of the ATQA signal collected with a 10M sample rate, their experiments showed effective results in distinguishing normal and relayed NFC signals with a high accuracy of 99%.

Lee et al. [4] experimented with three different kinds of deep neural networks, namely feedforward neural network (FNN), convolutional neural network (CNN), and recurrent neural network (RNN), and achieved 96.16% accuracy in distinguishing 50 NFC cards. With up to 40M sample rate adopted, only a segment of normalized ATQA signal, with the data scaled within 0 to 1, is used as training data, thus proving that a signal segment corresponding to one data bit could be employed to extract RF fingerprints of NFC devices.

To increase scalability and practicability, Yang et al. [2] proposed a novel method compared to traditional RF fingerprinting identification (RFFI) systems. They devised a chain of tag response amplitudes (TRA) to identify each tag. The underlying principle is that the amplitude of the ATQA signal is unique for each NFC tag, and this uniqueness is maintained under multiple continuous wave (CW) frequencies within the range of 13.5-13.76MHz. By hopping over this frequency band with a specific step (1kHz in their experiments), they estimated the scalability limit of this scheme to be 7.99×10^223 based on an 8-bit analog-to-digital converter (ADC). They achieved as low as 3.7% FRR and 4.1% FAR for over 600 tags in their experiments. Moreover, since only the amplitude is employed, they reduce the cost by purchasing high-end SDR with a high sample rate.

However, the RFFI system has long suffered from fingerprint instability due to environmental changes or random noise in the SDR device [2], [12]. For instance, inconsistencies in the fingerprint can arise from different card positions, and variations may occur across multiple measurements of the same tag due to signal noise in the generic SDR device or NFC reader. Many researchers have attempted to address this problem in the RF fingerprint field.

Shen et al. [3] proposed a channel-robust RFFI (Radio Frequency Fingerprinting) system by combining channel-independent spectrograms with specific data augmentation techniques, resulting in a scalable authentication and identification system for LoRa devices. To prevent the network from capturing channel-specific features instead of device-specific features, they introduced data augmentation methods based on the characteristics of LoRa communication, such as the Doppler effect. Additionally, they employed deep metric learning techniques, such as triplet loss, to force the network to cluster features extracted from data under different channel simulation conditions. This approach enhanced the network's generalization ability,

leading to over 90% average accuracy in cross-domain tasks.

Han et al. [12] pointed out that the environment-sensitive characteristics of RF fingerprinting result in both false positives and false negatives. To address this issue, they proposed Butterfly, an environment-independent data collection method for RFID devices. In this approach, signals from a pair of reference tags and a test tag positioned at a close distance are collected. Since these two tags experience the same environmental or channel effects, the difference between the reference signal and the test signal is used for further processing. This effectively mitigates the environmental effects and yields over 90% accuracy across different positions and rooms.

While the aforementioned methods yield promising results, they may not be directly applicable in the field of NFC due to the inherently short range of NFC channels and the limitation of generic NFC readers, which typically do not support the simultaneous collection of signals from two devices. Nevertheless, these approaches shed light on addressing the environmental effects in the field of RF fingerprinting.

Similarly to the approach mentioned in [12], Yang et al. [2] introduced a factor nulling method where they used the first TRA sample as the reference to mitigate feature drift in RF fingerprinting induced by environmental changes. According to their theory, given the extremely short signal acquisition time of 0.2 seconds, factors such as the relative position of the tag to the reader and thermal effects remain consistent across all TRA samples collected. By normalizing subsequent samples with this reference, they were able to achieve results comparable to those presented in [12].

Johnson et al. [13] also introduced the variance fractal dimension trajectory (VFDT) as a data representation input for deep neural networks. According to their theory, the variance fractal dimension can be estimated by calculating the Hurst exponent based on a power law relationship. By selecting a 45M sample rate and optimizing parameters for VFDT estimation, such as window size and stride, their results demonstrated that models trained with this kind of data representation input maintained an accuracy of 60-70% when evaluated on datasets collected from random positions. In contrast, when tested under the same conditions, traditional methods provided test accuracies under 20%.

Additionally, Lee et al. [4] proposed a simple yet effective method to reduce the likelihood of deep learning networks capturing channel information. By diversifying the training data with datasets collected under various environmental conditions, the model is compelled to learn device-specific features. Therefore, data augmentation is much more important than different manual feature extraction methods since deep learning networks provide a much more powerful extraction ability.

As demonstrated in [4], a signal segment corresponding to one data bit can be effective for feature extraction in NFC device identification tasks. A classifier should be capable of detecting device-specific features regardless of their position within the signal, whether it's

the 1st or 15th data bit in the ATQA sequence [15]. Wang et al. [14] introduced the *stochastic permutation* method, which randomly swaps signal segments since device-specific features should be mutually independent.

The methods previously described focus on RF fingerprinting for LoRa, Wi-Fi, and NFC device identification. However, these methods either involve modifying the normal NFC interrogation signals or require expensive signal acquisition devices with a high sample rate, which significantly hinders the widespread application of RF Fingerprint Identification (RFFI) systems for NFC technology. This project aims to reduce the cost of RF fingerprinting while maintaining a high level of generalization capability through the use of RF fingerprinting and deep learning techniques.

4 Theory

What is Radio Frequency Fingerprinting?

Radio Frequency (RF) fingerprinting is a technique used for the identification or authentication of devices by analyzing the features in the physical-layer (PHY) signal caused by radio circuitry [1]-[2]. Due to imperfections in analog circuitry, extracting a hardware fingerprint containing the inherent and device-specific hardware properties of the device is possible.

These unique fingerprints have been utilized for various purposes, including intrusion detection [3], device identification [4], and attack detection [5]. The most significant advantage of employing RF fingerprinting for identification is that it is unclonable and does not need the modification of the normal interrogation signals, which might interrupt the normal communication process [1], [2].

What is Near-Field Communication?

Near-field communication (NFC) technology is commonly utilized in access control and payment systems, playing an increasingly significant role in people's daily lives [2]. However, this technology faces significant security challenges since its security heavily depends on the extremely short communication distance and duration [5].

While this characteristic makes it difficult for attackers to access the signal during communication as long as users maintain distance from others, it also means that all users within the effective distance can receive transmitted information since the communication duration limits the complexity of the authentication algorithm [5]. For instance, a relay attack can be executed using a sniffer antenna embedded in the reader in advance [5], or attackers can duplicate card information and clone it onto blank cards using common NFC readers or even smart card applications on their phones [2].

Why RF Fingerprinting for NFC?

As shown in Fig. 1 below, with the help of common smart card applications, such as MIFARE Classic Tool [9], a 1st generation NFC card can be easily duplicated. An attacker can then copy the sensitive information to any empty 2nd generation smart card [1]. To prevent information leakage caused by card abuse, many cryptographic algorithms have been proposed and applied to encrypt NFC card data [2]. However, due to the limitation of power supply and memory capability of NFC cards, the wide application seems implausible, since the advanced algorithms require more power supply from the NFC reader, which also leads to longer charging time when communicating with the reader (e.g. vendor machine).

Fig. 1 Sensitive Information Leakage

Thus, with the help of RF fingerprinting, which only requires the raw signal during the communication and does not need the sensitive data stored in the card, such as UID, the NFC card identification system without any modification of the interrogation signal is proposed.

Why Deep Learning for RF Fingerprint?

Recently, deep learning has been employed in many applications across various domains due to its impressive ability to extract features effectively [5]. In the field of RF fingerprinting, previous studies have showcased the potential of deep learning in automatically extracting RF fingerprints from raw signal data, thereby reducing the need for manual extraction based on expert domain knowledge.

Moreover, due to the maturity of NFC card manufacturing, the distortion in signal is extremely small, which leads to a powerful Software Defined Radio (SDR) device being required. However, the cost of the overall system is relatively high, impeding the practical application of RF fingerprinting for NFC. However, with the help of deep learning techniques, it is possible to extract the PHY features hidden in high-dimension space even without expensive signal acquisition devices.

NFC Technology Background

As an international standard for contactless smart cards, ISO/IEC 14443 discusses the contactless interaction mechanisms and related protocols for NFC readers and cards. The two communication signal interfaces, type A and type B, are specified in 14443-2. NFC-A is a widely used type of NFC and is compatible with the 14443 type A standard. Therefore, the proposed method in this project is focused on NFC-A and is compatible with the ISO/IEC 14443A standard. ISO/IEC 14443-3 describes the whole communication process between NFC cards and readers [11]. The process is shown in Fig. 2 below.

When the NFC card approaches the operating RF field of the reader, the reader initiates the process of establishing communication with the card. The communication flow includes three steps: request, anti-collision loop, and select acknowledge. To detect whether a card enters the reader energy field, the reader continuously sends the request command, type A (REQA), until it receives answers to request type A (ATQA) from at least one NFC card. During this step, the reader has to wait for the card to harvest enough energy from the high-power carrier

wave (CW) signal. If any complex cryptographic algorithm is employed, the time for energy harvesting will increase. After harvesting enough energy from the reader, the card then shifts from the IDLE state to the READY state.

Next, if the card successfully passes the anticollision process, it enters the ACTIVE state. In the ACTIVE state, the card starts waiting for messages from the higher layer that belong to specific applications. It's important to note that for type A NFC cards, every transaction starts with the REQA and ATQA signals, and the ATQA response is identical for every type A NFC card (0x04:00), regardless of the card UID. Thus, the ATQA signal can be utilized as a generic data segment for extracting RF fingerprints. Moreover, as shown in Fig. 2, the ATQA signal precedes the exchange of any sensitive data, implying that this system can successfully protect user privacy.

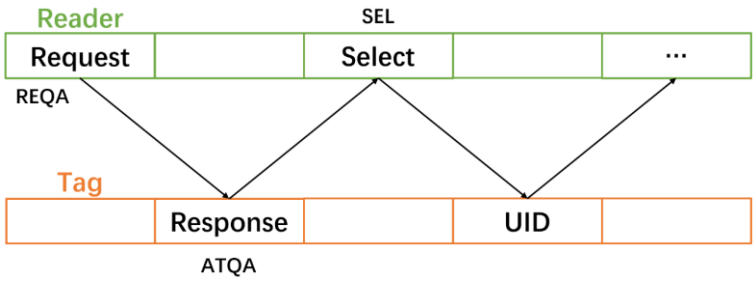


Fig. 2 Communication Process Between Card and Reader

In the ISO-14443A protocol, the carrier frequency transmitted by the reader to the card is 13.56 MHz, with a common bit rate of 106 kbps [16]. The interrogation signal is modulated using 100% amplitude shift keying (ASK) with load modulation, and the subcarrier signal from card to reader is 848 kHz. Both the REQA and ATQA signals are transmitted pairwise. For data transmission, a modified Miller coding scheme is used from the reader to the card, and Manchester coding is used from the card to the reader.

5 Design & Methods

Testbed Setup

To prepare enough labeled data for the supervised learning, we built an SDR-based testbed for data acquisition and collected ATQA signal segments of the MIFARE Classic 1K NFC card at the physical layer. The signal acquisition testbed includes a low-end SDR device, a sniffing antenna, an NFC reader, several common NFC cards, and a computer as shown in Fig. 3.

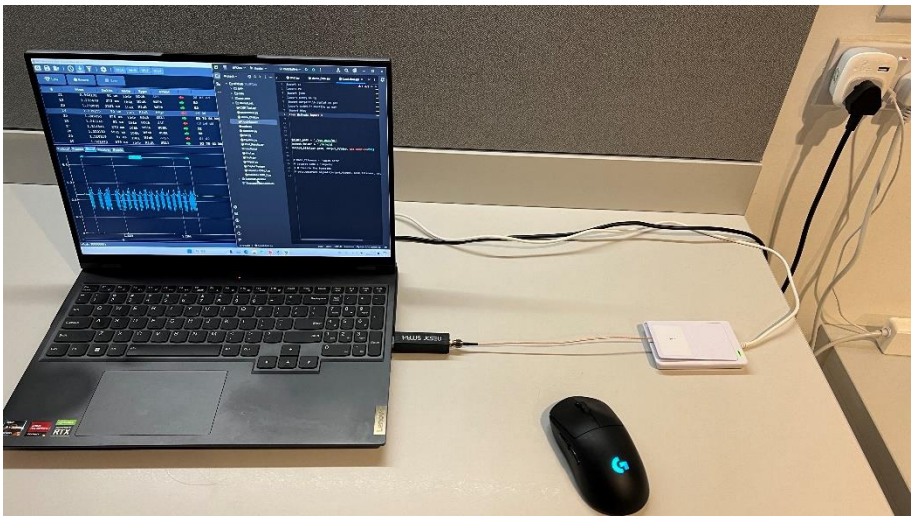


Fig. 3 Signal Acquisition Testbed Setup

SDR is a radio communication system that utilizes software to perform tasks that traditionally require specific hardware. In this project, an SDR device is utilized to capture the signal and transmit processed samples of the baseband signal to the PC. To avoid implementing specific modifications to the normal communication process, this project employed an open-source project available on GitHub called *NFC Laboratory* [17].

NFC Laboratory is developed for the Windows system, providing a user-friendly interface suitable for practical applications. It offers a Windows binary version, enabling the whole system to be functional after the plugin. The creator designed the signal acquisition and demodulation algorithm for RTL-SDR and AirSpy Mini or R2. For the cheapest one, RTL-SDR, it works by sampling at the second harmonic of the carrier frequency, 27.12MHz. In this project, RTL-SDR is adopted as the SDR hardware platform, supporting up to a 3.2 MSPS sampling rate, with a 2.4 MSPS sample rate being utilized [17]. As mentioned in the previous section, the highest frequency component of the baseband signal down-converted by the SDR is lower than half the sample rate of 1.2MHz. Therefore, this sample rate is acceptable for this project.

A sniffing antenna is employed to capture the transmitted NFC signals between the reader and the NFC cards. This antenna is connected to the SDR device via a wire. Fig. 4 displays the measurement setup for the acquisition of NFC signals. Additionally, to avoid the inconsistent

positioning of the sniffing antenna affecting the experiment about tag position change, the sniffing antenna is attached to the reader.

The NFC reader utilized in the experiments is a common NFC card reader, ACR122U, which supports ISO 14443 and ISO 18092 standards [18]. The reader is used to send request signals like the common NFC reader on the vendor machine, which initiates the communication process between the reader and the card.

In our experiments, eight NFC cards are utilized as legitimate devices, which means these devices are previously registered in the system. Additionally, four NFC cards serve as rogue devices for system evaluation, each possessing the same UID as one of the legitimate devices, achieved with the smart card application mentioned earlier. All twelve devices are from the same batch and manufacturer, featuring YWBL-WH MIFARE Classic 1K chips. They share identical ATQA content (04:00) and adhere to the communication protocol ISO/IEC 14443A, with the RF interface operating at 13.56MHz.

Fig. 4 Signal Acquisition Testbed Demonstration

After the samples are collected and stored in the buffer of the SDR device, the computer receives the data from the buffer and stores them in the file format proposed by the NFC Laboratory. Further processes, such as signal data augmentation, neural network training, and system testing, are both performed on the PC.

The components involved in this project are listed in the table below:

Table 1 Experiment Components List

Reader	ACR122U USB NFC Reader
Antenna	MIKORE Regular NFC 40x30 Antenna
Cards	MIFARE Classic 1K RFID Smart Cards
Cable	UFL to SMA Male Bulkhead, RG178, 250mm -
SDR	Nooelec RTL-SDR v5 SDR

Dataset Creation

The output files contain a .wav file containing the magnitudes of collected IQ samples and a .js file containing the decoded information. Therefore, we designed a simple Python script to automatically find the files and extract the ATQA segments, which are located by the dictionary stored in .js file. Fig. 5 illustrates one of the collected samples loaded by the Python script.

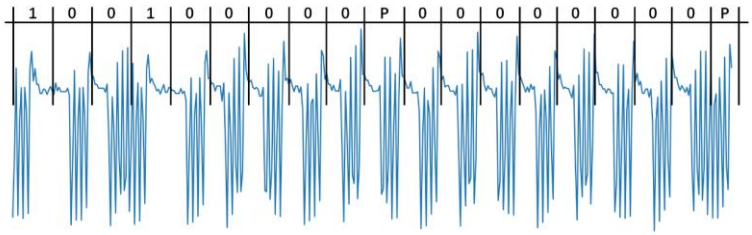


Fig. 5 Example of Signal Magnitude of ATQA signal

In this project, the proposed system should be capable of both detecting the rogue device intrusion and discriminating between the eight legitimate devices. Additionally, we also want to address the stability problem of RF fingerprints. Therefore, the datasets should contain both the legitimate and rogue cards under different environments and card positions. According to [17], the sample rate adopted is 2.4MSPS. As specified in [16], the time period of each bit of data is $9.4\ \mu s$. Equation (1) gives the estimation of the number of sampling points of the full ATQA segment:

$$N = \frac{N_b}{\frac{f_c}{128}} \cdot f_s$$

Where the N is the number of sampling points, N_b is the number of bits in ATQA command, 19 bits in this case (16 binary bits, 1 starting bit, and 2 parity bits), f_c is the carrier frequency, which is 13.56MHz, and f_s is the sample rate adopted. The resultant N is rounded to 431, given $N_b = 19, f_c = 13.56MHz, f_s = 2.4MSPS$. We have decided to set each data sample to contain 440 sampling points, padding the final sampling point's value as needed for alignment.

Dataset Description:

- Dataset volume: 13,200 (12,800 samples for 8 legitimate devices / 400 samples for 4 rogue devices);
- Sample rate: 2.4M samples/s;
- Data support for each class: 1800 samples with 440 sampling points each;
- Content of data: ATQA segments (0x04:00 in Manchester encoding);
- Total size of raw data: 25.6G Bytes;
- Dataset Setup: Dataset 1-4 position 1 / Dataset 5 position 2 / Dataset 6 position 3 / Dataset 7-9 random positions (Shown in Fig. 6)

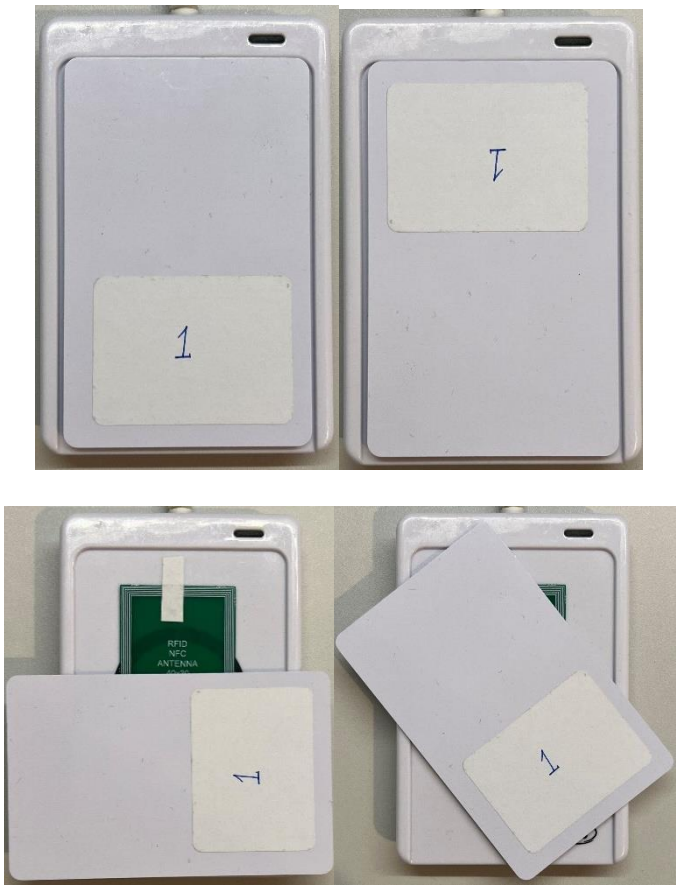


Fig. 6 Tag Positions (a) Position 1 (b) Position 2 (c) Position 3 (d) Example of Random Positions

Design for Generalization Ability

Although many data augmentation methods have been proposed to increase network generalization ability, this project is designed without data augmentation due to the practical application scenario of NFC technology. First, several data augmentation methods are introduced.

For instance, [3] processed the signal segment by dividing the root-mean-square (RMS) value, which depends on the signal transmitter, such as the NFC reader, rather than on inherent hardware impairments. This approach aims to mitigate the effect of position changes, which cause variations in the power harvested by the tag. However, experimental results show that this method is not applicable in this project.

Another method was proposed by Wang et al. in [14]. They introduced the stochastic permutation method, which randomly swaps signal segments since device-specific features should be mutually independent among the signal segments. However, this method introduces new variables into the system. Differences in the swapping process between different devices might be captured by the network.

Considering the real-world application scenario of this project, a simple NFC card slot is designed. The 3D modeling of the slot is shown in Fig. 7 below:

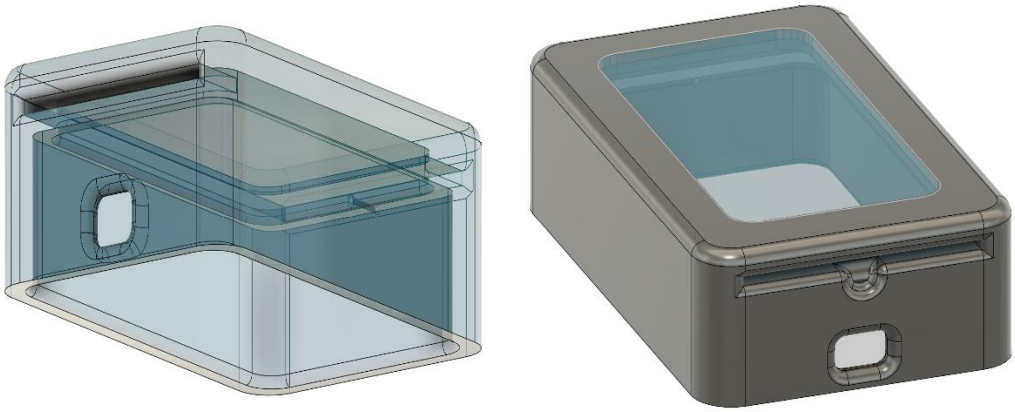


Fig. 7 NFC Card Slot

Only high-security required places might adopt this system, which indicates that the number of legitimate devices might be limited and the security check process might be strict. Therefore, users can insert their cards into the slot and proceed with the following security check instead of simply scanning the card on the reader. With this card slot, consistent positioning of the cards is achieved.

Thus, as long as inconsistent positioning of the NFC cards is the only or major factor contributing to the instability of RF fingerprint, this system remains stable and practical.

Deep Learning Network Structures

Previous research [2, 4, 5] has shown that convolutional neural networks (CNNs) outperform other network structures in extracting device-specific features. Due to the maturity of NFC card manufacturing, it is challenging to identify unique signal features resulting from hardware imperfections. However, the deeper network structures provided by CNNs have the potential to uncover inherent features hidden in higher-dimensional space. In this project, common deep learning networks are regarded as a combination of a feature extractor and a feature classifier. Typical networks comprise several convolutional layers, such as conv1d or conv2d in the PyTorch framework, along with a fully connected layer, which categorizes the features into their respective classes. Therefore, each network structure has two different sub-structures, with the FC layer and without FC layer. The with fc layer sub-structure is designed for training with classification loss function, typically Cross-Entropy loss, and the without fc layer sub-structure is designed for metric learning with triplet loss function or contrastive loss, which focuses on the distance between the generated feature samples within class or cross classes.

The first network structure is a modified version of the network employed in [5]. The input for the network is a single-channel signal magnitude, which means conv1d is adopted in this network. The CNN structure contains three convolutional layers (conv1d) followed by one fully connected layer, which expands the feature space from 128 to 512. The first convolutional

layer has 32 kernels with size of 5, followed by the second convolutional layer having 64 kernels with size of 3, and the third convolutional layer containing 128 kernels with size of 3. The FC layer contained 512 hidden neurons. Each convolutional layer is followed by a batch normalization layer and ReLU activation function. To evaluate the performance of this network structure in the metric learning training method, an additional fully connected layer with 8 hidden neurons is used to output the classification prediction corresponding to 8 legitimate devices. The proposed architecture is shown in Fig. 8.

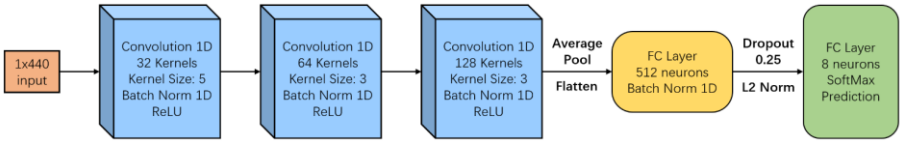


Fig. 8 Network Structure 1

Although the experiment results showed that the first network structure can yield good recognition accuracy on the test dataset, we proposed a modified network structure based on the network structure employed in [3]. The network structure is shown in Fig. 8 below.

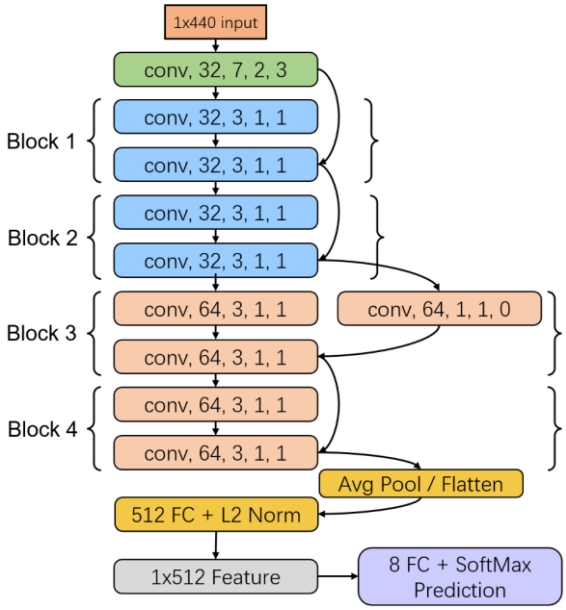


Fig. 9 Network Structure 2

As illustrated by Fig. 9, one convolutional block contains two 1D convolutional layers. Each convolutional layer is followed by a 1D batch normalization layer and ReLU activation function. For example, “conv, 32, 7, 2, 3” means this layer is a 1D convolutional layer with 32 output channels, kernel size of 7, stride of 2, and padding of 3. One special convolutional layer “conv, 64, 1, 1, 0” is added to extend the output of Block 2 with the channel of 32 to the channel of 64, which makes the output of Block 2 able to be added to the output of Block 3.

In this structure, the residual shortcut is adopted for two main reasons. The first reason is to make the training process for deeper networks easier. Based on the backpropagation algorithm, a deeper network structure might encounter the problem of vanishing gradients

due to the chain rule used to calculate the gradient. The chain rule takes the form of a continuous multiplication, so with the number of layers increasing, gradients will propagate in exponential form. The issues of vanishing gradients and exploding gradients typically become more obvious with an increase in the depth of the network, which will significantly make the training harder. When updating the weights of deep networks based on the calculated loss function through gradient backpropagation, the obtained gradient values approach zero or become particularly large, resulting in gradient vanishing or exploding.

The first reason is to prevent the degradation. As illustrated in Fig. 10 below, the blue region shows the basic network structure (few layers), while the green region shows the new network structure (More layers), and the orange region shows the final network structure (Most layers).

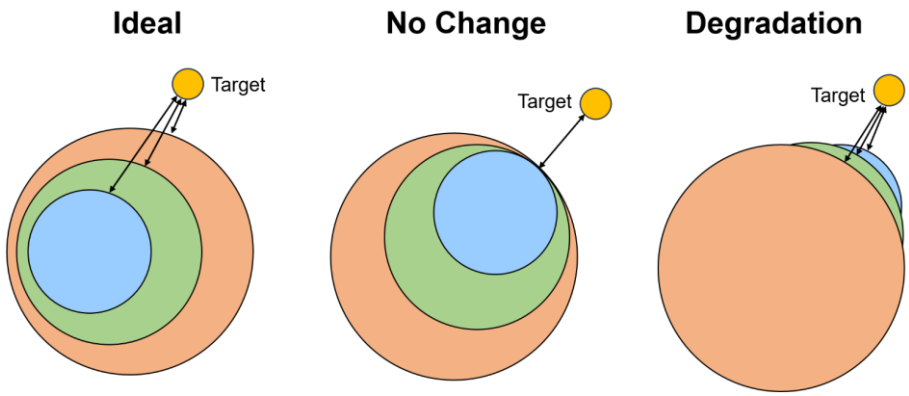


Fig. 10 Network Degradation

In the ideal case, adding more layers to create a deeper network structure, should lead to an improvement in the "distance" between the trained network and the target. However, in the other two cases, as the number of layers increases, this "distance" actually becomes larger, potentially resulting in poorer performance. To prevent this degradation and ensure that at most only a "no change" scenario occurs, a residual shortcut is proposed in [20], where the input of one layer is added to the output of this layer to create the input for the subsequent layer. By implementing this technique, even deeper networks can perform at least as well as the basic structure.

Loss Functions

The table below shows the loss functions employed in this project.

Table 4: Formula of Loss functions

Cross Entropy Loss	$l_{CE}(x_i) = -\log\left(\frac{exp(w_{y_i}^T \cdot x_i)}{\sum_j^C exp(w_j^T \cdot x_i)}\right)$
Triplet Loss	$l_{triplet}(x_i, x_j, x_k) = \left[\delta + x_i^T x_k - x_i^T x_j\right]_+$
SoftTriple Loss [21]	$l_{SoftTriple}(x_i) = -\log\left(\frac{\exp\left(\lambda(S'_{i,y_i} - \delta)\right)}{\exp\left(\lambda(S'_{i,y_i} - \delta)\right) + \sum_{j \neq y_i} \exp(\lambda S'_{i,j})}\right)$

As mentioned earlier, each structure has two versions: one includes a fully connected layer to generate class predictions, while the other only outputs the extracted features or embeddings from the input data. This difference arises from the type of loss function employed during the training process. For instance, if the network is trained using classification loss functions like Cross Entropy loss, it is expected to output logits, representing confidence scores for all classes. On the other hand, when trained with metric learning loss functions such as Triplet loss, the network should only output embeddings for each input data, which are high-dimensional combinations of features extracted from the input.

Since the following anomaly detection function depends on the differences in distances of extracted embeddings between samples from the same device and samples from different devices, the ability of the trained network to cluster the embeddings is important in this project. Therefore, the loss functions applied during the training process make a great difference due to different convergence conditions. In this project, three different loss functions are employed and tested, namely, Cross Entropy (CE) loss, Triplet loss, and SoftTriple loss [21].

For Cross Entropy loss, although this loss function is not designed for feature extraction or so-called metric learning, where the final goal of training is to maximize and minimize the distance between the embeddings from different classes, the trained network can be assessed by removing the last fully-connected layer and directly outputting the embeddings. The loss function can be calculated as follows:

$$l_{CE}(x_i) = -\log\left(\frac{exp(w_{y_i}^T \cdot x_i)}{\sum_j^C exp(w_j^T \cdot x_i)}\right)$$

Where x_i denotes the embedding of i-th data input and the true label is y_i. C denotes the number of classes and $w_1 \ldots w_C$ are the weights of the last fully-connected layer.

Triplet loss is a loss function specifically designed for metric learning, primarily used to train

samples with small differences, making it suitable for RF fingerprint extraction from NFC signals. As demonstrated previously, hardware impairment is not distinctive among NFC devices. Therefore, the difference between ATQA signal segments is not significant. Additionally, since rogue device detection relies on the distances between samples from unknown devices and samples from legitimate devices, the strong clustering feature of Triplet loss is considered advantageous.

The input comprises a triplet, including Anchor (A) examples, Positive (P) examples, and Negative (N) examples. Here, P represents a sample of the same class as A, while N represents a sample of a different class than A. By optimizing the distance between the embedding of A and the embedding of P to be smaller than the distance between the embedding of A and the embedding of N, the embeddings from the same class are drawn closer while being pushed away from embeddings from different classes. Moreover, a constant, margin, is introduced to determine the convergence condition for this loss function.

The samples can be categorized into three types: Easy Triplets, Hard Triplets, and Semi-Hard Triplets.

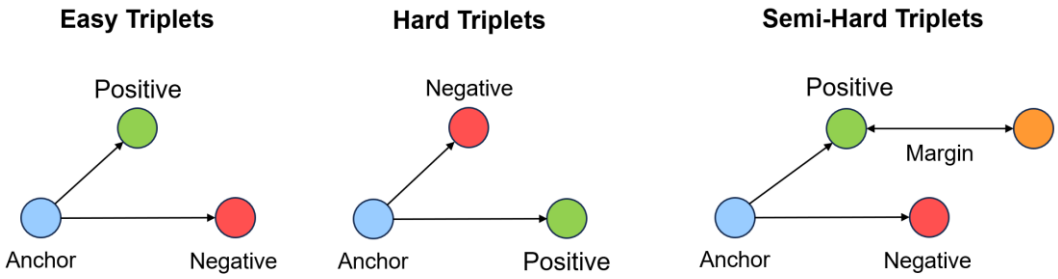


Fig. 11 Triplets Categories

As shown in Fig. 11 above, the first case Easy Triplets does not require any improvement, but the following two categories require optimization.

To prevent the model from taking shortcuts and outputting negative and positive embeddings that are very similar, a constant margin is introduced. Without a margin, the Triplet loss formula becomes:

$$l_{triplet}(x_i, x_j, x_k) = \left[x_i^T x_k - x_i^T x_j\right]_+$$

Where x_i stands for the anchor embedding, while x_k, x_j stands for positive and negative embeddings. Then as long as the network outputs $x_i^T x_k = x_i^T x_j$, the convergent condition is achieved. Therefore, by setting a constant margin, the network is forced to learn effectively. Due to the existence of the margin, Triplet loss introduces an additional hyperparameter, and the size of the margin needs to be carefully adjusted. If the margin is too large, the loss of the model will be very high, making it challenging for the loss to approach 0 at the end of the learning process, and it might even cause the network to fail to converge. However, larger margins enable the model to more confidently distinguish relatively similar samples; that is, it's easier to distinguish between A and N. Conversely, if the margin is too small, the loss can easily approach 0, making the model easier to train, but it becomes difficult to distinguish A

from N.

However, according to [22], a major drawback of the Triplet loss is that with the growing size of the dataset, the possible number of triplets increases cubically, resulting in difficulty in the training process. Moreover, the model quickly learns to correctly fit most easy triplets, making a significant number of all triplets uninformative. Thus, mining hard triplets becomes important for learning. Taking a vivid example, repeatedly telling one that individuals with clothes in different colours are different persons does not convey meaningful information, whereas being taught with similarly-looking but distinct individuals (hard negatives), or images of the same person in extremely different poses (hard positives), greatly aids in grasping the concept of "same person". However, only focusing on the hardest triplets, which contain the most outliers, is harmful for the network to learn the normal associations. Therefore, it is common practice to mine only moderate negatives or moderate positives. BatchHard mining strategy is proposed to effectively mine the informative triplets. For each sample in the batch, the hardest positive and hardest negative samples are selected within the batch when forming triplets for computing the loss.

The last kind of loss function employed is SoftTriple loss, which is proposed in [21] to combine the Cross Entropy loss, or called SoftMax loss, and Triplet loss. According to [21], the metric learning algorithm suffers from the constraints of the limited number of samples within one batch. For example, selecting the hardest negatives can in practice result in fitting in local minimum or outliers at the early stage of the training [22]. Therefore, an effective sampling strategy is essential to select informative triplets. Moreover, recent research also provided proxy-based loss for deep metric learning, which simply means by selecting the proxies or centres of each class, the influence of noise or outliers can be alleviated.

As mentioned before, the RF fingerprint is suffering from its instability caused by position changes or environmental changes. Lee et al. [4] provided a simple solution to increase the generalization ability by diversifying the training dataset. Therefore, this practice is actually bringing more centres or proxies for each class, which means the embeddings of samples from the same class but collected from different environments will ultimately converge to different proxies. By setting proxies for each class, the convergence might be improved and the performance of the network trained on a diverse dataset will be enhanced.

According to [21], optimizing CE loss with L2 normalization, which normalize the magnitude of embeddings to unit length, is equivalent to optimizing the triplet constraints consisting of an original example and two centers. Thus, the assumption that the model trained by CE loss can also be assessed in clustering task is verified. Moreover, SoftTriple loss assumes that each class has *K* centers, which might be caused by environmental changes. Then, since the embeddings are normalized the dot products become equivalent to cosine similarity. The hard similarity between sample x_i and the true label c can be defined as:

$$S_{i,c} = \max_k x_i^T w_c^k$$

Due to the similar drawbacks of *BatchHard* online mining strategy, the author introduced an alternative operator to replace the max operator in the equation above. The relaxed similarity is given as:

$$S'_{i,c} = \sum_k \frac{\exp\left(\frac{1}{\gamma} x_i^T w_c^k\right)}{\sum_k \exp\left(\frac{1}{\gamma} x_i^T w_c^k\right)} \cdot x_i^T w_c^k$$

This relaxed similarity gives a better convergent condition for samples. As shown in Fig. 12, the weights of different centers are both considered in the expression, therefore, achieving better performance.

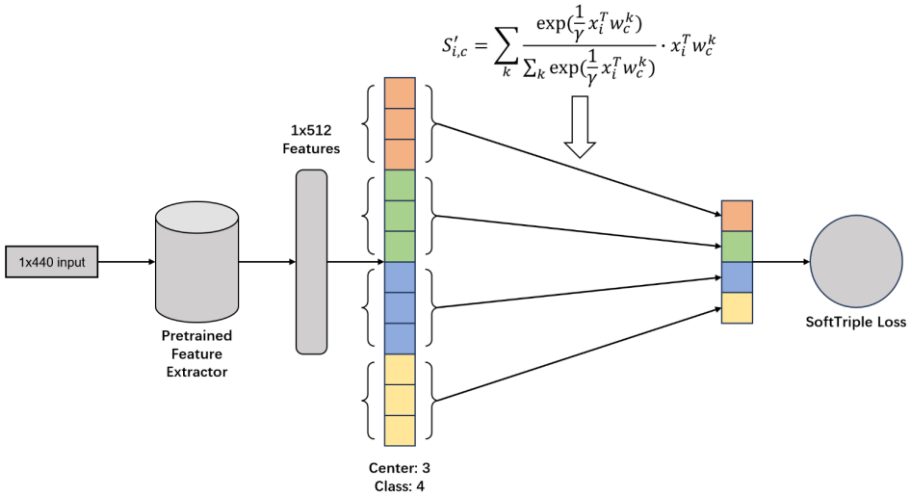


Fig. 12 SoftTriple Loss

And the final expression is given as:

$$l_{SoftTriple}(x_i) = -\log\left(\frac{\exp\left(\lambda\left(S'_{i,y_i} - \delta\right)\right)}{\exp\left(\lambda\left(S'_{i,y_i} - \delta\right)\right) + \sum_{j \neq y_i} \exp\left(\lambda S'_{i,j}\right)}\right)$$

Where λ is the scale factor corresponding to the L2 normalization and δ is the *margin* defined for triplets.

Rogue Device Detection

Rogue device detection can be divided into four phases: UID verification, Threshold Generation, Embedding Extraction, and Samples Authentication. The unknown signal packets should be passed through these four phases sequentially.

According to [2], due to the limitation of SDR devices, such as 8-bit ADC, the embedding space is constrained. Assuming the TRA is the only device-specific and stable feature the network can learn; the extensive experiments show that the TRA only varies within 0.05 [2]. Therefore, the single frequency, 13.56MHz, TRA can potentially differentiate only $0.05 \cdot 2^8 \approx 13$ tags. Thus, without information on other frequencies, it is hard to extend the fingerprint

scalability only based on deep learning techniques. For instance, increasing the complexity of the network structure can extend the embedding space by separating the embeddings from different classes more. However, this practice will definitely impend the practical application due to the computation cost and time consumption. However, the UID can effectively distinguish NFC devices at the first stage and then pass to the embedding extraction phase. By filtering out the devices with UIDs that are not registered in the system in advance, the system can effectively address the problem of scalability. The "feature space" of the UID technique is about $16^8 = 4.3 \times 10^{10}$ considering each bit is a hexadecimal bit. Therefore, since the *NFC Laboratory* provides the decoded UID and ATQA signal segment at the same time, UID verification is still implemented.

Next, the system will generate the threshold dictionary used to determine whether the unknown packets come from a genuine device or a forged device. This stage only needs to be done once, every time a new device is added to the legitimate devices because the generated dictionary can be saved on the PC to save time for computation next time. The purpose of this stage is to set more precise thresholds for each legitimate device. Since the network cannot perform equally on every class even with the help of a designed sampler, setting an overall threshold to determine whether the unknown packets belong to a genuine device is unreasonable. In our experiments, the distances between the embeddings from the devices vary from class to class. For example, the distances between embeddings from device1 might be centered at 0.1, while those from device2 might be centered at 0.2. At the same time, the distances between embeddings from device1 and embeddings from other devices might be centered at 0.5 and those from device2 might be centered at 1.0. Although the separation performance of the network on device1 and device2 are identical since 0.5/0.1 equals 1.0/0.2, setting a threshold at 0.3 is stricter for device1 and looser for device2. When encountered with extreme variation between devices, the overall threshold might result in a 100% True Negative Rate (TNR) for packets that might belong to one device and 0% TNR for another device. Thus, this project proposed a new method called adaptive threshold, which generated a threshold dictionary containing all the device-specific thresholds based on all the samples of legitimate devices. During this process, the distances between the samples in the training dataset will be calculated and the threshold is given as:

$$Threshold = median + \alpha \cdot \sigma$$

Where *median* is the median value of the distances between the samples from the same class, while α is an adjustable parameter to control the strictness of authentication and σ is the standard deviation of the distances.

Then, then the unknown packets will be fed into the pretrained network to generate embeddings, which will be used to find which device is the most possible device within the legitimate device. By using a *k* nearest neighbourhood (KNN) classifier fitted on the legitimate dataset (training dataset). This process does not take much time since KNN is a kind of instance-based supervised learning algorithm that does not require training itself and will not obtain a model that summarizes the data characteristics. It only needs to select the appropriate parameter *k* for initiation.

Then the system enters the last phase. In practice, the system will assume the unknown packets belong to the same device with the nearest *k* embeddings. For example, if k=5, the surrounding embeddings of the unknown embedding consist of 4 embeddings from device1 and 1 embedding from device2, the system will assume the unknown embedding belongs to device1. Then, the device-specific threshold will be selected based on the KNN's prediction from the threshold dictionary. If the distance between the unknown packet and one of the surrounding embeddings from the predicted legitimate device larger than the selected threshold, the unknown packet will be identified as an anomaly or rogue device. Otherwise, the system will output the label predicted by KNN as the final prediction of the unknown packets.

The whole process is shown in Fig. 13 below:

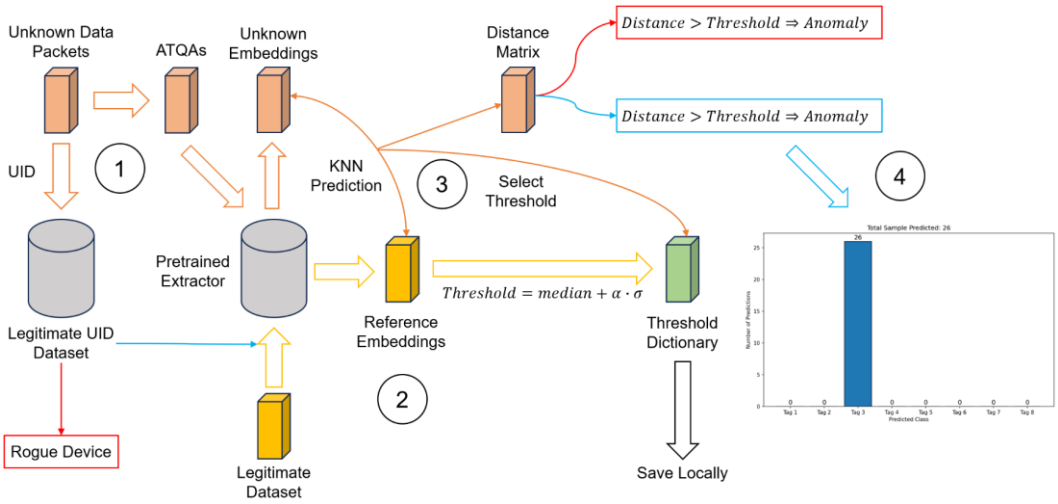


Fig. 13 Rogue Device Detection

There is one potential limit of the adjustable parameter α. A small value will definitely bring low FPR, but at the same time it will bring a high false negative rate (FNR), which means there is a high possibility that many packets from legitimate devices are also identified as "anomalies". Therefore, considering the practical application scenario, small α is recommended since the system will compare the number of "legitimate" packets and "anomaly" packets and if the number of "legitimate" packets is larger, the system will still identify the device as legitimate regardless of the number of packets identified as "anomaly". This practice sacrifices the FNR while maintaining the correct final decision and high FPR.

6 Results

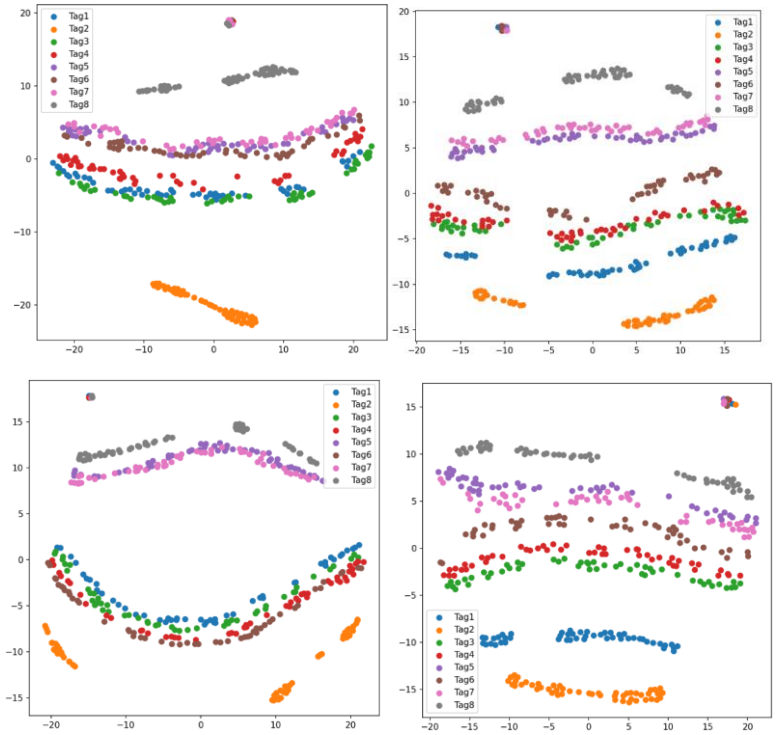
Environmental Changes & Data Augmentation

This section will demonstrate the distribution of data under different environmental changes and data augmentation methods. The datasets are designed as follows:

Dataset 1: Position 1 / Room Temperature 1 / Location 1
Dataset 2: Position 1 / Room Temperature 2 / Location 2
Dataset 3: Position 1 / Room Temperature 3 / Location 3
Dataset 4: Position 1 / Room Temperature 4 / Location 1
Dataset 5: Position 2 / Room Temperature 5 / Location 1
Dataset 6: Position 3 / Room Temperature 5 / Location 1
Dataset 7: Random Positions / Room Temperature 5 / Location 1
Dataset 8: Random Positions / Room Temperature 5 / Location 1

Position 1, 2, 3 are shown in the previous sections. The positions of different tags in this dataset are consistent. “Random positions” means the tags were randomly put on the reader. The positions of different tags in this dataset are inconsistent. “Location 1,2,3” means the datasets are collected under different electromagnetic environments.

The visualization is achieved by t-distributed stochastic neighbor embedding (TSNE), a statistical method to visualize the high-dimensional data. Firstly, the raw data distribution patterns are shown:



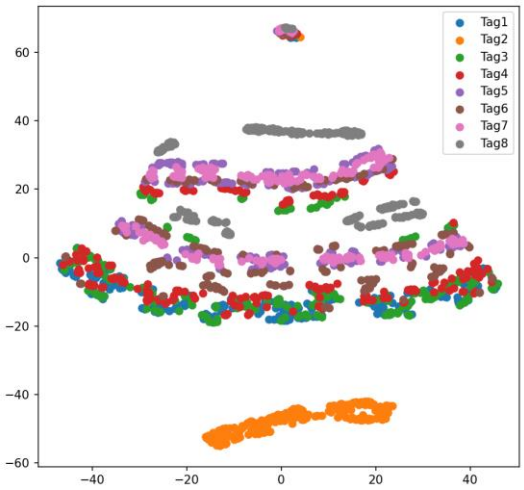
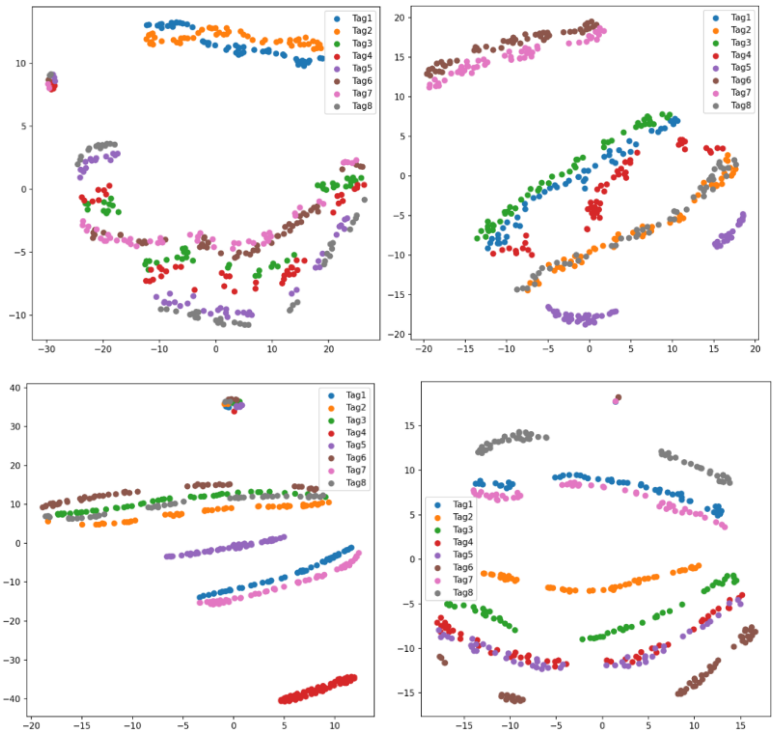


Fig. 14 (a-d) Raw Data Distribution Patterns for dataset 1-4 (e) Total Pattern

As shown above, the raw data distribution patterns are relatively consistent under different room temperatures and electromagnetic environments. It is obvious in the last figure that the samples from Tag2 are distinguishable and well-clustered even without a pretrained feature extractor. The detailed reason for this phenomenon will be introduced in the *Discussion* section. Then, the distribution patterns of dataset 5-8 are shown below.



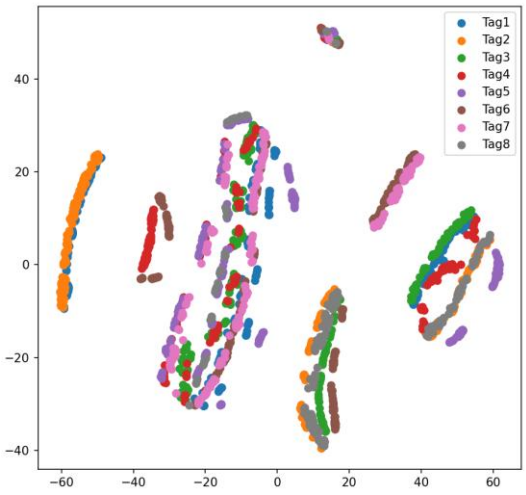
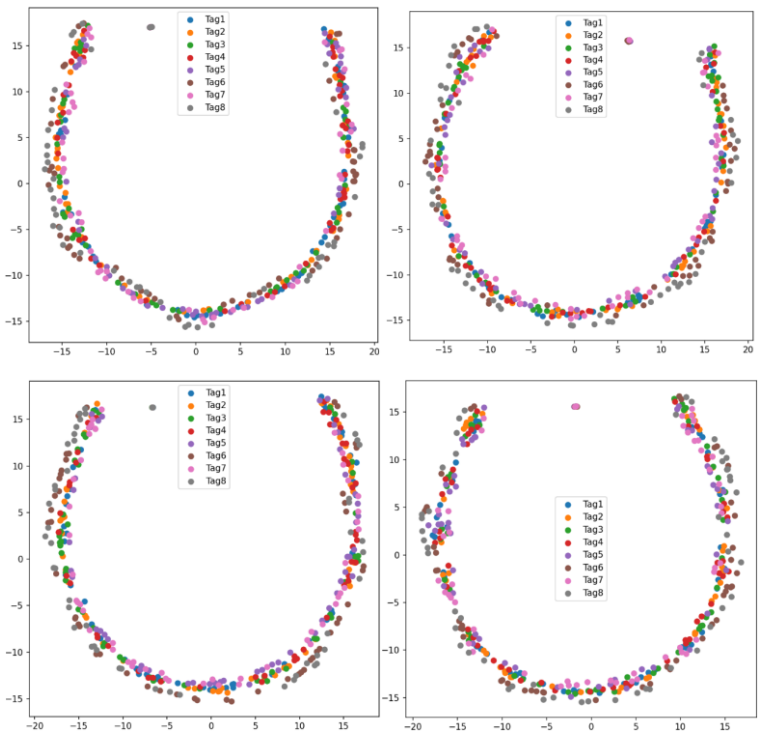


Fig. 15 (a-d) Raw Data Distribution Patterns for dataset 5-8 (e) Total Pattern

It is transparent that the samples from Tag2, which is consistent and well grouped in dataset 1-4, are distributed at random positions. And the other samples are also inconsistent. Therefore, it is predicable that the network trained on dataset 1-4 cannot perform well on dataset 5-8.

Then, the following results show the effect of normalizing the magnitude of the signal to 0 to 1, which is adopted in [4,5]. To correctly refer each method, this type of normalization is called Method 1.



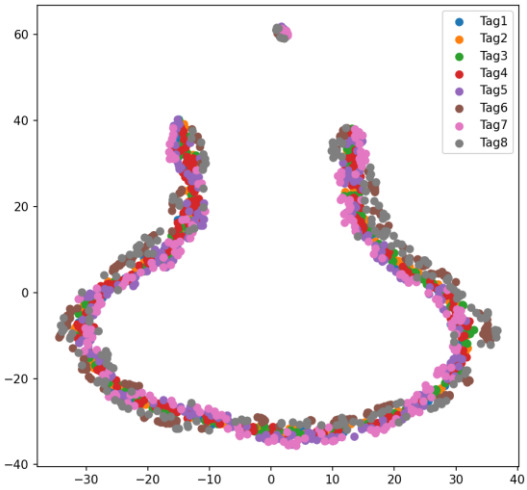
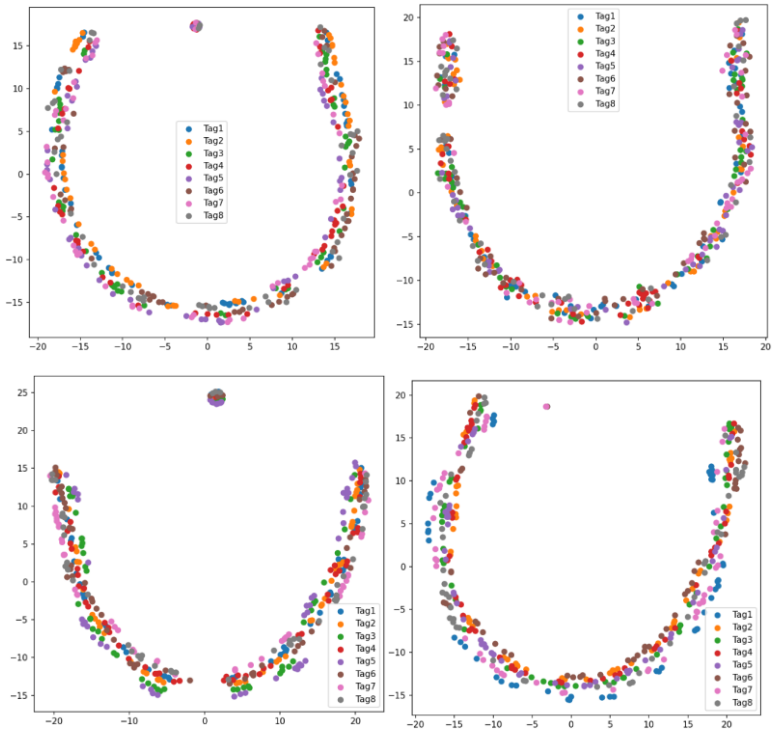


Fig. 16 (a-d) Method 1 Data Distribution Patterns for dataset 1-4 (e) Total Pattern

It is obvious that this normalization eliminates some device-specific features. Assuming it is the device-specific features that keep the samples distinguishable from each other when no normalization method applied, the samples become less distinguishable by applying this normalization method because this method eliminates some information about the absolute amplitude of the signal. The deeper reason is stated in *Discussion* section. Moreover, this method fails to mitigate the effect of inconsistent positioning as shown below.



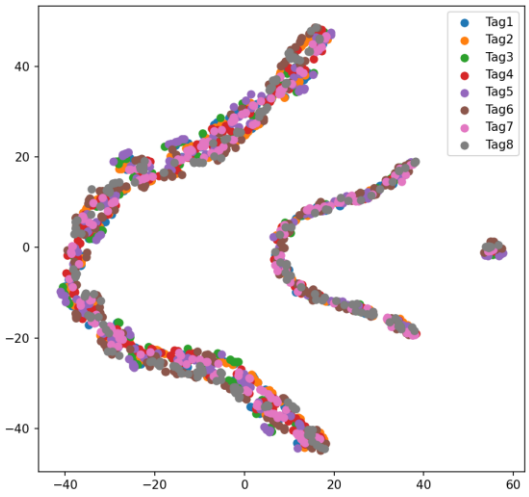


Fig. 17 (a-d) Method 1 Data Distribution Patterns for dataset 5-8 (e) Total Pattern

Considering this normalization not only eliminates some valuable (device-specific) features but also fails to mitigate the impacts of inconsistent positioning, this method is not employed. The next normalization is dividing the signal by this RMS value employed in [3]. Since this method also fails to improve the generalization ability the results are briefly shown below (this method is called Method 3):

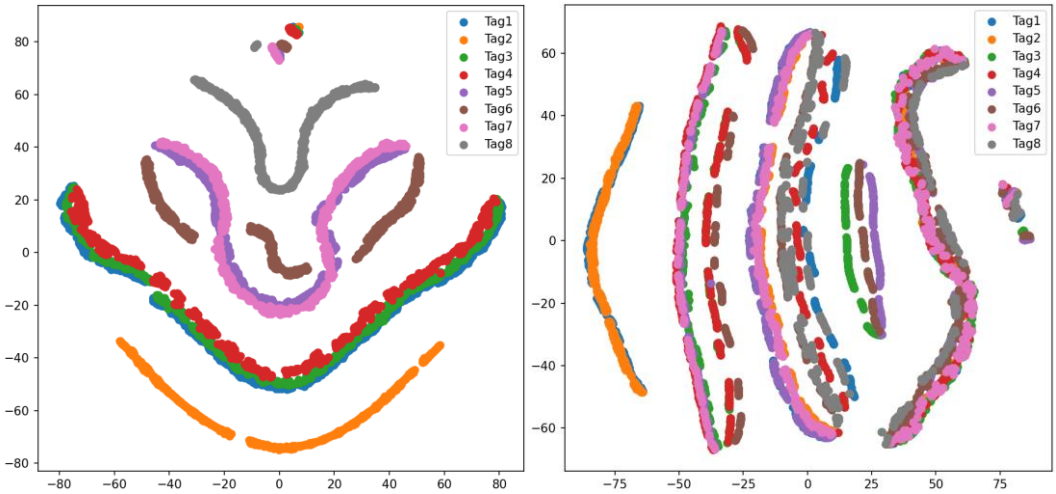


Fig. 18 Method 3 Data Distribution (a) Dataset 1-4 (b) Dataset 5-8

The last data augmentation method is proposed in [24], calculating the variance fractal dimension trajectory (VFDT), which is similar to the short-time Fourier transform. By sliding a window along the signal to calculate a series of data representing the change of the signal magnitude in the time domain. Since the signal is discrete, the length of the windowed segment, Δw, is used as the time interval Δt, and the VFDT of a given windowed segment i of size Δw can be estimated as

$$D(i) = 2 - log[var(\Delta x)]/(2log(\Delta w))$$

The effects of this augmentation are shown below:

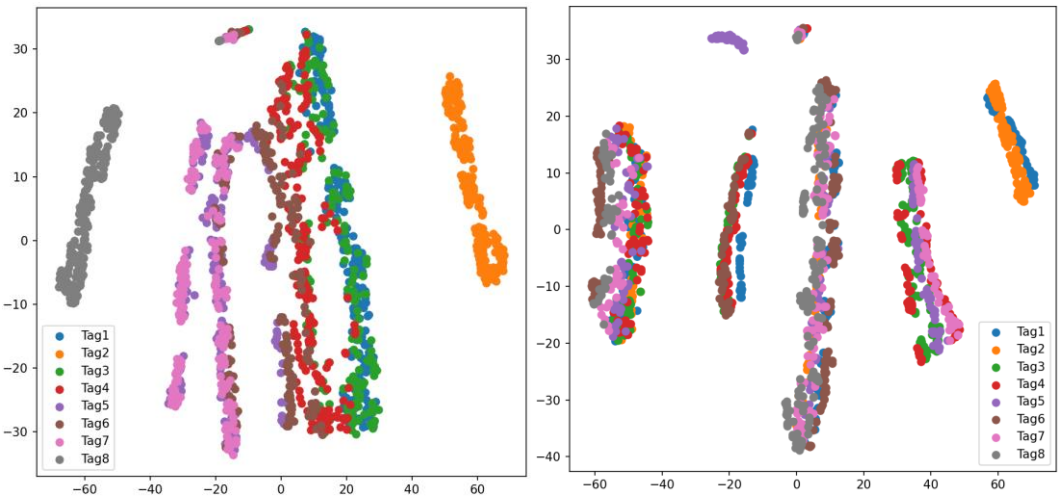


Fig. 19 VFDT Data Distribution (a) Dataset 1-4 (b) Dataset 5-8

The problem of this method is similar to Method 3. Sliding a window along the signal loses some useful information about the details and results in poorer performance.

Finally, as demonstrated in the Methodology section, the card slot is proposed to eliminate the effect brought by inconsistent positioning, and the following results will also show that inconsistent positioning is the major factor contributing to the instability of RF fingerprint. However, Method 3 (RMS value) is still adopted since the samples are better clustered in Fig. 18.

Network Performances

As mentioned before, the problem of inconsistent positioning is eliminated by applying a designed card slot. Therefore, the following results are focused on the performance of the model trained on datasets with the same tag position. In this section, all the results are obtained after achieving the same degree of convergence. The hyper-parameters and other information are shown in the table below:

Table 5 Hyper-Parameters

Batch Size	256
Learning Rate (LR)	1×10^{-4}
Decay Rate	0.987
Epochs	200
Patience	20
Margin	0.2
Optimizer	Adam
LR Scheduler	Exponential
Parameters for SoftTriple	$\lambda = 0.5; \gamma = 0.1; \tau = 0.2; K = 3$

The results are listed below, the accuracy results are obtained on the test dataset, where the

data is inaccessible during the training process.

Table 6 Test Accuracy Results

Structure & Loss	Accuracy on Dataset 1-4	Accuracy on Dataset 5-8
Model 1 + CE	96.563%	36.125%
Model 2 + CE	97.344%	35.859%
Model 1 + Triplet	97.513%	36.422%
Model 2 + Triplet	97.656%	36.656%
Model 1 + SoftTriple	98.344%	36.125%
Model 2 + SoftTriple	**98.438%**	**37.188%**

The best performance is achieved with network structure 2 and SoftTriple loss function. The detailed accuracy is shown in the confusion matrix in Fig. 20:

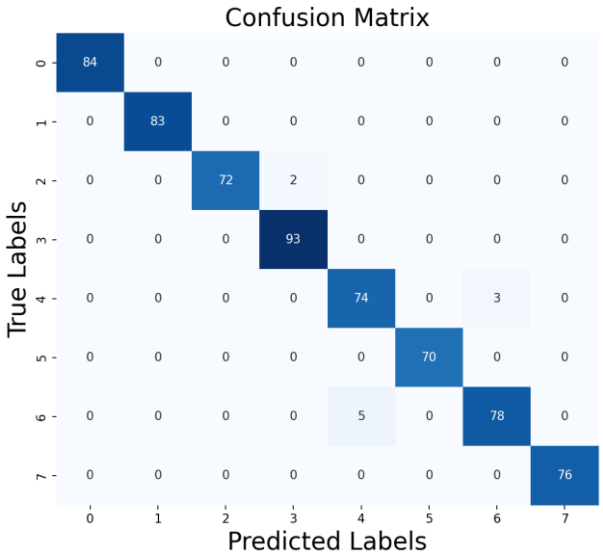
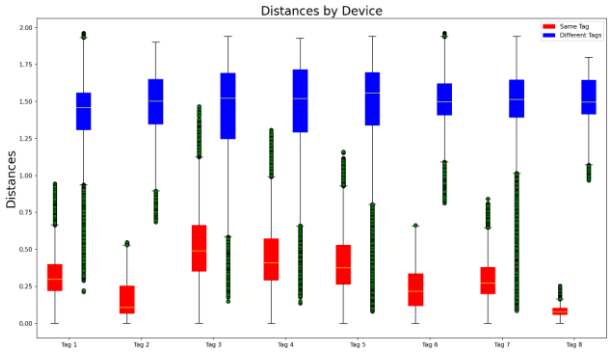


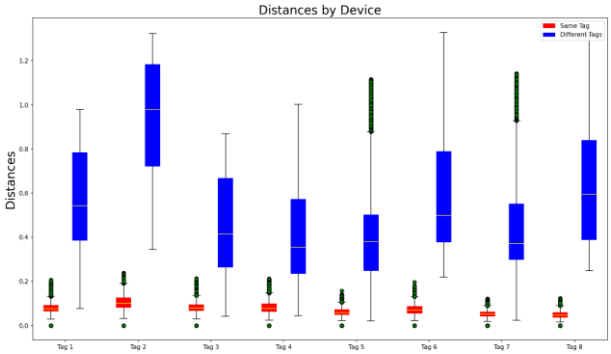
Fig. 20 Confusion Matrix of Model 2+SoftTriple Loss

However, the results shown above are focused on the identification task, where the accuracy of recognition is desired, but this system should also be able to detect rogue devices, where the ability to cluster the embeddings is desired. Although the differences between a legitimate device and a rogue device are much larger than the differences between two legitimate devices, which will definitely make the detection easier, by observing the boxplot of the distances between the embeddings of a legitimate device we can estimate the clustering ability of the pretrained network. The results are shown below:

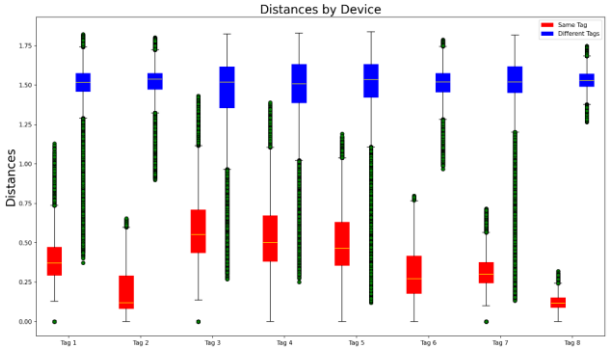
Model 1+CE



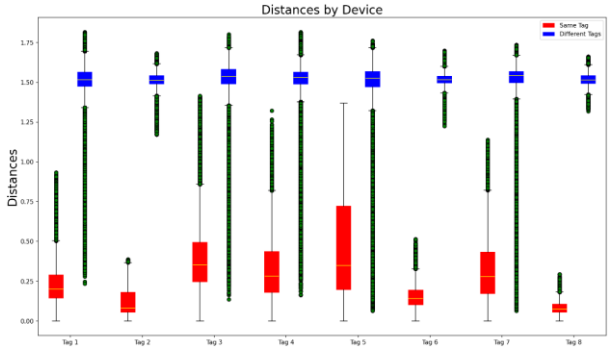
Model 1+Triplet



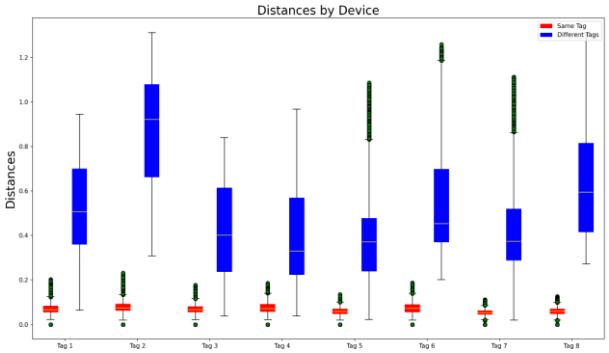
Model 1+SoftTriple



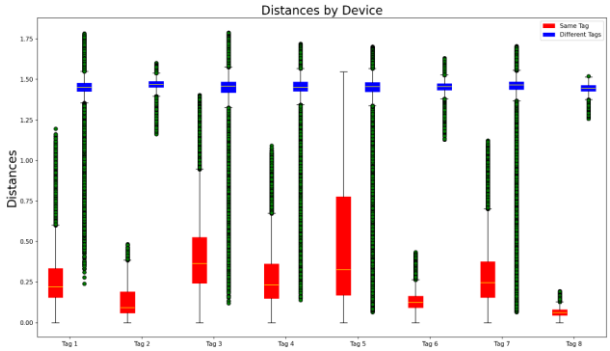
Model 2+CE



Model 2+Triplet



Model 2+SoftTriple



Based on the results shown above, it is obvious that the Cross Entropy loss does not perform well on clustering the embeddings when compared with Triplet loss and SoftTriple loss. The reason for this phenomenon is mentioned. According to [21], optimizing the CE loss is actually optimizing a relaxed triplet with two proxies, which might result in the embeddings converging at different centers. Therefore, the distances between any pair of two embeddings from the same tag might be unstable since it is unpredictable the embedding converges at which center, which leads to bad performance of clustering tasks. A detailed assessment of this result is provided in the next section.

The results about rogue device detection are not provided in this report, since the thresholds are adjustable, which makes the results not suitable to assess. But with the design, sacrificing a bit of TPR and compensating by ignoring some packets from legitimate devices but identified as “anomalies” mistakenly, the system can provide promising results.

7 Discussion

Assessment of Model Performance

The performance of two different network structures is consistent in both the identification task and the authentication task. Model 2 (Structure 2) is obviously better than Structure 1. There are two main reasons for this result. On one hand, the depth of the neural network is more important than the width. On the other hand, the residual shortcut might prevent potential network degradation.

As for the first reason, I believe there are three potential explanations. First, it may be that the randomness of the training process leads to different degrees of convergence of the network. Although we used the early stopping technique during the training process and set the same patience value to control the fairness of the comparison, the network might still have different convergence results. Then it may be a coincidence that Model 2 performs better than Model 1.

Second, as Bengio and LeCun stated [25]: "We claim that most functions that can be represented compactly by deep architectures cannot be represented by a compact shallow architecture." The basic modules of the popular deep learning network structure are convolution, pooling, and activation, which are standard nonlinear transformation modules. A deeper model implies the addition of more nonlinear layers, which might also directly lead to better results. The increased nonlinear expression ability enables the network to learn more complex changes and fit more complex inputs.

Third, the network being deeper means that each layer has simpler tasks. Taking this project as an example, the ATQA signal segment contains lots of kinds of information, including text information (0x04:00), hardware feature information of the SDR device, hardware feature information of the NFC tag, etc. The first convolutional layer may have learned the text information of the signal, then, the second layer may have learned the hardware feature information of the SDR device, and the third layer may have only begun to learn the hardware feature information of the tag. In a similar way, a deeper network is likely to directly mean that the network can learn more complex expressions. If there is only one layer, no matter how complex this layer is, it will be difficult to accurately capture the characteristics of the device from many characteristics, namely, the unique distortions that are independent of the environment and positioning effect.

For the second reason, the explanation can be quite straightforward. As illustrated in the figure provided previously, it is possible that since this project is relatively simple, the first layer has already successfully extracted certain tag features. However, the following layers in Model 1 cause degradation in the performance due to the lack of residual shortcuts. Conversely, in Model 2 the addition of subsequent layers will not result in model degradation. Consequently, Model 2 enhanced its nonlinear expression capabilities without suffering from model degradation. Therefore, the performance of Model 2 is generally better.

Assessment of Data Augmentation

In our extensive experiments, it can be concluded that no data augmentation methods provided effective improvement. The reason for this is simple, the signal acquisition software adopted in this project, NFC Laboratory, does not provide raw IQ samples, but only magnitude samples of the signal are provided. Therefore, we lost the phase information at the beginning. Based on the previous research, phase information, such as IQ imbalance, is the most effective factor to distinguish the devices. And the magnitude of the signal is actually the most vulnerable factor. For example, in the RFFI system for LoRa devices, the device can even move during the signal acquisition process, which directly influences the energy received by the SDR device, and then results in inconsistent signal magnitude. So, the question becomes: Why magnitude can be used to distinguish in this project?

According to [2], the data exchange between the reader and the tag is achieved by modifying the RF energy field created by the reader. The tag is passive in this communication process, which is also the reason why NFC devices are hard to distinguish. Fig. 21 gives a simplified model of this communication:

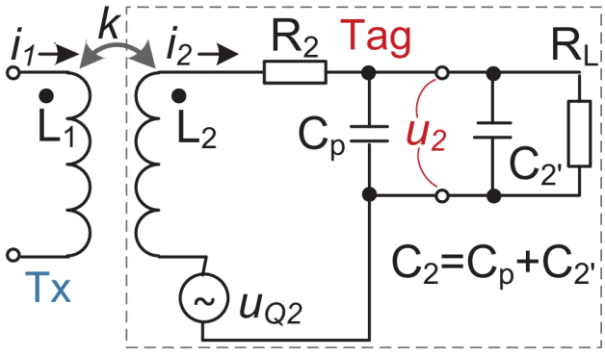


Fig. 21 Simplified Circuit Diagram [2]

To be simple, it is obvious that the magnitude of voltage u_2 depends on both the energy harvested by the tag circuit and the resistance & capacitance of the tag circuit. The detailed calculation is given as:

$$u_2 = \frac{2\pi f \cdot k \cdot \sqrt{L_1 L_2} \cdot i_1}{\sqrt{\left(\frac{2\pi f L_2}{R_L} + 2\pi R_2 C_2\right)^2 + \left[1 - (2\pi f)^2 L_2 C_2 + \frac{R_2}{L_2}\right]^2}}$$

where f is the CW frequency, k is the coupling coefficient between the tag and Tx antenna coils. L_1 and L_2 refer to the conduction loops of the Tx antenna and tag coil, respectively. i_1 represents the Tx antenna's current. C_1, C_2, and RL denote the capacitor, coil resistance, and load resistor of the tag, respectively. Assuming the energy delivered by the reader is constant at this stage, there are only 2 relevant factors left, coupling coefficient k, affecting the energy harvested by the tag circuit, and tag circuit resistance & capacitance.

By applying the card slot and attaching the sniffing antenna to the reader, the coupling

coefficient k is regarded as a constant. So, the magnitude is only affected by the circuitry alone. Therefore, the magnitude of the signal can be used as the RF fingerprint in this project. However, the assumption made before should be verified. We calculate the RMS value of each sample during one signal acquisition process, the results are shown below:

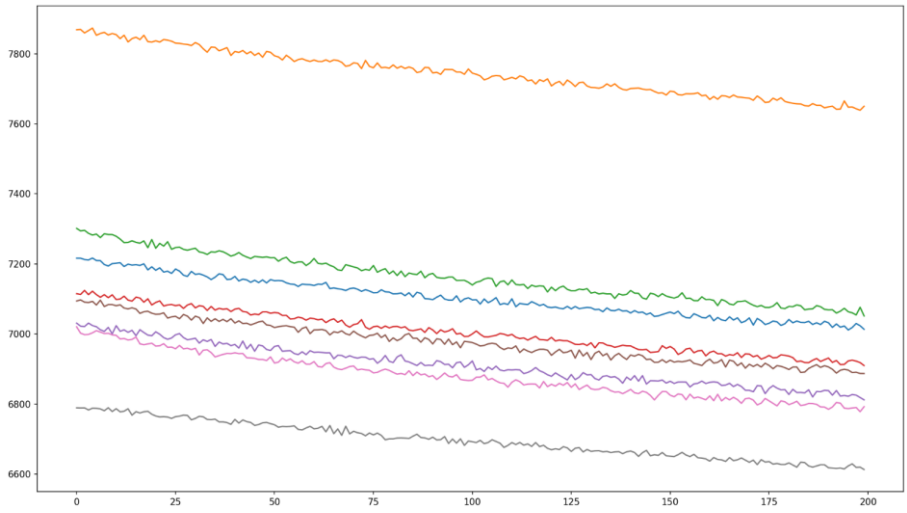


Fig. 22 RMS Value Curves

The different colours stand for different devices and 200 samples are collected for each device. It is obvious that the RMS values decline along the acquisition process. Therefore, the assumption is not correct all the time, which leads to the decision to employ Method 3 during the data augmentation process. However, it is interesting that the RMS curves show clear boundaries between devices, making us think about whether RMS value can be adopted as part of the RF fingerprint. Unfortunately, the RMS value curves are not consistent even with the fixed tag position. The results are shown below:

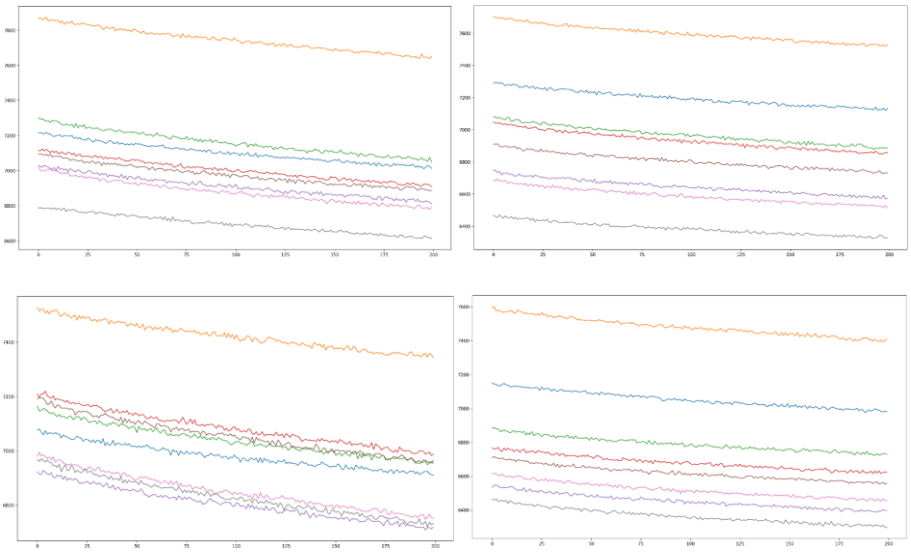


Fig. 23 RMS Value Curves for Dataset 1-4

Assessment of Loss Function

As shown in the results, the CE loss function and SoftTriple loss function perform poorly on the authentication task, where the clustering ability is assessed. The detailed reasons will be demonstrated here.

Recall the formula for Cross Entropy loss:

$$l_{CE}(x_i) = -\log\left(\frac{\exp\left(w_{y_i}^T \cdot x_i\right)}{\sum_j \exp\left(w_j^T \cdot x_i\right)}\right)$$

Since we applied L2 normalization in both two network structures, the formula has the normalized form as:

$$l_{CE_{norm}}(x_i) = -\log\left(\frac{\exp\left(\lambda w_{y_i}^T \cdot x_i\right)}{\sum_j \exp\left(\lambda w_j^T \cdot x_i\right)}\right)$$

Because the magnitude of x_i is normalized, by applying a scaling factor λ, the normalized CE loss is given above. According to [21], optimizing the normalized CE loss can be equivalent to optimizing a smoothed triplet loss. Since the magnitude of both x_i and w_j is normalized, they successfully proved that the goal of optimization is shifted from finding the biggest dot product to finding appropriate w_j that make the biggest cosine similarity between x_i and w_j. Therefore, there is a form describing the goal of CE loss similar to Triplet loss:

$$\text{For CE: } \forall i, j \quad x_i^T w_{y_i} - x_i^T w_j \geq 0$$

$$\text{For Triplet: } \forall a, p, n \quad x_a^T x_p - x_a^T x_n \geq \delta$$

Remember that the a, p, n stands for anchor, positive, and negative. Comparing these two forms, it is obvious that similar to the goal of optimizing Triplet loss, the CE loss optimizes the triplet constraints including an original sample and two centers, such as, (x_i, w_{y_i}, w_j). And the final SoftTriple loss is also designed for precise classification tasks, where the variance between samples from the same class is large.

So, the model trained by these two types of loss functions can be employed at the same time. During the authentication process, the model trained by Triplet loss will be used to generate the embeddings for rogue device detection because Triplet loss has better clustering ability. Then, after the unknown packets pass through the authentication mode, the embedding will be replaced by generating new embeddings by the model pretrained by SoftTriple loss. However, adopting this combination makes the whole process cost much more time. But since the differences between legitimate devices and rogue devices are much larger than those within the legitimate devices, in real practice the models trained by both loss functions provide promising authentication accuracy.

Original Specifications

The specifications proposed at the early stage of project are found to be not practical as the project procced. This report will discuss on each of the specification as following:

Specifications	Results	Explanation
Accuracy over 95%	98.438%	This point is achieved by apply more complex network structure and powerful loss functions.
FPR (False positive rate) below 5%	Achieved	This specification is hard to assess, since this project provides adjustable parameters to generate the threshold, $Threshold = median + \alpha \cdot \sigma$. Therefore, theoretically the FPR can be 0%. But the false negative rate (FNR) will also increase. However, this problem is clearly demonstrated in the *Methodology* section.
Generalization ability	Achieved	Although the deep learning techniques failed to make great improvements in generalization ability, we designed the NFC card slot successfully to solve the generalization problem caused by the instability of the RF fingerprint.
Time cost below 0.5s	3.379s	The time cost of authentication and identification depends on the number of unknown packets collected. Typically, only 50 samples will be collected for authentication. The time for threshold dictionary calculation is about 0.535s and the time for authentication and identification is 2.679s. Assuming the generation of the dictionary is required each time, the whole process costs 3.379s.

8 Reflection

The most valuable skill I acquired during this project is time management. My supervisor, Junqing Zhang, afforded me significant freedom, from choosing the project topic to the details of the following work, always supporting my decisions. This freedom helps allow me to undertake my first independent project. Initially, I underestimated the complexity of designing a signal synchronization algorithm independently, which led to significant errors in my project planning, which was shown in the Gantt Chart included in the initial Specification report where I allocated insufficient time for signal collection. This underestimation resulted in large delays during the first semester.

In the first four weeks, I devoted minimal effort to research, focusing on the Specification report. During weeks 4 and 8, I attempted to use GNU Radio Companions and Python to control a HackRF One for signal collection. At this phase, I encountered numerous challenges such as IQ imbalance, gain setting, and carrier frequency offset (CFO). The lack of knowledge in wireless signal systems forced me to spend time learning about SDR devices, which, while educational, further delayed the project timeline. It was in Week 11 that I decided to switch to a software-supported SDR device like RTL-SDR after I found the open-source NFC Laboratory project. This time delay forced me to double my efforts in the last two weeks of the first semester to catch up and learn about Deep Learning techniques.

Reflecting on the first semester, I consider my time management a significant failure for two main reasons. First, I lacked a thorough understanding of the project and did not take the Specification Report seriously enough, maintaining the foolish notion that I should start the project before fully understanding what it needed. This led to confusion during the signal collection phase, despite my supervisor's repeated suggestions to use available signal collection software, which were ignored at that time. Second, my insistence on developing independently often resulted in long periods of stagnation on specific problems, which was impractical from a project completion standpoint. Achieving overall project goals should have taken precedence over perfecting individual components.

After finishing this project, I believe I will perform thorough interviews and preparation by either consulting the supervisor or searching online before I take the project. And this responsibility will be definitely beneficial to my career development. Furthermore, the ability to finish a formal project independently will help me to become calmer when encountering difficulties. These transferable skills are significant since they are learned in a self-directed lifelong learning way.

9 Conclusion

Throughout this project, I acquired numerous technical and non-technical skills that emphases on both the challenges encountered and the personal development achieved. This project is an educational trip into the complexities of NFC technology and its applications, with a special focus on enhancing communication security through advanced RF fingerprinting technology.

The project included several phases. Initially, it was determined that RF fingerprinting technology would be utilized to enhance NFC security. Following a comprehensive review of recent literature in this domain, deep learning techniques were employed to extract device-specific features, leveraging the ability of deep neural networks (DNNs) in feature extraction and significantly reducing the need for expert knowledge to extract features manually. Then, with an understanding of NFC protocols, the ATQA signal segment was chosen as the target signal, as it initiates every Type-A communication.

After multiple attempts, NFC Laboratory software was utilized to simplify signal synchronization, significantly accelerating subsequent processes. A comprehensive experimental testbed was established to collect signals for later supervised learning, due to the requirement for large data volume. Various data augmentation methods were proposed and implemented to enhance generalization. However, after evaluating all data augmentation methods and their results, only the method of dividing RMS value was employed, which, while not substantially improving generalization ability, performed well in feature clustering. Furthermore, it was determined that the inconsistent tag positioning primarily contributed to RF fingerprint instability. To address this issue, a card slot was designed and 3D modeled to fix the tag's position.

Following data collection, two deep learning network structures, same in parameter amount, were proposed and evaluated. The deeper model, featuring residual shortcuts, proved better in this project. Three different loss functions—Cross Entropy loss, Triplet loss, and SoftTriple loss—were evaluated. Based on extensive experimental data, the Triplet loss, employing a BatchHard online mining strategy, outperformed in clustering embeddings, and the SoftTriple loss was proved to be optimal for the device recognition task. Given the effectiveness of Triplet loss in embedding clustering, a rogue device detection system was designed with adjustable parameters to control detection strictness. To make use of the effectiveness of SoftTriple loss in the identification task, a synthesized system was proposed, utilizing another model trained with this loss function for final predictions.

In conclusion, completing this project has been hugely beneficial for both professional growth and personal development. The obtained insights are beneficial not only to future technical endeavors but also to any professional scenario that requires a combination of technical expertise, time management, and adaptive management. The ability to synthesize complex technical concepts into practical applications has significantly advanced, establishing a strong foundation for future challenges and opportunities in the NFC technology field.

Reference

[1] N. Soltanieh, Y. Norouzi, Y. Yang and N. C. Karmakar, "A Review of Radio Frequency Fingerprinting Techniques," in IEEE Journal of Radio Frequency Identification, vol. 4, no. 3, pp. 222-233, Sept. 2020, doi:10.1109/JRFID.2020.2968369.

[2] Y. Yang, J. Cao, Z. An, Y. Wang, P. Hu and G. Zhang, "NFChain: A Practical Fingerprinting Scheme for NFC Tag Authentication," IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, New York City, NY, USA, 2023, pp. 1-10, doi: 10.1109/INFOCOM53939.2023.10229040.

[3] G. Shen, J. Zhang, A. Marshall and J. R. Cavallaro, "Towards Scalable and Channel-Robust Radio Frequency Fingerprint Identification for LoRa," in IEEE Transactions on Information Forensics and Security, vol. 17, pp. 774-787, 2022, doi: 10.1109/TIFS.2022.3152404.

[4] W. Lee, S. Y. Baek and S. H. Kim, "Deep-Learning-Aided RF Fingerprinting for NFC Security," in IEEE Communications Magazine, vol. 59, no. 5, pp. 96-101, May 2021, doi: 10.1109/MCOM.001.2000912.

[5] Wang, Y.; Zou, J.; Zhang, K., "Deep-Learning-Aided RF Fingerprinting for NFC Relay Attack Detection," *Electronics* 2023, *12*, 559. https://doi.org/10.3390/electronics12030559

[6] NFC Forum. (n.d.). Covid-19 resources. [Online]. Available: https://nfc-forum.org/covid-19-resources. [Accessed: March 28, 2024].

[7] MarketWatch. "Near field communication market 2021 to 2025 growth factors, market characteristics, opportunities by type analysis and forecast." [Online]. Available: https://www.marketwatch.com/press-release/near-field-communication-nfc-market-2021-to-2025-growth-factors-market-characteristics-opportunities-by-type-analysis-and-forecast-2021-06-10. [Accessed: April 1, 2024].

[8] Healthcare IT News. "New Zealand pilots NFC tags for COVID-19 tracking." [Online]. Available: https://www.healthcareitnews.com/news/anz/new-zealand-pilots-nfc-tags-covid-19-tracking. [Accessed: April 1, 2024].

[9] Ikarus23, "MifareClassicTool," GitHub. [Online]. Available: https://github.com/ikarus23/MifareClassicTool. [Accessed: April 3, 2024].

[10] B. Danev, T. S. Heydt-Benjamin, and S. Čapkun, "Physical-layer identification of RFID devices," in Proceedings of the 18th conference on USENIX security symposium (SSYM'09), USENIX Association, USA, 2009, pp. 199–214.

[11] ISO/IEC 14443-3:2016; Identification Cards—Contactless Integrated Circuit Cards—Proximity Cards—Part 3: Initialization and Anticollision. ISO, Geneva, Switzerland, 2016.

[12] J. Han, C. Qian, Y. Yang, G. Wang, H. Ding, X. Li, and K. Ren, "Butterfly: Environment-Independent Physical-Layer Authentication for Passive RFID," Proc. ACM Interact. Mob. Wearable Ubiquitous Technol., vol. 2, no. 4, Art. no. 166, Dec. 2018, doi: 10.1145/3287044.

[13] B. Johnson and B. Hamdaoui, "On the Domain Generalizability of RF Fingerprints Through Multifractal Dimension Representation," 2023 IEEE Conference on Communications and Network Security (CNS), Orlando, FL, USA, 2023, pp. 1-9, doi: 10.1109/CNS59707.2023.10289056.

[14] W. Wang, C. Luo, J. An, L. Gan, H. Liao and C. Yuen, "Semisupervised RF Fingerprinting With Consistency-Based Regularization," in IEEE Internet of Things Journal, vol. 11, no. 5, pp. 8624-8636, 1 March1, 2024, doi: 10.1109/JIOT.2023.3320975.

[15] S. Riyaz, K. Sankhe, S. Ioannidis and K. Chowdhury, "Deep Learning Convolutional Neural Networks for Radio Identification," in IEEE Communications Magazine, vol. 56, no. 9, pp. 146-152, Sept. 2018, doi: 10.1109/MCOM.2018.1800153.

[16] Vedat Coskun; Kerem Ok; Busra Ozdenizci, "NFC Essentials," in Near Field Communication (NFC): From Theory to Practice, Wiley, 2012, pp.73-114, doi: 10.1002/9781119965794.ch3.

[17] J. V. Campos, "nfc-laboratory," GitHub. [Online]. Available: https://github.com/josevcm/nfc-laboratory. [Accessed: April 5, 2024].

[18] ACS Company, "ACR122U USB NFC Reader Technical Specifications V3.06". [Online]. Available: https://www.acs.com.hk/en/products/3/acr122u-usb-nfc-reader/. [Accessed: April 6, 2024].

[19] B. Cha and J. Kim, "Design of NFC Based Micro-payment to Support MD Authentication and Privacy for Trade Safety in NFC Applications," 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems, Taichung, Taiwan, 2013, pp. 710-713, doi: 10.1109/CISIS.2013.127.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in ArXiv, vol. abs/1512.03385, 2015.

[21] Q. Qian et al., "SoftTriple Loss: Deep Metric Learning Without Triplet Sampling," 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, Korea (South), 2019, pp. 6449-6457, doi: 10.1109/ICCV.2019.00655.

[22] Hermans, A., Beyer, L., and Leibe, B., "In Defense of the Triplet Loss for Person Re-Identification," ArXiv, vol. 1703.07737, 2017.

[23] Schroff, F., Kalenichenko, D., & Philbin, J. (2015). "FaceNet: A Unified Embedding for Face

Recognition and Clustering," ArXiv. https://doi.org/10.1109/CVPR.2015.7298682

[24] B. Johnson and B. Hamdaoui, "On the Domain Generalizability of RF Fingerprints Through Multifractal Dimension Representation," 2023 IEEE Conference on Communications and Network Security (CNS), Orlando, FL, USA, 2023, pp. 1-9, doi: 10.1109/CNS59707.2023.10289056.

[25] Léon Bottou; Olivier Chapelle; Dennis DeCoste; Jason Weston, "Scaling Learning Algorithms toward AI," in Large-Scale Kernel Machines, MIT Press, 2007, pp.321-359.

Appendix

Revised Project Milestones

Phase	WP	Deliverable	Milestone
Preliminary Work	\	A detailed Project Specification Report demonstrating the overall arrangement of the project	Specification Report (15 Oct. 2023)
Data Acquisition	WP1 (3.1)	A comprehensive experiment testbed that is capable of capturing the raw RF signal	Crowdfunding Pitch (Week 12)
	WP2 (3.2)	A complete signal receiving system that can read the raw data transmitted by the SDR device	
	WP3 (3.3)	A comprehensive dataset that can be employed to train the neural network	
Algorithm Design	WP4 (3.4)	A dataset that can be employed directly, generated by appropriate data augmentation method based on the selected loss function	
	WP5 (3.5)	A synthesized system that can display the result of anomaly detection and device recognition by using the model trained by Triplet loss	Final System Testing (Week 21)
Other work	Bench Inspection	A simple DEMO that can be used to display the result of counterfeit detection by using the trained neural network	Bench Inspection (Week 22)
	Final Report	A complete system that can be used to display the result of anomaly detection and device recognition by using the model trained by different loss functions	Final Report (Week 24)

Revised Project Gantt Chart:

Source Codes

Data Augmentation Methods

```
import os
import re
import json
import numpy as np
import matplotlib.pyplot as plt
import scipy.io.wavfile as wav
import h5py
import torch
from scipy import signal
```

```
def cal_diff(sample, max_len = 130):
    mean_value = np.mean(sample)
    crossing_diff = []
    for i in range(1, len(sample)):
        if (sample[i - 1] < mean_value and sample[i] >= mean_value) or (
                sample[i - 1] > mean_value and sample[i] <= mean_value):
            crossing_diff.append(sample[i]-sample[i-1])
    return crossing_diff[:max_len]


def method_0(folder_path, output_folder, max_samples=300):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            hdf5_filename = os.path.join(output_folder, base_name + '.hdf5')
            _, signal = wav.read(signal_dir)
            signals = []
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
                atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
                sak_starts = [frame['sampleStart'] for frame in frames if "08:B6:DD" in
frame['frameData']]
            for i in range(min(len(atqa_starts), len(sak_starts))):
                if i >= max_samples:
                    break
                atqa_signal = signal[atqa_starts[i]:atqa_starts[i]+410]
                sak_signal = signal[sak_starts[i]:sak_starts[i]+620]
                if len(atqa_signal) < 410 or len(sak_signal) < 620:
                    continue
                # combined_signal = np.concatenate((atqa_signal, sak_signal))
                # normalized_signal = normalize(atqa_signal)
                signals.append(atqa_signal)
            print('Sample Collected:', len(signals))
            with h5py.File(hdf5_filename, 'w') as hdf5_file:
```

```
                    dataset = hdf5_file.create_dataset("signals", (len(signals), 410),
dtype='float32')
                    for i, signal in enumerate(signals):
                        dataset[i] = signal
                # np.save(numpy_filename,signals)
                print(f"Processed {base_name} to HDF5.")
def auto_subtract_signals(folder_path, base_filename, dataset_name):
    # Path to the base file
    base_filepath = os.path.join(folder_path, base_filename)
    # Load the base dataset from the base file
    with h5py.File(base_filepath, 'r') as base_file:
        base_dataset = base_file[dataset_name][()]
    # List all hdf5 files in the folder except the base file
    hdf5_files = [f for f in os.listdir(folder_path) if f.endswith('.hdf5') and f != base_filename]
    # Go through each hdf5 file and perform the subtraction
    for hdf5_file in hdf5_files:
        hdf5_filepath = os.path.join(folder_path, hdf5_file)
        with h5py.File(hdf5_filepath, 'r+') as h5_file:
            if dataset_name in h5_file:
                h5_file[dataset_name][:] -= base_dataset
                print(f"Subtraction performed for {hdf5_file}")
            else:
                print(f"Dataset '{dataset_name}' not found in {hdf5_file}")


def normalize1(signal):
    signal = signal.astype(np.float32)
    mean = np.mean(signal)
    shifted_signal = signal - mean
    signal_max = max(np.max(shifted_signal), abs(np.min(shifted_signal)))
    if signal_max == 0:
        return np.zeros_like(shifted_signal)
    normalized_signal = shifted_signal / signal_max
    return normalized_signal


def method_1(folder_path, output_folder, max_samples=300):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
```

```
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            hdf5_filename = os.path.join(output_folder, base_name + '.hdf5')
            _, signal = wav.read(signal_dir)
            signals = []
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
                atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
                for i in range(len(atqa_starts)):
                    if i >= max_samples:
                        break
                    atqa_signal = signal[atqa_starts[i]:atqa_starts[i]+410]
                    atqa_norm = normalize1(atqa_signal)
                    signals.append(atqa_norm)
            print('Sample Collected:', len(signals))
            with h5py.File(hdf5_filename, 'w') as hdf5_file:
                dataset = hdf5_file.create_dataset("signals", (len(signals), 410),
dtype='float32')
                for i, signal in enumerate(signals):
                    dataset[i] = signal
            print(f"Processed {base_name} to HDF5.")


def atqa_curves(folder_path, output_folder, max_samples=10):
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    samples_means = []
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            _, signal = wav.read(signal_dir)
            signals = []
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
                atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
```

```
                    for i in range(min(len(atqa_starts), max_samples)):
                        atqa_signal = signal[atqa_starts[i]:atqa_starts[i]+410]
                        atqa_signal = atqa_signal.astype(np.float32)
                        atqa_norm = normalize_method3(atqa_signal)
                        signals.append(atqa_norm)
                    samples_mean = np.mean([np.sqrt(np.mean(np.square(signal))) for signal in
signals])
                    samples_means.append(samples_mean)
    print(np.std(samples_means)/np.mean(samples_means))

    plt.figure(figsize=(8, 6))
    plt.plot(range(1, len(samples_means)+1), samples_means, marker='o')
    plt.title('Average of averages atqa_norm for Each Tag')
    plt.xlabel('Dataset')
    plt.ylabel('Average of averages atqa_norm')
    plt.show()


def normalize_method2(signal):
    signal_max = np.max(signal)
    signal_min = np.min(signal)
    if signal_max == signal_min:
        return np.zeros_like(signal)
    normalized_signal = (signal - signal_min) / (signal_max - signal_min)
    return normalized_signal


def method_2(folder_path, output_folder, max_samples=200):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            hdf5_filename = os.path.join(output_folder, base_name + '.hdf5')
            _, signal = wav.read(signal_dir)
            signals = []
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
```

```
                    atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
                    sak_starts = [frame['sampleStart'] for frame in frames if "08:B6:DD" in
frame['frameData']]
                for i in range(len(atqa_starts)):
                    if i >= max_samples:
                        break
                    atqa_signal = signal[atqa_starts[i]:atqa_starts[i]+410]
                    atqa_norm = normalize_method2(atqa_signal)
                    signals.append(atqa_norm)
                print('Sample Collected:', len(signals))
                print('Sample Length:', len(signals[0]))

                with h5py.File(hdf5_filename, 'w') as hdf5_file:
                    dataset = hdf5_file.create_dataset("signals", (len(signals), 410),
dtype='float32')
                    for i, signal in enumerate(signals):
                        dataset[i] = signal
                print(f"Processed {base_name} to HDF5.")


def cal_diff_v3(sample, max_len=120):
    mean_value = np.mean(sample)
    crossing_diff = []
    i = 0    # Start from the first sample
    while i < len(sample) - 1 and len(crossing_diff) < max_len:
        # Move to the next point if current is within 10% of the mean or on the same side as
the next
        next_i = i + 1
        while next_i < len(sample) and (sample[next_i] - mean_value) * (sample[i] -
mean_value) >= 0:
            next_i += 1
        if next_i < len(sample):
            # Calculate difference if we've found a point on the other side of the mean
            diff = sample[next_i] - sample[i]
            average = (sample[next_i] + sample[i]) / 2
            crossing_diff.append(diff / average)
            i = next_i    # Jump to the next point that was on the other side
        else:
            break    # Exit if no more points are found
    return crossing_diff[:max_len]


def plot_power_curve(folder_path, output_folder, max_samples=300):
```

```
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    all_RMSs = []
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            _, signal = wav.read(signal_dir)
            RMSs = []
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
                atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
            for i in range(len(atqa_starts)):
                if i >= max_samples:
                    break
                atqa_signal = signal[atqa_starts[i]:atqa_starts[i] + 410]
                atqa_signal = atqa_signal.astype(np.float32)
                RMSs.append(np.sqrt(np.mean(np.square(atqa_signal))))
            all_RMSs.append(RMSs)
    for rms in all_RMSs:
        plt.plot(rms)
    plt.show()


def plot_dataset_sample(folder_path, output_folder, max_samples=300):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        raw_means = []
        norm_means = []
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            _, signal = wav.read(signal_dir)
```

```
                with open(json_dir, 'r', encoding='utf8') as fp:
                    json_data = json.load(fp)
                    frames = json_data['frames']
                    atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
                    sak_starts = [frame['sampleStart'] for frame in frames if "08:B6:DD" in
frame['frameData']]
                    for i in range(len(atqa_starts)):
                        if i >= max_samples:
                            break
                        atqa_signal = signal[atqa_starts[i]:atqa_starts[i] + 410]
                        sak_siganl = signal[atqa_starts[i]:atqa_starts[i] + 410]
                        atqa_norm = atqa_signal / np.mean(atqa_signal)
                        raw_means.append(np.mean(atqa_signal))
                        norm_means.append(np.mean(atqa_norm))
            plt.plot(raw_means)
            plt.plot(norm_means+np.mean(raw_means))
            plt.show()
def normalize_method3(signal):
    signal = signal.astype(np.float32)
    rms = np.sqrt(np.mean(np.square(signal)))
    if rms == 0:
        return signal
    normalized_signal = signal / rms
    return normalized_signal


def method_3(folder_path, output_folder, max_samples=200):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            hdf5_filename = os.path.join(output_folder, base_name + '.hdf5')
            _, signal = wav.read(signal_dir)
            signals = []
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
```

```
                    atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
                for i in range(len(atqa_starts)):
                    if i >= max_samples:
                        break
                    atqa_signal = signal[atqa_starts[i]:atqa_starts[i]+410]
                    atqa_signal = atqa_signal.astype(np.float32)
                    atqa_norm = normalize_method3(atqa_signal)
                    signals.append(atqa_norm)
                print('Sample Collected:', len(signals))
                print('Sample Length:', len(signals[0]))
                with h5py.File(hdf5_filename, 'w') as hdf5_file:
                    dataset = hdf5_file.create_dataset("signals", (len(signals), 410),
dtype='float32')
                    for i, signal in enumerate(signals):
                        dataset[i] = signal
                print(f"Processed {base_name} to HDF5.")



def calculate_VFDT(signal, window_size=20, stride=2, samp_rate=2.4e6):
    delta_w = window_size / samp_rate
    VFDT = []
    for i in range(0, len(signal) - window_size, stride):
        window = signal[i:i + window_size]
        delta_x = np.diff(window)
        var_delta_x = np.var(delta_x)
        log_var_x = np.log(var_delta_x)
        VFDT.append(2 - log_var_x / (2 * np.log(delta_w)))
    return VFDT


def normalize_method4(signal):
    signal = signal.astype(np.float32)
    rms = np.sqrt(np.mean(np.square(signal)))
    if rms == 0:
        return signal
    normalized_signal = signal / rms
    return normalized_signal


def method_4(folder_path, output_folder, max_samples=200):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
```

```
    wav_files = [f for f in os.listdir(folder_path) if f.endswith('.wav')]
    js_files = [f for f in os.listdir(folder_path) if f.endswith('.js') or f.endswith('.json')]
    for wav_file in wav_files:
        base_name = re.sub(r'.wav$', '', wav_file)
        js_file = base_name + '.js' if base_name + '.js' in js_files else base_name + '.json'
        if js_file in js_files:
            signal_dir = os.path.join(folder_path, wav_file)
            json_dir = os.path.join(folder_path, js_file)
            hdf5_filename = os.path.join(output_folder, base_name + '.hdf5')
            _, signal = wav.read(signal_dir)
            signals = []
            reference_sig = None
            with open(json_dir, 'r', encoding='utf8') as fp:
                json_data = json.load(fp)
                frames = json_data['frames']
                atqa_starts = [frame['sampleStart'] for frame in frames if "04:00" in
frame['frameData']]
            for i in range(len(atqa_starts)):
                if i >= max_samples:
                    break
                atqa_signal = signal[atqa_starts[i]:atqa_starts[i]+410]
                atqa_norm = calculate_VFDT(atqa_signal)
                signals.append(atqa_norm)
            print('Sample Collected:', len(signals))
            print('Sample Length:', len(signals[0]))
            with h5py.File(hdf5_filename, 'w') as hdf5_file:
                dataset = hdf5_file.create_dataset("signals", (len(signals), 195),
dtype='float32')
                for i, signal in enumerate(signals):
                    dataset[i] = signal
            print(f"Processed {base_name} to HDF5.")
```

Load Data

```
from Methods import *

folder_path = './raw_data/d4'
output_folder = './data/d4'
plot_power_curve(folder_path, output_folder, max_samples=200)
```

Triplet Loss Training

```
from torch.utils.data import DataLoader, random_split
import numpy as np
```

```
import random
import torch
from torch.optim import Adam, SGD
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns

from net import model_1_tri, model_2_tri
from tools import MyDataset, EarlyStopping, TripletLoss, SoftTriple
from demo_extractor import testset_eval, get_embeddings


def setup_seed(seed):
     torch.manual_seed(seed)
     torch.cuda.manual_seed_all(seed)
     np.random.seed(seed)
     random.seed(seed)
     torch.backends.cudnn.deterministic = True
setup_seed(1)


# Hyper Parameters
num_epochs = 200
LR = 1e-4
patience = 20
batch_size = 256
train_ratio = 0.9
margin = 0.2
decayRate = 0.987
model_save_path = './trained_net/model_1_ST.pth'
model = model_1_tri().cuda()


optimizer = Adam(model.parameters(), lr=LR, weight_decay=0.0001)
# loss_function = TripletLoss(margin=margin)
loss_function = SoftTriple(la=0.5, gamma=0.1, tau=0.2, margin=margin, dim=512, cN=8, K=1)
early_stopping = EarlyStopping(patience=patience, verbose=True, path=model_save_path)
lr_scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer=optimizer,
gamma=decayRate)
dataset = MyDataset(root_dir='./data/d1234')
train_size = int(train_ratio * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
```

```
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=256, shuffle=False)



def adjust_learning_rate(optimizer, epoch, warmup_epochs=20, base_lr=LR):
    """Adjusts learning rate for each epoch based on the given parameters"""
    if epoch <= warmup_epochs:
        lr = base_lr * (epoch / warmup_epochs)
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
    else:
        lr_scheduler.step()



def train(model, train_loader, val_loader, optimizer, triplet_loss, num_epochs=50,
model_save_path=None):
    with open('results.txt', 'w') as file:
        for epoch in range(num_epochs):
            adjust_learning_rate(optimizer, epoch, warmup_epochs=20, base_lr=LR)
            model.train()
            total_train_loss = 0.0
            for inputs, labels in train_loader:
                inputs = inputs.unsqueeze(1).cuda()
                labels = labels.cuda()
                optimizer.zero_grad()
                embed = model(inputs)
                loss = triplet_loss(embed, labels)
                loss.backward()
                optimizer.step()
                total_train_loss += loss.item()
            model.eval()
            total_val_loss = 0.0
            with torch.no_grad():
                for inputs, labels in val_loader:
                    inputs = inputs.unsqueeze(1).to('cuda')
                    labels = labels.to('cuda')
                    embed = model(inputs)
                    loss = triplet_loss(embed, labels)
                    total_val_loss += loss.item()
            avg_train_loss = total_train_loss / len(train_loader)
            avg_val_loss = total_val_loss / len(val_loader)
            print(f'Epoch {epoch + 1}/{num_epochs}')
```

```
            print(f'Train Total Loss: {avg_train_loss:.4f}')
            print(f'Validation Total Loss: {avg_val_loss:.4f}')
            print('==' * 30)
            file.write(f"{epoch + 1},{avg_train_loss:.4f},{avg_val_loss:.4f}\n")
            early_stopping(avg_val_loss, model, epoch + 1)
            if early_stopping.early_stop:
                print(f"Early stopping triggered at epoch {early_stopping.best_epoch}.")
                print(
                    f"Loading best model from epoch {early_stopping.best_epoch} with loss
{early_stopping.val_loss_min}")
                break



def test():
    model = model_1_tri()
    model.load_state_dict(torch.load(model_save_path))
    model.eval()
    Train_embeddings, Train_labels = get_embeddings(model, train_loader)
    k = 1
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(Train_embeddings, Train_labels)
    testset_eval(Tri_model=model, test_loader=val_loader, knn=knn)



# train(model, train_loader, val_loader, optimizer, loss_function, num_epochs,
model_save_path)
test()
```

Distance Boxplot

```
import torch
import numpy as np
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import random
from sklearn.metrics import roc_curve, auc

from tools import MyDataset
from net import model_1_tri, model_2_tri, model_2_feature, model_1_feature


def get_embeddings(model, dataloader):
```

```
    embeddings = []
    labels = []
    with torch.no_grad():
        for inputs, classes in dataloader:
            inputs = inputs.unsqueeze(1).cuda()
            emb = model(inputs)
            embeddings.append(emb.cpu().numpy())
            labels.extend(classes.numpy())
    return np.vstack(embeddings), np.array(labels)



test_dataset = MyDataset(root_dir='./data/d1234')
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)

CE_model_path = './trained_net/model_2_CE.pth'
Tri_model_path = './trained_net/model_2_ST.pth'

num_classes = 8
CE_model = model_2_feature(num_classes).cuda()
Tri_model = model_2_tri().cuda()

CE_model.load_state_dict(torch.load(CE_model_path))
Tri_model.load_state_dict(torch.load(Tri_model_path))
CE_model.eval()
Tri_model.eval()



CE_embeddings, CE_labels = get_embeddings(CE_model, test_loader)
Tri_embeddings, Tri_labels = get_embeddings(Tri_model, test_loader)



def calculate_distance(embeddings, labels, num_samples=40):
    all_genuine_distances = []
    all_forged_distances = []

    # Iterate over each class
    for genuine_class_id in range(num_classes):
        # Get embeddings for the genuine class
        genuine_class_embeddings = embeddings[labels == genuine_class_id]

        if len(genuine_class_embeddings) >= num_samples:
            # Randomly select num_samples of embeddings for the genuine class
            selected_genuine_embeddings = genuine_class_embeddings[
```

```
                random.sample(range(len(genuine_class_embeddings)), num_samples)]
        else:
            # If there are not enough samples, use what's available
            selected_genuine_embeddings = genuine_class_embeddings

        genuine_distances = []
        forged_distances = []

        # Calculate distances from each selected genuine embedding to other samples
        for selected_emb in selected_genuine_embeddings:
            # For the genuine class
            distances = np.linalg.norm(selected_emb - genuine_class_embeddings, axis=1)
            genuine_distances.extend(distances)

            # For forged classes
            for forged_class_id in range(num_classes):
                if forged_class_id != genuine_class_id:
                    forged_class_embeddings = embeddings[labels == forged_class_id]
                    if len(forged_class_embeddings) >= num_samples:
                        selected_forged_embeddings = forged_class_embeddings[
                            random.sample(range(len(forged_class_embeddings)),
num_samples)]
                    else:
                        selected_forged_embeddings = forged_class_embeddings
                    distances = np.linalg.norm(selected_emb -
selected_forged_embeddings, axis=1)
                    forged_distances.extend(distances)

        all_genuine_distances.append(genuine_distances)
        all_forged_distances.append(forged_distances)

    return all_genuine_distances, all_forged_distances



def box_plot(all_genuine_distances, all_forged_distances):
    fig, ax = plt.subplots()
    positions = np.array(range(num_classes)) * 2
    genuine_positions = positions - 0.4
    forged_positions = positions + 0.4
    plt.boxplot(all_genuine_distances, positions=genuine_positions, widths=0.4,
patch_artist=True,
                boxprops=dict(facecolor="red", color="red"),
                medianprops=dict(color="yellow"), showfliers=True,
```

```
flierprops=dict(markerfacecolor='g', marker='o'))
    plt.boxplot(all_forged_distances, positions=forged_positions, widths=0.4,
patch_artist=True,
                boxprops=dict(facecolor="blue", color="blue"),
                medianprops=dict(color="yellow"), showfliers=True,
flierprops=dict(markerfacecolor='g', marker='o'))
    ax.set_xticks(positions)
    ax.set_xticklabels([f'Tag {i + 1}' for i in range(num_classes)])
    ax.set_ylabel('Distances', fontsize=20)
    ax.set_title('Distances by Device', fontsize=20)
    from matplotlib.patches import Patch
    legend_elements = [Patch(facecolor='red', label='Same Tag'),
                       Patch(facecolor='blue', label='Different Tags')]
    ax.legend(handles=legend_elements, loc='upper right')
    plt.show()



all_genuine_distances, all_forged_distances = calculate_distance(Tri_embeddings, Tri_labels,
num_samples=400)

box_plot(all_genuine_distances, all_forged_distances)
```

DEMO

```
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, random_split
import os
import json
import scipy.io.wavfile as wav
from scipy.spatial.distance import pdist, cdist
import time

from tools import MyDataset
from net import model_1_tri, model_2_tri
from Methods import normalize_method3
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
```

```
def testset_eval(Tri_model, test_loader, knn):
    test_embeddings, test_labels = get_embeddings(Tri_model, test_loader)
    test_labels_predicted = knn.predict(test_embeddings)
    test_accuracy = accuracy_score(test_labels, test_labels_predicted)
    print(f"Test set accuracy: {test_accuracy}")
    cm = confusion_matrix(test_labels, test_labels_predicted)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.xlabel('Predicted Labels', fontsize=20)
    plt.ylabel('True Labels', fontsize=20)
    plt.title('Confusion Matrix', fontsize=20)
    plt.show()


def compute_thresholds(registered_embeddings, registered_labels):
    thresholds = {}
    for label in np.unique(registered_labels):
        label_embeddings = registered_embeddings[registered_labels == label]
        distances = pdist(label_embeddings, 'euclidean')
        median = np.median(distances)
        std = np.std(distances)
        thresholds[label] = median + 1.5 * std
    return thresholds


def predict():
    thresholds = compute_thresholds(Train_embeddings, Train_labels)
    folder_path = './APP'
    signal_dir = None
    json_dir = None
    for filename in os.listdir(folder_path):
        if filename.endswith('.wav') and signal_dir is None:
            signal_dir = folder_path + '/' + filename
        elif filename.endswith('.json') and json_dir is None:
            json_dir = folder_path + '/' + filename
        if signal_dir is not None and json_dir is not None:
            break

    _, signal = wav.read(signal_dir)

    atqa_starts = []
    with open(json_dir, 'r', encoding='utf8') as fp:
        json_data = json.load(fp)
        frames = json_data['frames']
```

```
            for i in range(len(frames)):
                if "04:00" in frames[i]['frameData']:
                    atqa_starts.append(frames[i]['sampleStart'])

        genuine_count = 0
        anomaly_count = 0
        predictions = []

        for i in range(len(atqa_starts) - 1):
            atqa_signal = signal[atqa_starts[i]:atqa_starts[i] + 410]
            norm_signal = normalize_method3(atqa_signal)
            input = torch.tensor(norm_signal).unsqueeze(0).unsqueeze(1)
            test_embedding = Tri_model(input)
            test_embedding = test_embedding.detach().cpu()
            predicted_label = knn.predict(test_embedding)[0]
            predictions.append(predicted_label)
            label_indices = np.where(knn.classes_ == predicted_label)[0]
            distances, indices = knn.kneighbors(test_embedding, n_neighbors=len(label_indices))
            distances = distances.flatten()
            mean_distance = np.mean(distances)
            if mean_distance > thresholds[predicted_label]:
                anomaly_count += 1
            else:
                genuine_count += 1
        categories = ['Genuine', 'Anomaly']
        quantities = [genuine_count, anomaly_count]

        plt.figure(figsize=(8, 6))
        bar_positions = np.arange(len(categories))
        plt.bar(bar_positions, quantities, color=['skyblue', 'salmon'])
        plt.xticks(bar_positions, categories)
        plt.ylabel('Count')
        plt.title('Prediction of Genuine & Anomaly Packets')
        plt.grid(axis='y', linestyle='--', alpha=0.7)
        # plt.show()

        if anomaly_count < genuine_count:
            plt.figure(figsize=(10, 6))
            counts, bins, patches = plt.hist(predictions, bins=np.arange(-0.5, 8.5, 1),
edgecolor='black', rwidth=0.8)
            for count, bin, patch in zip(counts, bins, patches):
                height = patch.get_height()
                plt.text(patch.get_x() + patch.get_width() / 2, height + 0.1, f'{int(count)}',
ha='center', va='bottom',
```

```
                    fontsize=12)

        plt.xticks(range(8), ['Tag 1', 'Tag 2', 'Tag 3', 'Tag 4', 'Tag 5', 'Tag 6', 'Tag 7', 'Tag 8'])
        plt.title('Total Sample Predicted: {}'.format(len(predictions)))
        plt.xlabel('Predicted Class')
        plt.ylabel('Number of Predictions')
        # plt.show()



def get_embeddings(model, dataloader):
    embeddings = []
    labels = []
    with torch.no_grad():
        for inputs, classes in dataloader:
            inputs = inputs.unsqueeze(1)
            emb = model(inputs)
            embeddings.append(emb.numpy())
            labels.extend(classes.numpy())
    return np.vstack(embeddings), np.array(labels)



train_dataset = MyDataset(root_dir='./data/d1234')
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=False)
test_dataset = MyDataset(root_dir='./data/d5678')
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)

# train_ratio = 0.01
# dataset = MyDataset(root_dir='./data/d1234')
# train_size = int(train_ratio * len(dataset))
# test_size = len(dataset) - train_size
# train_dataset, test_dataset = random_split(dataset, [train_size, test_size])
# train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
# test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)



if __name__ == '__main__':
    Tri_model_path = './trained_net/model_2_TRI.pth'
    Tri_model = model_2_tri()
    Tri_model.load_state_dict(torch.load(Tri_model_path))
    Tri_model.eval()
    Train_embeddings, Train_labels = get_embeddings(Tri_model, train_loader)
```

```
    k = 1
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(Train_embeddings, Train_labels)
    # testset_eval(Tri_model, test_loader, knn)
    predict()
```

Model Test 1

```
import torch
from torch.utils.data import Dataset, DataLoader, random_split
import seaborn as sns
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import numpy as np

from tools import MyDataset
from net import model_1, model_2


dataset = MyDataset(root_dir='./data/d5678')
loader = DataLoader(dataset, batch_size=256, shuffle=False)

num_classes = 8
model = model_2(num_classes)


model_save_path = './trained_net/model_2_CE.pth'
model.load_state_dict(torch.load(model_save_path))


correct = 0
total = 0
model.eval()
all_predicted = []
all_true_labels = []
with torch.no_grad():
    for inputs, labels in loader:
        inputs = inputs.unsqueeze(1)
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        all_predicted.extend(predicted.tolist())
        all_true_labels.extend(labels.tolist())
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

```
print(f'Accuracy of the model on the new data: {100 * correct / total}%')
confusion = confusion_matrix(all_true_labels, all_predicted)
plt.figure(figsize=(num_classes, num_classes))
sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", xticklabels=True,
yticklabels=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

CE Loss Training

```
import torch
from torch.utils.data import DataLoader, random_split
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

from tools import MyDataset, EarlyStopping
from net import model_1, model_2



def setup_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)
    torch.backends.cudnn.deterministic = True
setup_seed(1)



dataset = MyDataset(root_dir='./data/d1234')

train_ratio = 0.8
train_size = int(train_ratio * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
```

```
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=256, shuffle=False)


device = torch.device("cuda")

num_classes = 8
model = model_2(num_classes).to(device)
model_save_path = './trained_net/model_2_CE.pth'


criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, weight_decay=0.0001)

decayRate = 0.987
lr_scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer=optimizer,
gamma=decayRate)
# lr_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer=optimizer, T_max=32)
early_stopping = EarlyStopping(patience=20, verbose=True, path=model_save_path)

num_epochs = 200


def adjust_learning_rate(optimizer, epoch, warmup_epochs=20, base_lr=1e-4):
    """Adjusts learning rate for each epoch based on the given parameters"""
    if epoch <= warmup_epochs:
        lr = base_lr * (epoch / warmup_epochs)
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr


def train():
    with open('results.txt', 'w') as file:
        for epoch in range(num_epochs):

            train_losses = []
            test_losses = []
            train_accs = []
            test_accs = []

            model.train()
            train_loss = 0.0
            correct_train = 0
```

```
            total_train = 0
            for inputs, labels in train_loader:
                inputs, labels = inputs.to('cuda'), labels.to('cuda')
                inputs = inputs.unsqueeze(1)
                optimizer.zero_grad()
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()

                train_loss += loss.item() * inputs.size(0)
                _, predicted = torch.max(outputs.data, 1)
                total_train += labels.size(0)
                correct_train += (predicted == labels).sum().item()
            train_accuracy = correct_train / total_train
            average_train_loss = train_loss / len(train_loader)

            correct_test = 0
            total_test = 0
            test_loss = 0.0
            model.eval()
            with torch.no_grad():
                for inputs, labels in val_loader:
                    inputs, labels = inputs.to('cuda'), labels.to('cuda')
                    inputs = inputs.unsqueeze(1)
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    test_loss += loss.item() * inputs.size(0)
                    _, predicted = torch.max(outputs.data, 1)
                    total_test += labels.size(0)
                    correct_test += (predicted == labels).sum().item()

            test_accuracy = correct_test / total_test
            average_test_loss = test_loss / len(val_loader)

            file.write(
                f'{epoch + 1} {average_train_loss:.4f} {train_accuracy * 100:.2f}
{average_test_loss:.4f} {test_accuracy * 100:.2f}\n')

            print(f'Epoch {epoch + 1}/{num_epochs}')
            print(f'Train Loss: {average_train_loss:.4f}, Train Accuracy: {train_accuracy *
100:.2f}%')
            print(f'Test Loss: {average_test_loss:.4f}, Test Accuracy: {test_accuracy * 100:.2f}%')
```

```
            train_losses.append(average_train_loss)
            test_losses.append(average_test_loss)
            train_accs.append(train_accuracy)
            test_accs.append(average_test_loss)

            lr_scheduler.step()

            early_stopping(average_test_loss, model, epoch + 1)
            if early_stopping.early_stop:
                print(f"Early stopping triggered at epoch {early_stopping.best_epoch}.")
                print(
                    f"Loading best model from epoch {early_stopping.best_epoch} with loss
{early_stopping.val_loss_min}")
                break

def test():
    model.load_state_dict(torch.load(model_save_path))
    correct = 0
    total = 0
    model.eval()
    all_predicted = []
    all_true_labels = []
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to('cuda'), labels.to('cuda')
            inputs = inputs.unsqueeze(1)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            all_predicted.extend(predicted.tolist())
            all_true_labels.extend(labels.tolist())
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the model on the new data: {100 * correct / total}%')
    confusion = confusion_matrix(all_true_labels, all_predicted)
    plt.figure(figsize=(num_classes, num_classes))
    sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", xticklabels=True,
yticklabels=True)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()
```

```
if __name__ == '__main__':
    train()
    test()
```

Models

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary


class model_1(nn.Module):
    def __init__(self, num_classes):
        super(model_1, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5, stride=1,
padding=2)
        self.bn1 = nn.BatchNorm1d(32)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3, stride=1,
padding=1)
        self.bn2 = nn.BatchNorm1d(64)
        self.conv3 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, stride=1,
padding=1)
        self.bn3 = nn.BatchNorm1d(128)
        self.adaptive_pool = nn.AdaptiveAvgPool1d(1)
        self.fc1 = nn.Linear(128, 512)
        self.dropout = nn.Dropout(0.25)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.adaptive_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.normalize(x, p=2, dim=1)
        x = self.dropout(x)
        x = self.fc2(x)
        return x


class model_1_feature(nn.Module):
```

```
    def __init__(self, num_classes):
        super(model_1_feature, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5, stride=1,
padding=2)
        self.bn1 = nn.BatchNorm1d(32)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3, stride=1,
padding=1)
        self.bn2 = nn.BatchNorm1d(64)
        self.conv3 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, stride=1,
padding=1)
        self.bn3 = nn.BatchNorm1d(128)
        self.adaptive_pool = nn.AdaptiveAvgPool1d(1)
        self.fc1 = nn.Linear(128, 512)
        self.dropout = nn.Dropout(0.25)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.adaptive_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.normalize(x, p=2, dim=1)
        return x



class model_1_tri(nn.Module):
    def __init__(self):
        super(model_1_tri, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5, stride=1,
padding=2)
        self.bn1 = nn.BatchNorm1d(32)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3, stride=1,
padding=1)
        self.bn2 = nn.BatchNorm1d(64)
        self.conv3 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, stride=1,
padding=1)
        self.bn3 = nn.BatchNorm1d(128)
        self.adaptive_pool = nn.AdaptiveAvgPool1d(1)
        self.fc1 = nn.Linear(128, 512)

    def forward(self, x):
```

```
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.adaptive_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.normalize(x, p=2, dim=1)
        return x



class ResBlock_model2(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResBlock_model2, self).__init__()

        self.layer = nn.Sequential(
            nn.Conv1d(in_channels, out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(out_channels),
            nn.ReLU(),
            nn.Conv1d(out_channels, out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(out_channels)
        )

        if in_channels != out_channels:
            self.res_layer = torch.nn.Conv1d(in_channels, out_channels, kernel_size=1,
stride=1, padding=0)
        else:
            self.res_layer = None

    def forward(self, x):
        if self.res_layer is not None:
            residual = self.res_layer(x)
        else:
            residual = x
        return F.relu(self.layer(x) + residual)



class model_2(nn.Module):
    def __init__(self, num_classes):
        super(model_2, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=7, stride=2,
padding=3)
        self.block1 = ResBlock_model2(32, 32)
        self.block2 = ResBlock_model2(32, 32)
```

```
        self.block3 = ResBlock_model2(32, 64)
        self.block4 = ResBlock_model2(64, 64)

        self.avgpool = nn.AdaptiveAvgPool1d(1)
        self.dense = nn.Linear(64, 512)

        self.dropout = nn.Dropout(0.25)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.dense(x)
        x = F.normalize(x, p=2, dim=1)    # L2 normalization
        x = self.dropout(x)
        x = self.fc2(x)
        return x



class model_2_feature(nn.Module):
    def __init__(self, num_classes):
        super(model_2_feature, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=7, stride=2,
padding=3)
        self.block1 = ResBlock_model2(32, 32)
        self.block2 = ResBlock_model2(32, 32)
        self.block3 = ResBlock_model2(32, 64)
        self.block4 = ResBlock_model2(64, 64)

        self.avgpool = nn.AdaptiveAvgPool1d(1)
        self.dense = nn.Linear(64, 512)

        self.dropout = nn.Dropout(0.25)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.block1(x)
```

```
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.dense(x)
        x = F.normalize(x, p=2, dim=1)  # L2 normalization
        return x



class model_2_tri(nn.Module):
    def __init__(self):
        super(model_2_tri, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=7, stride=2,
padding=3)
        self.block1 = ResBlock_model2(32, 32)
        self.block2 = ResBlock_model2(32, 32)
        self.block3 = ResBlock_model2(32, 64)
        self.block4 = ResBlock_model2(64, 64)

        self.avgpool = nn.AdaptiveAvgPool1d(1)
        self.dense = nn.Linear(64, 512)

    def forward(self, x):
        x = self.conv1(x)
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.dense(x)
        x = F.normalize(x, p=2, dim=1)  # L2 normalization
        return x


if __name__ == '__main__':
    input_size = (1, 410)
    model = model_2(8).cuda()
    summary(model, input_size)
```

Tools

```
import h5py
import os
import re
import torch
from torch.utils.data import Dataset
import librosa
import numpy as np
from torch import nn
from torch.nn.parameter import Parameter
from torch.nn import init
import torch.nn.functional as F
import math


class MyDataset(Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.file_paths = []
        self.labels = []

        for filename in os.listdir(root_dir):
            if filename.endswith('.hdf5'):
                label = int(re.search(r'tag(\d+)', filename).group(1)) - 1
                file_path = os.path.join(root_dir, filename)
                with h5py.File(file_path, 'r') as hdf:
                    num_signals = len(hdf['signals'])
                    self.file_paths.extend([file_path] * num_signals)
                    self.labels.extend([label] * num_signals)

    def __len__(self):
        return len(self.file_paths)

    def __getitem__(self, idx):
        file_path = self.file_paths[idx]
        label = self.labels[idx]
        with h5py.File(file_path, 'r') as hdf:
            data = hdf['signals'][idx % len(hdf['signals'])]
        data_tensor = torch.tensor(data, dtype=torch.float32)
        label_tensor = torch.tensor(label, dtype=torch.long)
        return data_tensor, label_tensor
```

```
class EarlyStopping:
    def __init__(self, patience=10, verbose=False, path='./checkpoint.pth'):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.val_loss_min = float('inf')
        self.path = path
        self.early_stop = False
        self.best_epoch = 0

    def __call__(self, val_loss, model, epoch):
        if val_loss < self.val_loss_min:
            self.save_checkpoint(val_loss, model, epoch)
            self.val_loss_min = val_loss
            self.best_epoch = epoch
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
                if self.verbose:
                    print(f"EarlyStopping: Stop training —— The loss has not improved
after {self.patience} epochs.")

    def save_checkpoint(self, val_loss, model, epoch):
        if self.verbose:
            print(f'Loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), self.path)



class TripletDataset(Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.data = []
        self.labels = []
        self.label_to_indices = {}
        for filename in os.listdir(root_dir):
            if filename.endswith('.hdf5'):
                label = int(re.search(r'tag(\d+)', filename).group(1)) - 1
                file_path = os.path.join(root_dir, filename)
                with h5py.File(file_path, 'r') as hdf:
                    signals = hdf['signals'][:]
                    for signal in signals:
```

```
                idx = len(self.data)
                self.data.append(signal)
                self.labels.append(label)
                if label not in self.label_to_indices:
                    self.label_to_indices[label] = []
                self.label_to_indices[label].append(idx)
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        data_anchor = self.data[idx]
        label = self.labels[idx]
        positive_index = idx
        while positive_index == idx:
            positive_index = np.random.choice(self.label_to_indices[label])
        negative_label = np.random.choice(list(set(self.labels) - {label}))
        negative_index = np.random.choice(self.label_to_indices[negative_label])
        data_positive = self.data[positive_index]
        data_negative = self.data[negative_index]
        data_anchor = torch.tensor(data_anchor, dtype=torch.float32)
        data_positive = torch.tensor(data_positive, dtype=torch.float32)
        data_negative = torch.tensor(data_negative, dtype=torch.float32)
        return data_anchor, data_positive, data_negative, label


# https://github.com/h1yuol/pytorch-triplet-loss
class TripletLoss(nn.Module):
    def __init__(self, margin=0.2):
        super(TripletLoss, self).__init__()
        self.margin = margin
        self.ranking_loss = nn.MarginRankingLoss(margin=margin)

    def forward(self, inputs, targets):

        n = inputs.size(0)  # Batch Size

        dist = torch.pow(inputs, 2).sum(dim=1, keepdim=True).expand(n, n)
        dist = dist + dist.t()
        dist.addmm_(inputs, inputs.t(), beta=1, alpha=-2)
        dist = dist.clamp(min=1e-12).sqrt()  # for numerical stability

        # For each anchor, find the hardest positive and negative
        mask = targets.expand(n, n).eq(targets.expand(n, n).t())
        dist_ap, dist_an = [], []
        for i in range(n):
```

```
            hard_positive = dist[i][mask[i]].max().unsqueeze(0)
            hard_negative = dist[i][mask[i] == 0].min().unsqueeze(0)
            dist_ap.append(hard_positive)
            dist_an.append(hard_negative)

        dist_ap = torch.cat(dist_ap)
        dist_an = torch.cat(dist_an)

        y = torch.ones_like(dist_an)
        loss = self.ranking_loss(dist_an, dist_ap, y)
        return loss



# https://github.com/idstcv/SoftTriple
class SoftTriple(nn.Module):
    def __init__(self, la, gamma, tau, margin, dim, cN, K):
        # margin: Triplet Margin / dim: dimension of embeddings / cN: Class Numbers / K:
number of centers
        super(SoftTriple, self).__init__()
        self.la = la
        self.gamma = 1./gamma
        self.tau = tau
        self.margin = margin
        self.cN = cN
        self.K = K
        self.fc = Parameter(torch.Tensor(dim, cN*K))
        self.weight = torch.zeros(cN*K, cN*K, dtype=torch.bool).cuda()
        for i in range(0, cN):
            for j in range(0, K):
                self.weight[i*K+j, i*K+j+1:(i+1)*K] = 1
        init.kaiming_uniform_(self.fc, a=math.sqrt(5))

        self.to(torch.device('cuda'))
        return

    def forward(self, input, target):
        centers = F.normalize(self.fc, p=2, dim=0)
        simInd = input.matmul(centers)
        simStruc = simInd.reshape(-1, self.cN, self.K)
        prob = F.softmax(simStruc*self.gamma, dim=2)
        simClass = torch.sum(prob*simStruc, dim=2)
        marginM = torch.zeros(simClass.shape).cuda()
        marginM[torch.arange(0, marginM.shape[0]), target] = self.margin
```

```
        lossClassify = F.cross_entropy(self.la*(simClass-marginM), target)
        if self.tau > 0 and self.K > 1:
            simCenter = centers.t().matmul(centers)
            reg = torch.sum(torch.sqrt(2.0+1e-5-
2.*simCenter[self.weight]))/(self.cN*self.K*(self.K-1.))
            return lossClassify+self.tau*reg
        else:
            return lossClassify
```

Visualization

```
import torch
import numpy as np
from torch.utils.data import DataLoader
from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from scipy.spatial.distance import cdist
from sklearn.metrics import confusion_matrix
import seaborn as sns

from tools import MyDataset
from net import model_2_feature, model_2_tri



test_dataset = MyDataset(root_dir='./data/d5678')
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

train_dataset = MyDataset(root_dir='./data/d1234')
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=False)


CE_model_path = './trained_net/model_2_ce.pth'
Tri_model_path = './trained_net/model_2_ST.pth'


num_classes = 8
CE_model = model_2_feature(num_classes).cuda()
Tri_model = model_2_tri().cuda()

CE_model.load_state_dict(torch.load(CE_model_path))
Tri_model.load_state_dict(torch.load(Tri_model_path))
```

```
CE_model.eval()
Tri_model.eval()

def extract_features_and_labels(model, loader):
    features = []
    labels = []

    with torch.no_grad():
        for data, target in loader:
            data = data.unsqueeze(1).cuda()
            output = model(data)
            # inputs = torch.flatten(data, 1)
            features.extend(output.cpu().numpy())

            labels.extend(target.numpy())

    return np.array(features), np.array(labels)


CE_train_features, CE_train_labels = extract_features_and_labels(CE_model, train_loader)
Tri_train_features, Tri_train_labels = extract_features_and_labels(Tri_model, train_loader)
CE_test_features, CE_test_labels = extract_features_and_labels(CE_model, test_loader)
Tri_test_features, Tri_test_labels = extract_features_and_labels(Tri_model, test_loader)

CE_train_features = CE_train_features.squeeze()
Tri_train_features = Tri_train_features.squeeze()


def visualize_features_subplot(CE_features, Tri_features, labels):
    tsne = TSNE(n_components=2, random_state=0)
    CE_tsne_results = tsne.fit_transform(CE_features)
    Tri_tsne_results = tsne.fit_transform(Tri_features)

    fig, axes = plt.subplots(1, 2, figsize=(16, 7))

    for i in range(num_classes):
        axes[0].scatter(CE_tsne_results[labels == i, 0], CE_tsne_results[labels == i, 1],
label=f"Tag{i+1}")
        axes[1].scatter(Tri_tsne_results[labels == i, 0], Tri_tsne_results[labels == i, 1],
label=f"Tag{i+1}")

    # axes[0].set_title("CE Model Features", fontsize=20)
    # axes[0].set_title("Raw Data Distribution", fontsize=20)
```

```
    # axes[1].set_title("Triplet Loss Model Features", fontsize=20)
    axes[0].legend()
    axes[1].legend()
    # plt.savefig('./tSNE.png', dpi=300)
    plt.show()

# visualize_features_subplot(CE_train_features, Tri_train_features, Tri_train_labels)



def calculate_recall_corrected(features, labels, num_classes):
    dists = cdist(features, features, metric='euclidean')
    np.fill_diagonal(dists, np.inf)
    recall_per_class = np.zeros(num_classes)

    for i in range(num_classes):
        class_indices = np.where(labels == i)[0]
        if len(class_indices) > 1:
            correct_retrievals = 0
            for idx in class_indices:
                nearest_idx = np.argmin(dists[idx, :])
                if labels[nearest_idx] == i:
                    correct_retrievals += 1
            recall_per_class[i] = correct_retrievals / len(class_indices)
    return recall_per_class

recall_ce_corrected = 100*calculate_recall_corrected(CE_test_features, CE_test_labels,
num_classes)
recall_tri_corrected = 100*calculate_recall_corrected(Tri_test_features, Tri_test_labels,
num_classes)

for i in range(num_classes):
    print(f"Class {i}: Corrected CE Recall = {recall_ce_corrected[i]:.3f}, Corrected Triplet Recall =
{recall_tri_corrected[i]:.3f}")

def plot_recall_comparison(recall_ce, recall_tri, num_classes):
    labels = range(num_classes)
    x = np.arange(len(labels))
    width = 0.35
    fig, ax = plt.subplots()
    rects1 = ax.bar(x - width/2, recall_ce, width, label='CE Model')
    rects2 = ax.bar(x + width/2, recall_tri, width, label='Triplet Model')
    ax.set_ylabel('Recall')
    ax.set_title('Recall by class and model type')
```

```
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()

    def autolabel(rects):
        for rect in rects:
            height = rect.get_height()
            ax.annotate('{}'.format(round(height,2)),
                        xy=(rect.get_x() + rect.get_width() / 2, height),
                        xytext=(0, 3),    # 3 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom')
    autolabel(rects1)
    autolabel(rects2)
    fig.tight_layout()
    plt.show()
plot_recall_comparison(recall_ce_corrected, recall_tri_corrected, num_classes)
```

Print Training Results

```
import matplotlib.pyplot as plt

file_path = 'results.txt'
Triplet = True

if Triplet:
    epochs = []
    train_losses = []
    val_total_losses = []

    with open(file_path, 'r') as file:
        next(file)
        for line in file:
            epoch, train_loss, val_total_loss = line.strip().split(',')
            epochs.append(int(epoch))
            train_losses.append(float(train_loss))
            val_total_losses.append(float(val_total_loss))

    plt.figure(figsize=(10, 8))
    plt.plot(epochs, train_losses, label='Train Loss')
    plt.plot(epochs, val_total_losses, label='Validation Total Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Total Loss')
```

```
        plt.legend()
        plt.tight_layout()
        plt.show()

else:
        epochs = []
        train_losses = []
        val_losses = []
        train_accs = []
        val_accs = []
        with open(file_path, 'r') as file:
                for line in file:
                        epoch, train_loss, train_acc, val_loss, val_acc = line.split()
                        epochs.append(int(epoch))
                        train_losses.append(float(train_loss))
                        val_losses.append(float(val_loss))
                        train_accs.append(float(train_acc))
                        val_accs.append(float(val_acc))

        plt.figure(figsize=(10, 15))
        plt.subplot(2, 1, 1)
        plt.plot(epochs, train_losses, label='Train Loss')
        plt.plot(epochs, val_losses, label='Validation Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Training and Validation Loss')
        plt.legend()

        plt.subplot(2, 1, 2)
        plt.plot(epochs, train_accs, label='Train Accuracy')
        plt.plot(epochs, val_accs, label='Validation Accuracy')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy (%)')
        plt.title('Training and Validation Accuracy')
        plt.legend()
        plt.show()
```