Clemson University
TigerPrints

All Dissertations Dissertations

8-2023

# Generative Neural Network-Based Defense Methods Against Cyberattacks for Connected and Autonomous Vehicles

M Sabbir Salek
*Clemson University*, msalek@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

 Part of the Civil Engineering Commons, and the Transportation Engineering Commons

## Recommended Citation

Salek, M Sabbir, "Generative Neural Network-Based Defense Methods Against Cyberattacks for Connected and Autonomous Vehicles" (2023). *All Dissertations*. 3373.
https://tigerprints.clemson.edu/all_dissertations/3373

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

# GENERATIVE NEURAL NETWORK-BASED DEFENSE METHODS AGAINST CYBERATTACKS FOR CONNECTED AND AUTONOMOUS VEHICLES

---

A Dissertation
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Civil Engineering

---

by
M Sabbir Salek
August, 2023

---

Accepted by:
Dr. Mashrur Chowdhury, Committee Chair
Dr. Yao Wang
Dr. Feng Luo
Dr. Long Cheng
Dr. Sakib Mahmud Khan

# ABSTRACT

The rapid advancement of communication and artificial intelligence technologies is propelling the development of connected and autonomous vehicles (CAVs), revolutionizing the transportation landscape. However, increased connectivity and automation also present heightened potential for cyber threats. Recently, the emergence of generative neural networks (NNs) has unveiled a myriad of opportunities for complementing CAV applications, including generative NN-based cybersecurity measures to protect the CAVs in a transportation cyber-physical system (TCPS) from known and unknown cyberattacks. The goal of this dissertation is to explore the utility of the generative NNs for devising cyberattack detection and mitigation strategies for CAVs. To this end, the author developed (i) a hybrid quantum-classical restricted Boltzmann machine (RBM)-based framework for in-vehicle network intrusion detection for connected vehicles and (ii) a generative adversarial network (GAN)-based defense method for the traffic sign classification system within the perception module of autonomous vehicles. The author evaluated the hybrid quantum-classical RBM-based intrusion detection framework on three separate real-world Fuzzy attack datasets and compared its performance with a similar but classical-only approach (i.e., a classical computer-based data preprocessing and RBM training). The results showed that the hybrid quantum-classical RBM-based intrusion detection framework achieved an average intrusion detection accuracy of 98%, whereas the classical-only approach achieved an average accuracy of 90%. For the second study, the author evaluated the GAN-based adversarial defense method for traffic sign classification against different white-box adversarial attacks, such as the fast gradient sign

method, the DeepFool, the Carlini and Wagner, and the projected gradient descent attacks. The author compared the performance of the GAN-based defense method with several traditional benchmark defense methods, such as Gaussian augmentation, JPEG compression, feature squeezing, and spatial smoothing. The findings indicated that the GAN-based adversarial defense method for traffic sign classification outperformed all the benchmark defense methods under all the white-box adversarial attacks the author considered for evaluation. Thus, the contribution of this dissertation lies in utilizing the generative ability of existing generative NNs to develop novel high-performing cyberattack detection and mitigation strategies that are feasible to deploy in CAVs in a TCPS environment.

**Keywords**: Artificial Intelligence, Generative neural network, Cybersecurity, Defense method, Connected vehicle, Autonomous vehicle, Quantum AI, Controller area network, and Traffic sign classification

# DEDICATION

I wholeheartedly dedicate my Ph.D. dissertation to my beloved father, Md. Shahidul Islam, my lovely mother, Shoheli Jahan, and my closest friend and life partner, Rizwana Akter. Their unwavering love and support have been instrumental in guiding me through the challenges of my Ph.D. journey. They are my constant source of inspiration, aspiration, and happiness.

# ACKNOWLEDGMENTS

I seize this moment with profound gratitude, an opportunity to extend my heartfelt appreciation to all whose influence, whether direct or indirect, has paved the path to this remarkable achievement. My words are insufficient to express my indebtedness to the exceptional mentorship and unwavering support of Dr. Mashrur Chowdhury. Gratitude abounds as I acknowledge the invaluable contributions of my esteemed Ph.D. committee members: Dr. Yao Wang, Dr. Feng Luo, Dr. Long Cheng, and Dr. Sakib Mahmud Khan. Their scholarly insights and constructive feedback have propelled me throughout this academic excellence. I have been fortunate to be surrounded by remarkable colleagues and friends at Clemson University, whose encouragement and mental fortitude bolstered me through every obstacle I encountered. This transformative odyssey toward my Ph.D. has been nothing short of an extraordinary expedition.

This work is based upon the work partially supported by the National Center for Transportation Cybersecurity and Resiliency (TraCR) (a U.S. Department of Transportation National University Transportation Center) headquartered at Clemson University, Clemson, South Carolina, USA. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the author and do not necessarily reflect the views of TraCR, and the U.S. Government assumes no liability for the contents or use thereof.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

# CHAPTER ONE

# INTRODUCTION

## 1.1 Background and Motivation

The global transportation sector is undergoing a swift and remarkable transformation, driven by groundbreaking advancements in information and communication technologies ("ICT for Transport," n.d.). Yesterday's traditional transportation systems are now being complemented by the evolving realm of transportation cyber-physical systems (TCPS). These systems seamlessly connect diverse modes of physical transportation systems with cutting-edge cyber systems to leverage accelerated and highly efficient computational processes, as well as robust and fortified data storage capabilities (McGregor et al., 2018). This paradigm shift has been facilitated by recent breakthroughs in the field of artificial intelligence (AI), which have empowered TCPS to seamlessly integrate automation into a myriad of applications that were previously deemed arduous or perilous for human involvement. Among these remarkable advancements, the rapid progress in connected and autonomous vehicles (CAVs) stands out prominently.

As the name suggests, CAVs encompass two aspects, i.e., connectivity and autonomy ("Connected/Automated Vehicles," n.d.). If a vehicle is equipped with an onboard communication device that enables the vehicle to communicate with the other neighboring vehicles via vehicle-to-vehicle (V2V) communication, or with infrastructures via vehicle-to-infrastructure (V2I) communication, or with a diverse range of other entities

(e.g., pedestrians, bicyclists or other vulnerable road users) via vehicle-to-everything (V2X) communication, then it is referred to as a connected vehicle (CV). Conversely, an autonomous vehicle (AV) is characterized by a suite of sensors (e.g., cameras, ultrasonic sensors, radio detection and ranging or Radar sensors, and light detection and ranging or LiDAR sensors), sophisticated software, and supporting hardware that collectively enables the vehicle to undertake driving tasks autonomously, without the involvement of a human driver. When a vehicle combines both these capabilities—connectivity and automation—it is regarded as a CAV, signifying a remarkable convergence of technological advancements.

However, as the reliance on connectivity and autonomy intensifies, so does the vulnerability to potential cyber threats (Sun et al., 2022). Attackers now have unprecedented opportunities to exploit these systems, jeopardizing the confidentiality, integrity, and availability of the TCPS. The consequences range from the theft of sensitive information and invasion of privacy to the creation and dissemination of false/altered information and the injection of malicious data and software into the TCPS, such as CAVs, thereby posing significant risks to human safety, property, security, and privacy. As a result, there is a growing imperative to develop attack detection and defense strategies to enhance the security of the evolving TCPS, including CAVs, against both known and unknown cyberattacks. The focus lies on detecting cyberattacks as well as fortifying these systems to be inherently resilient against attacks, ensuring their robustness and ability to withstand potential breaches.

The current surge in AI, particularly in the realm of generative neural networks (NNs), has opened new windows for data-driven approaches to attack detection and mitigation (Sarker, 2021). Today, generative NNs possess the remarkable capacity to generate diverse forms of content, ranging from textual compositions to intricate images, captivating music, and even compelling artworks. Harnessing the immense generative potential of these NNs, researchers have begun developing strategies for detecting and mitigating cyberattacks that were inconceivable merely a decade ago. By training generative NNs on authentic, non-attack datasets, they can be effectively employed to identify and counteract cyber threats with remarkable ease and accuracy. This represents a significant leap forward in the field of cybersecurity, leveraging the creative abilities of generative NNs to safeguard against existing and emerging threats.

In this dissertation, the focus is to utilize such generative NNs to develop innovative attack detection and mitigation strategies for CAVs operating in a TCPS environment. In particular, the research endeavors presented in this dissertation are directed to (i) develop a generative NN-based cyberattack detection strategy for CV's in-vehicle network that connects multiple in-vehicle sensors and systems, and (ii) develop a generative NN-based defense method to protect the perception module of an AV from cyberattacks that is responsible for perceiving and comprehending the surrounding environment of an AV to facilitate precise navigate through roadways.

### 1.2 Research Hypotheses

The hypotheses of the research endeavors of this dissertation are as follows,

1. A hybrid quantum-classical generative NN-based intrusion detection system (IDS) performs better in detecting intrusions in the in-vehicle network of a CV compared to a classical-only IDS in a TCPS environment, and
2. A generative NN-based attack-resilient AV traffic sign classification system performs better in mitigating the effects of cyberattacks compared to other deep NN-based defense methods.

## 1.3 Research Objectives

The objectives of the research endeavors of this dissertation are as follows,

1. To explore how the generative ability of generative NNs can be utilized to devise an in-vehicle network intrusion detection framework for CVs in a TCPS environment, and
2. To explore how the generative ability of generative NNs can be utilized to develop an attack-resilient traffic sign classification system that requires no prior knowledge about adversarial attack models and samples.

CHAPTER TWO

# HYBRID QUANTUM-CLASSICAL RESTRICTED BOLTZMANN MACHINE-BASED IN-VEHICLE CONTROLLER AREA NETWORK INTRUSION DETECTION FOR CONNECTED VEHICLES

## 2.1 Background and Motivation

Controller area network (CAN) is a de facto standard for the broadcast-based in-vehicle message communication system to provide a dedicated, reliable, and efficient communication channel for all in-vehicle connected electronic control units or ECUs. Although CAN is widely popular among in-vehicle networks, it lacks common security features, such as authentication. Attackers can easily inject false messages into a vehicle's CAN via the on-board diagnostic (OBD-II) port, the infotainment system, or wireless communication. Thus, intrusion detection system (IDS) has been widely studied in recent years due to the inherent vulnerabilities of CAN communication to cyberattacks (Lokman et al., 2019; Wu et al., 2020; Young et al., 2019a). Researchers presented various IDSs based on different machine learning and deep learning techniques (Lo et al., 2022; Moulahi et al., 2021; Song et al., 2020; Minawi et al., 2020; Hossain et al., 2020). Besides the variation in machine learning or deep learning techniques, different features and their combinations have been attempted by researchers to improve CAN intrusion detection accuracy. Some common features used in the existing studies include message timing (e.g., message frequency/rate and interval) (Moore et al., 2017), signatures (e.g., ID, time interval, and correlation) (Jin et al., 2021), and anomaly (Lo et al., 2022).

Beyond the classical computer-based machine learning and deep learning algorithms, quantum computing can be used for CAN intrusion detection to detect the increasing number of cyberattacks. Dong et al., 2022 presented a quantum beetle swarm optimization-based extreme learning machine or ELM (i.e., a neural network (NN) where randomly selected input weights and hidden layer biases are utilized for faster learning) for network intrusion detection. The ELM in (Dong et al., 2022) provided higher detection accuracy and faster convergence than several other classical intrusion detection systems, such as backpropagation, support vector machine, improved rough ELM, particle swarm optimization, and genetic algorithm optimization-based ELMs. Chen et al., 2020 applied quantum computing for k-means clustering combined with a quantum-inspired ant lion optimization algorithm for intrusion detection. Their approach improved the convergence of k-means clustering to the global optimal solution. Caivano et al., 2022 presented a quantum annealing or QA-based IDS for CAN that achieved similar detection accuracy for denial of service and fuzzy attacks as of a classical classification technique with significantly shorter training and prediction time than the classical technique.

This study presents a hybrid quantum-classical CAN intrusion detection framework that utilizes a classical computer for data preprocessing to generate CAN images with embedded labels and a quantum computer for restricted Boltzmann machine or RBM-based CAN image reconstruction and classification to detect CAN intrusions. A restricted Boltzmann machine or RBM is a widely used energy-based generative stochastic NN model. The training process of an RBM can be done using different algorithms, such as contrastive divergence (CD) (Hinton, 2012) and quantum annealing (QA) (Adachi and

Henderson, 2015). QA provides more accurate gradient estimates for training RBMs compared to CD-based training for problems with high energy gaps between modes, as shown by Korenkevych et al., 2016. Dixit et al., 2021 trained an RBM model using D-Wave 2000Q QA with 64 visible and 64 hidden units, a task difficult to achieve using a gate-based approach. Such QA-based training can also be utilized for training RBM models to detect intrusions in an in-vehicle CAN, which is the motivation for this study.

### 2.2 Contribution

In a transportation cyber-physical system or TCPS environment, classical and quantum computers can be used together in a hybrid fashion for CAN intrusion detection. For example, Islam et al., 2022 presented a hybrid quantum-classical NN-based framework for CAN intrusion detection that outperformed both the classical-only and quantum-only approaches by overcoming the limitations of each of them. However, to the best of the author's knowledge, a hybrid approach of a classical NN and a quantum RBM has not been undertaken for CAN intrusion detection yet. Besides, existing studies on NN-based CAN IDS do not consider embedding labels directly into the corresponding CAN images to leverage the image generation capability of generative NNs for an image classification-based CAN IDS. Utilizing the QA-based training of an RBM, which enables sampling from the original probability distribution of the model, the CAN image (embedded with dedicated labeling pixels) reconstruction-based CAN intrusion detection framework offers more efficient learning (i.e., faster convergence with high detection accuracy) compared to the existing generative NN-based CAN IDSs. Thus, this study contributes to the existing body of CAN IDS literature by presenting a hybrid quantum-classical framework for CAN

intrusion detection leveraging the image generation capability of generative NNs to reconstruct the embedded labels in CAN images, which can then be used for image classification-based CAN intrusion detection.

### 2.3 Related Work

CAN intrusion detection has been widely studied by researchers in recent years because of the inherent vulnerabilities of CAN communication due to its broadcast-based nature. As a result, the existing body of literature is quite vast, and there are also several survey studies on CAN intrusion detection systems (Buscemi et al., 2023; Jo and Choi, 2022; Lampe and Meng, 2023; Lokman et al., 2019; Rajapaksha et al., 2023; Young et al., 2019b). Since the author developed a hybrid quantum-classical framework that utilizes a generative NN, in this section, the author explicitly focuses on reviewing studies that used generative NN models for CAN intrusion detection. The studies reviewed here are presented in chronological order.

Seo et al., 2018 developed a generative adversarial network (GAN)-based IDS for in-vehicle networks that is able to detect unknown attacks while using only normal data (i.e., non-attack data) for training. The generator in their GAN-based IDS generates fake CAN images to train the discriminator to distinguish between normal and fake CAN images. The authors in (Seo et al., 2018) evaluated their GAN-based IDS for denial of service (DoS), FUZZY, RPM, and GEAR attack datasets and obtained 97.9%, 98%, 98%, and 96.2% accuracies, respectively.

Xie et al., 2021 developed a GAN-based CAN IDS utilizing an enhanced GAN model to overcome the limitation of generating rough CAN message blocks in other GAN-

based IDSs. The authors in (Xie et al., 2021) tested their CAN IDS against DoS, injection, masquerade, and data tampering attacks and achieved approximately 99% precision, recall, and F1 score for the tested attack types.

Nam et al., 2021 developed a generative pretrained transformer (GPT)-based CAN IDS that can learn normal CAN ID sequences to detect any small changes in the sequence due to an attack. The authors used two GPT NNs arranged in a bi-directional manner to learn both past and future CAN ID sequences. The authors in (Nam et al., 2021) evaluated their CAN IDS for flooding, spoofing, replay, and fuzzing attacks, which showed a minimum 95% attack detection F-measure.

Zhang et al., 2021 developed a CAN fuzz testing method to filter fuzzy messages using a GAN to generate fuzzy messages and an Adaptive Boosting-based detection system to detect anomalies in CAN communication due to the fuzzy message injection. The Adaptive Boosting-based anomaly monitor in (Zhang et al., 2021) was shown to be able to detect even slight anomalies in CAN communication.

Q. Zhao et al., 2022 developed a CAN IDS based on Auxiliary Classifier GAN (ACGAN) and out-of-distribution detection. Their proposed IDS consists of two stages of classifiers. In the first stage, an ACGAN-based multi-class classifier is responsible for classifying normal and known attacks and filtering out-of-distribution samples. In the second stage, a binary classifier is responsible for detecting unknown attacks from the out-of-distribution samples found in the first-stage classifier. The authors in (Q. Zhao et al., 2022) achieved an average of 99% recall, 99% precision, and 99% F1 score for DoS, fuzzy, GEAR spoofing, and RPM attack detections.

Y. Zhao et al., 2022 developed a novel CAN intrusion attack method called the same origin method execution (SOME) attack and a GAN-based CAN IDS. Their proposed CAN IDS utilizes one-hot encoding with an adopted GAN network known as GANomaly (Akcay et al., 2019). The authors in (Y. Zhao et al., 2022) tested their CAN IDS against spoofing, bus-off, masquerade, and SOME attacks and achieved a minimum of 91% and 93% detection accuracy for two test vehicles under all four types of attacks mentioned above.

Although the studies listed in this section utilize one or more generative NN models for either training their IDS or detecting in-vehicle CAN intrusions, or doing both, none of the studies have considered embedding the label as a set of pixels directly into the corresponding CAN image, and then reconstructing the CAN image using a quantum RBM for CAN intrusion detection. This study developed a hybrid quantum-classical CAN intrusion detection framework leveraging RBM's generative ability to reconstruct the embedded labeling pixels in a CAN image and then utilize the reconstructed CAN image for CAN intrusion detection.

### 2.4 Hybrid Quantum-Classical Framework for CAN Intrusion Detection

In this section, the author presents the hybrid quantum-classical framework for CAN intrusion detection using a classical computer for data preprocessing and a quantum computer for image reconstruction and classification. The steps involved in the hybrid quantum-classical framework for CAN intrusion detection are presented in Figure 2.1.

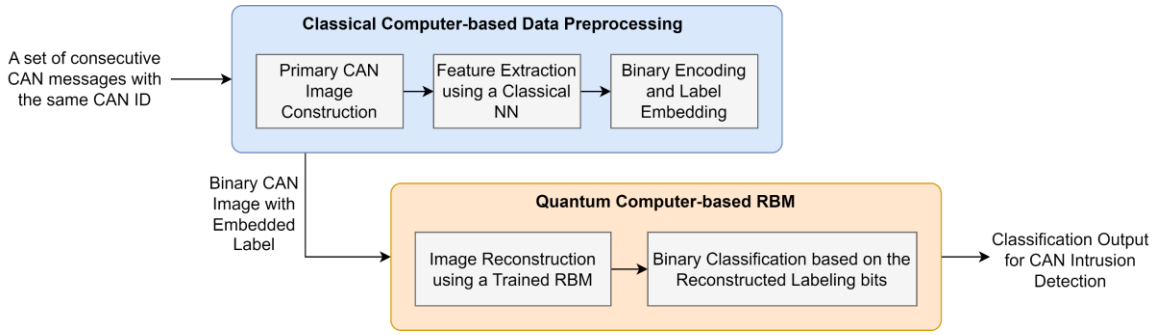



**FIGURE 2.1** A Hybrid Quantum-Classical CAN Intrusion Detection Framework.

### *2.4.1 Classical Computer-based Data Preprocessing*

CAN messages can include different data fields, such as timestamp, CAN arbitration ID (i.e., an ID allocated to an in-vehicle system based on its CAN message broadcasting priority), data length code (i.e., a code that represents the length of the data contained in a CAN message), data (i.e., a string that contains various information in an encoded format related to the system that is broadcasting the CAN message), cyclic redundancy check or CRC sequence (i.e., an error-detecting code), and acknowledgment. In the hybrid quantum-classical CAN intrusion detection framework, the author converts a set of CAN messages into a CAN image that not only contains the information included in the CAN messages but also contains a label representing whether the CAN messages are normal messages or attack messages (i.e., injected false messages by an attacker). The steps to convert the CAN messages into label-embedded CAN images are as follows, 1) primary CAN image construction, 2) feature extraction using a classical NN, and 3) binary encoding and label embedding. Figure 2.2 presents the details of data preprocessing based on a classical computer.

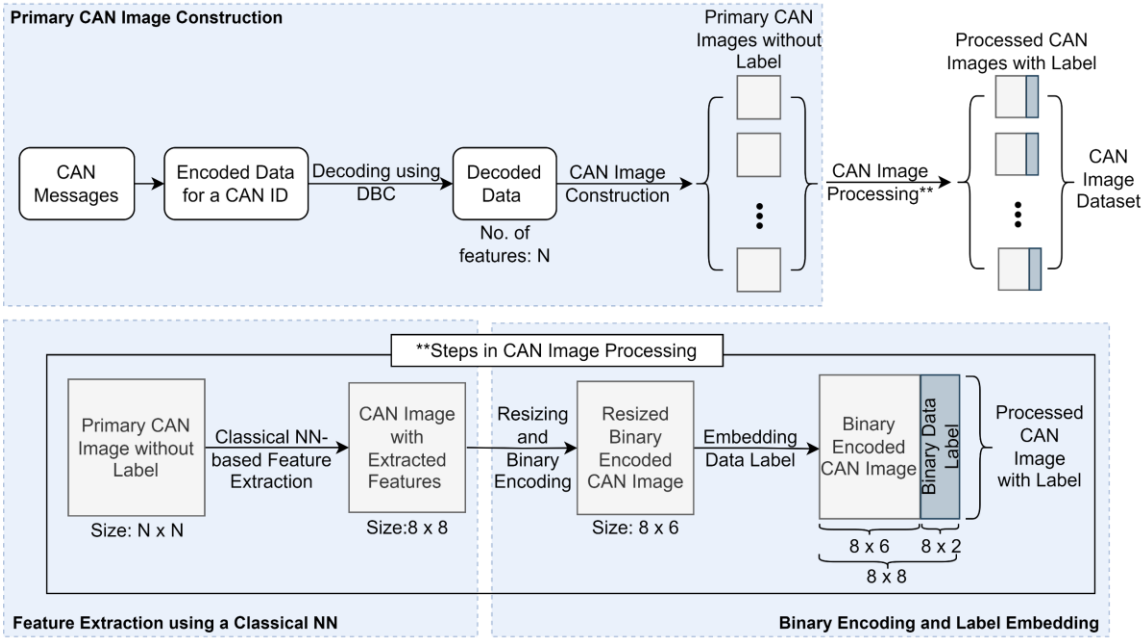



**FIGURE 2.2** Steps in Classical Computer-based Data Preprocessing.

2.4.1.1 Primary CAN Image Construction

The data contained in a CAN message is typically encoded (e.g., HEX-encoded). Thus, the first step for primary CAN image construction is to decode the encoded data using the corresponding database CAN (DBC); a DBC contains relevant information to decode CAN messages that may vary based on a vehicle's make, model, and year. Once decoded, a set of features containing data from different in-vehicle sensors is obtained. Then, the author constructs an N × N primary CAN image using a set of N consecutive CAN messages with the same CAN ID, where N is the number of decoded features present in a CAN message with that CAN ID. Thus, in a primary CAN image, a row represents a single CAN message, whereas a column represents a feature.

2.4.1.2 Feature Extraction using a Classical NN

The author uses a classical NN to extract features from an N × N primary CAN image to create an 8 × 8 secondary CAN image following the feature extraction procedure presented in (Islam et al., 2022). In a later stage, when the author utilizes a QA-based RBM, the author considers 64 neurons in each layer. Thus, the motivation to create 8 × 8 CAN images from N × N primary CAN images is to be able to map each pixel of an image to a neuron of the visible layer of an RBM, which the author will discuss in section 2.4.2. The feature extraction using a classical NN can be described as follows,

$$L_{8\times8} = L_{p-1} \bigcirc L_{p-2} \bigcirc L_{p-3} \bigcirc \ldots\ldots \bigcirc L_1 \bigcirc L_0 \quad (1)$$

$$L_n : x_{n-1} \rightarrow x_n = \phi\,(W_n x_{n-1} + v_n) \quad (2)$$

Here, $L_{8\times8}$ denotes the output of a classical NN, $p$ denotes the number of layers, $L_n$ denotes the $n^{th}$ layer of the classical NN, $x_{n-1}$ denotes the input vector of $L_n$, $x_n$ denotes the output vector of $L_n$, $W_n$ denotes the weight, $v_n$ denotes a bias vector, and $\phi$ denotes a nonlinear function. The model parameters ($W_n, d, v_n$) are to be optimized while training the classical NN.

2.4.1.3 Binary Encoding and Label Embedding

The author resize the 8 × 8 secondary CAN images into 8 × 6 reduced-size secondary CAN images to allocate two rightmost columns, i.e., a total of 16 bits of each image, for embedding the corresponding label to indicate whether the image is a normal image or an attack image. Then, binary encoding is performed on each 8 × 6 image. In a binary CAN image, each row represents a six-bit binary string: $x_m = (b_1, b_2, \ldots, b_6)$, where $b_i \in \{0,1\}\ \forall\ i = 1, 2, \ldots, 6$, and $m$ represents the row number. Each bit is a binary

representation of a pixel in an 8 × 6 reduced-size secondary CAN image, e.g., $b_1 = 1$ in $x_m$ indicates the first feature is present in the $m$-th row, whereas $b_1 = 0$ indicates the first feature is absent in the $m$-th row. Binary image thresholding with a fixed threshold value of 0.5 is used to generate a binary CAN image $x_m$ from an 8 × 6 reduced-size secondary CAN image (Islam et al., 2022). After performing binary encoding, each binary CAN image of 8 × 6 size is embedded with the corresponding image label, i.e., whether the image represents an attack image or a normal image. This embedding is either an 8 × 2 matrix of all ones when an image represents an attack image, or an 8 × 2 matrix of all zeroes when an image represents a normal image. Then, this 8 × 2 matrix is concatenated horizontally with the corresponding 8 × 6 binary CAN image giving each final processed binary CAN image with the embedded label an overall size of 8 × 8.

*2.4.2 Quantum RBM for CAN Image Reconstruction and Binary Classification*

The final processed binary CAN images with embedded labels are reconstructed by a quantum RBM and then used for binary classification based on the reconstructed dedicated bits in the images for labeling. In this framework, the author considers an adiabatic quantum computer offered by D-Wave, which is based on superconducting electronics and allows QA-based sampling ("What is Quantum Annealing? — D-Wave System Documentation documentation," n.d.) for image reconstruction and classification. In this subsection, the author starts with the motivation for using quantum computers for training RBM models and presents the details of RBM-based CAN image reconstruction.

Quantum computing utilizes the principles of quantum mechanics to process information. As opposed to classical computers that use classical bits (i.e., 0 and 1),

quantum computers use quantum bits (qubits) represented by photons, atoms, ions, etc., to process information. Besides, due to quantum phenomena, such as superposition and entanglement, quantum computing has the potential to process information at a much higher rate compared to classical computers. Unlike a classical bit that can only take a value of 0 or 1, a qubit can be in a state of 0, 1, or any combination of 0 and 1, known as superposition. A classical system with four bits can be used to represent only one out of $2^4$ or 16 combinations at once, whereas a quantum computer with four qubits can represent all 16 combinations simultaneously using superposition. On the other hand, the entanglement of two qubits refers to a quantum phenomenon that enables a quantum computer to instantaneously determine the state of an entangled qubit by only measuring the state of the other entangled qubit. Because of such uniqueness, it has been theorized over the last few decades by researchers that quantum computers could potentially solve complex problems exponentially faster than classical computers. Indeed, in 1994, Daniel Simon came up with an algorithm known as Simon's algorithm, which was among the first of its kind to prove that a quantum algorithm is capable of exponentially speeding up the computations compared to a classical computer. Peter Shor discovered Shor's algorithm the same year, which is considered one of the most famous and influential quantum algorithms so far, to factorize a given integer in polynomial time.

Quantum computing has also shown tremendous potential in speeding up complex optimization problems; for example, Stokes et al., 2020 presented a generalized optimization framework using quantum natural gradient descent, which was proved to significantly speed up an optimization problem compared to its classical counterpart. The

gradient descent-based model parameter update rule is among the most fundamental algorithms for developing most ML and DL models nowadays. However, for a complex non-linear optimization problem, gradient descent-based ML or DL model training often suffers from non-convergence issues due to getting stuck at a local minimum. Thus, reaching the global minimum for such problems is sometimes challenging and computationally expensive for classical computers, which can be eased down by using a quantum approach.

In this study, the author is particularly interested in developing RBM models using a quantum computer. RBM is an energy-based stochastic generative NN model, which can be represented by a bipartite graph (i.e., only nodes from alternate layers between two layers can be connected) consisting of two layers of nodes known as visible layer and hidden layer nodes (as shown in Figure 2.3). Each connection is associated with a weight, while the corresponding nodes are associated with biases. The energy function of an RBM is given by,

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{3}$$

where, $v_i$ and $h_j$ are two visible and hidden layer nodes, and $a_i$ and $b_j$ are their associated biases, respectively; and $w_{ij}$ is the weight of the connection between $v_i$ and $h_j$. Here, the probability of a given state $(v,h)$ is given by,

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)} \tag{4}$$

where, $Z$ is a partition function used for normalization and is given by,

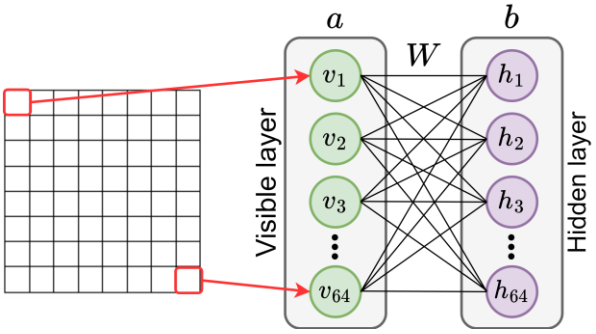



**FIGURE 2.3** Schematic of an RBM Architecture.

$$Z = \sum_{(v,h)} e^{-E(v,h)} \tag{5}$$

As it is difficult to compute all the possible combinations of $v$ and $h$, computing $Z$ is a computationally expensive process. In CD-based training, this problem is simplified by assuming that the variables are independent. The readers are referred to (Hinton, 2012) for CD-based training.

Alternatively, an RBM model can be mapped to a binary quadratic model (BQM), in which the variables are essentially binary, and the model is a combination of linear and quadratic terms. The objective function of a BQM is given by the Ising model, which is shown in the following equation,

$$E_{ising}(\boldsymbol{s}) = \sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j} s_i s_j \tag{6}$$

where, $\boldsymbol{s}$ is a vector of binary variables representing the spins, i.e., $s_i \in \{-1, +1\}$, and $h$ denotes the linear coefficients associated with the qubit biases, and $J$ denotes the quadratic coefficients associated with the coupling strengths. A similar way to represent the BQM

models in computer science is the quadratic unconstrained binary optimization (QUBO) model, where the objective function is given by the following equation,

$$f(\boldsymbol{x}) = \sum_i Q_{ii} x_i + \sum_{i<j} Q_{ij} x_i x_j \tag{7}$$

where, $\boldsymbol{x}$ is a vector of binary variables such that $x_i \in \{0,1\}$ $\boldsymbol{Q}$ is an N × N upper triangular matrix consisting of weights, i.e., $Q_{ij}$ represents the element of the $i^{th}$ row and $j^{th}$ column of $\boldsymbol{Q}$, and $Q_{ii}$ represents the diagonal element of the $i^{th}$ row of $\boldsymbol{Q}$; and $x_i$ and $x_j$ are the $i^{th}$ and the $j^{th}$ elements of $\boldsymbol{x}$, which is a vector of binary variables. Note that, the conversion between the functions presented in (6) and (7) is trivial as (7) simply performs a linear transformation to change the spins ($s_i$) to a binary variable $x_i$, i.e., $x_i = \frac{1}{2}(1 + s_i)$. Thus, the energy function of an RBM in (3) can also be mapped to the objective function of a QUBO problem in (7).

As presented in (Lucas, 2014), a QUBO problem as in (7) can be expressed as a Hamiltonian given by the following equation,

$$H(\boldsymbol{x}) = -\sum_i Q_{ii} \sigma_i^z - \sum_{i<j} Q_{ij} \sigma_i^z \sigma_j^z \tag{8}$$

where, $\sigma_i^z$ denotes a Pauli-Z gate applied on the $i$-th qubit. In (Kadowaki and Nishimori, 1998), the authors presented how to solve such problems using QA by extending the Hamiltonian in (8) with a transverse field. QA is an optimization technique to find the global optimum of an objective function from a given set of candidates (Nonnenmann and Bogomolec, 2021). The Hamiltonian with a transverse field can be written as follows,

$$H(\boldsymbol{x}) = -A(\boldsymbol{x}) \sum_i \sigma_i^x - B(\boldsymbol{x}) \left[ \sum_i Q_{ii} \sigma_i^z + \sum_{i<j} Q_{ij} \sigma_i^z \sigma_j^z \right] \quad (9)$$

where, $A$ and $B$ are two weighting functions, and $\sigma_i^x$ denotes a Pauli-X gate applied on the $i$-th qubit. In (Kadowaki and Nishimori, 1998), the authors proved that using the Hamiltonian in (9), QA could lead to fast convergence (i.e., reaching the ground state with the lowest energy in (9)) with much higher probability than its classical counterpart.

D-Wave is a commercially available QA system that can be utilized to solve such QUBO problems using the Hamiltonian given in (9) (D-Wave Systems Inc., n.d.). Thus, the author chose to utilize a D-Wave QA system (i.e., D-Wave Advantage 4.1 system with over 5,000 qubits) for training the quantum RBM models in this study. Besides, using D-wave's quantum sampler, the author can obtain accurate samples from the original probability distribution of the model given in (4) (Kurowski et al., 2021). Obtaining accurate samples from the original probability distribution is a computationally expensive task for classical computers. However, unlike CD-based training, assuming that the variables are independent is unnecessary while using QA-based training. The author refers the readers to (Kurowski et al., 2021) for the details of an RBM implementation using QUBO and QA-based training, which the author adopted in this framework.

### 2.5 Evaluation

In this section, the author presents an evaluation of the hybrid quantum-classical CAN intrusion detection framework based on data collected from a real-world vehicle. To evaluate the efficacy of the framework, the author compares the intrusion detection performance of the framework developed in this study with a similar but classical-only

framework, i.e., all the steps of the classical-only framework are accomplished in a classical computer, including the RBM-based image reconstruction. In this section, first, the author will discuss the datasets the author used for this evaluation. Next, the author will explain the details of CAN intrusion detection based on the framework presented in section 2.4. Finally, the author will present the results obtained from the evaluation.

### *2.5.1 Dataset*

For evaluation, the author used a CAN intrusion dataset created by the Hacking and Countermeasure Research Lab (HCRL) (Han et al., 2018). The datasets in (Han et al., 2018) include different CAN intrusion datasets, such as fuzzy, malfunction, and replay attack datasets. For this study, the author used a fuzzy attack dataset created by injecting randomly generated CAN messages into the CAN bus of a KIA Soul vehicle. Thus, the dataset includes both the injected CAN messages as well as the normal CAN messages. As shown in Figure 2.4, the dataset contains the following fields: (i) timestamp, (ii) CAN ID (in Hex), (iii) data length code (DLC), (iv) data (encoded as a Hex string), and (v) flag ('R' represents a normal message, and 'T' represents an injected message). The author divided the messages into different datasets based on the associated CAN IDs and selected three datasets based on randomly chosen CAN IDs, i.e., dataset 1 (with CAN ID: 0x220), dataset

| Timestamp | ID | DLC | Data | Flag |
|---|---|---|---|---|
| 1.514E+09 | 04B0 | 8 | 00 00 00 00 00 00 00 00 | R |
| 1.514E+09 | 164 | 8 | 00 08 00 00 00 00 02 0A | R |
| 1.514E+09 | 517 | 8 | 00 00 00 00 00 00 00 00 | R |
| 1.514E+09 | 07A8 | 8 | A2 A0 97 26 D6 A1 66 9A | T |
| 1.514E+09 | 00F3 | 8 | 44 53 C2 73 33 4D 5D 73 | T |
| 1.514E+09 | 175 | 8 | 5F 06 43 32 8A E7 51 2E | T |

**FIGURE 2.4** CAN Fuzzy Attack Dataset.

2 (with CAN ID: 0x316), and dataset 3 (with CAN ID: 0x329), that contain both the normal and injected messages. Each CAN ID is dedicated to broadcasting a particular set of information. Thus, the three datasets used here contain different sets of information encoded as Hex strings. Details of the datasets are presented in Table 2.1.

**TABLE 2.1** Details of the CAN Datasets

| **Dataset** | **Size** | **No. of normal messages** | **No. of attack (injected) messages** | **No. of features** | **Some example features** |
|---|---|---|---|---|---|
| 1 | 2,384 | 1,192 | 1,192 | 14 | LAT_ACCEL, LONG_ACCEL, CYL_PRES, YAW_RATE, YAW_RATE_DIAG, and ESP12_Checksum |
| 2 | 2,684 | 1,324 | 1,324 | 13 | SWI_IGK, F_N_ENG, ACK_TCS, PUC_STAT, TQ_COR_STAT, and TQFR |
| 3 | 1,800 | 900 | 900 | 19 | MUL_CODE, TEMP_ENG, ACK_ES, |

| Dataset | Size | No. of normal messages | No. of attack (injected) messages | No. of features | Some example features |
|---|---|---|---|---|---|
| | | | | | TPS, ACC_ACT, ENG_CHR, and ENG_VOL |

*2.5.2 CAN Intrusion Detection*

First, the author decoded the encoded data fields in each dataset using a generic Database CAN (DBC) file for KIA vehicles from the OpenDBC repository ("OpenDBC," n.d.). After decoding, the author obtained several features containing data from different in-vehicle sensors. Table 2.1 lists some example features for each dataset. Next, the author constructed primary CAN images from the decoded CAN messages in each dataset. Each primary CAN image is obtained by vertically stacking $N$ consecutive decoded CAN messages from a dataset, where $N$ is the number of decoded features in that dataset. Then, the author trained a classical NN to extract features from the primary CAN images in the form of $8 \times 8$ secondary CAN images, as explained in section 2.4.1.2. The $8 \times 8$ secondary CAN images were resized to $8 \times 6$ to allocate the two rightmost columns for the labeling bits. After embedding the labels into the $8 \times 6$ CAN images, the author obtained the final processed $8 \times 8$ CAN images with embedded labels, as explained in section 2.4.1.3. Figure 2.5 provides some examples of the binary encoded CAN images with embedded labels.

The final processed CAN images in each dataset were divided into a training dataset (including randomly shuffled 80% of the CAN images) and a test dataset (including the remaining 20% of the CAN images). For each training dataset, the author trained a classical RBM model using CD-based training and a quantum RBM model using QA-based training. The QA-based training of QRBM models was performed using the D-Wave Advantage 4.1 System, whereas the CD-based training for the classical RBM models was performed on a classical computer. The hyperparameters (i.e., learning rate, number of epochs, weights, and biases) of each RBM model were optimized to yield the best CAN intrusion detection performance. Both classical and quantum RBM models included 64 visible layer nodes and 64 hidden layer nodes that resulted in 64 visible layer biases, 64 hidden layer biases, and 64 × 64 weights to be trained. The same training and test datasets were used for training

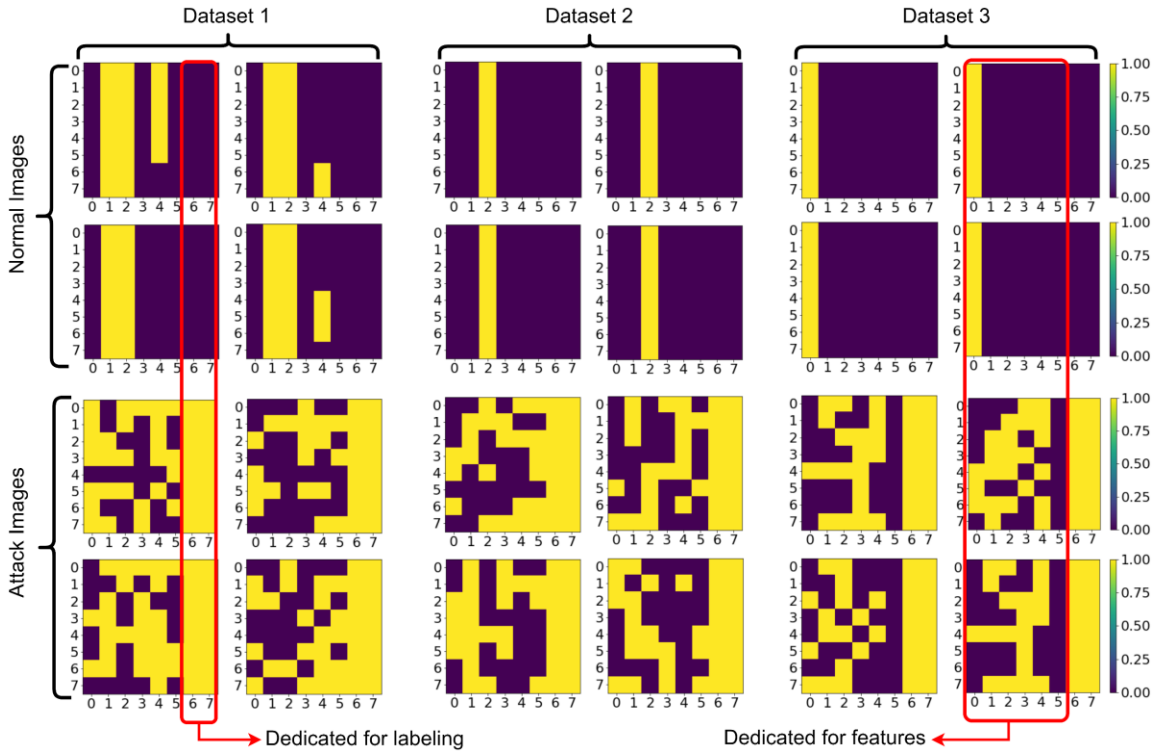



**FIGURE 2.5** Examples of Processed Binary CAN Images with Embedded Labels.

both the classical and the quantum RBM models for comparison. The source code is provided in Appendix A and GitHub (Salek, 2022).

For evaluation, the labeling bits of each CAN image in a test dataset are first replaced by random binary bits. The trained RBM models are then used for reconstructing the CAN images in the test datasets. A reconstructed image is classified as a normal image if most of the bits among the 16 bits dedicated for labeling indicate a normal image; otherwise, the reconstructed image is classified as an attack image.

### *2.5.3 Evaluation Metrics*

The CAN intrusion detection task in this study (i.e., fuzzy attack detection) falls under the category of binary classification (i.e., attack data or normal data). Therefore, classification accuracy (i.e., CAN intrusion detection accuracy) is considered the primary evaluation metric in this study. Recall is considered the secondary evaluation metric since it provides a measure of correctly detected attack data among all the attack data. Binary classification accuracy and recall are given by,

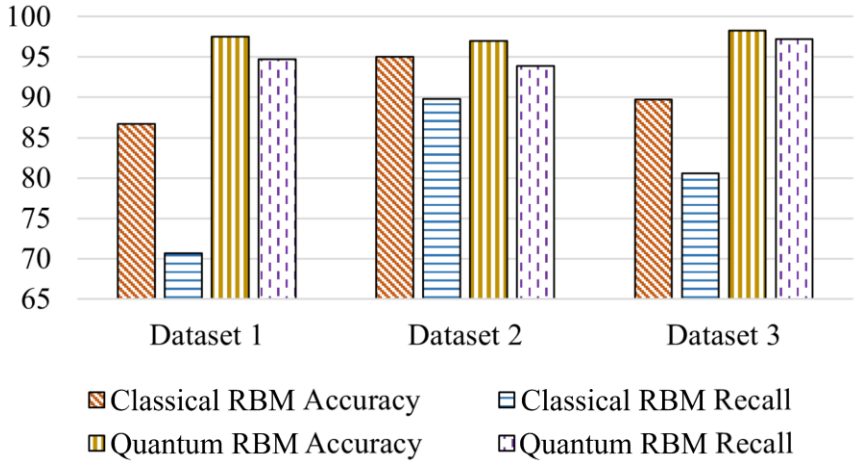
$$Accuracy = \frac{TP + FN}{TP + TN + FP + FN} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

where, TP, TN, FP, and FN denote the total number of true positives, the total number of true negatives, the total number of false positives, and the total number of false negatives, respectively.

*2.5.4 Evaluation Results*

Figure 2.6 presents the accuracies and recalls of the classical RBM, and the quantum RBM approaches for each dataset. As observed from Figure 2.6, the quantum RBM approach outperformed the classical RBM approach for all three datasets used in this study. Among the three datasets, the minimum and maximum CAN intrusion detection accuracies while using the quantum RBM approach were 97% and 98.3%, respectively, whereas the minimum and maximum CAN intrusion detection accuracies for the classical RBM approach were 86.7% and 95%, respectively. On the other hand, the minimum and maximum recall for the quantum RBM approach were 93.9% and 97.2%, respectively, whereas the minimum and maximum recall for the classical RBM approach were 70.7% and 89.8%. Thus, the hybrid quantum-classical framework was able to improve both the accuracies and recalls for CAN intrusion detection on all three datasets used in this study




| | Classical RBM | | Quantum RBM | |
|---|---|---|---|---|
| | Accuracy | Recall | Accuracy | Recall |
| Dataset 1 | 86.7 | 70.7 | 97.5 | 94.7 |
| Dataset 2 | 95 | 89.8 | 97 | 93.9 |
| Dataset 3 | 89.7 | 80.6 | 98.3 | 97.2 |

**FIGURE 2.6** Comparison of CAN Intrusion Detection Performance.

compared to the classical-only framework. This improvement in intrusion detection performance while using the quantum RBM can be attributed to several factors. Quantum DL models have been reported in the literature to achieve similar or better classification performance while being trained on a much smaller dataset compared to their classical counterparts (Caro et al., 2022). This implies that while being trained on the same dataset and using the same DL model architecture with the same number of model parameters, quantum DL models might achieve better classification performance compared to the classical DL models, which aligns with the observations of this study. Training ML or DL models heavily utilizes optimization-based approaches for updating the model parameters. However, a classical computing-based optimization process might get stuck at some local minima, which can be overcome by utilizing a quantum optimization approach as it leverages quantum tunneling to bypass local minima and reach the global minimum quickly. Quantum tunneling enables atoms, electrons, or photons to pass through potential energy barriers, which helps in bypassing local minima to reach the global minimum. In addition, the hybrid framework utilized D-Wave's QA-based sampling for the RBM that enables accurate sampling from the original probability distribution of the model, unlike the CD-based RBM used in the classical-only framework that samples from the conditional probability distribution, as discussed in section 2.4.2. Also, unlike other generative NNs (e.g., generative adversarial network or GAN and generative pretrained transformer or GPT) that would require rigorous training to obtain a well-performing CAN IDS, the hybrid framework utilizes a simpler generative NN architecture of an RBM that can quickly learn to detect attacks by learning the patterns of normal and attack CAN images embedded

with labeling pixels. All of the quantum RBM models in this study converged within a comparable number of epochs while yielding an overall higher attack detection accuracy and recall than the classical RBM, which proves the efficacy of the hybrid quantum-classical CAN intrusion detection framework.

### 2.6 Discussion

In this study, the author presented a hybrid quantum-classical CAN intrusion detection framework utilizing a classical NN and a quantum RBM. In this framework, data preprocessing is done in a classical computer to generate CAN images with embedded labeling pixels from CAN messages. A quantum RBM is used in the framework to reconstruct each CAN image along with its labeling pixels, which is then applied for image classification-based CAN intrusion detection. The author evaluated the hybrid quantum-classical CAN intrusion detection framework on three different real-world fuzzy attack datasets and compared the CAN intrusion detection performance of the hybrid framework with a similar but classical-only framework (i.e., classical computer-based data preprocessing and RBM training). Based on the experiments conducted on the datasets, the minimum accuracy and recall for the hybrid framework were 97% and 93.9%, respectively. In contrast, for the similar but classical-only framework, the minimum CAN intrusion detection accuracy and recall were 86.7% and 70.7%, respectively.

It should be noted that although the hybrid quantum-classical CAN intrusion detection framework utilizes a quantum computer to train the RBM, once the RBM is trained to yield a desired intrusion detection performance, the quantum computer is not used anymore. The trained model can then be transferred to an in-vehicle computing unit

where the entire process of CAN intrusion detection will take place. This will help minimize the end-to-end latency in CAN intrusion detection to a point where it can perform as a real-time cyberattack detection application.

# CHAPTER THREE

# AR-GAN: GENERATIVE ADVERSARIAL NETWORK-BASED DEFENSE METHOD AGAINST ADVERSARIAL ATTACKS ON THE TRAFFIC SIGN CLASSIFICATION SYSTEM OF AUTONOMOUS VEHICLES

## 3.1 Background and Motivation

Autonomous vehicles (AVs) perform the autonomous driving task with the help of a suite of sensors and software. Sensors, such as cameras, light detection and ranging (LiDAR) sensors, radio detection and ranging (Radar) sensors, and ultrasonic sensors, help AVs perceive their surrounding environment like humans do using their sensory systems (Marti et al., 2019). These sensors feed their sensed data to the AV perception module, where the necessary information for autonomous navigation is extracted. This information includes recognizing traffic signs and signals and detecting lane markings, surrounding vehicles, pedestrians, obstacles, etc. Any alteration to this information due to compromised security may result in severe consequences, such as fatal crashes. In addition, if AVs are connected with other vehicles via vehicle-to-vehicle (V2V) communication, with infrastructures via vehicle-to-infrastructure (V2I) communication, and with other entities via vehicle-to-everything (V2X) communication, then the potential attack surfaces increase dramatically (Sharma and Gillanders, 2022; Sun et al., 2022).

Nowadays, even many human-driven vehicles have dashboard camera-based built-in traffic sign classification systems. These systems are typically highly dependent on machine learning (ML) or deep learning (DL) models, especially deep NNs (DNNs) (Wali et al., 2019). However, since AVs rely on such systems to realize roadway regulations and

act accordingly, such as performing a braking or acceleration maneuver, compromised information regarding roadway traffic signs can be more hazardous for AVs compared to any human-driven vehicles. For example, if an AV's traffic sign classification system misclassifies a STOP sign as a SPEED LIMIT sign and fails to stop at a stop-controlled intersection, it may crash on other vehicles or run over a vulnerable road user like a pedestrian. Thus, researchers have emphasized developing DNN-based accurate traffic sign classification systems over the past few years. Some of these classification systems were reported to perform exceptionally well under uncompromised security situations.

However, DNN-based classification systems have some cybersecurity vulnerabilities. For example, an adversarial attack can introduce slight perturbations to the input images or video frames fed to a traffic sign classification system and cause the underlying DNN models to misclassify the signs on the roadway with very high confidence. These perturbations can be so minimal that they are almost imperceptible to regular human eyes. However, they can be very effective in fooling the DNN models used in AVs' traffic sign classification systems. In this study, the author aims to develop an AV traffic sign classification system resilient to such adversarial attacks.

Adversarial attacks can be targeted or untargeted. In a targeted attack, the attack model aims to make a classification system predict a specific target label. For example, a targeted attack can be crafted to force an AV traffic sign classification system to misclassify the STOP signs on the roadway as YIELD signs. In contrast, an untargeted attack aims to force a classification system to misclassify without any specific target label. Another way to categorize adversarial attacks is based on the knowledge the attack model or the attacker

has about the DNN models responsible for classification. If the attacker has no knowledge about the DNN models, e.g., its architecture, parameters, or its defense methods, then it is called a black-box attack. If the attacker has knowledge about the DNN models, e.g., its architecture and parameters, but does not have any information about the defense methods, then it is known as a gray-box attack. On the other hand, if the attacker has complete knowledge of the DNN models and its defense methods, then the attack is called a white-box attack. Without a doubt, the victim, in this case, the DNN-based classification system, is at a maximum disadvantage during a white-box attack because the attacker might craft an adversarial attack in a way so that the DNN-based classification system and its defense methods remain utterly unaware of the fact that there was an attack. In this study, the goal is to develop an AV traffic sign classification system that is resilient to such white-box attacks because, in a connected and AV environment, it is only reasonable to assume that an attacker might get access to this information even without physically accessing the AV.

Different defense methods have been proposed by researchers over the past few years to protect image classification systems from adversarial attacks (Khamaiseh et al., 2022), such as modification of the DNNs (Papernot et al., 2016; Ross and Doshi-Velez, 2018), adversarial training (Bai et al., 2021), input transformation (Liu et al., 2019; Xu et al., 2018), and input reconstruction (Jin et al., 2019; Laykaviriyakul and Phaisangittisagul, 2023; Samangouei et al., 2018). The recent breakthroughs in generative adversarial networks (GANs) have opened opportunities to utilize GANs for defense against adversarial attacks. For example, Samangouei et al., 2018 introduced a Wasserstein GAN (WGAN)-based defense method, known as the Defense-GAN, that can protect image

classification systems against known and unknown adversarial attacks by reconstructing the input images before feeding them to a classifier. The generator model in the Defense-GAN method was trained on to generate samples similar to the unperturbed (legitimate) images given a random input latent vector. The random input latent vector is determined by solving an optimization problem to minimize the reconstruction error of the generator. However, WGAN is known for suffering from issues associated with the weight clipping method it uses, such as vanishing gradient and non-convergence of the discriminator, which makes it training an appropriate WGAN model very difficult (Gulrajani et al., 2017). In (Jin et al., 2019), Jin et al. developed a defense method for image classification systems against adversarial attacks called the Adversarial Perturbation Elimination with GAN (APE-GAN). The generator of the APE-GAN was trained with adversarial examples to eliminate adversarial perturbations by making changes to the input images. However, the authors in (Jin et al., 2019) also utilized the loss function of WGAN, which has the same issues as mentioned above. Besides, adversarial training-based defense methods work well for known attacks only. Laykaviriyakul and Phaisangittisagul, 2023 proposed an adversarial defense framework for image classification systems based on the DiscoGAN architecture (Kim et al., 2017), where the authors utilized an attacker model to create adversarial examples from the training data and a defender model to reconstruct unperturbed images from the adversarial images. These two models were trained in tandem to play a competing game with each other. However, the authors in (Laykaviriyakul and Phaisangittisagul, 2023) also used an adversarial training-based defense approach, which may not perform well under unknown attacks on which the models were not trained.

Besides, all these studies considered benchmark datasets like CIFAR-10, MNIST, and Fashion MNIST. Thus, their performance on real-world datasets, such as a real-world traffic sign dataset, is not explored yet.

In this study, the author developed an attack-resilient GAN-based defense method for an AV traffic sign classification system, which the author refers to as the AR-GAN in this study, that can protect the perception module of an AV from unknown attacks. The author used a WGAN-based loss function with gradient penalty (WGAN-GP) to train the GAN models, which was shown to overcome common issues with GAN/WGAN training, such as mode collapse and vanishing gradient. The author trained the GAN models and classifiers on the unperturbed traffic sign images only so that all types of adversarial attacks are unknown to the models. For evaluation, the author only considered white-box attacks with full knowledge of the AR-GAN models to put the AR-GAN method in a situation with maximum disadvantage.

### 3.2 Contribution

Much work has been done on DNN-based traffic sign classification systems in the literature (Gudigar et al., 2016; Saadna and Behloul, 2017; Wali et al., 2019; Lim et al., 2023). However, to the best of the author's knowledge, none of the existing studies utilized a GAN-based adversarial defense method for traffic sign classification. In this study, the author developed an adversarial attack-resilient traffic sign classification system based on GAN (AR-GAN) for AVs, which does not require any prior knowledge about adversarial attacks. The AR-GAN method utilizes a WGAN-GP-based loss function to overcome typical convergence issues with GANs, such as mode collapse and vanishing gradient. The

generator in the AR-GAN method is based on the DCGAN architecture (Radford et al., 2016) and trained to generate unperturbed samples from adversarial samples before feeding them to the classifier. The classifier in the AR-GAN method is based on the ResNet9 architecture (He et al., 2016) and trained on traffic sign images reconstructed by the generator. Besides, AR-GAN uses a particular training framework (discussed in section 3.5 in detail) to obtain the models to ensure that the models used in the final traffic sign classification system perform the best. Thus, the generator and the classifier in the AR-GAN traffic sign classification system can achieve high traffic sign classification accuracies under no-attack scenario as well as under different types of white-box adversarial attacks, which the author evaluated with a real-world traffic sign dataset in this study. Also, the AR-GAN traffic sign classification system can provide very consistent classification performance under different perturbation magnitudes, unlike the traditional defense methods.

### 3.3 Related Work

Significant progress has been made in traffic sign classification in recent years, with a growing focus on using DNN-based techniques. In this section, the author describes some notable contributions in this area.

Among the earlier studies on traffic sign classification, Zhang et al., 2017 presented a shallow Convolutional Neural Network (CNN) for traffic sign recognition consisting of feature extraction stages, fully connected layers, and a Softmax-loss layer. The authors performed subsampling using a combination of max pooling and average pooling, which allowed the network to learn discriminative features automatically. The proposed network

in (Zhang et al., 2017) achieved an accuracy of 99.84% on the German Traffic Sign Recognition Benchmark (GTSRB) dataset (Houben et al., 2013). However, a limitation of this network indicated by the authors was its dependency on a fixed input image size. Li and Wang, 2019 conducted a study on traffic sign recognition using the MobileNet (Howard et al., 2017) CNN architecture. The authors incorporated batch normalization, ReLU activation, and a Softmax layer to calculate confidence probabilities for traffic sign classification. The experiments were performed on the GTSRB dataset, and the model achieved a classification accuracy of 99.66%.

Among the more recent studies in this area, Kerim and Efe, 2021 developed a hybrid NN to classify traffic signs using various features, including Histograms of Oriented Gradients (HOG) and a combination of color, HOG, and Local Binary Patterns (LBP). The authors employed the GTSRB and Chinese Traffic Sign Recognition Dataset (TSRD) for training and evaluation. The hybrid NN consisted of nine individual NNs, each analyzing traffic signs based on specific image attributes. In addition, the authors employed data augmentation to improve their model performance. The method, including color, HOG, and LBP features, demonstrated an accuracy of 95%, which outperformed the approach that solely employed HOG features. Kheder and Mohammed, 2023 improved the traditional LeNet-5 CNN model architecture (Lecun et al., 1998) by increasing the number of layers and incorporating various common image preprocessing algorithms to enhance performance. The authors in (Kheder and Mohammed, 2023) trained the model using the GTSRB and extended GTSRB (EGTSRB) (i.e., GTSRB combined with Belgium Traffic Sign dataset) datasets. Compared to the other state-of-the-art approaches, such as Viola-

Jones and Inception-V3 CNN architecture (Jose et al., 2019) and improved LeNet-5 CNN architectures (RADU et al., 2020; Zaibi et al., 2021), the results demonstrated high classification accuracy, with 99.12% achieved on the GTSRB dataset and 99.78% on the EGTSRB dataset.

Shanmugavel et al., 2023 developed a model called Ensemble-based LeNet, VGGNet, and DropoutNet (ELVD) for real-time traffic sign classification, object tracking, and recognition. The authors used a combination of CNN architectures, including LeNet, VGGNet (Simonyan and Zisserman, 2015), and DropoutNet (Volkovs et al., 2017), and trained and tested the models on the GTSRB and the TSRD datasets. The proposed ELVD model demonstrated fast detection time compared to traditional models, such as MCDNN (Han et al., 2016), Mask R-CNN (Bharati and Pramanik, 2020), Support Vector Machine or SVM (Creusen et al., 2010), HOG (Yang et al., 2016), and Single Shot multi-box Detector or SSD (Liu et al., 2016), with over 99% classification accuracy. Pandurangan et al., 2023 introduced a traffic signal recognition model by using preprocessing techniques, such as median filtering and histogram equalization, and employing ML and DL algorithms, including SVM, Extreme Learning Machine (ELM), Linear Discriminant Analysis (LDA), Principal Component Analysis (PCA), and CNN-General Regression Neural Network (GRNN). The proposed model achieved a high accuracy of 99.41% on the GTSRB dataset compared to other traditional methods.

In addition to the studies mentioned above, many others developed DNN architectures to classify traffic signs using various datasets (Dilek and Dener, 2023; Lim et al., 2023). However, only a few studies have developed DNN-based defense methods to

improve the resilience and robustness of traffic sign classification systems against adversarial attacks. Among them, Li et al., 2021 proposed a defense method based on an attention mechanism to address the vulnerability of traditional NNs to adversarial attacks for traffic sign recognition. The authors utilized a spatial transformation module that extracts affine coordinate parameters of target objects, redraws them using a coordinate mapping model, and applies an attention mechanism to filter pixels through interpolation. The authors evaluated their proposed model against various attack methods on traffic sign adversarial samples generated on the GTSRB dataset. The model demonstrated an average accuracy of 73.95% when challenged with untargeted white-box Fast Gradient Sign Method (FGSM) attacks. These adversarial samples were generated using a ResNet50 classifier trained on the GTSRB dataset. Hashemi et al., 2022 developed a novel cost function, which the authors refer to as the Regularized Guided Complement Entropy (RGCE), to increase the performance of CNNs used for traffic sign recognition. They employed the information from the Softmax layer in addition to some of the extracted features from convolutional layers in their optimization process. The authors evaluated the RGCE on two datasets, the CIFAR-10 dataset and the GTSRB dataset, highlighting its improved robustness against various adversarial attacks while maintaining or enhancing performance on clean images. The RGCE in (Hashemi et al., 2022) was reported to achieve an accuracy of 90.24% and 74.44% when targeted by the FGSM and the Projected Gradient Descent (PGD) L-infinite norm attacks, respectively, with a perturbation magnitude of 0.04. The adversarial samples in (Hashemi et al., 2022) were generated using a ResNet18 classifier trained on the GTSRB dataset.

Khan et al., 2022 proposed a DNN-based hybrid defense method for improving the resilience of traffic sign classification against adversarial attacks. They developed the hybrid defense method based on Inception-V3 and Resnet-152 DNN models and incorporated random filtering, ensembling, and local feature mapping defense strategies. The authors evaluated their proposed hybrid defense method on a modified subset version of the extended LISA traffic sign database(Møgelmose et al., 2015), which showed 99% classification accuracy on average in the absence of attacks and 88% classification accuracy on average against various adversarial attacks, such as FGSM, Momentum Iterative Method (MIM), PGD, and Carlini and Wagner (C&W) attacks. The hybrid defense method in (Khan et al., 2022) was reported to perform better than some other traditional defense methods, such as feature squeezing, JPEG filtering, binary filtering, and random filtering. Majumder et al., 2021 combined classical and quantum neural layers to develop hybrid classical-quantum DL models for increasing the resiliency of traffic sign image classification models in AVs against adversarial attacks. The authors employed a pre-trained ResNet18 CNN for the classical part and incorporated quantum gates to employ mechanical features for the quantum layer. The authors in (Majumder et al., 2021) proposed two hybrid models and evaluated them using a modified subset version of the LISA traffic sign image dataset. The second hybrid model performed better under the PGD attack than the classical model; however, under the FGSM attack, the classical model outperformed the hybrid model.

Although the abovementioned studies proposed different DNN-based traffic sign classification systems for defending against adversarial attacks, none considered a

generative DNN-based method. With the recent breakthroughs in GANs and high-performance in-vehicle computational units, GANs have introduced an unprecedented window of opportunities to be deployed for AV applications, such as an AV traffic sign classification application. However, to the best of the author's knowledge, none of the existing studies attempted to develop a GAN-based adversarial attack-resilient AV traffic sign classification system, which is the focus of this study.

## 3.4 Attack Models

Numerous adversarial attack models have been developed by researchers in the past few years to target NN-based classifiers (Khamaiseh et al., 2022). In this study, the author only considers white-box attacks, in which an attacker, having complete knowledge about the classification system, aims to find a perturbation $\delta$ that will cause misclassification by the classifier when added to legitimate input $x \in \mathbb{R}^n$, i.e., $\tilde{x} = x + \delta$, where $\tilde{x}$ is the modified input or the adversarial example that may cause misclassification. In this section, the author discusses the different white-box attack models used in this study.

### *3.4.1 Fast Gradient Sign Method (FGSM) Attack*

FGSM is a simple but effective attack proposed by Goodfellow et al., 2015. FGSM utilizes the gradient of the loss (cost) function with respect to the input image to generate adversarial examples. The "Fast" in the name refers to the efficiency and simplicity of the attack. Despite its simplicity, FGSM has become one of the most popular adversarial attacks due to its effectiveness in causing misclassification with high confidence (Goodfellow et al., 2015). Given an input image $x$ and a target classifier with parameters $\theta$, FGSM attack aims to generate an adversarial example $\tilde{x} = x + \delta$, where the added

perturbation $\delta$ is determined by computing the gradient of the loss function with respect to the input $x$ as follows,

$$\delta = \varepsilon . sign\big(\nabla_x J(\theta, x, y)\big) \tag{12}$$

where, $J$ denotes the loss function, $y$ denotes the output class, $\nabla_x$ is a differential operator with respect to $x$, and $\varepsilon$ denotes the magnitude of perturbation chosen by the attacker.

*3.4.2 DeepFool Attack*

DeepFool is another simple yet effective optimization-based iterative white-box attack model proposed by Moosavi-Dezfooli et al., 2016, which has been reported by the authors to be more effective than the FGSM attack on MNIST and CIFAR-10 datasets. Given a binary classifier model, this attack aims to find the minimum perturbation $\delta^*$ that would cause misclassification by shifting the input $x$ to the other side of the classification boundary. The minimum perturbation $\delta^*$ is determined through an optimization problem as follows,

$$\delta^* = \arg\min_{\delta} \|\delta\|_2 \tag{13}$$

$$\text{subject to: } sign\big(f(x+\delta)\big) \neq sign\big(f(x)\big)$$

where, $f$ is an arbitrary binary classification model. At $i^{th}$ iteration, DeepFool updates $\delta$ by linearizing the classification boundary around the current point $x_i$. In order to ensure that the final perturbation $\hat{\delta}$ yields misclassification, it is multiplied by a constant $(1+\eta)$, where $\eta \ll 1$. The authors in (Moosavi-Dezfooli et al., 2016) extended this approach to multi-class classifiers as well.

*3.4.3 Carlini and Wagner (C&W) Attack*

Carlini and Wagner, 2017 introduced an optimization-based iterative adversarial attack known as the C&W attack. The C&W attack is a powerful attack that has been reported to cause very low classification accuracy on benchmark datasets, such as MNIST and CIFAR datasets (Carlini and Wagner, 2017). In a C&W attack, given an input image $x \in \mathbb{R}^n$, the goal of the attack is to determine an optimal perturbation $\delta^*$ so that the adversarial example $\tilde{x} = x + \delta^*$ is misclassified by a target classifier. In an $\ell_2$-norm C&W attack, the optimal perturbation $\delta^*$ is determined through the following optimization problem,

$$\delta^* = \arg\min_{\delta} \|\delta\|_2 + c.f(x + \delta) \tag{14}$$

$$\text{subject to: } x + \delta \in [0, 1]^n$$

Here, $\|.\|_2$ denotes the $\ell_2$-norm, $c > 0$ is an arbitrary constant, and $f$ is an objective function that helps the input image $x$ to be misclassified, which is chosen based on the knowledge of the target classifier model. Apart from the $\ell_2$-norm attack explained above, C&W attacks can also be performed using $\ell_0$ and $\ell_\infty$ norms.

*3.4.4 Projected Gradient Descent (PGD) Attack*

PGD is a powerful white-box attack model that also utilizes an optimization-based iterative approach to determine the optimal perturbation $\delta$ that can cause misclassification when added to an input image $x$. The authors in (Madry et al., 2017) showed the effectiveness of PGD attacks on MNIST and CIFAR-10 datasets, in which PGD was able to yield lower classification accuracy on the datasets compared to FGSM and C&W

attacks. Given an input image $x$ and a perturbation $\delta$ to be optimized for a PGD attack, the optimization problem can be written as,

$$\delta^* = \arg\min_{\delta} \|\delta\|_2 \tag{15}$$

$$\text{subject to: } x + \delta \in [x_{min}, x_{max}]$$

Here, $\|.\|_2$ denotes the $\ell_2$-norm, and $x_{min}$ and $x_{max}$ are the minimum and maximum values of each pixel. The perturbation $\delta$ is updated in each iteration as follows,

$$\delta_{t+1} = \delta_t + \varepsilon.sign\big(\nabla_x J(\theta, x + \delta, y)\big) \tag{16}$$

where, $J$ denotes the loss function, $y$ denotes the output class, $\nabla_x$ is a differential operator with respect to $x$, and $\varepsilon$ denotes the magnitude of perturbation.

### 3.5 AR-GAN for Adversarial Attack Resilience in Traffic Sign Classification

In this section, the author formally introduces a GAN-based adversarial attack resilience method for traffic sign classification, which the author refers to as the attack-resilient GAN (AR-GAN). The final traffic sign classification system of the AR-GAN includes a generator model and a classifier model obtained from the AR-GAN training. The generator is used to reconstruct any input images of traffic signs, which, in the process of reconstruction, helps denoise the traffic sign images from adversarial noise, and the classifier, trained on reconstructed traffic sign images by the generator, helps classify the denoised traffic sign images. The author will discuss the details of training the GAN and the classifier models in this section, but before that, the author presents an overview of the AR-GAN training framework that helps obtain the models in the traffic sign classification system of the AR-GAN.

The training framework to obtain the models in the final traffic sign classification system of AR-GAN is depicted in Figure 3.1. First, the author trains a classifier model to classify unperturbed (i.e., legitimate or without attack) images in a traffic sign image dataset. In this study, the author used a 9-layer deep residual learning architecture known as ResNet9 (He et al., 2016). Once the classifier is trained to classify unperturbed traffic sign images with acceptable accuracy, the author calls it Classifier #1, which will later be

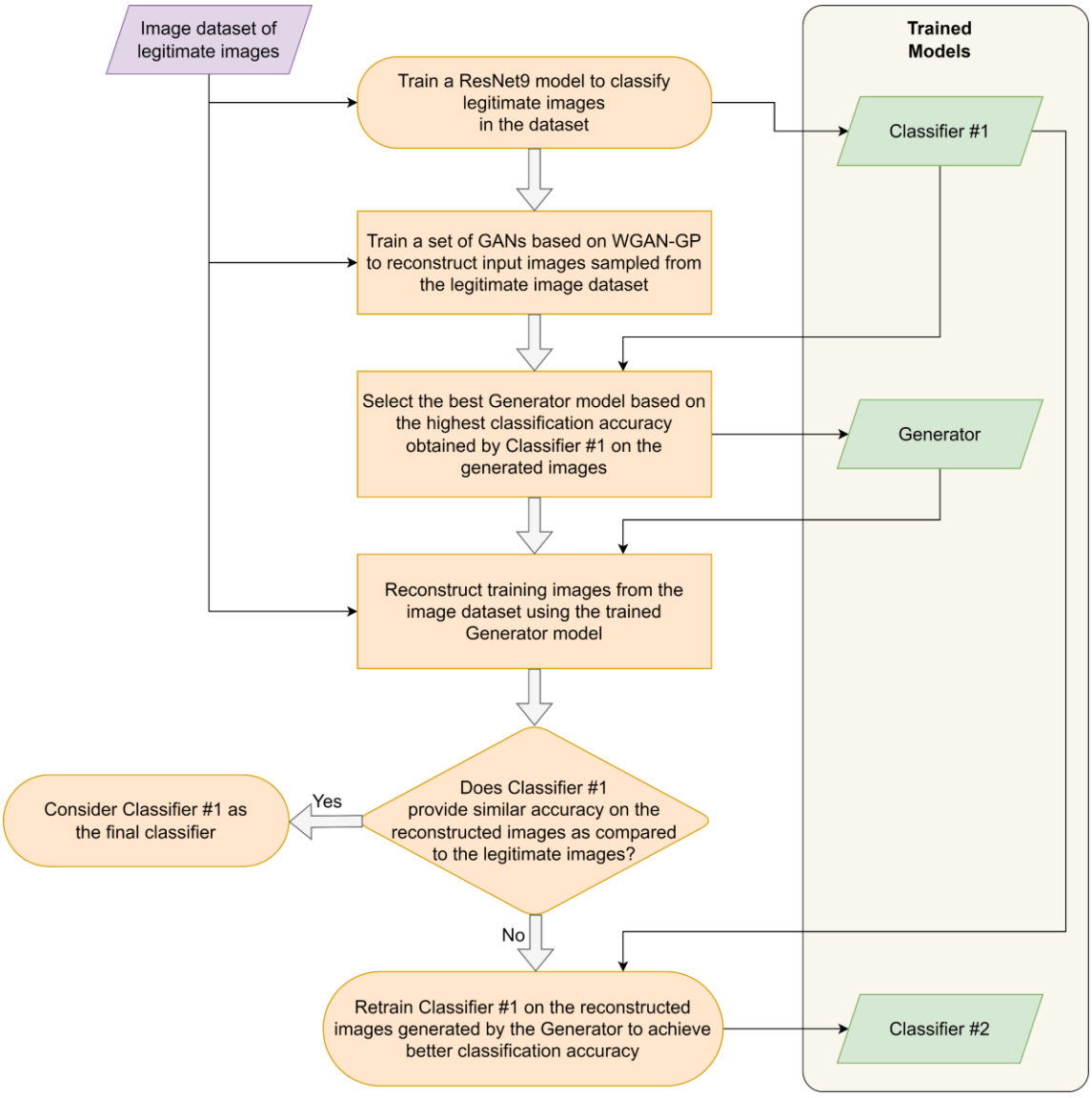



**FIGURE 3.1** AR-GAN Training Framework.

used to select the best generator model. Next, the author trains a set of GANs based on the Wasserstein GAN architecture with a gradient penalty (WGAN-GP) using the same unperturbed traffic sign images from the dataset. Once the GANs are trained, the author utilizes Classifier #1 to select the best generator model from the set of trained GANs that would yield the highest classification accuracy on the reconstructed traffic sign images of the test dataset. Then, the author uses the selected generator model to reconstruct all the unperturbed traffic sign images in the dataset. In an ideal case, if the GANs are trained to a point when the reconstructed traffic sign images look identical to the unperturbed traffic sign images, Classifier # 1 should be able to classify the reconstructed traffic sign images with similar accuracy to that of the unperturbed traffic sign images. However, if that is not the case, then the author retrains Classifier #1 on the reconstructed traffic sign images to achieve better accuracy. The author calls this retrained classifier model Classifier #2. Finally, the AR-GAN traffic sign classification system is built with the best generator model (to reconstruct and denoise any input traffic sign images) and Classifier #2 (to classify the reconstructed traffic sign images).

### *3.5.1 Classifier Model*

The classifiers in the AR-GAN method are based on the ResNet9 architecture. ResNet9 is a 9-layer deep NN, including eight convolutional layers and one linear layer. Table 3.1 presents the model architecture of ResNet9 used in the AR-GAN method, assuming that the input traffic sign images have three color channels, i.e., red, green, and blue, each with $32 \times 32$ pixels. The output dimension of each layer is presented as $C \times H \times W$ in Table 3.1, where $C$, $H$, and $W$ represent the number of channels, the height

(i.e., the number of pixels vertically), and the width (i.e., the number of pixels horizontally) of the output feature map of that layer, respectively. The column representing the operations in Table 3.1 is formatted as $(k \times k)$, $s$, $p$, where $(k \times k)$ is the kernel size, $s$ is the number of strides, and $p$ is the amount of zero-padding in horizontal and vertical directions. The different types of layers utilized in the architecture are explained below.

Conv2D: Conv2D refers to a 2D convolution operation applied to an input 2D matrix. A convolutional layer in Conv2D consists of a kernel or filter with a specified height and width. These kernels are typically set to be smaller than the input images and are moved across the images to extract features.

BatchNorm2D: BatchNorm2D refers to 2D batch normalization operation applied to an input 2D matrix. Batch normalization is a popular technique used in deep learning to normalize a batch of inputs. It can be applied to the activations of a previous layer or to the inputs directly. Batch normalization accelerates the training process and provides regularizations.

MaxPool2D: MaxPool2D refers to a 2D pooling operation that selects the maximum element from a region of a 2D feature map covered by a kernel. The output of a max-pooling layer consists of a feature map that includes the most prominent features from the previous feature map.

ReLU: Rectified Linear Unit or ReLU refers to a non-linear activation function used in DNNs. The function simply returns zero for any negative input and the input value itself for any positive input. ReLU introduces non-linearity to the data, enabling the DNN to learn complex patterns and representations.

**TABLE 3.1** Model Architecture of the Classifier

| | **Layer Type** | **Output Dimension** No. of channels × Height ×Width | **Operation** (Kernel size), No. of strides, Padding size |
|---|---|---|---|
| Layer 1 | Conv2D | 64 × 32 × 32 | (3 × 3), 1, 1 |
| | BatchNorm2D | 64 × 32 × 32 | Not applicable |
| | ReLU | 64 × 32 × 32 | Not applicable |
| Layer 2 | Conv2D | 128 × 32 × 32 | (3 × 3), 1, 1 |
| | BatchNorm2D | 128 × 32 × 32 | Not applicable |
| | ReLU | 128 × 32 × 32 | Not applicable |
| | MaxPool2D | 128 × 16 × 16 | (2 × 2), 2, 0 |
| Layer 3 | Conv2D | 128 × 16 × 16 | (3 × 3), 1, 1 |
| | BatchNorm2D | 128 × 16 × 16 | Not applicable |
| | ReLU | 128 × 16 × 16 | Not applicable |
| Layer 4 | Conv2D | 128 × 16 × 16 | (3 × 3), 1, 1 |
| | BatchNorm2D | 128 × 16 × 16 | Not applicable |
| | ReLU | 128 × 16 × 16 | Not applicable |
| Layer 5 | Conv2D | 256 × 16 × 16 | (3 × 3), 1, 1 |
| | BatchNorm2D | 256 × 16 × 16 | Not applicable |
| | ReLU | 256 × 16 × 16 | Not applicable |
| | MaxPool2D | 256 × 8 × 8 | (2 × 2), 2, 0 |

| | **Layer Type** | **Output Dimension**<br>No. of channels × Height ×Width | **Operation**<br>(Kernel size), No. of strides, Padding size |
|---|---|---|---|
| Layer 6 | Conv2D | 512 × 8 × 8 | (3 × 3), 1, 1 |
| | BatchNorm2D | 512 × 8 × 8 | Not applicable |
| | ReLU | 512 × 8 × 8 | Not applicable |
| | MaxPool2D | 512 × 4 × 4 | (2 × 2), 2, 0 |
| Layer 7 | Conv2D | 512 × 4 × 4 | (3 × 3), 1, 1 |
| | BatchNorm2D | 512 × 4 × 4 | Not applicable |
| | ReLU | 512 × 4 × 4 | Not applicable |
| Layer 8 | Conv2D | 512 × 4 × 4 | (3 × 3), 1, 1 |
| | BatchNorm2D | 512 × 4 × 4 | Not applicable |
| | ReLU | 512 × 4 × 4 | Not applicable |
| Layer 9 | MaxPool2D | 512 × 1 × 1 | (4 × 4), 4, 0 |
| | Flatten | 512 | Not applicable |
| | Dropout | 512 | Not applicable |
| | Linear | 2 | Not applicable |

*3.5.2 Generator Model*

To obtain the generator model for the final traffic sign classification system in the AR-GAN method, the author utilizes a WGAN-GP-based training method and extends it

to reconstruct an input image $x$ at inference time. In this subsection, the author starts by explaining the loss function used in traditional GANs. Then, the author discusses how WGAN-GP modified it, and finally, the author discusses the extension adopted from the Defense-GAN method (proposed by Samangouei et al., 2018) to reconstruct a given image at inference time.

GANs, first introduced by Goodfellow et al., 2014, consist of two NNs, known as the generator ($G$) and the discriminator ($D$). $G: \mathbb{R}^k \rightarrow \mathbb{R}^n$ takes a low-dimensional latent vector $z \in \mathbb{R}^k$ as the input and maps it to high-dimensional sample space of $x \in \mathbb{R}^n$. The discriminator, $D$ is a binary classifier that distinguishes between real samples and fake (i.e., generated) samples. $G$ and $D$ are trained in tandem to simultaneously optimize both NNs. While $G$ aims to generate images that are identical to the real images, $D$ helps $G$ by discriminating between real samples $x$ and fake samples $G(z)$. $G$ and $D$ are trained alternatively to optimize the following min-max loss function defined by Goodfellow et al., 2014,

$$\min_G \max_D V(D,G) = \mathrm{E}_{x\sim P_r(x)}[\log D(x)] + \mathrm{E}_{z\sim P_g(z)}\left[\log\left(1 - D\big(G(z)\big)\right)\right] \qquad (17)$$

where, $P_r(x)$ and $P_g(z)$ denote the real sample distribution and the generated sample distribution, respectively, and E denotes the expected value. The optimal GAN is obtained when these two distributions become the same. However, in reality, it turned out to be very difficult to train GANs to achieve this optimality due to issues such as mode collapses and vanishing gradients.

To resolve these issues, Arjovsky et al., 2017 proposed a variant of the GAN, known as the Wasserstein GAN (WGAN), that utilizes the concepts of Wasserstein

distance and Kantorovich-Rubinstein duality (Rachev, 1990), resulting in an alternative loss function given by,

$$\min_G \max_D V_W(D,G) = \mathrm{E}_{x\sim P_r(x)}[\log D(x)] - \mathrm{E}_{z\sim P_g(z)}\left[\log\left(D\left(G(z)\right)\right)\right] \tag{18}$$

WGAN also removed the sigmoid function from the discriminator of the original GAN proposed in (Goodfellow et al., 2014) to interpret the output of $D$ in terms of probability to indicate how "real" the generated images are. The authors in (Arjovsky et al., 2017) renamed the modified discriminator as "critic". However, WGAN still suffered from some convergence issues due to the hard constraints set by the weight clipping method used by the authors to enforce the Lipschitz condition. This resulted in an improved version of the WGAN, proposed by Gulrajani et al., 2017, known as WGAN with gradient penalty (WGAN-GP). WGAN-GP utilizes a soft version of the constraints by penalizing the model if the norm of the gradient deviates from its target norm value of 1 to meet the Lipschitz condition as follows,

$$\begin{aligned}\min_G \max_D V_W(D,G) = {} & \mathrm{E}_{x\sim P_r(x)}[\log D(x)] - \mathrm{E}_{z\sim P_g(z)}\left[\log\left(D\left(G(z)\right)\right)\right] \\ & + \lambda \mathrm{E}_{\hat{x}\sim P_{\hat{x}}}[\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1]^2\end{aligned} \tag{19}$$

Here, $\lambda$ is set to 10, $\nabla_{\hat{x}}$ is a differential operator with respect to $\hat{x}$, and $\hat{x}$ is sampled from $x$ and $G(x)$ using the following linear equation,

$$\hat{x} = tG(x) + (1-t)x \tag{20}$$

where, $t$ is uniformly sampled between 0 and 1, i.e., $0 \leq t \leq 1$. The WGAN-GP, proposed by Gulrajani et al., 2017 also removed the batch normalization steps from the critic as it was reported to affect the effectiveness of the gradient penalty.

Thus, WGAN-GP provides us with a modified loss function from that of the traditional GAN that is able to resolve traditional GAN training issues, such as mode collapse and vanishing gradient, as well as the tractability issues with WGAN. However, to be able to reconstruct an input image with minimum reconstruction error, the author requires a further extension. The authors in (Samangouei et al., 2018) proposed a simple extension to achieve this by optimizing the input latent vector $z$ that will be fed to the generator to reconstruct an input image $x$, which the author adopts in the AR-GAN method. The optimal latent vector $z^*$ for reconstructing an input image $x$ is obtained by solving the following optimization problem (Samangouei et al., 2018),

$$z^* = \arg\min_z \|G(z) - x\|_2^2 \tag{21}$$

The optimization problem in (21) is solved in a gradient descent-based iterative approach. Because (21) is highly non-convex, the authors in (Samangouei et al., 2018) utilized a fixed number of gradient descent steps along with a given number of random initializations of the latent vector $z$.

Tables 3.2 and 3.3 presents the architectures of the generator and the discriminator used in the AR-GAN method. The generator architecture is based on the deep convolutional GAN (DCGAN) architecture (Radford et al., 2016), whereas the critic or discriminator architecture is kept the same as the WGAN architecture. The different types of layers used in these architectures are similar to that discussed in section 2.5.1, except the ConvTranspose2D layers used in the generator architecture and the LeakyReLU layers used in the discriminator architecture, which are explained below.

ConvTranspose2D: The ConvTranspose2D layer is an extension of the traditional convolutional layer but with a reversed operation. While a regular convolutional layer performs a sliding window operation on the input to produce a feature map, the ConvTranspose2d layer performs an inverse operation.

LeakyReLU: LeakyReLU refers to an extension of the ReLU. Instead of returning zero for a negative input, LeakyReLU utilizes a small non-zero negative gradient for negative inputs. This solves the "dying ReLU" issue of the original ReLU that occurs when ReLUs get stuck in a negative state during training and fail to update their weights.

**TABLE 3.2** Model Architecture of the Generator

| | **Layer Type** | **Output Dimension**<br>No. of channels × Height ×Width | **Operation**<br>(Kernel size), No. of strides, Padding size |
|---|---|---|---|
| Layer 1 | ConvTranspose2D | $512 \times 4 \times 4$ | $(4 \times 4)$, 1, 0 |
| | BatchNorm2D | $512 \times 4 \times 4$ | Not applicable |
| | ReLU | $512 \times 4 \times 4$ | Not applicable |
| Layer 2 | ConvTranspose2D | $256 \times 8 \times 8$ | $(4 \times 4)$, 2, 1 |
| | BatchNorm2D | $256 \times 8 \times 8$ | Not applicable |
| | ReLU | $256 \times 8 \times 8$ | Not applicable |
| Layer 3 | ConvTranspose2D | $128 \times 16 \times 16$ | $(4 \times 4)$, 2, 1 |
| | BatchNorm2D | $128 \times 16 \times 16$ | Not applicable |
| | ReLU | $128 \times 16 \times 16$ | Not applicable |

| | Layer Type | Output Dimension<br>No. of channels × Height ×Width | Operation<br>(Kernel size), No. of strides, Padding size |
|---|---|---|---|
| Layer 4 | ConvTranspose2D | $3 \times 32 \times 32$ | $(4 \times 4)$, 2, 1 |
| | Tanh | $3 \times 32 \times 32$ | Not applicable |

**TABLE 3.3** Model Architecture of the Discriminator

| | Layer Type | Output Dimension<br>No. of channels × Height ×Width | Operation<br>(Kernel size), No. of strides, Padding size |
|---|---|---|---|
| Layer 1 | Conv2D | $32 \times 16 \times 16$ | $(4 \times 4)$, 2,1 |
| | LeakyReLU | $32 \times 16 \times 16$ | Not applicable |
| Layer 2 | Conv2D | $64 \times 8 \times 8$ | $(4 \times 4)$, 2, 1 |
| | LeakyReLU | $64 \times 8 \times 8$ | Not applicable |
| Layer 3 | Conv2D | $128 \times 4 \times 4$ | $(4 \times 4)$, 2, 1 |
| | LeakyReLU | $128 \times 4 \times 4$ | Not applicable |
| Layer 4 | Conv2D | $256 \times 1 \times 1$ | $(4 \times 4)$, 1, 0 |
| | LeakyReLU | $256 \times 1 \times 1$ | Not applicable |
| Layer 5 | Flatten | 256 | Not applicable |
| | Linear | 1 | Not applicable |

Finally, the AR-GAN traffic sign classification system consists of the trained generator, the classifier, and the optimizer that optimizes the input latent vector applied to the generator to reconstruct an input traffic sign image. Figure 3.2 presents the AR-GAN traffic sign classification system with a flow diagram.

## 3.6 Evaluation Method

This section discusses the evaluation method, specifically the traffic sign dataset, and the traditional preprocessing-based defense strategies the author used in this study for comparison.

### *3.6.1 Traffic Sign Dataset*

The author reviewed the existing literature to select a comprehensive traffic sign dataset for the US traffic signs and found the LISA traffic sign dataset (created by Mogelmose et al., 2012) the most appropriate since it contains data collected from the real world. The LISA dataset covers 49 types of US traffic signs with 7,855 annotations on 6,610 frames. The traffic sign images in the LISA dataset were extracted from video frames captured by multiple vehicles' dashboard cameras while the vehicles were driven around

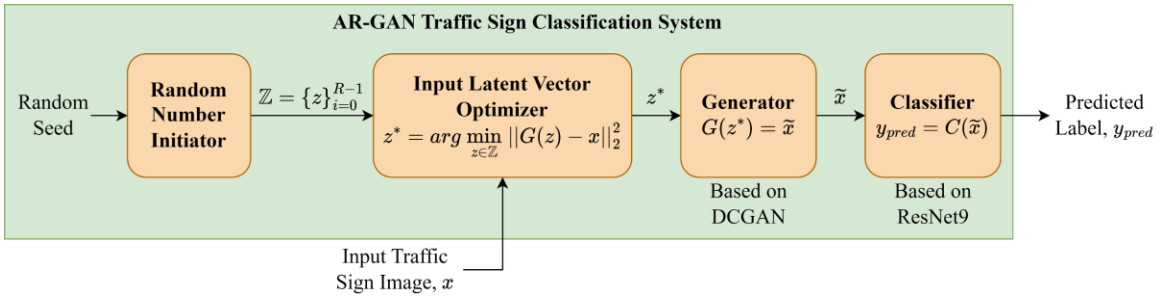



**FIGURE 3.2** AR-GAN Traffic Sign Classification System.

San Diego, California. The original video frames exhibit varying resolutions, ranging from 640×480 pixels to 1024×522 pixels. The annotations of the traffic signs within these frames have dimensions spanning from 6×6 pixels to 167×168 pixels and include both color and grayscale images.

However, the LISA dataset does not contain enough images for each type of traffic sign to train GAN models. Also, some of the images have resolutions that are too small to be used for training purposes. Therefore, the author created a subset of the LISA dataset containing only STOP signs and SPEED LIMIT signs. The original LISA dataset contains different types of SPEED LIMIT signs, such as 15, 25, 30, 35, 40, 45, 50, and 65 miles per hour (mph) signs. The author combined all these SPEED LIMIT signs into one class to create a balanced dataset. Thus, the subset of the LISA dataset used in this study for evaluation of the AR-GAN method contains a total of 1,562 traffic sign images and includes two classes of traffic signs, i.e., 805 images under the STOP sign class and 757 images under the SPEED LIMIT sign class. The author applied cropping and resizing to ensure that all the images in the dataset have the same dimension, i.e., each image has three channels for red, green, and blue colors, and each channel contains 32×32 pixels. Figure 3.3 presents some sample images from the dataset used in this study. As observed from the figure, the images are not very clean and contain some noise, which makes them even harder to classify under adversarial perturbations.

### *3.6.2 Traditional Preprocessing-based Defense Strategies*

The AR-GAN method utilizes a generator model (i.e., trained on unperturbed legitimate images only) to denoise traffic sign images through reconstruction before


**FIGURE 3.3** Image Samples from the LISA Traffic Sign Dataset.

feeding them to the classifier, which can be considered as a GAN-based image preprocessing step. Figure 3.4 shows a set of sample traffic sign images taken from the dataset used in this study before and after the AR-GAN generator-based preprocessing (i.e., reconstruction). Therefore, the author chose several traditional preprocessing-based defense strategies that can be used as benchmarks to compare with the classification performance of the AR-GAN traffic sign classification system. In this subsection, the author discusses these traditional defense strategies.

3.6.2.1 Gaussian Augmentation

Gaussian augmentation is a simple yet very effective preprocessing technique for improving the robustness of image classification systems against adversarial attacks. Gaussian augmentation applies random noise to every pixel of an input image (Grandvalet and Canu, 1997). By training a classifier on these images with added noise, the classifier grows robustness against Gaussian noise. The author applied independent and identically distributed Gaussian noise sampled from a zero-mean normal distribution, $\mathcal{N}(0, \sigma^2)$, to each pixel of the input images, where the standard deviation of the distribution $\mathcal{N}$ is set to

$\sigma = 1$ (Grandvalet and Canu, 1997). Figure 3.4 shows the effect of Gaussian augmentation on a set of sample traffic sign images taken from the dataset used in this study.

3.6.2.2 JPEG Compression

JPEG compression is another popular preprocessing-based adversarial defense method, which was reported to effectively reduce the effects of several powerful adversarial attacks, such as FGSM and DeepFool (Das et al., 2017; Dziugaite et al., 2016). Adversarial attacks typically aim to introduce perturbations to images resulting in high-

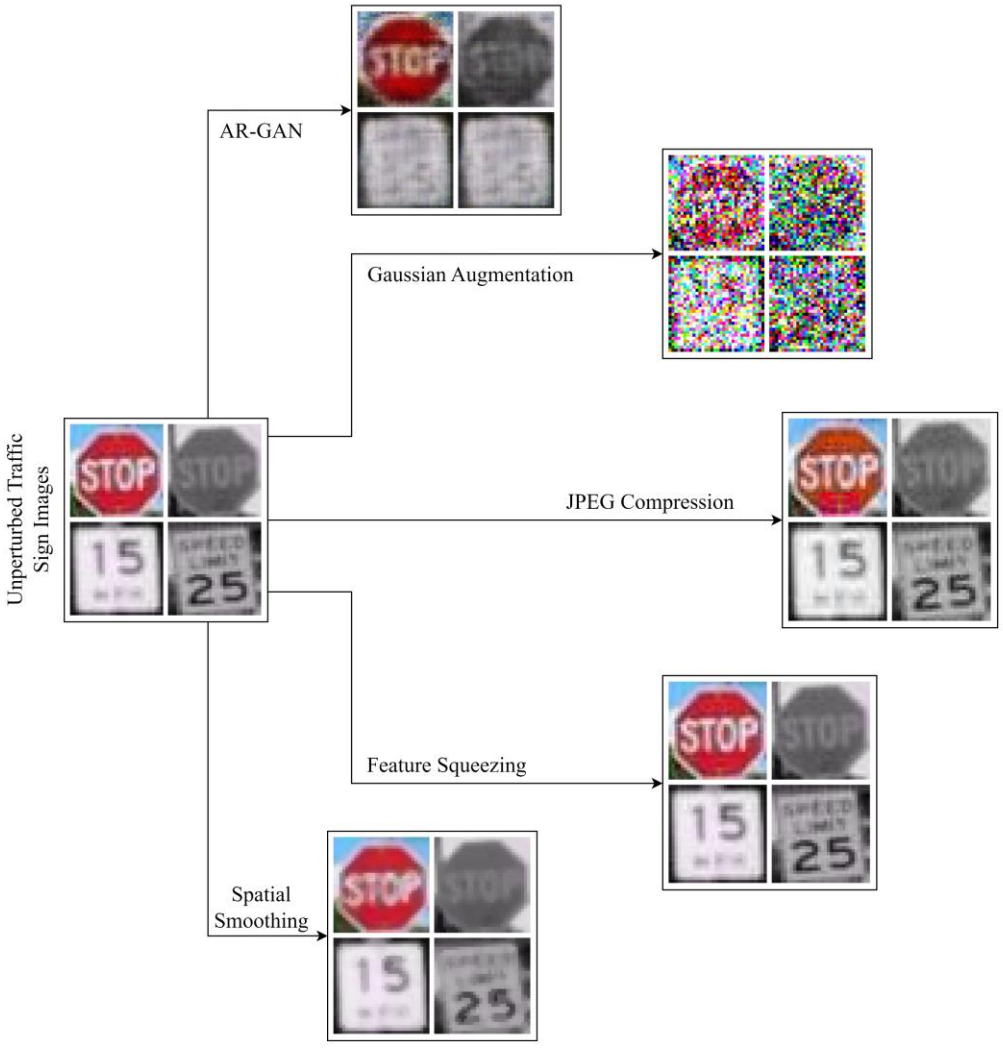



**FIGURE 3.4** Examples of Preprocessed Images using Different Defense Methods.

frequency components that are imperceptible to human eyes. However, these high-frequency components are picked up by NN-based classifiers and result in misclassification. JPEG compression eliminates these high-frequency components from the images before feeding them to the classifier, which is similar to a selective blurring of the images to help remove the additive noise due to an adversarial attack. In this study, the author sets the JPEG-compressed image quality to 50% because earlier studies reported that this setting provides an effective defense against adversarial attacks (Liu et al., 2019). Figure 3.4 shows the effect of JPEG compression on a set of sample traffic sign images taken from the dataset used in this study.

#### 3.6.2.3 Feature Squeezing

Feature squeezing, first introduced by Xu et al., 2018, is a process of reducing the color bit depth of each pixel of an input image, which is alternatively referred to as "bit squeezing" since it involves reducing the number of bits necessary to represent the color value of a pixel. This reduction of the feature space is particularly beneficial in defending against adversarial attacks since it transforms diverse feature vectors from the original space into more similar samples. The author sets the bit depth value to 4 since it provided the highest image classification accuracy for the dataset under the different types of adversarial attacks considered in this study. Figure 3.4 shows the effect of feature squeezing on a set of sample traffic sign images taken from the dataset used in this study.

#### 3.6.2.4 Spatial Smoothing

Spatial smoothing, alternatively known as blurring, is another variant of the feature squeezing technique, also proposed by Xu et al., 2018, that reduces differences at the pixel

level. In image processing, spatial smoothing is a popular technique for reducing image noise. In this study, the author utilized a median smoothing technique that was reported in (Xu et al., 2018) to be effective in mitigating adversarial attacks. In median smoothing, a sliding window is moved across an image while the center pixel of the window is replaced by the median value of its neighboring pixels. Thus, the additive noise due to adversarial attacks gets reduced in spatial smoothing, which helps in achieving better classification accuracy with a classifier. Figure 3.4 shows the effect of spatial smoothing on a set of sample traffic sign images taken from the dataset used in this study.

For all four traditional preprocessing-based adversarial defense methods used in this study, separate classifier models were trained based on the same ResNet9 architecture presented in Table 3.1. The author presents an evaluation of these models on the unperturbed (legitimate) images in the dataset used in this study in section 3.7.1.1.

### 3.7 Analysis and Results

This section presents the analysis and results based on the AR-GAN method and compares them with several traditional preprocessing-based defense methods discussed in section 3.6.2. The author divides the evaluation scenarios into two categories, i.e., (i) evaluation on unperturbed (legitimate) traffic sign images and adversarial images and (ii) evaluation under different perturbation magnitudes. The dataset discussed in section 3.6.1 was split into three groups, i.e., train set (containing 60% of the images), validation set (containing 20% of the images), and test set (containing the remaining 20% of the images), after applying random shuffling on all the images on the dataset. The same train, validation,

and test sets were used for all the evaluation scenarios to present a fair comparison among the different defense methods used in this study.

As mentioned in section 3.5.2, the AR-GAN traffic sign classification system utilizes a gradient descent-based optimization with a fixed number of gradient descent steps and random initializations to determine the input latent vector that would minimize the reconstruction error for an input image. The author conducted a sensitivity analysis of the AR-GAN traffic sign classification system's classification accuracy and end-to-end delay (i.e., the time required to perform all the steps shown in Figure 3.2) for an input unperturbed image with respect to the number of gradient descent steps and the number of random initializations (as shown in Figure 3.5). Based on this analysis, the author selected 2,250 as the fixed number of gradient descent steps and 20 as the fixed number of random initializations because these parameters yielded the highest traffic sign classification accuracy with an end-to-end delay of 0.6 seconds per image. As reported by Liu and Deng, 2021, the average delay for human drivers in recognizing traffic signs ranges from 0.5

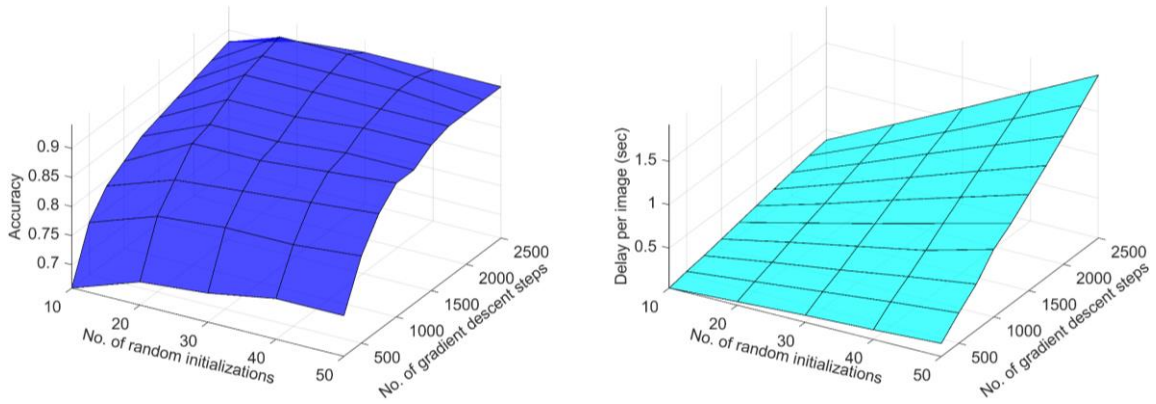



**FIGURE 3.5** Results of Sensitivity Analysis.

seconds to 2.0 seconds. Therefore, the author considered the 0.6-second end-to-end delay of the AR-GAN traffic sign classification system feasible for real-world implementations.

In this study, the author used Pytorch packages ("PyTorch," n.d.) to implement the GAN and classifier models, and the Adversarial Robustness Toolbox ("Adversarial Robustness Toolbox (ART) v1.15," n.d.) to implement the attack models and traditional preprocessing-based adversarial defense methods. The source codes are provided in Appendix B and GitHub (Salek, 2023). All the NN models in this study were trained using Nvidia Tesla A100 GPUs available in the Palmetto Cluster nodes at Clemson University ("About the Palmetto Cluster | RCD Documentation," 2023). These GPUs have a capacity of 312 trillion floating point operations per second (TFLOPS) ("NVIDIA A100 TENSOR CORE GPU," n.d.). However, these models should be implementable in real-world AVs using in-vehicle computational units. One of the recent in-vehicle computational units developed by NVIDIA is the NVIDIA Drive Thor, offering a GPU-based computational capacity of 2,000 TFLOPS ("NVIDIA Unveils DRIVE Thor — Centralized Car Computer Unifying Cluster, Infotainment, Automated Driving, and Parking in a Single, Cost-Saving System," n.d.), which is well above the capacity of the A100 GPUs utilized in this study. Besides, the in-vehicle computational units should only be responsible for running pretrained models for traffic sign image classification, whereas the training task can take place separately beforehand. Thus, the models developed under the AR-GAN method are considered feasible to be implemented in real-world AVs in terms of in-vehicle computational capacity.

### *3.7.1 Performance Evaluation on Unperturbed and Adversarial Traffic Sign Images*

In this subsection, the author presents the evaluation results obtained using unperturbed and adversarial traffic sign images generated. The adversarial images were generated under FGSM, DeepFool, C&W, and PGD attacks. The author used precision, recall, F1-score, and accuracy for performance comparison among the different types of adversarial defense methods along with the AR-GAN method. Among the performance metrics, accuracy was calculated globally for all the images in the test set, whereas the other metrics were calculated for each class, and then a weighted average was taken to present a global value.

#### 3.7.1.1 Evaluation on Unperturbed Traffic Sign Images

The author evaluated all the adversarial defense methods used in this study on the unperturbed test images. As observed from Table 3.4, all the defense methods achieved high precision, recall, F1-score, and accuracy on the unperturbed images. This proves that the classifier models used in all the defense methods in this study are well-trained to accurately classify the traffic sign images of the dataset. Although the AR-GAN method achieved about 94% classification accuracy on the unperturbed images, it was lower than the other methods because, unlike the other preprocessing-based defense methods that transform or modify an input image, the AR-GAN completely reconstructs any input images. Gaussian augmentation achieved the second-lowest classification performance compared to the other defense methods, which is also expected because this defense method itself adds some Gaussian noise to the images as part of its adversarial defense strategy.

**TABLE 3.4** Comparison of Defense Methods on Unperturbed Images

| Defense Method | Precision | Recall | F1-score | Accuracy |
|---|---|---|---|---|
| Gaussian Augmentation | 95.1% | 94.9% | 94.9% | 94.9% |
| JPEG Compression | 99.7% | 99.7% | 99.7% | 99.7% |
| Feature Squeezing | 99.4% | 99.4% | 99.4% | 99.4% |
| Spatial Smoothing | 98.8% | 98.7% | 98.7% | 98.7% |
| AR-GAN | 93.7% | 93.6% | 93.6% | 93.6% |

3.7.1.2 Evaluation on Traffic Sign Images under the FGSM Attack

The FGSM attack was implemented with an $\varepsilon = 0.1$ perturbation magnitude, as recommended by Ye and Zhu, 2018. The results are presented in Table 3.5. As observed from the table, the FGSM was able to reduce the classification performance of all the defense methods to some extent. However, the FGSM attack is not as powerful as the other attacks used in this study. Therefore, the performance metrics of the traditional preprocessing-based defense methods ranged from approximately 68% to 81%. However, the AR-GAN method was able to improve all the performance metrics by about 9-10% compared to the second-best defense method, i.e., the Gaussian augmentation.

**TABLE 3.5** Comparison of Defense Methods under the FGSM Attack

| Defense Method | Precision | Recall | F1-score | Accuracy |
|---|---|---|---|---|
| Gaussian Augmentation | 81.1% | 80.5% | 80.4% | 80.5% |
| JPEG Compression | 75.4% | 75.4% | 75.4% | 75.4% |

| Defense Method | Precision | Recall | F1-score | Accuracy |
|---|---|---|---|---|
| Feature Squeezing | 68.7% | 68.7% | 68.7% | 68.7% |
| Spatial Smoothing | 68.5% | 68.4% | 68.3% | 68.4% |
| AR-GAN | 90.2% | 90.1% | 90.1% | 90.1% |

3.7.1.3 Evaluation on Traffic Sign Images under the DeepFool Attack

The $\ell_2$-norm DeepFool attack was performed with a perturbation magnitude of $\varepsilon = 0.1$ for this evaluation scenario. The results obtained from the defense methods are presented in Table 3.6. As observed, the DeepFool attack was more effective than the FGSM attack in terms of degrading the classification performance of the traditional preprocessing-based defense methods. Feature squeezing and spatial smoothing performed the worst among all the defense methods. Gaussian augmentation was able to achieve about 74% classification accuracy, which was outperformed by the AR-GAN method with a classification accuracy of about 90% under the DeepFool attack.

**TABLE 3.6** Comparison of Defense Methods under the DeepFool Attack

| Defense Method | Precision | Recall | F1-score | Accuracy |
|---|---|---|---|---|
| Gaussian Augmentation | 74.0% | 73.8% | 73.8% | 73.8% |
| JPEG Compression | 62.0% | 61.3% | 60.8% | 61.3% |
| Feature Squeezing | 32.3% | 32.3% | 32.3% | 32.3% |
| Spatial Smoothing | 42.1% | 42.8% | 41.6% | 42.8% |
| AR-GAN | 90.7% | 90.4% | 90.4% | 90.4% |

3.7.1.4 Evaluation on Traffic Sign Images under the C&W Attack

The $\ell_2$-norm C&W attack was performed using a learning rate of 0.01 with a maximum of 10 iterations. The results are presented in Table 3.7. Feature squeezing and spatial smoothing provided the worst traffic sign classification accuracies among all the defense methods. Gaussian augmentation performed well compared to traditional preprocessing-based defense methods, with a 78.6% classification accuracy. However, it was outperformed by the AR-GAN method, which achieved approximately 90% classification accuracy under the C&W attack.

**TABLE 3.7** Comparison of Defense Methods under the C&W Attack

| **Defense Method** | **Precision** | **Recall** | **F1-score** | **Accuracy** |
|---|---|---|---|---|
| Gaussian Augmentation | 79.0% | 78.6% | 78.5% | 78.6% |
| JPEG Compression | 62.9% | 61.7% | 60.8% | 61.7% |
| Feature Squeezing | 26.2% | 26.2% | 26.2% | 26.2% |
| Spatial Smoothing | 38.1% | 38.7% | 37.9% | 38.7% |
| AR-GAN | 90.3% | 89.8% | 89.7% | 89.8% |

3.7.1.5 Evaluation on Traffic Sign Images under the PGD Attack

The $\ell_2$-norm PGD attack was performed using a maximum iteration number of 100 and a perturbation magnitude of $\varepsilon = 0.1$. The results are presented in Table 3.8. Under the PGD attack, the traffic sign classification accuracies of almost all the traditional preprocessing-based defense methods dropped below 60%, except for the Gaussian augmentation preprocessing, which achieved about 75% accuracy. The AR-GAN method

outperformed all the traditional preprocessing-based defense methods with a classification accuracy of approximately 91%.

**TABLE 3.8** Comparison of Defense Methods under the PGD Attack

| Defense Method | Precision | Recall | F1-score | Accuracy |
|---|---|---|---|---|
| Gaussian Augmentation | 75.6% | 75.4% | 75.4% | 75.4% |
| JPEG Compression | 57.1% | 56.9% | 56.6% | 56.9% |
| Feature Squeezing | 43.1% | 43.1% | 43.0% | 43.1% |
| Spatial Smoothing | 49.2% | 49.2% | 49.2% | 49.2% |
| AR-GAN | 90.8% | 90.7% | 90.7% | 90.7% |

*3.7.2 Performance Evaluation under Different Perturbation Magnitudes*

To evaluate how well the AR-GAN method performs under different perturbation magnitudes, the author varied $\varepsilon$ from 0.05 to 0.2 with a 0.05 step size for the FGSM, DeepFool, and PGD attacks following the previous studies (Khan et al., 2022; Gao and Oates, 2019; Pan et al., 2019).

Figures 3.6 to 3.8 present the results of this evaluation under different perturbation magnitudes. As observed from Figures 3.6 and 3.8, the accuracies of the traditional preprocessing-based defense methods dropped abruptly as the author increased the perturbation magnitudes of FGSM and PGD attacks. In Figure 3.7, it is observed that these drops in traffic sign classification performance happen gradually for the traditional preprocessing-based defense methods under the DeepFool attack. However, the AR-GAN method achieved above 88% classification accuracy in all these cases, except under the

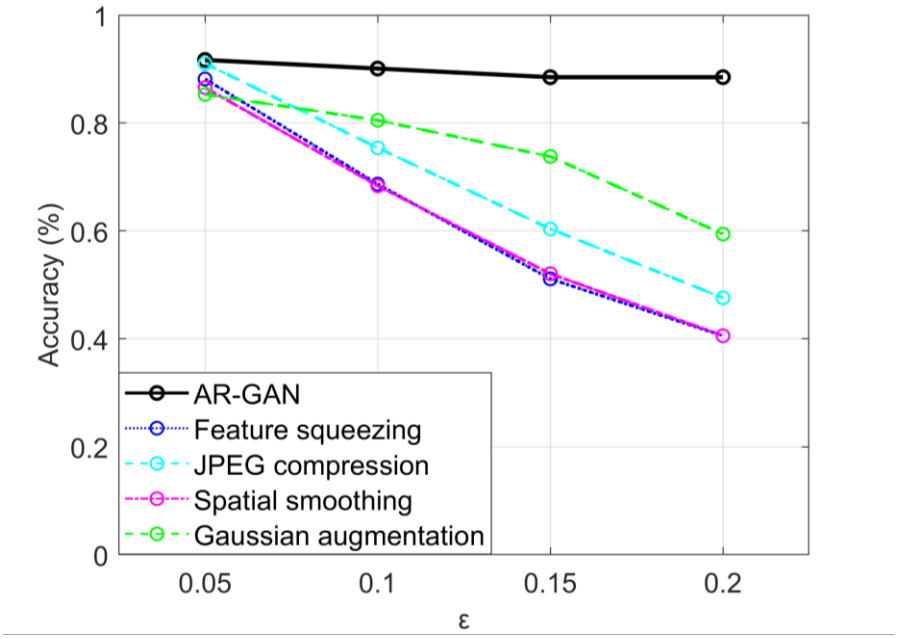



**FIGURE 3.6** Performance under the FGSM Attack with Varied Perturbations.

PGD attacks with $\varepsilon = 0.15$ and 0.2, where its accuracy dropped to about 85%. This consistency in traffic sign classification performance is achievable with the AR-GAN method because the generator in the AR-GAN method was trained to generate samples close to the distribution of the unperturbed traffic sign images. Thus, the AR-GAN method developed in this study is capable of effectively denoising the traffic sign images by reconstructing them with a generator trained on the unperturbed traffic sign images. This shows the potential of the AR-GAN method as an adversarial attack-resilient AV traffic sign classification system.

### 3.8 Discussion

The AR-GAN method developed in this study is a GAN-based adversarial attack-resilient traffic sign classification system. The GAN models in the AR-GAN method are trained based on the WGAN-GP loss function, where the generator model is based on the

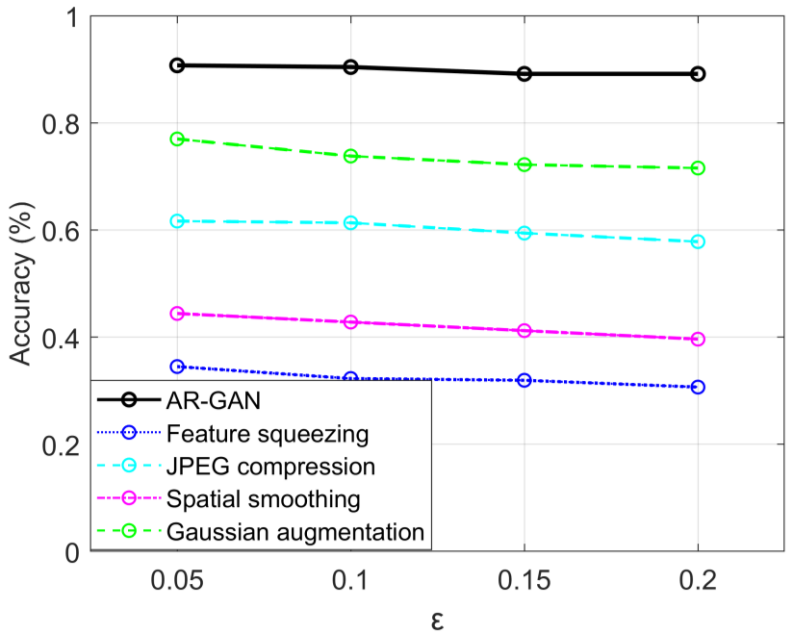



**FIGURE 3.7** Performance under the DeepFool Attack with Varied Perturbations.

DCGAN architecture, and the discriminator or critic model is based on the WGAN architecture. The generator of the AR-GAN method is trained to reconstruct any input traffic sign images close to a sample distribution of unperturbed traffic sign images, while a ResNet9-based classifier is trained to classify the traffic sign images reconstructed by the generator model. Thus, in the AR-GAN method, the generator serves the purpose of denoising the traffic sign images with adversarial perturbations through reconstruction.

The author evaluated the AR-GAN method with a real-world traffic sign dataset under no-attack and under white-box attack scenarios and compared its performance with several benchmark preprocessing-based adversarial defense methods. The results indicate that the AR-GAN method can consistently achieve high traffic sign classification performance under both no-attack and white-box attack scenarios. The AR-GAN method

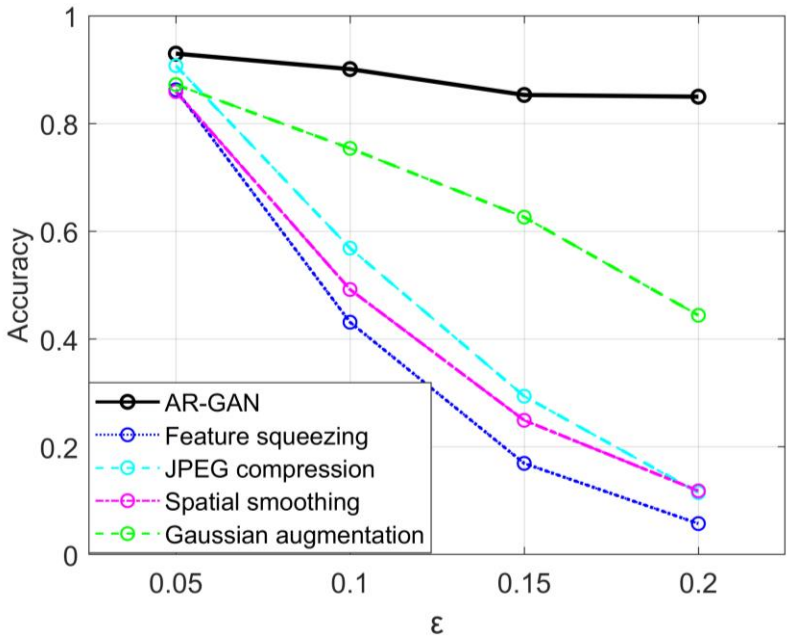



**FIGURE 3.8** Performance under the PGD Attack with Varied Perturbations.

outperformed all the benchmark preprocessing-based adversarial defense methods used in this study under different white-box adversarial attacks, such as FGSM. DeepFool, C&W, and PGD attacks. Under varied magnitudes of adversarial perturbations for the FGSM, the DeepFool, and the PGD attacks, the AR-GAN method showed consistent traffic sign classification accuracies, unlike the other preprocessing-based adversarial defense methods. This proves the efficacy of the AR-GAN method as an adversarial attack-resilient traffic sign classification system for AVs.

# CHAPTER FOUR

# SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

## 4.1 Summary

The rapid development of AI, particularly generative AI, unveils an unprecedented array of opportunities for both malicious attackers as well as for vigilant defenders. In this dissertation, the aim was to delve into different generative AI-based methods for formulating effective cyberattack detection and mitigation strategies for CAVs operating in a TCPS environment. To this end, the author developed and evaluated a generative AI-based CAN IDS for the in-vehicle CAN of a CV in Chapter 2 and a generative AI-based adversarial defense method for the perception module of an AV in Chapter 3. In both studies, the results indicated that generative AI holds a tremendous potential to safeguard CAVs from known and unknown cyberattacks. In this chapter, the author concludes based on the methods and findings of the research endeavors presented in Chapters 2 and 3 and provides recommendations for future studies focusing on generative AI-based cyberattack detection mitigation strategies.

## 4.2 Conclusions

### *4.2.1 A Hybrid Quantum-Classical RBM-based Framework for in-vehicle CAN IDS*

In this study, the author developed a hybrid quantum-classical RBM-based CAN IDS for the in-vehicle CAN of CVs. First, the author utilized a classical computer to develop CAN images from the CAN messages in a dataset. A classical NN was then used to extract features from the CAN images. Then, the author embedded the labels of the CAN

images (i.e., a CAN image represents an attack image or a normal image) directly into the images using the two rightmost columns of the images- this concludes the classical computer-based CAN image preprocessing. Next, the author trained a quantum RBM using the D-wave's Advantage 4.1 system (i.e., a semi-conductor-based quantum annealing machine) to reconstruct any CAN images with the embedded labeling pixels. After the quantum RBM is trained properly, the author replaced the labeling pixels of the test CAN image dataset with random binary bits (i.e., 0 or 1) and fed them to the quantum RBM for CAN image reconstruction. Then, the reconstructed labeling pixels were used to classify the CAN images in the test dataset as attack and normal CAN images. The author compared the hybrid quantum-classical CAN IDS with a similar but classical-only approach. The findings showed that the hybrid quantum-classical RBM-based CAN IDS outperformed the classical-only CAN IDS across all the performance metrics used for evaluating the image classification-based CAN IDSs. This proves the potential of the hybrid quantum-classical RBM-based approach to be used as the CAN IDS of a CV in a TCPS environment. Besides, as quantum computers are still in the development stage, the author found such hybrid quantum-classical cyberattack detection approaches the optimal way to leverage the efficacies of both classical and quantum computers while exceeding the detection performance of traditional classical-only attack detection approaches for CAN intrusions.

A limitation of this study is that the author only considered a specific type of intrusion attack in this study, known as the Fuzzy attack, due to some resource access limitations related to the D-wave's Advantage 4.1 system. In the future, the author aims to

explore other types of CAN intrusion attacks, such as denial-of-service and spoofing attacks, and evaluate the detection performance of the hybrid CAN IDS against them.

*4.2.2 AR-GAN*

In this study, the author developed a GAN-based adversarial defense method for AV's perception module. In particular, the focus of this study was to protect the traffic sign classification system within an AV perception module from unknown adversarial attacks. The author provided a training framework for training a generator model and a classifier model that comprise the attack-resilient traffic sign classification system of the AR-GAN method. The generator model used in this study was trained using a WGAN-GP-based loss function with a NN architecture similar to a DCGAN. The discriminator or critic used to support the training of the generator was based on the WGAN architecture. On the other hand, the classifier was trained based on the ResNet9 architecture. All these models were trained on unperturbed (i.e., legitimate) traffic sign images only to ensure that all types of adversarial attacks are unknown to the models.

The traffic sign classification system of the AR-GAN defense method utilizes generator-based image reconstruction, which helps remove adversarial perturbations from the input traffic sign images. Once the traffic sign image is reconstructed by the generator, it is fed to the classifier to identify what type of traffic sign it is. The author evaluated the AR-GAN traffic sign classification system against widely used white-box adversarial attacks, such as FGSM, DeepFool, C&W, and PGD attacks, and compared its performance with benchmark traditional adversarial defense methods, such as Gaussian augmentation, JPEG compression, feature squeezing, and spatial smoothing. The AR-GAN method

outperformed all the traditional defense methods under all attack categories considered in this study. Besides, the AR-GAN method was able to consistently achieve high traffic sign classification performance under a range of adversarial perturbation magnitudes, whereas the performance for the other traditional defense methods dropped abruptly at increased perturbation levels. This shows the potential of the AR-GAN method to be deployed as a robust attack-resilient traffic sign classification system for AVs.

A limitation of this study is that the author focused on only two classes of traffic signs, i.e., STOP signs and SPEED LIMIT signs, due to a limitation of sufficient real-world training data for the other types of US traffic signs. In the future, the author will extend this work to include all types of traffic signs by collecting traffic sign image data from the real world.

## 4.3 Recommendations

### *4.3.1 A Hybrid Quantum-Classical RBM-based Framework for in-vehicle CAN IDS*

In this subsection, the author presents some recommendations for future research endeavors related to quantum RBM-based CAN intrusion detection strategies.

1. The hybrid quantum-classical RBM-based CAN IDS was reported to outperform the classical-only RBM-based CAN IDS in this study. However, the CAN intrusion detection performance of a similar but quantum-only CAN IDS was not explored in this study. Future studies can focus on developing a CAN IDS based on quantum computers entirely, including the data preprocessing, the RBM-based CAN image reconstruction, and the classification steps.

2. Although the dataset used in this study was obtained from a real-world vehicle's CAN, the hybrid quantum-classical CAN IDS was not tested for real-time intrusion detection with a real-world CV. Further studies are necessary to test this IDS for real-time CAN intrusion detection and to explore novel strategies to ensure that the IDS is capable of detecting intrusions within the latency requirement of CV mobility and safety applications.
3. This study used quantum annealing (QA)-based training for developing the RBM model in the hybrid quantum-classical CAN IDS. Future studies should also focus on developing gate-based RBMs for CAN IDS and compare them with the QA-based RBMs.

*4.3.2 AR-GAN*

In this subsection, the author presents some recommendations for future research endeavors related to GAN-based defense methods for protecting the traffic sign classification system of an AV perception module.

1. The AR-GAN method developed in this study utilized a subset of the extended LISA traffic sign dataset that contains only US traffic signs. The focus of future studies can include improving the models in the AR-GAN method by training them with other benchmark traffic sign datasets, such as the German Traffic Sign Recognition Benchmark (GTSRB), the Belgium Traffic Sign Dataset (BEL-TSD), the Traffic Sign Recognition Multi-Task Dataset (TSR), and the Brazilian Traffic Sign Recognition Benchmark (BR-TSD).

2. Although the AR-GAN method outperformed all the other traditional preprocessing-based adversarial defense methods used in this study, the author observed a 10-14% drop in classification accuracy under the adversarial attacks compared to that of the unperturbed images. This drop in performance is not unexpected because the author did not train the models on any adversarial examples. Thus, future work can aim to include adversarial training of the AR-GAN models to develop more robust attack-resilient defense models that can achieve similar traffic sign classification performance to no-attack scenarios under adversarial attacks.
3. The AR-GAN method was not evaluated in real-world driving scenarios yet. Thus, future studies should focus on rigorous field testing and further modification of the AR-GAN method to obtain a deployment-ready AV traffic sign classification system that is computationally lightweight yet capable of achieving state-of-the-art traffic sign classification performance under different known and unknown adversarial attacks.

# APPENDICES

# Appendix A

## Python Codes Related to Chapter Two

### Sample Python Code (in IPYNB Format) for Decoding CAN Messages

```
import numpy as np
import pandas as pd
from IPython.display import display

import cantools
import can
from pprint import pprint

#Load dataset from .txt file
Attack_ds = pd.read_csv('Fuzzy_dataset_KIA_SOUL.csv',

names=['Timestamp','ID','DLC','allData','Flag'],
                          sep=',', low_memory=False)

#No. of rows in the dataset
rowCount = Attack_ds.count()[0]
print('No. of rows in dataset: ', rowCount)

Attack_ds_filtered = Attack_ds[Attack_ds['ID'] == '220']
Attack_ds_filtered.loc[Attack_ds_filtered['Flag'] == 'R', 'Flag'] = 0
Attack_ds_filtered.loc[Attack_ds_filtered['Flag'] == 'T', 'Flag'] = 1

Attack_ds_filtered.reset_index(drop=True, inplace=True)

#No. of rows in the dataset
rowCount = Attack_ds_filtered.count()[0]
print('No. of rows in dataset: ', rowCount)

for idx in range(rowCount):
    for j in range(Attack_ds_filtered['DLC'][idx]+1):
        if j == 0:
            allData = Attack_ds_filtered['allData'][idx][3*j:2+3*j]
        elif j == Attack_ds_filtered['DLC'][idx]:
            Attack_ds_filtered.loc[idx,'allData'] = allData
        else:
            allData = allData +
Attack_ds_filtered['allData'][idx][3*j:2+3*j]

#Display Flag options
print('################################################')
print('Flags listed in the dataset: ')
display(Attack_ds_filtered.groupby(['Flag'])['Flag'].count())
```

```
def merge_dicts(dict1, dict2):
    return(dict1.update(dict2))

def create_atk_ds(atk_ds, ID):
    global msg_count, nonAtk_msg, Atk_msg, error
    msg_count = 0
    nonAtk_msg = 0
    Atk_msg = 0
    error = 0
    LOG_EVERY_N = 1000

    dbc =
cantools.database.load_file(r'C:\Users\sabbi\Desktop\QCProject\opendbc\
hyundai_kia_generic.dbc')

    for idx, row in atk_ds.iterrows():
        if row['ID'] == ID:
            try:
                temp_dict =
{'Attack':row['Flag'],'Timestamp':row['Timestamp']
                            #'ID':row['ID'], #'RTR':row['RTR'],
                            #'DLC':row['DLC']
                           }

                raw_data = row['allData']
                decoded_data = dbc.decode_message(0x220,
bytes.fromhex(raw_data))
                merge_dicts(temp_dict,decoded_data)

                #print('Row {} contains data:
{}'.format(idx,temp_dict))
                #print('Timestamp: {}, Raw data:
{}'.format(row['Timestamp'], raw_data))

                if msg_count == 0:
                    Processed_ds = pd.DataFrame(temp_dict, index = [0])
                else:
                    ds_row = pd.DataFrame(temp_dict, index = [0])
                    Processed_ds = pd.concat([Processed_ds, ds_row],
ignore_index=True)


                if row['Flag'] == 1:
                    Atk_msg += 1
                else:
                    nonAtk_msg += 1

                msg_count += 1

                '''
```

```
                if msg_count == 10:
                    break
                '''
            except:
                error += 1

            if msg_count and msg_count % LOG_EVERY_N == 0:
                print(f"logging : {msg_count}")


    print('{} CAN messages decoded successfully!'.format(msg_count))
    print('{} CAN messages were injected and {} CAN messages were
authentic!'.format(Atk_msg, nonAtk_msg))
    print('{} CAN messages could not be decoded!'.format(error))

    return Processed_ds


Processed_ds = create_atk_ds(Attack_ds_filtered, ID = '###')


with pd.ExcelWriter(/data/KiaSoulFuzzy_ID_220.xlsx') as writer:
    Processed_ds.to_excel(writer, index = False)
```

## Sample Python Code (in IPYNB Format) for NN-based Feature Extraction

```
import tensorflow as tf

import cirq
import sympy
import numpy as np
import seaborn as sns
import collections
import random as random

import pandas as pd
from sklearn.preprocessing import MinMaxScaler,StandardScaler

from tensorflow.keras.models import Model
from sklearn.preprocessing import minmax_scale

# visualization tools
%matplotlib inline
import matplotlib.pyplot as plt

#Data Preprocessing
```

```
df = pd.read_excel('Data.xlsx')

df_noAtk_all = df[df['Attack']==0]
df_Atk = df[df['Attack']==1]

df_noAtk_all = df_noAtk_all.reset_index(drop=True)
df_Atk = df_Atk.reset_index(drop=True)

col_names = list(df_noAtk_all.columns)

row_noAtk_all = df_noAtk_all.count()[0]
row_Atk = df_Atk.count()[0]

row_smaller = min(row_noAtk_all,row_Atk)
num_CAN_img = int(row_smaller/(len(col_names)-2))

df_noAtk = df_noAtk_all[:][-row_Atk:].reset_index(drop=True)

init_image_size = len(col_names) - 2
train_noAtk_size = int(row_Atk*0.8)

train_noAtk = df_noAtk[:][0:train_noAtk_size]
test_noAtk = df_noAtk[:][train_noAtk_size:]

train_Atk = df_Atk[:][0:train_noAtk_size]
test_Atk = df_noAtk[:][train_noAtk_size:]

train_noAtk = train_noAtk.reset_index(drop=True)
test_noAtk = test_noAtk.reset_index(drop=True)
train_Atk = train_Atk.reset_index(drop=True)
test_Atk = test_Atk.reset_index(drop=True)

train_df = pd.concat([train_noAtk,train_Atk],
ignore_index=True).reset_index(drop=True)
test_df = pd.concat([test_noAtk,test_Atk],
ignore_index=True).reset_index(drop=True)

col_names = list(df_noAtk_all.columns)
col_names.remove('Attack')
col_names.remove('Timestamp')

df_x = train_df.drop(columns = ['Attack','Timestamp'])
df_y = train_df.drop(columns = col_names)

df_x_test = test_df.drop(columns = ['Attack','Timestamp'])
df_y_test = train_df.drop(columns = col_names)

min_max_scaler = MinMaxScaler()
```

```
scaler = StandardScaler()
df_x = pd.DataFrame(scaler.fit_transform(df_x), columns=df_x.columns)
df_x = pd.DataFrame(min_max_scaler.fit_transform(df_x),
columns=df_x.columns)

df_x_test = pd.DataFrame(scaler.fit_transform(df_x_test),
columns=df_x.columns)
df_x_test = pd.DataFrame(min_max_scaler.fit_transform(df_x_test),
columns=df_x.columns)

#Build CAN Image Dataset

x = []
y = []

np_x = df_x.to_numpy()
np_y = df_y.to_numpy()

for i in range(len(np_x)-init_image_size):
    if(i%init_image_size==0):
        img =
np_x[i:i+init_image_size,:].reshape(init_image_size,init_image_size,1)
        label = 1 if 1 in np_y[i:i+13,:] else 0
        x.append(img)
        y.append(label)

x_train = np.array(x)
y_train = np.array(y)

x = np.array(x_train)
y = np.array(y_train)

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.3, random_state=42)

#Train a NN
def create_full_classical_model(init_image_size):
    # A simple model based off LeNet
    model = tf.keras.Sequential()

model.add(tf.keras.layers.Flatten(input_shape=(init_image_size,init_ima
ge_size,1)))
    model.add(tf.keras.layers.Dense(100, activation='relu'))
    model.add(tf.keras.layers.Dense(64))
    model.add(tf.keras.layers.Dense(16))
    model.add(tf.keras.layers.Dense(1))
    return model
```

```
model = create_full_classical_model(init_image_size)
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True)
,
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()

EPOCHS = 100
BATCH_SIZE = 8

fair_history = model.fit(x_train,
          y_train,
          batch_size=BATCH_SIZE,
          epochs=EPOCHS,
          verbose=1,
          validation_data=(x_test, y_test))

fair_nn_results = model.evaluate(x_test, y_test)

plt.plot(fair_history.history['accuracy'])
plt.plot(fair_history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()
# summarize history for loss
plt.plot(fair_history.history['loss'])
plt.plot(fair_history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

def extract_feat(model, x_in):
    layer_name = model.layers[-3].name
    intermediate_layer_model = Model(inputs=model.input,

outputs=model.get_layer(layer_name).output)
    intermediate_output =
intermediate_layer_model.predict(x_test[0][None, :, :, :])
    foo_norm = minmax_scale(intermediate_output, feature_range=(0,1),
axis=1)
    return foo_norm

x_trans = []
y_trans = []
```

```
total = len(x_train)
for i in range(len(x_train)):
    x_trans.append(extract_feat(model,x_train[i]))
    y_trans.append(y_train[i])

print ('Done')

x_trans_test = []
y_trans_test = []

for i in range(len(x_test)):
    x_trans_test.append(extract_feat(model,x_test[i]))
    y_trans_test.append(y_test[i])

print ('Done')

x_trans = np.array(x_trans)
x_trans_test = np.array(x_trans_test)

x_trans = x_trans.reshape(len(x_trans),8,8)
x_trans_test = x_trans_test.reshape(len(x_trans_test),8,8)

#Perform Encoding
THRESHOLD = 0.5

x_train_bin = np.array(x_trans > THRESHOLD, dtype=np.float32)
x_test_bin = np.array(x_trans_test > THRESHOLD, dtype=np.float32)

print (x_train_bin.shape)
print (x_test_bin.shape)

import random as random

idx = random.randint(0,len(x_train_bin))
print('index:', idx , ' label : ', y_train[idx])

plt.imshow(x_train_bin[idx,:,:])
plt.colorbar()

#Save Training Data
count = 0
x_train_bin_squeezed = np.squeeze(x_train_bin)

for y in y_train:
    y_train_bin = np.expand_dims(y*np.ones([8,2]), axis=0)
    x_train_y_train = np.concatenate((x_train_bin_squeezed[count].T,
np.squeeze(y_train_bin).T), axis=0).T

    if count == 0:
        Train_all = x_train_y_train
```

```
    else:
        Train_all = np.concatenate((Train_all, x_train_y_train),
axis=0)

    count += 1

comb_final = pd.DataFrame(Train_all)

with pd.ExcelWriter('Train.xlsx') as writer:
    comb_final.to_excel(writer)

#Save Test Data
count = 0
x_test_bin_squeezed = np.squeeze(x_test_bin)

for y in y_test:
    if y == 0:
        y = 1
    else:
        y=0

    y_test_bin = np.expand_dims(y*np.ones([8,2]), axis=0)
    x_test_y_test = np.concatenate((x_test_bin_squeezed[count].T,
np.squeeze(y_test_bin).T), axis=0).T

    if count == 0:
        Test_all = x_test_y_test
    else:
        Test_all = np.concatenate((Test_all, x_test_y_test), axis=0)

    count += 1

test_final = pd.DataFrame(Test_all)

with pd.ExcelWriter('Test.xlsx') as writer:
    test_final.to_excel(writer)
```

## Sample Python Code (in IPYNB Format) for Quantum RBM Training and Classification

```
import numpy as np
import timeit
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook as tqdm

from qrbm.MSQRBM import MSQRBM
from qrbm.classicalRBM import classicalRBM
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['image.cmap'] = 'gray'

import pandas as pd

from skimage import data, color
from skimage.transform import rescale, resize, downscale_local_mean
from skimage import img_as_bool

import cv2 as cv
import random
from numpy import genfromtxt


train_data_df = pd.read_excel('Train.xlsx',header=None)
train_data = np.array(train_data_df)
test_data_df = pd.read_excel('Test.xlsx',header=None)
test_data = np.array(test_data_df)

flat_train_data = []

for i in range(int(len(train_data)/8)):
    x = []
    for k in range(8):
        for j in range(8):
            x.append(train_data[8*i+j][k])
    flat_train_data.append(x)

flat_test_data = []

for i in range(int(len(test_data)/8)):
    x = []
    for k in range(8):
        for j in range(8):
            x.append(test_data[8*i+j][k])
    flat_test_data.append(x)


#Presets

image_height = 8
image_width = 8

len_x = image_height * image_width
len_y = 0
```

```
n_visible = 64
n_hidden = 64

epochs = 50
lr = 0.1
lr_decay = 0

result_picture_tab = []

for i in range(int(len(test_data)/8)):
    x = []
    for k in range(8):
        for j in range(8):
            x.append(test_data[8*i+j][k])
    result_picture_tab.append(x)

bm = MSQRBM(n_visible=n_visible, n_hidden=n_hidden, qpu=True)
bm.image_height = image_height
bm.tqdm = tqdm
bm.result_picture_tab = result_picture_tab

bm.train(flat_train_data, len_x, len_y, epochs = epochs, lr = lr,
lr_decay = lr_decay)

weights_biases = bm.get_weights()

np.savetxt("vishid.csv", weights_biases[0], delimiter=",")
np.savetxt("visbiases.csv", weights_biases[1], delimiter=",")
np.savetxt("hidbiases.csv", weights_biases[2], delimiter=",")

def compute_metrics(bm):
    rand_label_test = []

    for i in range(len(test_data)):
        rand_mat = np.random.randint(2, size=[2,1])
        #rand_mat = random.choice([0, 1])
        x =
np.concatenate((test_data[i][0:6],np.squeeze(rand_mat)),axis =
0).tolist()
        rand_label_test.append(x)

    rand_label_test = np.array(rand_label_test)

    rand_label_picture_tab = []

    for i in range(int(len(rand_label_test)/8)):
        x = []
        for k in range(8):
            for j in range(8):
```

```
                x.append(rand_label_test[8*i+j][k])
        rand_label_picture_tab.append(x)

    result_picture_tab = []

    for i in range(int(len(test_data)/8)):
        x = []
        for k in range(8):
            for j in range(8):
                x.append(test_data[8*i+j][k])
        result_picture_tab.append(x)

    result_picture_tab = result_picture_tab[0:80]
    rand_label_picture_tab = rand_label_picture_tab[0:80]

    TP = 0
    TN = 0
    FP = 0
    FN = 0

    for i in tqdm(range(len(result_picture_tab))):
        True_image = np.reshape(result_picture_tab[i], (image_width,
image_height))
        True_label = True_image[7][0]
        #print('True label: {}'.format(True_label))

        Recon_image = bm.generate(rand_label_picture_tab[i])
        Recon_image = np.reshape(Recon_image, (image_width,
image_height))
        Pred_label = int(sum(sum(Recon_image[6:8][:]))/16 > 0.5)
        #print('Predicted label: {}'.format(Pred_label))

        if True_label == Pred_label:
            if True_label == 1:
                TN += 1
            else:
                TP += 1
        else:
            if True_label == 1:
                FP += 1
            else:
                FN += 1

    print('TP (test): {}'.format(TP))
    print('TN (test): {}'.format(TN))
    print('FP (test): {}'.format(FP))
    print('FN (test): {}'.format(FN))

    Accuracy = (TP + TN)/(TP + TN + FP + FN)
    Precision = TP/(TP + FP)
```

```
    Recall = TP/(TP + FN)
    F1 = 2*Precision*Recall/(Precision + Recall)

    print('Accuracy (test): {}'.format(Accuracy))
    print('Precision (test): {}'.format(Precision))
    print('Recall (test): {}'.format(Recall))
    print('F1 score (test): {}'.format(F1))


saved_weights = genfromtxt('vishid_.csv', delimiter=',')
saved_visbiases = genfromtxt('visbiases.csv', delimiter=',')
saved_hidbiases = genfromtxt('hidbiases.csv', delimiter=',')

bm = MSQRBM(n_visible=n_visible, n_hidden=n_hidden, qpu=False)
bm.image_height = image_height
bm.result_picture_tab = result_picture_tab
bm.set_weights(saved_weights,saved_visbiases,saved_hidbiases)
bm.tqdm = tqdm

compute_metrics(bm)
```

## Appendix B

## Python Codes Related to Chapter Three

### Sample Python Code (in IPYNB Format) for Training the Classifier

```
import os
import torch
import torchvision
import numpy as np
from torch.utils.data import DataLoader, ConcatDataset
from torchvision.utils import make_grid
import torchvision.transforms as T

import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

matplotlib.rcParams['figure.facecolor'] = '#ffffff'

random_seed = 42
torch.manual_seed(random_seed);

import pickle
from Split_data import random_split

data_directory = "./data"
filename = "stop_speed.pkl"
file_path = os.path.join(data_directory, filename)

# Load the data from the file
with open(file_path, "rb") as file:
    data = pickle.load(file)

train_ds, val_ds, test_ds = random_split(data)

from collections import Counter
train_classes = [label for _, label in train_ds]
val_classes = [label for _, label in val_ds]
test_classes = [label for _, label in test_ds]

train_class_size = Counter(train_classes)
val_class_size = Counter(val_classes)
test_class_size = Counter(test_classes)
all_class_size = train_class_size + val_class_size + test_class_size

print(f'Size of train classes: {train_class_size}')
print(f'Size of validation classes: {val_class_size}')
```

```
print(f'Size of test classes: {test_class_size}')
print(f'Size of all classes in train, val, and test sets:
{all_class_size}')

# PyTorch data loaders
n_cores = os.cpu_count()
batch_size = 64
stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
num_classes = 2

train_dl = DataLoader(train_ds, batch_size, shuffle=True,
num_workers=int(n_cores/2), pin_memory=True)
val_dl = DataLoader(val_ds, batch_size*2, shuffle=True,
num_workers=int(n_cores/2), pin_memory=True)

def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]

def show_images(images, nmax=64):
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denorm(images.detach()[:nmax]),
nrow=8).permute(1, 2, 0))
    #ax.imshow(make_grid(images.detach()[:nmax], nrow=8).permute(1, 2,
0))

def show_batch(dl, nmax=64):
    for images, _ in dl:
        show_images(images, nmax)
        break

show_batch(train_dl)

# Move data to GPU
from deviceSelector import DeviceDataLoader, to_device

torch.cuda.empty_cache()
train_dl = DeviceDataLoader(train_dl)
val_dl = DeviceDataLoader(val_dl)

print(f'train dataloader device: {train_dl.device}')
print(f'validation dataloader device: {val_dl.device}')

# Instantiate ResNet9 Model
from ResNet9 import ResNet9
model = to_device(ResNet9(3,num_classes),device='cuda')

# Define training parameters
epochs = 8
```

```
max_lr = 0.01
grad_clip = 0.1
weight_decay = 1e-4
opt_func = torch.optim.Adam

from ResNet9 import *

history = [evaluate(model, val_dl)]

%%time
history += fit_one_cycle(epochs, max_lr, model, train_dl, val_dl,
                             grad_clip=grad_clip,
                             weight_decay=weight_decay,
                             opt_func=opt_func)

from plot_history import *

plot_accuracies(history)
plot_losses(history)
plot_lrs(history)

# Save the model
torch.save(model.state_dict(),
'./trained_models/ResNet9/resnet9_m20.pth')

def predict_image(img, model, device='cuda'):
    xb = to_device(img.unsqueeze(0), device)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return preds[0].item()

from sklearn.metrics import classification_report

def eval_test(test_ds, model, device='cuda'):
    with torch.no_grad():
        correct = 0
        total = 0
        y_true = []
        y_pred = []

        for img, label in test_ds:
            xb = to_device(img.unsqueeze(0), device)
            yb = model(xb)
            _, preds  = torch.max(yb, dim=1)
            total += 1
            correct += (preds[0] == label).sum().item()
            predicted=preds[0].to('cpu')
            y_true.append(label)
            y_pred.append(predicted)
```

```
        print('Test Accuracy: {}%'.format(100 * correct / total))

        # Generate a classification report
        print(classification_report(y_true, y_pred))

# Evaluate the model on the test dataset
eval_test(test_ds, model_m)
```

**Supporting Python Scripts for Training the Classifier**

*Script 1: Split_data.py*

```
import random
import torch
from torchvision import transforms

random.seed(42)

def random_split(data):
    random.seed(42)

    # Define the mean and standard deviation for normalization
    stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)

    # Create the transformation for normalization
    transform = transforms.Compose([
        transforms.Normalize(*stats)
    ])

    # Apply normalization to the data
    normalized_data = [(transform(tensor), label) for tensor, label in
data]

    random.shuffle(normalized_data)

    # Calculate the lengths of train, validation, and test sets based on
the ratios
    train_ratio = 0.6
    val_ratio = 0.2
    test_ratio = 0.2

    total_samples = len(normalized_data)
    train_samples = int(train_ratio * total_samples)
    val_samples = int(val_ratio * total_samples)
    test_samples = total_samples - train_samples - val_samples

    # Split the normalized data into train, validation, and test sets
    train_data = normalized_data[:train_samples]
```

```
    val_data = normalized_data[train_samples : train_samples +
val_samples]
    test_data = normalized_data[train_samples + val_samples:]

    return train_data, val_data, test_data
```

*Script 2: ResNet9.py*

```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F


def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

class ImageClassificationBase(nn.Module):
    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc  =  torch.stack(batch_accs).mean()           #  Combine
accuracies
        return      {'val_loss':      epoch_loss.item(),      'val_acc':
epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, val_loss:
{:.4f}, val_acc: {:.4f}".format(
            epoch,       result['lrs'][-1],       result['train_loss'],
result['val_loss'], result['val_acc']))


def conv_block(in_channels, out_channels, pool=False):
```

```
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3,
padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True)]
    if pool: layers.append(nn.MaxPool2d(2))
    return nn.Sequential(*layers)


class ResNet9(ImageClassificationBase):
    def __init__(self, in_channels, num_classes):
        super().__init__()

        self.conv1 = conv_block(in_channels, 64)
        self.conv2 = conv_block(64, 128, pool=True)
        self.res1 = nn.Sequential(conv_block(128, 128), conv_block(128,
128))

        self.conv3 = conv_block(128, 256, pool=True)
        self.conv4 = conv_block(256, 512, pool=True)
        self.res2 = nn.Sequential(conv_block(512, 512), conv_block(512,
512))

        self.classifier = nn.Sequential(nn.MaxPool2d(4),
                                        nn.Flatten(),
                                        nn.Dropout(0.2),
                                        nn.Linear(512, num_classes))

    def forward(self, xb):
        out = self.conv1(xb)
        out = self.conv2(out)
        out = self.res1(out) + out
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.res2(out) + out
        out = self.classifier(out)
        return out


@torch.no_grad()
def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']

def fit_one_cycle(epochs, max_lr, model, train_loader, val_loader,
                  weight_decay=0,                         grad_clip=None,
opt_func=torch.optim.SGD):
    torch.cuda.empty_cache()
    history = []
```

```
    # Set up cutom optimizer with weight decay
    optimizer        =        opt_func(model.parameters(),        max_lr,
weight_decay=weight_decay)
    # Set up one-cycle learning rate scheduler
    sched   =   torch.optim.lr_scheduler.OneCycleLR(optimizer,   max_lr,
epochs=epochs,

steps_per_epoch=len(train_loader))

    for epoch in range(epochs):
        # Training Phase
        model.train()
        train_losses = []
        lrs = []
        for batch in train_loader:
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()

            # Gradient clipping
            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
            optimizer.zero_grad()

            # Record & update learning rate
            lrs.append(get_lr(optimizer))
            sched.step()

        # Validation phase
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['lrs'] = lrs
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

*Script 3: deviceSelector.py*

```
import torch

def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    """Move tensor(s) to chosen device"""
```

```
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device=get_default_device()):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

*Script 4: plot_history.py*

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')

matplotlib.rcParams['figure.facecolor'] = '#ffffff'

def plot_accuracies(history):
    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs');

def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. No. of epochs');

def plot_lrs(history):
    lrs = np.concatenate([x.get('lrs', []) for x in history])
    plt.plot(lrs)
    plt.xlabel('Batch no.')
    plt.ylabel('Learning rate')
    plt.title('Learning Rate vs. Batch no.');
```

**Sample Python Code (in IPYNB Format) for Training the Generator**

```
import os

import torch
import torchvision
from torch.utils.data import DataLoader
from torchsummary import summary

import pickle
from Split_data import random_split

from WGAN_GP import Generator, Discriminator
from Train_WGAN_GP import train_WGANGP

random_seed = 42
torch.manual_seed(random_seed);

# Parameters
INPUT_LATENT = 128
batch_size = 128
N_CORES = os.cpu_count()

# load dataset
data_file_path = os.path.join("./data", "stop_speed.pkl")

# Load the data from the file
with open(data_file_path, "rb") as data_file:
    reduced_data = pickle.load(data_file)

train_ds, val_ds, test_ds = random_split(reduced_data)

train_loader = DataLoader(
    train_ds,
    batch_size,
    shuffle=True,
    num_workers=int(N_CORES/2),
    pin_memory=True
)

val_loader = DataLoader(
    test_ds,
    batch_size*2,
    num_workers=int(N_CORES/2),
    pin_memory=True
)

# Set compute devices
device_D = torch.device('cuda')
```

```
device_G = torch.device('cuda')

# load generator model
netG = Generator()
summary(netG, input_size = (INPUT_LATENT, 1, 1), device = 'cpu')

# load discriminator model
netD = Discriminator()
summary(netD, input_size = (3, 32, 32), device = 'cpu')

# set folder to save model checkpoints
model_folder = os.path.abspath('./trained_models/WGAN_GP')
if not os.path.exists(model_folder):
    os.mkdir(model_folder)

# set folder to save generated images
img_folder = os.path.abspath('./Generated_imgs')
if not os.path.exists(img_folder):
    os.mkdir(img_folder)

# Load last saved models (if any)
check_point_path =  './trained_models/WGAN_GP/model_snapshots.pth'

if os.path.exists(check_point_path):
    checkpoint = torch.load(check_point_path)

    inital_epoch = checkpoint['epoch']

    netG.load_state_dict(checkpoint['netG_state_dict'])
    netD.load_state_dict(checkpoint['netD_state_dict'])

# Move models to GPU
netG = netG.to(device_G)
netD = netD.to(device_D)


# Train WGAN-GP
inital_epoch = 0

train_WGANGP(train_loader, val_loader, netD, netG, inital_epoch)
```

**Supporting Python Scripts for Training the Generator**

*Script 1: Train_WGAN_GP.py*

```
import os

import copy
import time
import pickle
import numpy as np

import torch
import torch.optim as optim
import torch.autograd as autograd
from torchvision.utils import save_image
import torchvision
import torchvision.transforms as T
from tqdm.notebook import tqdm

from WGAN_GP import Generator, Discriminator
from utils_WGAN_GP import adjust_lr, compute_gradient_penalty, denorm

random_seed = 42
torch.manual_seed(random_seed);


def train_WGANGP(train_loader, val_loader, netD, netG, inital_epoch):

    # Parameters
    ITERS = 400000
    INPUT_LATENT = 128
    LAMBDA = 10 # Gradient penalty lambda hyperparameter
    CRITIC_ITERS = 5 # Critic iterations per generator iteration

    device_D = 'cuda'
    device_G = 'cuda'

    batch_size = 128
    in_channel = 3
    height = 32
    width = 32

    learning_rate = 1e-4
    display_steps = 500

    check_point_path = './trained_models/WGAN_GP/model_snapshots.pth'

    # set optimizer for generator and discriminator
    optimizerD    =    optim.Adam(netD.parameters(),    lr=learning_rate,
betas=(0.5, 0.9))
    optimizerG    =    optim.Adam(netG.parameters(),    lr=learning_rate,
betas=(0.5, 0.9))
```

```
    print('Number of training batches: {}, Number of validation batches:
{}'.format(len(train_loader), len(val_loader)))

    save_losses = []
    dev_disc_costs = []

    if os.path.exists('./trained_models/WGAN_GP/lisa_losses_gp.pickle'):
        with   open   ('./trained_models/WGAN_GP/lisa_losses_gp.pickle',
'rb') as fp:
            save_losses = pickle.load(fp)

    if os.path.exists('./trained_models/WGAN_GP/dev_disc_costs.pickle'):
        with   open   ('./trained_models/WGAN_GP/dev_disc_costs.pickle',
'rb') as fp:
            dev_disc_costs = pickle.load(fp)

    one = torch.tensor(1, dtype=torch.float)
    mone = one * -1

    one = one.to(device_D)
    mone = mone.to(device_D)


    # Training
    print('Training starts ...')

    for iteration in range(inital_epoch, ITERS, 1):

        start_time = time.time()

        adjust_lr(optimizerD,   iteration,   init_lr   =   learning_rate,
total_iteration = ITERS)
        adjust_lr(optimizerG,   iteration,   init_lr   =   learning_rate,
total_iteration = ITERS)

        d_loss_real = 0
        d_loss_fake = 0

        #for iter_d in range(CRITIC_ITERS):
        for i, (imgs, _) in enumerate(tqdm(train_loader)):

            ############################
            # (1) Update D network
            ###########################
            for p in netD.parameters():
                p.requires_grad = True

            real_imgs = autograd.Variable(imgs.to(device_D))

            optimizerD.zero_grad()

            # Sample noise as generator input
            z       =       autograd.Variable(torch.randn(imgs.size(0),
INPUT_LATENT,1,1))
```

```
            z = z.to(device_G)

            # Generate a batch of images
            fake_imgs = netG(z).cpu()
            fake_imgs = fake_imgs.to(device_D)

            # Real images
            real_validity = netD(real_imgs)
            d_loss_real = real_validity.mean()
            d_loss_real.backward(mone)

            # Fake images
            fake_validity = netD(fake_imgs)
            d_loss_fake = fake_validity.mean()
            d_loss_fake.backward(one)

            # Gradient penalty
            gradient_penalty      =      compute_gradient_penalty(netD,
real_imgs.data, fake_imgs.data, device_D)
            gradient_penalty.backward()

            # Adversarial loss
            loss_D   =   d_loss_fake   -   d_loss_real   +   LAMBDA   *
gradient_penalty

            #loss_D.backward()
            optimizerD.step()
            optimizerG.zero_grad()

            del      real_validity,      fake_validity,      fake_imgs,
gradient_penalty, real_imgs

            # Train the generator every n_critic iterations

            if (i + 1)% CRITIC_ITERS == 0 or (i + 1) == len(train_loader):

                ############################
                # (2) Update G network
                ###########################
                for p in netD.parameters():
                    p.requires_grad = False  # to avoid computation

                # Generate a batch of images
                fake_imgs = netG(z).cpu()
                fake_imgs = fake_imgs.to(device_D)

                #  Loss  measures  generator's  ability  to  fool  the
discriminator
                # Train on fake images
                fake_validity = netD(fake_imgs)
                g_loss = fake_validity.mean()
                g_loss.backward(mone)
                loss_G = -g_loss
```

```
                #loss_G.backward()
                optimizerG.step()

                del fake_validity


        save_losses.append([loss_D.item(), loss_G.item()])

        if (iteration + 1) % display_steps == 0 or (iteration + 1) ==
ITERS:

            print('batch    {:>3}/{:>3},    D_cost    {:.4f},    G_cost
{:.4f}\r'.format(iteration + 1, ITERS,loss_D.item(), loss_G.item()))

            with  open('./trained_models/WGAN_GP/lisa_losses_gp.pickle',
'wb') as fp:
                pickle.dump(save_losses, fp)

            # snapshots for model
            modelG_copy = copy.deepcopy(netG)
            modelG_copy = modelG_copy.cpu()

            modelG_state_dict = modelG_copy.state_dict()

            modelD_copy = copy.deepcopy(netD)
            modelD_copy = modelD_copy.cpu()

            modelD_state_dict = modelD_copy.state_dict()

            torch.save({
                'netG_state_dict': modelG_state_dict,
                'netD_state_dict': modelD_state_dict,
                'epoch': iteration
                }, check_point_path)

            del     modelG_copy,     modelG_state_dict,     modelD_copy,
modelD_state_dict

        # save generator model after certain iteration
        if (iteration + 1) % display_steps == 0 :

            g_path     =     './trained_models/WGAN_GP/G_lisa_gp_'     +
str(iteration) + '.pth'

            model_copy = copy.deepcopy(netG)
            model_copy = model_copy.cpu()
            model_state_dict = model_copy.state_dict()
            torch.save(model_state_dict, g_path)

            del model_copy

        # save LISA generated images by generator model every 1000 time

        if (iteration + 1) % display_steps == 0 :
```

```
            denorm_fake_imgs = denorm(fake_imgs)
            save_image(denorm_fake_imgs.data,
'./Generated_imgs/sample_{}.png'.format(iteration), nrow=8)

            costs_avg = 0.0
            disc_count = 0

            # validate GAN model
            with torch.no_grad():
                for images,_ in val_loader:

                    imgs = images.to(device_D)

                    D = netD(imgs)

                    costs_avg += -D.mean().cpu().data.numpy()
                    disc_count += 1

                    del images, imgs

            costs_avg = costs_avg / disc_count

            dev_disc_costs.append(costs_avg)

            with  open('./trained_models/WGAN_GP/dev_disc_costs.pickle',
'wb') as fp:
                pickle.dump(dev_disc_costs, fp)

            print('batch   {:>3}/{:>3},   validation   disc   cost   :
{:.4f}'.format(iteration, ITERS, costs_avg))
```

*Script 2: utils_WGAN_GP.py*

```
import torch
import torch.autograd as autograd
import torchvision
import numpy as np


# Learning Rate Adjustment

def adjust_lr(optimizer, iteration, init_lr = 1e-4, total_iteration =
200000):

    gradient = (float(-init_lr) / total_iteration)
    lr = gradient * iteration + init_lr

    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

```
# Calculate Gradient Penalty Loss for WGAN-GP

def compute_gradient_penalty(D, real_samples, fake_samples, device):

    # Random weight term for interpolation between real and fake samples
    alpha = torch.Tensor(np.random.random((real_samples.size(0), 1, 1,
1)))
    alpha = alpha.expand(real_samples.size(0), real_samples.size(1),
real_samples.size(2), real_samples.size(3))
    alpha = alpha.to(device)

    # Get random interpolation between real and fake samples
    interpolates = (alpha * real_samples + ((1 - alpha) *
fake_samples)).requires_grad_(True)
    d_interpolates = D(interpolates)
    fake = autograd.Variable(torch.Tensor(real_samples.shape[0],
1).fill_(1.0), requires_grad=False)
    fake = fake.to(device)

    # Get gradient w.r.t. interpolates
    gradients = autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=fake,
        create_graph=True,
        retain_graph=True,
        only_inputs=True,
    )[0]
    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()

    return gradient_penalty


# Denormalize image tensors

def denorm(img_tensors):
    stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
    return img_tensors * stats[1][0] + stats[0][0]
```

*Script 3: WGAN_GP.py*

```
import torch
import torch.nn as nn

latent_size = 128

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
```

```
        convT1 = nn.Sequential(
            # in: latent_size x 1 x 1
            nn.ConvTranspose2d(latent_size,      512,      kernel_size=4,
stride=1, padding=0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True)
            # out: 512 x 4 x 4
        )

        convT2 = nn.Sequential(
            nn.ConvTranspose2d(512,    256,    kernel_size=4,    stride=2,
padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True)
            # out: 256 x 8 x 8
        )

        convT3 = nn.Sequential(
            nn.ConvTranspose2d(256,    128,    kernel_size=4,    stride=2,
padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True)
            # out: 128 x 16 x 16
        )

        convT4 = nn.Sequential(
            nn.ConvTranspose2d(128,     3,    kernel_size=4,    stride=2,
padding=1, bias=False),
            nn.Tanh()
            # out: 3 x 32 x 32
        )

        self.convT1 = convT1
        self.convT2 = convT2
        self.convT3 = convT3
        self.convT4 = convT4

    def forward(self, input):
        output = self.convT1(input)
        output = self.convT2(output)
        output = self.convT3(output)
        output = self.convT4(output)

        return output



class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        conv1 = nn.Sequential(
            # in: 3 x 32 x 32
```

```
            nn.Conv2d(3,   32,   kernel_size=4,   stride=2,   padding=1,
bias=False),
            #nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2, inplace=True)
            # out: 32 x 16 x 16
        )

        conv2 = nn.Sequential(
            nn.Conv2d(32,   64,   kernel_size=4,   stride=2,   padding=1,
bias=False),
            #nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            # out: 64 x 8 x 8
        )

        conv3 = nn.Sequential(
            nn.Conv2d(64,   128,   kernel_size=4,   stride=2,   padding=1,
bias=False),
            #nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True)
            # out: 128 x 4 x 4
        )

        conv4 = nn.Sequential(
            nn.Conv2d(128,   256,   kernel_size=4,   stride=1,   padding=0,
bias=False),
            #nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            # out: 256 x 1 x 1

            nn.Flatten()
        )

        self.conv1 = conv1
        self.conv2 = conv2
        self.conv3 = conv3
        self.conv4 = conv4
        self.linear = nn.Linear(256, 1)

    def forward(self, input):
        output = self.conv1(input)
        output = self.conv2(output)
        output = self.conv3(output)
        output = self.conv4(output)
        output = self.linear(output)
        return output
```

**Sample Python Code (in IPYNB Format) for Evaluating the AR-GAN**

```
import os
import numpy as np
import pickle
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torchvision.transforms as T
from torch.utils.data import DataLoader

from utils_AR_GAN import adjust_lr, get_z_sets, get_z_star,
Resize_Image
from Split_data import random_split

from WGAN_GP import Generator
from torchsummary import summary
import copy

batch_size = 128
in_channel = 3
height = 32
width = 32
num_classes = 2

display_steps = 20

# load dataset
data_file_path = os.path.join("./data", "stop_speed.pkl")

# Load the data from the file
with open(data_file_path, "rb") as data_file:
    reduced_data = pickle.load(data_file)

train_ds, val_ds, test_ds = random_split(reduced_data)

# Move data to GPU
from deviceSelector import DeviceDataLoader, to_device

torch.cuda.empty_cache()

n_cores = os.cpu_count()
test_loader = DataLoader(test_ds,
                      batch_size,
                      shuffle = False,
                      num_workers = int(n_cores/2),
                      pin_memory = True)
test_loader = DeviceDataLoader(test_loader)
```

```
from torchvision.utils import make_grid
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

matplotlib.rcParams['figure.facecolor'] = '#ffffff'

stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)

def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]

def show_images(images, nmax=64):
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denorm(images.cpu().detach()[:nmax]),
nrow=8).permute(1, 2, 0))
    #ax.imshow(make_grid(images.detach()[:nmax], nrow=8).permute(1, 2,
0))

def show_batch(dl, nmax=64):
    for images, _ in dl:
        show_images(images, nmax)
        break

show_batch(test_loader)

from deviceSelector import DeviceDataLoader, to_device
from ResNet9 import ResNet9

device_model = 'cuda'
model = to_device(ResNet9(3,num_classes), device='cuda')
model.load_state_dict(torch.load('./trained_models/ResNet9/resnet9_m19_
retrained.pth'))

learning_rate = 10.0
rec_iters = [1000]
rec_rrs = [20]
decay_rate = 0.1
global_step = 3.0
generator_input_size = 32

INPUT_LATENT = 128
device_generator = torch.device('cuda')

ModelG = Generator()
generator_path = './trained_models/WGAN_GP/G_lisa_gp_4519.pth'
ModelG.load_state_dict(torch.load(generator_path))
```

```
summary(ModelG, input_size = (INPUT_LATENT,1,1), device = 'cpu')

ModelG = ModelG.to(device_generator)
loss = nn.MSELoss()

model.eval()

running_corrects = 0
epoch_size = 0

is_input_size_diff = False

save_test_results = []

for rec_iter in rec_iters:
    for rec_rr in rec_rrs:

        for batch_idx, (inputs, labels) in enumerate(test_loader):

            # size change

            if inputs.size(2) != generator_input_size :

                target_shape = (inputs.size(0), inputs.size(1),
generator_input_size, generator_input_size)

                data = Resize_Image(target_shape, inputs)
                data = data.to(device_generator)

                is_input_size_diff = True

            else :
                data = inputs.to(device_generator)

            # find z*

            _, z_sets = get_z_sets2(ModelG, data, learning_rate, \
                                    loss, device_generator,
rec_iter = rec_iter, \
                                    rec_rr = rec_rr, input_latent =
INPUT_LATENT, global_step = global_step)

            z_star = get_z_star(ModelG, data, z_sets, loss,
device_generator)

            # generate data

            data_hat =
ModelG(z_star.to(device_generator)).cpu().detach()
```

```
            # size back

            if is_input_size_diff:

                target_shape = (inputs.size(0), inputs.size(1), height,
width)
                data_hat = Resize_Image(target_shape, data_hat)

            # classifier
            data_hat = data_hat.to(device_model)

            labels = labels.to(device_model)

            # evaluate

            outputs = model(data_hat)

            _, preds = torch.max(outputs, 1)

            # statistics
            running_corrects += torch.sum(preds == labels.data)
            epoch_size += inputs.size(0)

            if batch_idx % display_steps == 0:
                print('{:>3}/{:>3} average acc {:.4f}\r'\
                      .format(batch_idx+1, len(test_loader),
running_corrects.double() / epoch_size))

            del labels, outputs, preds, data, data_hat,z_star

        test_acc = running_corrects.double() / epoch_size

        print('rec_iter : {}, rec_rr : {}, Test Acc:
{:.4f}'.format(rec_iter, rec_rr, test_acc))

        save_test_results.append(test_acc)

del test_loader
```

**Supporting Python Script for Evaluating the AR-GAN**

*Script 1: utils_AR_GAN.py*

```
import torch
import torch.optim as optim
import numpy as np
from torchvision import transforms
```

```
import math

def adjust_lr(optimizer, cur_lr, decay_rate = 0.1, global_step = 1,
rec_iter = 200):

    lr = cur_lr * decay_rate ** (global_step / int(math.ceil(rec_iter *
0.8)))

    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

    return lr

"""
To get R random different initializations of z from L steps of Gradient
Descent.
rec_iter : the number of L of Gradient Descent steps
tec_rr : the number of different random initialization of z
"""

def get_z_sets(model, data, lr, loss, device, rec_iter = 1000, rec_rr =
20, input_latent = 128, global_step = 1):

    display_steps = 100

    # the output of R random different initializations of z from L steps
of GD
    z_hats_recs = torch.Tensor(rec_rr, data.size(0), input_latent,1,1)

    # the R random differernt initializations of z before L steps of GD
    z_hats_orig = torch.Tensor(rec_rr, data.size(0), input_latent,1,1)

    for idx in range(len(z_hats_recs)):

        z_hat = torch.randn(data.size(0), input_latent,1,1).to(device)
        z_hat = z_hat.detach().requires_grad_()

        cur_lr = lr

        optimizer = optim.SGD([z_hat], lr = cur_lr, momentum = 0.7)

        z_hats_orig[idx] = z_hat.cpu().detach().clone()

        for iteration in range(rec_iter):

            optimizer.zero_grad()

            fake_image = model(z_hat)

            fake_image = fake_image.view(-1, data.size(1), data.size(2),
data.size(3))

            reconstruct_loss = loss(fake_image, data)
```

```
            reconstruct_loss.backward()

            optimizer.step()

            cur_lr = adjust_lr(optimizer, cur_lr, global_step =
global_step, rec_iter= rec_iter)

        z_hats_recs[idx] = z_hat.cpu().detach().clone()

    return z_hats_orig, z_hats_recs

"""
To get z* so as to minimize reconstruction error between generator G and
an image x
"""

def get_z_star(model, data, z_hats_recs, loss, device):

    reconstructions = torch.Tensor(len(z_hats_recs))

    for i in range(len(z_hats_recs)):

        z = model(z_hats_recs[i].to(device))

        z = z.view(-1, data.size(1), data.size(2), data.size(3))

        reconstructions[i] = loss(z, data).cpu().item()

    min_idx = torch.argmin(reconstructions)

    return z_hats_recs[min_idx]


def Resize_Image(target_shape, images):

    batch_size, channel, width, height = target_shape

    Resize = transforms.Compose([
        transforms.ToPILImage(),
        transforms.Resize((width,height)),
        transforms.ToTensor(),
    ])

    result = torch.zeros((batch_size, channel, width, height),
dtype=torch.float)

    for idx in range(len(result)):
        result[idx] = Resize(images.data[idx])

    return result
```

# REFERENCES

About the Palmetto Cluster | RCD Documentation [WWW Document], 2023. URL https://docs.rcd.clemson.edu/palmetto/about (accessed 7.8.23).

Adachi, S.H., Henderson, M.P., 2015. Application of quantum annealing to training of deep neural networks. arXiv preprint arXiv:1510.06356.

Adversarial Robustness Toolbox (ART) v1.15 [WWW Document], n.d. URL https://github.com/Trusted-AI/adversarial-robustness-toolbox/tree/main (accessed 7.8.23).

Akcay, S., Atapour-Abarghouei, A., Breckon, T.P., 2019. GANomaly: Semi-supervised Anomaly Detection via Adversarial Training, in: Jawahar, C.V., Li, H., Mori, G., Schindler, K. (Eds.), Computer Vision – ACCV 2018, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 622–637. https://doi.org/10.1007/978-3-030-20893-6_39

Arjovsky, M., Chintala, S., Bottou, L., 2017. Wasserstein Generative Adversarial Networks, in: Proceedings of the 34th International Conference on Machine Learning. Presented at the International Conference on Machine Learning, PMLR, pp. 214–223.

Bai, T., Luo, J., Zhao, J., Wen, B., Wang, Q., 2021. Recent Advances in Adversarial Training for Adversarial Robustness. https://doi.org/10.48550/arXiv.2102.01356

Bharati, P., Pramanik, A., 2020. Deep Learning Techniques—R-CNN to Mask R-CNN: A Survey, in: Das, A.K., Nayak, J., Naik, B., Pati, S.K., Pelusi, D. (Eds.), Computational Intelligence in Pattern Recognition, Advances in Intelligent

Systems and Computing. Springer, Singapore, pp. 657–668. https://doi.org/10.1007/978-981-13-9042-5_56

Buscemi, A., Turcanu, I., Castignani, G., Panchenko, A., Engel, T., Shin, K.G., 2023. A Survey on Controller Area Network Reverse Engineering. IEEE Communications Surveys & Tutorials 1–1. https://doi.org/10.1109/COMST.2023.3264928

Caivano, D., De Vincentiis, M., Nitti, F., Pal, A., 2022. Quantum optimization for fast CAN bus intrusion detection, in: Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering, QP4SE 2022. Association for Computing Machinery, New York, NY, USA, pp. 15–18. https://doi.org/10.1145/3549036.3562058

Carlini, N., Wagner, D., 2017. Towards Evaluating the Robustness of Neural Networks, in: 2017 IEEE Symposium on Security and Privacy (SP). Presented at the 2017 IEEE Symposium on Security and Privacy (SP), pp. 39–57. https://doi.org/10.1109/SP.2017.49

Caro, M.C., Huang, H.-Y., Cerezo, M., Sharma, K., Sornborger, A., Cincio, L., Coles, P.J., 2022. Generalization in quantum machine learning from few training data. Nat Commun 13, 4919. https://doi.org/10.1038/s41467-022-32550-3

Chen, J., Qi, X., Chen, L., Chen, F., Cheng, G., 2020. Quantum-inspired ant lion optimized hybrid k-means for cluster analysis and intrusion detection. Knowledge-Based Systems 203, 106167. https://doi.org/10.1016/j.knosys.2020.106167

Connected/Automated Vehicles [WWW Document], n.d. . Institute of Transportation Engineers. URL https://www.ite.org/technical-resources/topics/connected-automated-vehicles/ (accessed 7.10.23).

Creusen, I.M., Wijnhoven, R.G.J., Herbschleb, E., de With, P.H.N., 2010. Color exploitation in hog-based traffic sign detection, in: 2010 IEEE International Conference on Image Processing. Presented at the 2010 IEEE International Conference on Image Processing, pp. 2669–2672. https://doi.org/10.1109/ICIP.2010.5651637

Das, N., Shanbhogue, M., Chen, S.-T., Hohman, F., Chen, L., Kounavis, M.E., Chau, D.H., 2017. Keeping the Bad Guys Out: Protecting and Vaccinating Deep Learning with JPEG Compression. https://doi.org/10.48550/arXiv.1705.02900

Dilek, E., Dener, M., 2023. Computer Vision Applications in Intelligent Transportation Systems: A Survey. Sensors 23, 2938. https://doi.org/10.3390/s23062938

Dixit, V., Selvarajan, R., Alam, M.A., Humble, T.S., Kais, S., 2021. Training restricted boltzmann machines with a d-wave quantum annealer. Front. Phys. 9: 589626. doi: 10.3389/fphy.

Dong, Y., Hu, W., Zhang, J., Chen, M., Liao, W., Chen, Z., 2022. Quantum beetle swarm algorithm optimized extreme learning machine for intrusion detection. Quantum Inf Process 21, 9. https://doi.org/10.1007/s11128-021-03311-w

D-Wave Systems Inc., n.d. Solving Problems with Quantum Samplers — D-Wave System Documentation [WWW Document]. URL https://docs.dwavesys.com/docs/latest/c_gs_3.html#qubo (accessed 12.28.22).

Dziugaite, G.K., Ghahramani, Z., Roy, D.M., 2016. A study of the effect of JPG compression on adversarial images. https://doi.org/10.48550/arXiv.1608.00853

Gao, H., Oates, T., 2019. Universal Adversarial Perturbation for Text Classification. https://doi.org/10.48550/arXiv.1910.04618

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y., 2014. Generative adversarial nets. Advances in neural information processing systems 27.

Goodfellow, I.J., Shlens, J., Szegedy, C., 2015. Explaining and Harnessing Adversarial Examples. https://doi.org/10.48550/arXiv.1412.6572

Grandvalet, Y., Canu, S., 1997. Noise injection for inputs relevance determination, in: Advances in Intelligent Systems. IOS Press, NLD, pp. 378–382.

Gudigar, A., Chokkadi, S., U, R., 2016. A review on automatic detection and recognition of traffic sign. Multimed Tools Appl 75, 333–364. https://doi.org/10.1007/s11042-014-2293-7

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.C., 2017. Improved Training of Wasserstein GANs, in: Advances in Neural Information Processing Systems. Curran Associates, Inc.

Han, M.L., Kwak, B.I., Kim, H.K., 2018. Anomaly intrusion detection method for vehicular networks based on survival analysis. Vehicular communications 14, 52–63.

Han, S., Shen, H., Philipose, M., Agarwal, S., Wolman, A., Krishnamurthy, A., 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream

Processing Under Resource Constraints, in: Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16. Association for Computing Machinery, New York, NY, USA, pp. 123–136. https://doi.org/10.1145/2906388.2906396

Hashemi, A.S., Mozaffari, S., Alirezaee, S., 2022. Improving adversarial robustness of traffic sign image recognition networks. Displays 74, 102277. https://doi.org/10.1016/j.displa.2022.102277

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep Residual Learning for Image Recognition. Presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778.

Hinton, G.E., 2012. A practical guide to training restricted Boltzmann machines, in: Neural Networks: Tricks of the Trade. Springer, pp. 599–619.

Hossain, M.D., Inoue, H., Ochiai, H., Fall, D., Kadobayashi, Y., 2020. Long Short-Term Memory-Based Intrusion Detection System for In-Vehicle Controller Area Network Bus, in: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). Presented at the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 10–17. https://doi.org/10.1109/COMPSAC48688.2020.00011

Houben, S., Stallkamp, J., Salmen, J., Schlipsing, M., Igel, C., 2013. Detection of traffic signs in real-world images: The German traffic sign detection benchmark, in: The 2013 International Joint Conference on Neural Networks (IJCNN). Presented at the

The 2013 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. https://doi.org/10.1109/IJCNN.2013.6706807

Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H., 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. https://doi.org/10.48550/arXiv.1704.04861

ICT for Transport [WWW Document], n.d. URL https://www.e-elgar.com/shop/gbp/ict-for-transport-9781783471287.html (accessed 7.10.23).

Islam, M., Chowdhury, M., Khan, Z., Khan, S.M., 2022. Hybrid Quantum-Classical Neural Network for Cloud-Supported In-Vehicle Cyberattack Detection. IEEE Sensors Letters 6, 1–4. https://doi.org/10.1109/LSENS.2022.3153931

Jin, G., Shen, S., Zhang, D., Dai, F., Zhang, Y., 2019. APE-GAN: Adversarial Perturbation Elimination with GAN, in: ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). Presented at the ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 3842–3846. https://doi.org/10.1109/ICASSP.2019.8683044

Jin, S., Chung, J.-G., Xu, Y., 2021. Signature-Based Intrusion Detection System (IDS) for In-Vehicle CAN Bus Network, in: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). Presented at the 2021 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401087

Jo, H.J., Choi, W., 2022. A Survey of Attacks on Controller Area Networks and Corresponding Countermeasures. IEEE Transactions on Intelligent Transportation Systems 23, 6123–6141. https://doi.org/10.1109/TITS.2021.3078740

Jose, A., Thodupunoori, H., Nair, B.B., 2019. A Novel Traffic Sign Recognition System Combining Viola–Jones Framework and Deep Learning, in: Wang, J., Reddy, G.R.M., Prasad, V.K., Reddy, V.S. (Eds.), Soft Computing and Signal Processing, Advances in Intelligent Systems and Computing. Springer, Singapore, pp. 507–517. https://doi.org/10.1007/978-981-13-3600-3_48

Kadowaki, T., Nishimori, H., 1998. Quantum annealing in the transverse Ising model. Phys. Rev. E 58, 5355–5363. https://doi.org/10.1103/PhysRevE.58.5355

Kerim, A., Efe, M.Ö., 2021. Recognition of Traffic Signs with Artificial Neural Networks: A Novel Dataset and Algorithm, in: 2021 International Conference on Artificial Intelligence in Information and Communication (ICAIIC). Presented at the 2021 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), pp. 171–176. https://doi.org/10.1109/ICAIIC51459.2021.9415238

Khamaiseh, S.Y., Bagagem, D., Al-Alaj, A., Mancino, M., Alomari, H.W., 2022. Adversarial Deep Learning: A Survey on Adversarial Attacks and Defense Mechanisms on Image Classification. IEEE Access 10, 102266–102291. https://doi.org/10.1109/ACCESS.2022.3208131

Khan, Z., Chowdhury, M., Khan, S.M., 2022. A hybrid defense method against adversarial attacks on traffic sign classifiers in autonomous vehicles. arXiv preprint arXiv:2205.01225.

Kheder, M.Q., Mohammed, A.A., 2023. Improved traffic sign recognition system (itsrs) for autonomous vehicle based on deep convolutional neural network. Multimed Tools Appl. https://doi.org/10.1007/s11042-023-15898-6

Kim, T., Cha, M., Kim, H., Lee, J.K., Kim, J., 2017. Learning to Discover Cross-Domain Relations with Generative Adversarial Networks, in: Proceedings of the 34th International Conference on Machine Learning. Presented at the International Conference on Machine Learning, PMLR, pp. 1857–1865.

Korenkevych, D., Xue, Y., Bian, Z., Chudak, F., Macready, W.G., Rolfe, J., Andriyash, E., 2016. Benchmarking quantum hardware for training of fully visible Boltzmann machines. arXiv preprint arXiv:1611.04528.

Kurowski, K., Slysz, M., Subocz, M., Różycki, R., 2021. Applying a Quantum Annealing Based Restricted Boltzmann Machine for MNIST Handwritten Digit Classification. Computational Methods in Science and Technology 27. https://doi.org/10.12921/cmst.2021.0000011

Lampe, B., Meng, W., 2023. A survey of deep learning-based intrusion detection in automotive applications. Expert Systems with Applications 221, 119771. https://doi.org/10.1016/j.eswa.2023.119771

Laykaviriyakul, P., Phaisangittisagul, E., 2023. Collaborative Defense-GAN for protecting adversarial attacks on classification system. Expert Systems with Applications 214, 118957. https://doi.org/10.1016/j.eswa.2022.118957

Lecun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 2278–2324. https://doi.org/10.1109/5.726791

Li, H., Zhang, B., Zhang, Y., Dang, X., Han, Y., Wei, L., Mao, Y., Weng, J., 2021. A defense method based on attention mechanism against traffic sign adversarial samples. Information Fusion 76, 55–65. https://doi.org/10.1016/j.inffus.2021.05.005

Li, J., Wang, Z., 2019. Real-Time Traffic Sign Recognition Based on Efficient CNNs in the Wild. IEEE Transactions on Intelligent Transportation Systems 20, 975–984. https://doi.org/10.1109/TITS.2018.2843815

Lim, X.R., Lee, C.P., Lim, K.M., Ong, T.S., Alqahtani, A., Ali, M., 2023. Recent Advances in Traffic Sign Recognition: Approaches and Datasets. Sensors 23, 4674. https://doi.org/10.3390/s23104674

Liu, K., Deng, H., 2021. The Analysis of Driver's Recognition Time of Different Traffic Sign Combinations on Urban Roads via Driving Simulation. Journal of Advanced Transportation 2021, e8157293. https://doi.org/10.1155/2021/8157293

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., Berg, A.C., 2016. SSD: Single Shot MultiBox Detector, in: Leibe, B., Matas, J., Sebe, N., Welling, M. (Eds.), Computer Vision – ECCV 2016, Lecture Notes in Computer Science.

Springer International Publishing, Cham, pp. 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

Liu, Z., Liu, Q., Liu, T., Xu, N., Lin, X., Wang, Y., Wen, W., 2019. Feature Distillation: DNN-Oriented JPEG Compression Against Adversarial Examples, in: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). Presented at the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 860–868. https://doi.org/10.1109/CVPR.2019.00095

Lo, W., Alqahtani, H., Thakur, K., Almadhor, A., Chander, S., Kumar, G., 2022. A hybrid deep learning based intrusion detection system using spatial-temporal representation of in-vehicle network traffic. Vehicular Communications 35, 100471. https://doi.org/10.1016/j.vehcom.2022.100471

Lokman, S.-F., Othman, A.T., Abu-Bakar, M.-H., 2019. Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review. J Wireless Com Network 2019, 184. https://doi.org/10.1186/s13638-019-1484-3

Lucas, A., 2014. Ising formulations of many NP problems. Frontiers in Physics 2.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A., 2017. Towards Deep Learning Models Resistant to Adversarial Attacks [WWW Document]. arXiv.org. URL https://arxiv.org/abs/1706.06083v4 (accessed 7.6.23).

Majumder, R., Khan, S.M., Ahmed, F., Khan, Z., Ngeni, F., Comert, G., Mwakalonge, J., Michalaka, D., Chowdhury, M., 2021. Hybrid Classical-Quantum Deep Learning Models for Autonomous Vehicle Traffic Image Classification Under Adversarial Attack. https://doi.org/10.48550/arXiv.2108.01125

Marti, E., de Miguel, M.A., Garcia, F., Perez, J., 2019. A Review of Sensor Technologies for Perception in Automated Driving. IEEE Intelligent Transportation Systems Magazine 11, 94–108. https://doi.org/10.1109/MITS.2019.2907630

McGregor, J.D., Silva, R.S., Almeida, E.S., 2018. 2 - Architectures of Transportation Cyber-Physical Systems, in: Deka, L., Chowdhury, M. (Eds.), Transportation Cyber-Physical Systems. Elsevier, pp. 21–49. https://doi.org/10.1016/B978-0-12-814295-0.00002-2

Minawi, O., Whelan, J., Almehmadi, A., El-Khatib, K., 2020. Machine Learning-Based Intrusion Detection System for Controller Area Networks, in: Proceedings of the 10th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications, DIVANet '20. Association for Computing Machinery, New York, NY, USA, pp. 41–47. https://doi.org/10.1145/3416014.3424581

Møgelmose, A., Liu, D., Trivedi, M.M., 2015. Detection of U.S. Traffic Signs. IEEE Transactions on Intelligent Transportation Systems 16, 3116–3125. https://doi.org/10.1109/TITS.2015.2433019

Mogelmose, A., Trivedi, M.M., Moeslund, T.B., 2012. Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey. IEEE transactions on intelligent transportation systems 13, 1484–1497.

Moore, M.R., Bridges, R.A., Combs, F.L., Starr, M.S., Prowell, S.J., 2017. Modeling inter-signal arrival times for accurate detection of CAN bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection, in: Proceedings of the 12th Annual Conference on Cyber and Information Security Research, CISRC '17.

Association for Computing Machinery, New York, NY, USA, pp. 1–4. https://doi.org/10.1145/3064814.3064816

Moosavi-Dezfooli, S.-M., Fawzi, A., Frossard, P., 2016. Deepfool: a simple and accurate method to fool deep neural networks. Presented at the Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2574–2582.

Moulahi, T., Zidi, S., Alabdulatif, A., Atiquzzaman, M., 2021. Comparative Performance Evaluation of Intrusion Detection Based on Machine Learning in In-Vehicle Controller Area Network Bus. IEEE Access 9, 99595–99605. https://doi.org/10.1109/ACCESS.2021.3095962

Nam, M., Park, S., Kim, D.S., 2021. Intrusion Detection Method Using Bi-Directional GPT for in-Vehicle Controller Area Networks. IEEE Access 9, 124931–124944. https://doi.org/10.1109/ACCESS.2021.3110524

Nonnenmann, P., Bogomolec, X., 2021. Quantum Technologies, in: Liermann, V., Stegmann, C. (Eds.), The Digital Journey of Banking and Insurance, Volume II: Digitalization and Machine Learning. Springer International Publishing, Cham, pp. 201–219. https://doi.org/10.1007/978-3-030-78829-2_12

NVIDIA A100 TENSOR CORE GPU [WWW Document], n.d. . https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf. URL http://nvidianews.nvidia.com/news/nvidia-unveils-drive-thor-centralized-car-computer-unifying-cluster-infotainment-automated-driving-and-parking-in-a-single-cost-saving-system (accessed 7.8.23).

NVIDIA Unveils DRIVE Thor — Centralized Car Computer Unifying Cluster, Infotainment, Automated Driving, and Parking in a Single, Cost-Saving System [WWW Document], n.d. . NVIDIA Newsroom. URL http://nvidianews.nvidia.com/news/nvidia-unveils-drive-thor-centralized-car-computer-unifying-cluster-infotainment-automated-driving-and-parking-in-a-single-cost-saving-system (accessed 7.8.23).

OpenDBC, n.d.

Pan, R., Islam, M.J., Ahmed, S., Rajan, H., 2019. Identifying Classes Susceptible to Adversarial Attacks. https://doi.org/10.48550/arXiv.1905.13284

Pandurangan, R., Jayaseelan, S.M., Rajalingam, S., Angelo, K.M., 2023. A novel hybrid machine learning approach for traffic sign detection using CNN-GRNN. Journal of Intelligent & Fuzzy Systems 44, 1283–1303. https://doi.org/10.3233/JIFS-221720

Papernot, N., McDaniel, P., Wu, X., Jha, S., Swami, A., 2016. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks, in: 2016 IEEE Symposium on Security and Privacy (SP). Presented at the 2016 IEEE Symposium on Security and Privacy (SP), pp. 582–597. https://doi.org/10.1109/SP.2016.41

PyTorch [WWW Document], n.d. URL https://www.pytorch.org (accessed 7.8.23).

Rachev, S.T., 1990. Duality theorems for Kantorovich-Rubinstein and Wasserstein functionals.

Radford, A., Metz, L., Chintala, S., 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. https://doi.org/10.48550/arXiv.1511.06434

RADU, M.D., COSTEA, I.M., STAN, V.A., 2020. Automatic Traffic Sign Recognition Artificial Inteligence - Deep Learning Algorithm, in: 2020 12th International Conference on Electronics, Computers and Artificial Intelligence (ECAI). Presented at the 2020 12th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), pp. 1–4. https://doi.org/10.1109/ECAI50035.2020.9223186

Rajapaksha, S., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Madzudzo, G., Cheah, M., 2023. AI-Based Intrusion Detection Systems for In-Vehicle Networks: A Survey. ACM Comput. Surv. 55, 237:1-237:40. https://doi.org/10.1145/3570954

Ross, A., Doshi-Velez, F., 2018. Improving the Adversarial Robustness and Interpretability of Deep Neural Networks by Regularizing Their Input Gradients. Proceedings of the AAAI Conference on Artificial Intelligence 32. https://doi.org/10.1609/aaai.v32i1.11504

Saadna, Y., Behloul, A., 2017. An overview of traffic sign detection and classification methods. Int J Multimed Info Retr 6, 193–210. https://doi.org/10.1007/s13735-017-0129-8

Salek, M.S., 2023. msabbirsalek/AR-GAN [WWW Document]. URL https://github.com/msabbirsalek/AR-GAN (accessed 7.12.23).

Salek, M.S., 2022. msabbirsalek/Restricted-Boltzmann-Machine-for-CAN-IDS.

Samangouei, P., Kabkab, M., Chellappa, R., 2018. Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models. https://doi.org/10.48550/arXiv.1805.06605

Sarker, I.H., 2021. Deep Cybersecurity: A Comprehensive Overview from Neural Network and Deep Learning Perspective. SN COMPUT. SCI. 2, 154. https://doi.org/10.1007/s42979-021-00535-6

Seo, E., Song, H.M., Kim, H.K., 2018. GIDS: GAN based Intrusion Detection System for In-Vehicle Network, in: 2018 16th Annual Conference on Privacy, Security and Trust (PST). Presented at the 2018 16th Annual Conference on Privacy, Security and Trust (PST), pp. 1–6. https://doi.org/10.1109/PST.2018.8514157

Shanmugavel, A.B., Ellappan, V., Mahendran, A., Subramanian, M., Lakshmanan, R., Mazzara, M., 2023. A Novel Ensemble Based Reduced Overfitting Model with Convolutional Neural Network for Traffic Sign Recognition System. Electronics 12, 926. https://doi.org/10.3390/electronics12040926

Sharma, P., Gillanders, J., 2022. Cybersecurity and Forensics in Connected Autonomous Vehicles: A Review of the State-of-the-Art. IEEE Access 10, 108979–108996. https://doi.org/10.1109/ACCESS.2022.3213843

Simonyan, K., Zisserman, A., 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://doi.org/10.48550/arXiv.1409.1556

Song, H.M., Woo, J., Kim, H.K., 2020. In-vehicle network intrusion detection using deep convolutional neural network. Vehicular Communications 21, 100198. https://doi.org/10.1016/j.vehcom.2019.100198

Stokes, J., Izaac, J., Killoran, N., Carleo, G., 2020. Quantum Natural Gradient. Quantum 4, 269. https://doi.org/10.22331/q-2020-05-25-269

Sun, X., Yu, F.R., Zhang, P., 2022. A Survey on Cyber-Security of Connected and Autonomous Vehicles (CAVs). IEEE Transactions on Intelligent Transportation Systems 23, 6240–6259. https://doi.org/10.1109/TITS.2021.3085297

Volkovs, M., Yu, G., Poutanen, T., 2017. DropoutNet: Addressing Cold Start in Recommender Systems, in: Advances in Neural Information Processing Systems. Curran Associates, Inc.

Wali, S.B., Abdullah, M.A., Hannan, M.A., Hussain, A., Samad, S.A., Ker, P.J., Mansor, M.B., 2019. Vision-Based Traffic Sign Detection and Recognition Systems: Current Trends and Challenges. Sensors 19, 2093. https://doi.org/10.3390/s19092093

What is Quantum Annealing? — D-Wave System Documentation documentation [WWW Document], n.d. URL https://docs.dwavesys.com/docs/latest/c_gs_2.html (accessed 6.29.22).

Wu, W., Li, R., Xie, G., An, J., Bai, Y., Zhou, J., Li, K., 2020. A Survey of Intrusion Detection for In-Vehicle Networks. IEEE Transactions on Intelligent Transportation Systems 21, 919–933. https://doi.org/10.1109/TITS.2019.2908074

Xie, G., Yang, L.T., Yang, Y., Luo, H., Li, R., Alazab, M., 2021. Threat Analysis for Automotive CAN Networks: A GAN Model-Based Intrusion Detection Technique. IEEE Transactions on Intelligent Transportation Systems 22, 4467–4477. https://doi.org/10.1109/TITS.2021.3055351

Xu, W., Evans, D., Qi, Y., 2018. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks, in: Proceedings 2018 Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2018.23198

Yang, Y., Luo, H., Xu, H., Wu, F., 2016. Towards Real-Time Traffic Sign Detection and Classification. IEEE Transactions on Intelligent Transportation Systems 17, 2022–2031. https://doi.org/10.1109/TITS.2015.2482461

Ye, N., Zhu, Z., 2018. Bayesian Adversarial Learning, in: Advances in Neural Information Processing Systems. Curran Associates, Inc.

Young, C., Zambreno, J., Olufowobi, H., Bloom, G., 2019a. Survey of Automotive Controller Area Network Intrusion Detection Systems. IEEE Design & Test 36, 48–55. https://doi.org/10.1109/MDAT.2019.2899062

Young, C., Zambreno, J., Olufowobi, H., Bloom, G., 2019b. Survey of Automotive Controller Area Network Intrusion Detection Systems. IEEE Design Test 36, 48–55. https://doi.org/10.1109/MDAT.2019.2899062

Zaibi, A., Ladgham, A., Sakly, A., 2021. A Lightweight Model for Traffic Sign Classification Based on Enhanced LeNet-5 Network. Journal of Sensors 2021, e8870529. https://doi.org/10.1155/2021/8870529

Zhang, H., Huang, K., Wang, J., Liu, Z., 2021. CAN-FT: A Fuzz Testing Method for Automotive Controller Area Network Bus, in: 2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI). Presented at the 2021 International Conference on Computer Information Science and Artificial

Intelligence (CISAI), pp. 225–231. https://doi.org/10.1109/CISAI54367.2021.00050

Zhang, J., Huang, Q., Wu, H., Liu, Y., 2017. A Shallow Network with Combined Pooling for Fast Traffic Sign Recognition. Information 8, 45. https://doi.org/10.3390/info8020045

Zhao, Q., Chen, M., Gu, Z., Luan, S., Zeng, H., Chakrabory, S., 2022. CAN Bus Intrusion Detection Based on Auxiliary Classifier GAN and Out-of-distribution Detection. ACM Trans. Embed. Comput. Syst. 21, 45:1-45:30. https://doi.org/10.1145/3540198

Zhao, Y., Xun, Y., Liu, J., Ma, S., 2022. GVIDS: A Reliable Vehicle Intrusion Detection System Based on Generative Adversarial Network, in: GLOBECOM 2022 - 2022 IEEE Global Communications Conference. Presented at the GLOBECOM 2022 - 2022 IEEE Global Communications Conference, pp. 4310–4315. https://doi.org/10.1109/GLOBECOM48099.2022.10001410