TÉCNICO LISBOA

Lightweight Machine Learning Models for Intrusion Detection

Mafalda Barbosa de Brito

Final Report of

2nd Cycle Integrated Project in Eletrical and Computer Engineering

Supervisor(s):	Prof. Paulo Rogerio Barreiros d'Almeida Pereira
	Prof. Naercio David Pedro Magaia

Month 2026

Contents

List of Tables

List of Figures

Glossary

AD	Anomaly-based Detection
AI	Artificial Intelligence
AIDS	Anomaly-based intrusion detection system
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
DoS	Denial-of-Service
DQN	Deep Q-Network
DS	Database Server
DSRC	Dedicated Short-Range Communications
FN	False Negative
FP	False Positive
HIDS	Host-based IDS
IDS	Intrusion Detection System
IoT	Internet of Things
IoV	Internet of Vehicles
kNN	K-Nearest Neighbors
LiteRT	Lite Runtime
LSTM	Long Short-Term Memory
MIDS	Mixed IDS
ML	Machine Learning
MLPs	Multi-Layer Perceptrons
MN	Managed Network
MS	Management Server
NAS	Neural Architecture Search
NBA	Network Behavior Analysis
NIDES	Next-generation Intrusion Detection Expert System
NIDS	Network-based IDS
OBU	On-Board Unit

OCSVM	One-Class SVMs
RL	Reinforcement Learning
RNN	Recurrent Neural Networks
SD	Signature-based Detection
SIDS	Signature-based intrusion detection system
SIEM	Security information and event management
SN	Standard Network
SPA	Stateful Protocol Analysis
SVM	Support Vector Machine
TinyML	Tiny Machine Learning
V2V	Vehicle-to-Vehicle
WIDS	Wireless-based IDS

Chapter 1

Introduction

The Internet of Things (IoT) represents a vast network of interconnected devices that exchange data to enable automation, efficiency, and intelligence across multiple domains, including smart homes, industrial systems, and autonomous vehicles. While IoT has brought remarkable **technological advancements**, its **growing interconnectivity** has also **expanded the attack surface**, exposing systems to numerous **cybersecurity vulnerabilities**.

To address these challenges, Intrusion Detection System (IDS) has emerged as a fundamental element of IoT security frameworks. These systems continuously **monitor network traffic**, **detect anomalies**, and **identify potential intrusions in real time**.

Within the broader IoT landscape, **Vehicle-to-Vehicle (V2V)** [1] communication has emerged as a vital research domain for **enhancing traffic efficiency**, **road safety**, and the **advancement of autonomous driving technologies**. V2V systems form a core component of the evolving Internet of Vehicles (IoV) ecosystem, enabling vehicles to exchange real-time information about their position, speed, and intent. However, as vehicular applications become increasingly complex, they also require more sophisticated network and security protocols to maintain reliability and trustworthiness.

Traditional IDS and Deep Learning architectures, while powerful, are **computationally expensive** and **unsuitable** for deployment in **resource-constrained** vehicular devices. These systems demand solutions that balance accuracy, speed, and efficiency. The **Lightweight Deep Learning (DL)** [2] paradigm addresses this challenge by enabling the design of **compact**, **optimized neural networks** capable of operating under tight hardware constraints.

Tiny Machine Learning (TinyML) [3] has emerged as a promising approach to achieve this balance. TinyML enables **efficient on-device inference** while **maintaining high detection performance**. In the vehicular context, deploying lightweight IDS models directly on On-Board Units (OBUs) **reduces latency**, **enhances privacy**, and **minimizes dependency on cloud resources** - crucial for real-time detection of fast-evolving cyberattacks.

1.1 Problem Context and Motivation

Ensuring information availability is essential for the reliable operation of any network [1]. In the context of vehicular networks, disruptions in availability can severely impact emerging vehicular services and applications, even posing risks to human safety when malicious actors inject large volumes of packets to overwhelm the system.

In V2V communication, timely and reliable data exchange enables vehicles to make critical rapid safety decisions, such as avoidance of collisions or route optimization. **Flooding attacks**, however, overwhelm the network with a massive number of fake or redundant packets, leading to **congestion**, **communication delays**, or **complete service disruption**. Such failures can **degrade the performance** of essential vehicular applications and, in extreme cases, **endanger human lives**.

For this reason, this thesis focuses on Denial-of-Service (DoS) attacks, specifically **SYN flood** and **UDP flood**, where malicious users send massive numbers of spurious or junk packets to consume resources (bandwidth, processing, energy) and disrupt the network.

To train the ML model, this work will employ datasets from the realistic simulations of UDP and SYN flooding attacks developed by Sousa *et al.* [1]. Unlike typical simulated datasets, these were obtained using three **MK5 On-Board Units (OBUs)** operating with the **IEEE 802.11p** standard, the core communication protocol of Dedicated Short-Range Communications (DSRC) systems. Consequently, the data more accurately reflects **realistic vehicular communication conditions**.

The datasets encompass **multiple experimental scenarios** and **diverse traffic patterns**, including both normal and malicious packets, to enhance heterogeneity and realism. Feature extraction using Tranalyzer [4] and Wireshark [5] produced 89 detailed features, including time-based metrics essential for effective attack detection. Baseline machine learning results indicate a realistic level of task difficulty (F1-scores between 0.771 and 0.829). Since the F1-score reflects model performance especially on imbalanced data where one class is more frequent than others - these results confirm the dataset's complexity, and its suitability for evaluating robust IDS models.

These devices, like other IoT elements, operate in **resource-constrained environments**, possessing **limited processing power**, **memory capacity**, and **energy availability**. Deploying traditional, complex IDS models in these settings is **often inadequate**. The existing baseline results used a Decision Tree ML algorithm. The future work explicitly calls for building novel ML models for intrusion detection using the collected real V2V data and comparing their performance with state-of-the-art solutions. Since the environment is resource-limited, these new models must adhere to the principles of Lightweight Deep Learning.

1.2 Objectives

The main objective of the thesis is to **design**, **implement**, and **evaluate lightweight Deep Learning (DL) models** for **intrusion detection against V2V flooding attacks** (SYN flood and UDP flood), utilizing the realistic datasets generated from MK5 OBU devices, with a focus on **optimizing models** for deployment on **resource-constrained hardware**.

1.3 Report Outline

Chapter 2

Background

2.1 Intrusion Detection Systems

Intrusion can be understood as any form of unauthorized activity that disrupts, damages, or exploits an information system. Such activities are considered intrusions because they endanger one or more of the system's core security principles: confidentiality, integrity, and availability.

Intrusion detection systems (IDSs) are the 'burglar alarms' (or rather 'intrusion alarms') of the computer security field [6]. They can be implemented as **software** or **hardware** solutions that automate the intrusion detection process in a **computer system** or **network**. The goal of an IDS is to identify different kinds of malicious network traffic and computer usage, which cannot be identified by a traditional firewall [7]. Traditional firewalls function as a barrier that blocks unauthorized access, while an IDS **monitors the network to detect and alert** on suspicious or malicious activity. In other words, a firewall acts like a locked door, and an IDS works like a **security camera**.

2.1.1 Detection Systems Methodologies

Intrusion detection methodologies are generally divided into three main categories: Signature-based intrusion detection system (SIDS), Anomaly-based intrusion detection system (AIDS), and Stateful Protocol Analysis (SPA).

Stateful Protocol Analysis is an intrusion detection method that monitors the state of network protocols - meaning it can track how a protocol operates over time (e.g., matching a request with its reply). SPA relies on vendor-developed profiles based on official protocol standards (like those from the IETF). Because of this, SPA is also called Specification-based Detection.

A **SIDS** relies on **pattern-matching** techniques to identify known attacks by comparing activity against a database of intrusion signatures. These systems achieve **high accuracy** for previously identified threats but **struggle with zero-day and polymorphic attacks** since no matching signature exists until added [7]. Traditional SIDS tools (e.g., Snort, NetSTAT) analyze individual network packets, but modern threats often span multiple packets, requiring more advanced methods. Due to the rise of sophisticated and zero-day attacks [8], SIDS are becoming less effective, and alternative approaches, such as anomaly-based IDS, are being explored.

An **AIDS** detects attacks by modeling normal system behavior and **flagging significant deviations as anomalies**. Unlike SIDS, AIDS **can identify zero-day attacks** and internal malicious activities because they do not rely on a signature database. They use **statistical**, **knowledge-based**, or **machine learning methods**, typically involving a training phase to **learn normal behavior** and a testing phase to **evaluate detection of unseen intrusions**. While AIDS provides stronger protection against novel threats, its main drawback is a **high false positive rate**, since unusual but legitimate behaviors may be misclassified as attacks.

Most IDSs use **multiple methodologies** to provide more extensive and accurate detection. For example, SD and AD are complementary methods, because the former concerns certain attacks/threats and the latter focuses on unknown attacks [9].

2.1.2 Detection Systems Approaches

Although detailed analyses of detection approaches are generally scarce, a few articles [9][7] have proposed a classification into five subclasses - Statistics-based, Pattern-based, Rule-based, State-based, and Heuristic-based - offering a comprehensive perspective on their distinctive characteristics.

Statistics-based approaches analyze network traffic using statistical algorithms, thresholds, mean, standard deviation, and probabilities to detect anomalies. They are simple but may require substantial statistical knowledge and can be less accurate in real-time scenarios.

Pattern-based detection identifies known attacks through string matching, character patterns, or forms in the data, often using hash functions for identification. It is easy to implement and focuses on known threats.

Rule-based techniques employ If–Then or If–Then–Else rules or attack signatures to model and detect intrusions. While they offer high detection rates and low false positives, they can be computationally expensive and require extensive rule sets.

State-based methods examine streams of events and exploit finite state machines derived from network behavior to detect attacks. They are probabilistic, self-trained, and achieve low false-positive rates.

Finally, **Heuristic-based** approaches draw on artificial intelligence or biological inspiration to identify abnormal activity. These methods rely on experience, experimental learning, and evolutionary strategies.

2.1.3 Technologies and Classification

There are many types of IDSs technologies that may be systematically classified, according to where they are deployed to inspect suspicious activities, and what event types they can recognize [10–13], as follows: Host-based IDS (HIDS), Network-based IDS (NIDS), Wireless-based IDS (WIDS), Network Behavior Analysis (NBA), and Mixed IDS (MIDS).

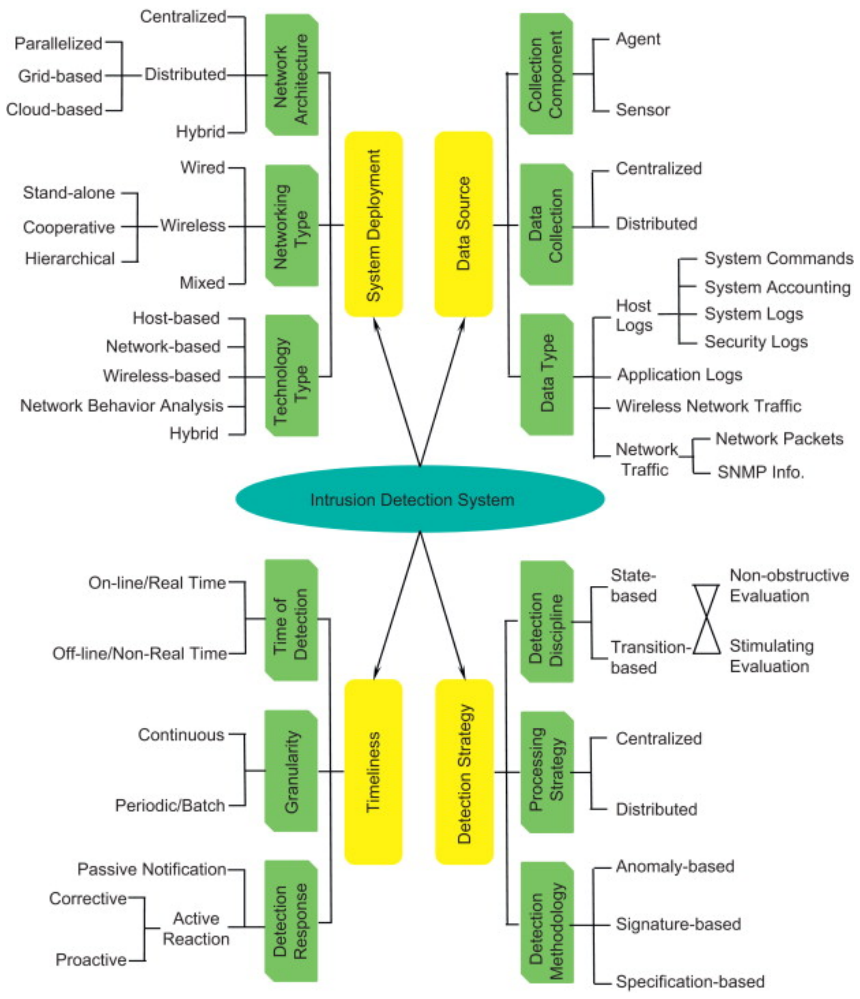

Figure 2.1: An overview of IDS taxonomy [9]

Host-based IDSs monitor and analyze activity on individual hosts, including sensitive systems and servers, by inspecting logs from the operating system, applications, firewalls, and databases. They are particularly effective at detecting insider attacks that do not generate network traffic [14].

Network-based IDSs monitor network traffic using sensors and data sources such as packet captures and NetFlow - a network protocol developed by Cisco that collects and records information about IP traffic flowing through a network - analyzing application and protocol activity to detect suspicious incidents. NIDS can oversee multiple hosts and identify external threats early, but may struggle with high-bandwidth networks [15]. When deployed alongside HIDS and firewalls, NIDS contributes to a multi-layered defense against both external and insider attacks.

Wireless-based IDS functions similarly to NIDS but monitors wireless traffic, including ad hoc, sensor, and mesh networks. **Network Behavior Analysis** detects attacks by analyzing unusual traffic flows. And **Mixed IDS** or **Hybrid IDS** combines multiple technologies for more comprehensive detection.

Components and Architecture

IDS systems use **sensors** (for NIDS, WIDS, NBA) or **agents** (software) (for HIDS) to collect data. Both the sensor and agent can deliver data to a **Management Server (MS)** for analysis and a **Database**

Server (DS) for storage.

Networks may be deployed as a **Managed Network (MN)**, an isolated network deployed for security software management to conceal the IDS information from intruders, or a **Standard Network (SN)**, a public network without protection, secured by virtual isolation [9].

Core Capabilities

Most intrusion detection systems commonly integrate four essential security mechanisms: information gathering, logging, anomaly detection, and intrusion prevention [9]. **Information gathering** collects information on hosts/networks from observed activities. **Logging**, where the related logging data for detected vents, can be used to validate the alerts and investigate incidents. **Detection** methodologies that in most IDSs usually need the sophisticated tuning to receive a higher accuracy. And some intrusion detection systems include **prevention** capabilities and are called Intrusion Prevention Systems (IPS). These systems possess all IDS functionalities while also being able to actively block threats in real-time [11].

Accuracy Challenges

A notable limitation of IDS technologies is that they **cannot achieve perfect detection**. The accuracy of an IDS is typically assessed using false positives (FP) - when benign activity is mistakenly classified as malicious - and false negatives (FN) - when malicious activity goes undetected.

Because the consequences of missing an attack are often more severe than dealing with extra alerts, security administrators usually **prioritize reducing FNs**, even if it results in more FPs. This means the IDS is tuned to be more sensitive, catching more potential threats at the cost of generating additional false alarms. Ho et al. (2012) [16] analyzed FP and FN cases from real-world traffic and reported three key findings: (1) **most false cases are FNs**, as many application behaviors and data formats are custom-defined rather than adhering to RFC standards; (2) the **majority of FP alerts arise from management policies** rather than actual security issues; and (3) **older, well-known types of cyberattacks**, such as buffer overflows, SQL server exploits, and the Worm Slammer attack, **tend to be missed more often by IDSs**, resulting in a high rate of FNs.

2.1.4 IDS Taxonomy

Figure 2.1 presents four aspects for classifying IDSs, which are described sequentially below [9].

System Deployment

The *Network architecture* of an IDS may be **centralized**, collecting and analyzing data from a single monitored system; **distributed**, gathering information from multiple monitored systems to detect coordinated or distributed attacks; or **hybrid**, combining both approaches. Modern distributed configurations often employ parallel, grid-based, or cloud-based architectures to enhance scalability and performance.

The *Networking Type* denotes the manner in which an IDS connects to the monitored system, which can be **wired**, **wireless**, or a **hybrid** of both. Wireless can be deployed in stand-alone, cooperative, or hierarchical environments. The *Technology Type* refers to the underlying IDS technology, as discussed in the subsection 2.1.3.

Data Source

The only subsection of Fig.2.1 not yet addressed is *Data Type* that can include (i) audit trails (e.g., system logs, system commands, etc.) on a host, (ii) network packets or connections, (iii) wireless network traffic, and (iv) application logs.

Timeliness

Timeliness concerns when and how often detection occurs. IDSs may operate in **real-time/on-line** or **offline/non-real-time** modes. Detection can be processed **continuously**, **periodically**, or in **batches**. And when it comes to responses, they can be **passive**, where the IDS only generates alerts, or **active**, where the system takes corrective or preventive measures (IPS).

Detection Strategy

This subsection highlights three related aspects of IDS strategy. The *Detection Discipline* can be **state-based**, evaluating whether a system is currently secure or insecure, or **transition-based**, which monitors changes between secure and insecure states over time. Both approaches can employ either a **stimulating** evaluation, actively testing the system, or a **non-obtrusive** evaluation, which observes passively without interfering with normal operations. In addition, the *Processing Strategy* describes how detection tasks are managed and can be either **centralized**, with analysis occurring at a single point, or **distributed**, with processing shared across multiple nodes. As to *Detection Methodology*, one of anomaly-based, signature-based and specification-based is as illustrated in subsection 2.1.1.

2.2 Traditional Machine Learning Methods

Before discussing deep learning approaches, it is essential to understand the foundational machine learning algorithms commonly employed in intrusion detection systems. Traditional ML methods include both supervised algorithms (Decision Trees, Random Forest, SVM, Naïve Bayes, kNN, Logistic Regression) and unsupervised methods (K-Means Clustering, Semi-Supervised Learning). Additionally, Reinforcement Learning offers a different paradigm where agents learn through interaction with their environment.

2.2.1 Supervised Learning Methods

Decision Trees are popular in IDS due to their **simplicity**, **interpretability**, and **efficiency** in classifying network traffic. They are non-parametric supervised learning algorithms [17] that recursively partition data, with nodes representing features, branches representing decision rules, and leaves representing class labels. They handle both **categorical** (e.g., protocol type, connection state) and **numerical data** (e.g., packet size, duration) to distinguish normal from malicious activity [18]. Decision trees are effective for binary and multi-class classification and provide an **interpretable framework** for IDS, though they may overfit, exhibit high variance, and be computationally expensive.

Random Forests improve IDS performance by combining **multiple decision trees** to produce a **single result**, enhancing accuracy and **controlling overfitting** [18, 19]. They handle **large, high-dimensional datasets** and class imbalance effectively, reducing false positives [18, 20]. Feature selection techniques further improve efficiency, though Random Forests can be time-consuming and resource-intensive compared to a single tree.

Support Vector Machines (SVM) are supervised machine learning algorithms that classify data by finding an optimal hyperplane that maximizes the distance between classes in an N-dimensional space [21]. They excel in **binary classification** and **high-dimensional datasets**. One-Class SVMs (OCSVM) are useful when only normal data is available, making them suitable for scenarios where labeled attack data is scarce [22].

Naïve Bayes is a probabilistic supervised classifier based on Bayes' theorem. It **efficiently** classifies network traffic using **labeled** datasets and can be combined with **feature selection** to improve detection performance [23, 24]. Its **simplicity** and **effectiveness** in processing labeled data make it suitable for real-time network security applications [25].

K-Nearest Neighbors (kNN) is a non-parametric, supervised learning classifier that uses proximity to make classifications. It works by finding the k nearest labeled training examples to a new data point and assigning the class label based on the majority vote among neighbors [26]. kNN adapts well to **dynamic IDS environments** and **different types of network attacks**.

Logistic Regression is a classification algorithm that predicts the probability that a given input belongs to a particular class [27]. It models this probability by applying a logistic function to a linear combination of input features. Logistic regression is employed to analyze network traffic features and **predict** the **likelihood** of an intrusion [18], with the ability to identify **various types of network attacks** effectively [28] and, combined with a multinominal regression model, can **enhance detection** performance and **reduce misclassification** [29].

2.2.2 Unsupervised and Semi-Supervised Learning Methods

K-Means Clustering is an unsupervised learning algorithm that groups unlabeled data points into clusters [30]. It is applied in IDS to identify patterns in network traffic, distinguishing between normal and potentially malicious activities based on feature similarities. Data points that fall into small or distant clusters may indicate **anomalies** or **potential intrusions**. K-Means is more **effective** for **initial pattern**

recognition in the **absence of labels** [31].

Semi-Supervised Learning combines supervised and unsupervised learning by leveraging both labeled and unlabeled data [32]. Self-training and co-training are common semi-supervised techniques. **Self-training** iteratively adds high-confidence pseudo-labeled data, while **co-training** uses multiple classifiers on different feature subsets. These techniques are beneficial in IDS applications as they allow utilization of **large amounts of unlabeled data**, which is easier and cheaper to obtain than labeled data [18].

2.2.3 Reinforcement Learning

Reinforcement Learning (RL) enables IDS to learn optimal actions through interaction with the environment, aiming to maximize cumulative rewards over time. Two main RL methods used in IDS are:

Q-Learning is a model-free RL algorithm that learns the best action for each state using a Q-table, updated by the Bellman equation. The Q-value represents the expected cumulative reward for taking a specific action in a given state. Q-learning is simpler and **suitable for smaller environments**. It helps IDS adapt to **new attacks** by updating detection policies based on feedback.

Deep Q-Networks (DQN) extend Q-learning by using a deep neural network to approximate Q-values, allowing the algorithm to handle high-dimensional state spaces. Through experience replay, DQNs learn complex patterns from raw data. DQN is more **powerful** and **scalable** but requires **greater computational resources** and **training time**.

Multi-Layer Perceptrons (MLPs), a specific type of artificial neural network with multiple interconnected layers, have been extensively applied in IDS using labeled datasets. MLPs consist of interconnected layers of neurons that learn to classify network traffic as normal or malicious through forward propagation and backpropagation [18].

2.3 Lightweight Deep Learning

Deep Learning (DL) is a subset of Machine Learning (ML), which itself is a branch of Artificial Intelligence (AI). While AI is about making machines do tasks that would require 'intelligence' if a human did them, such as decision-making, problem solving, learning, etc. - Machine Learning is about making AI systems **automatically learn from data** rather than relying on explicit programming [33][34].

2.3.1 Deep Learning

The primary goal of Deep Learning is to automatically **learn useful representations of data** to solve complex tasks such as classification, prediction, or generation. Unlike traditional machine learning methods, which rely heavily on handcrafted **features** - individual measurable properties or characteristics of the data used by the model to make predictions - deep learning architectures are capable of **extracting and refining hierarchical features directly from raw data**, such as pixels, text, or audio signals.

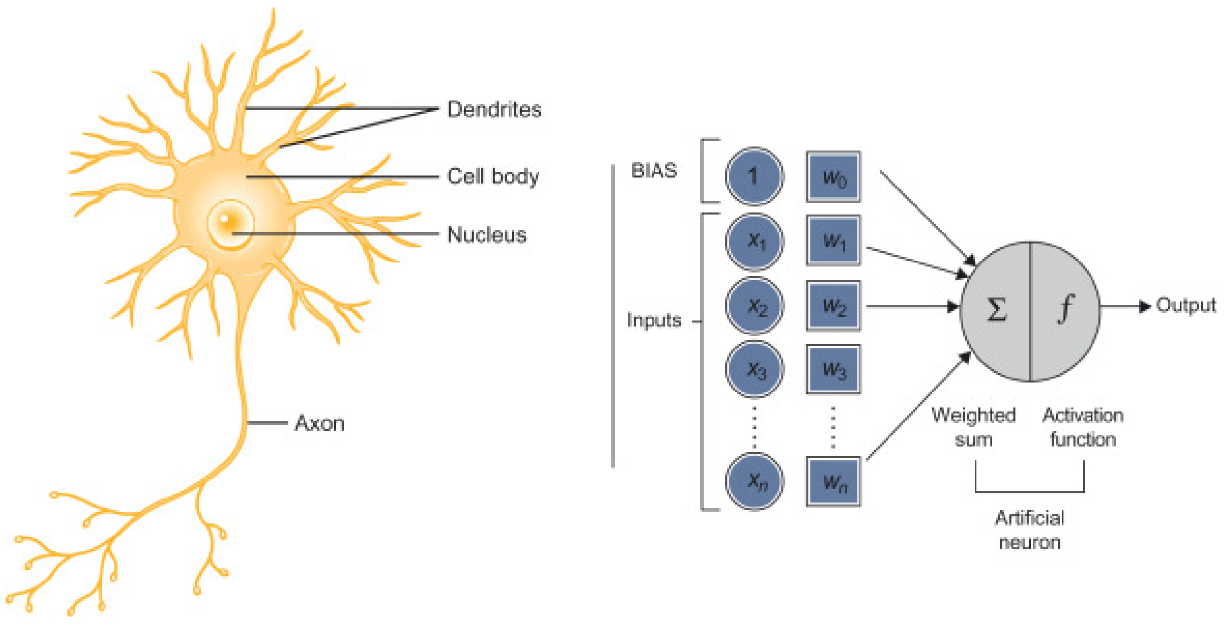

Figure 2.2: Biological neuron and computational perceptron [37].

Formally, Deep Learning can be narrowly defined as the optimization of Artificial Neural Network (ANN) with many layers, enabling the model to capture increasingly abstract representations. In a broader sense, Deep Learning encompasses all methods, architectures, and applications that involve multi-layered ANN representations. [35]

An **Artificial Neural Network** is a structure containing simple elements that are interconnected in many ways with hierarchical organization. It tries to interact with objects in the real world in the same way as the biological nervous system [36]. It is composed of layers of interconnected units called **perceptrons** (also called artificial neurons or nodes), which transform input data into output predictions through a series of weighted connections and nonlinear activation functions.

Just as a biological neuron receives input signals from other neurons through its dendrites, a perceptron receives data from preceding units through its **input nodes**. Each connection between an input node and the perceptron is associated with a **weight**, which represents the relative importance of that input. In biological systems, the dendrites transmit signals to the nucleus, where they are processed to generate an output. Analogously, the perceptron's processing unit performs a **weighted computation on its inputs and produces an output value**. In the brain, this output is carried away by the axon; in artificial neural networks, the perceptron's output is **propagated forward as input** to subsequent perceptrons (Fig. 2.2).

In ANNs, the connections between neurons are represented mathematically using vectors and matrices. A **vector** can be used to describe the set of inputs received by a perceptron, while a **matrix** organizes the weights that connect one layer of perceptrons to the next. For example, if a layer with m neurons is connected to a layer with n neurons, the weights of these connections can be stored in an $m \times n$ matrix. During the forward pass, the input vector is multiplied by the weight matrix, and a bias vector is added before applying the activation function. This matrix representation is fundamental because it **allows neural networks to efficiently perform large-scale computations** using well-established linear algebra operations.

Artificial Neural Networks operate in two main phases.

Training Phase

During training, the neural network learns from a dataset by adjusting its **parameters** (weights and biases) to minimize a loss function, which measures the difference between the predicted outputs and the true labels. This process involves **forward propagation**, where inputs pass through the network to generate predictions, and **backpropagation**, where gradients of the loss function are computed and used to update the weights via an optimization algorithm. The goal of training is to **capture the underlying patterns in the data** so that the network can generalize to unseen examples.

Training is performed over multiple **epochs**, where each epoch corresponds to one complete pass through the **training dataset**. To monitor the model's ability to generalize to unseen data, a separate **validation dataset** is used. The model's performance on this validation set is typically plotted alongside the training performance in a **learning curve**, which shows **how loss or accuracy evolves over epochs**. Learning curves help detect **underfitting**, where the model is too simple to capture the underlying patterns and performs poorly on both training and validation data, and **overfitting**, where the model fits the training data too closely (including noise), resulting in high training performance but poor validation performance. By observing learning curves on both training and validation sets, one can **adjust parameters**, **network architecture**, or **apply regularization techniques** to improve the model's generalization.

Inference Phase

Once the network is trained, it enters the inference phase, also called **testing** or **prediction**. During inference, the model uses the learned weights to process new input data and generate outputs. Unlike training, inference does not involve weight updates; it **only performs forward propagation to produce predictions**. This phase is typically faster and is what is deployed in real-world applications.

Layers

A unilayer ANN like that in Figure 2.2 has a low processing capacity by itself and its level of applicability is low; its true power lies in the interconnection of many ANNs, as happens in the human brain. This has motivated different researchers to propose various topologies (architectures) to connect neurons to each other in the context of an ANN.

From Figure 2.3 we can see that a multi-layer Artificial Neural Network is a directed graph whose nodes correspond to perceptrons and whose edges correspond to links between them. The model given is organized as several interconnected layers: the **input layer** - the set of neurons that directly receives the information coming from the external sources of the network - **hidden layers** - sets of

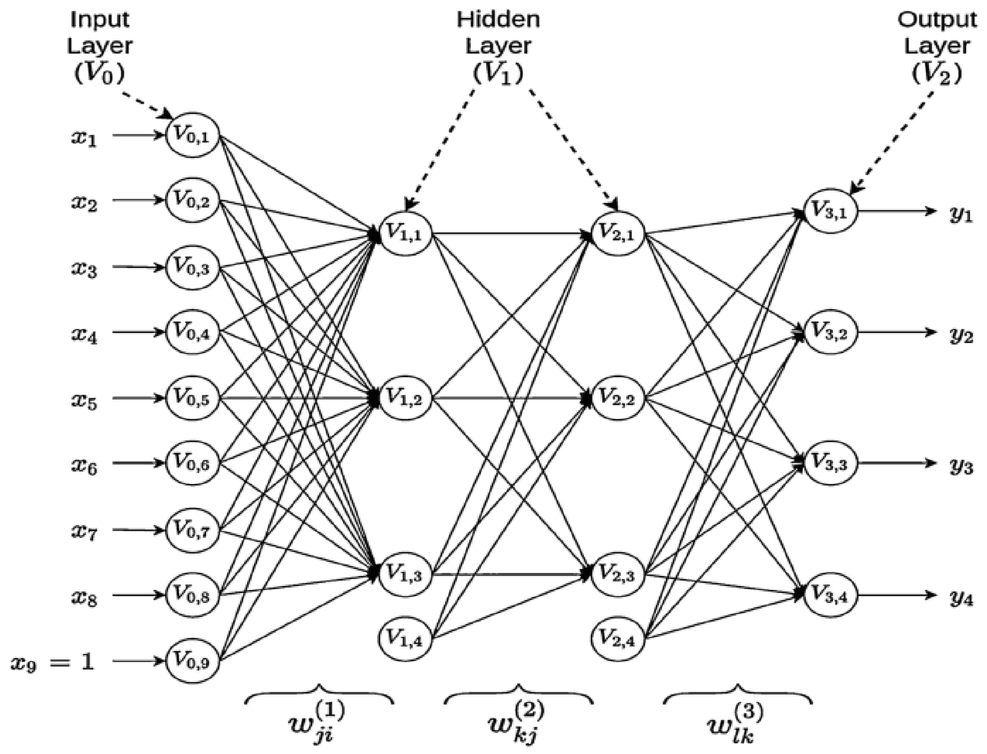

Figure 2.3: Artificial deep neural network with a feedforward neural network with eight input variables (x1, . . . ,x8), four output variables (y1, y2, y3, y4), and two hidden layers with three neurons each [38].

internal neurons of the network that do not have direct contact with the outside - and **output layer** - set of neurons that transfers the information that the network has processed to the outside [38].

An ANN with multiple hidden layers is called a Deep Neural Network (DNN). The adjective “deep” applies not to the acquired knowledge, but to the way in which the knowledge is acquired [39], since it stands for the idea of successive layers of representations. The “deep” of the model refers to the number of layers that contribute to the model. [38]

Convolutional Neural Network (CNN) is a specialized form of DNN for analyzing input data that contains some form of spatial structure [40]. They introduce the concept of **filters** (also called kernels). A filter is a **small matrix of trainable weights** that is applied across the input data to detect local patterns such as edges, textures, or more complex features. Just as the learning algorithm in fully connected networks adjusts synaptic weights, in CNN it adjusts the values of these filters during training. The effect is that the **network automatically learns which features are most relevant for the task**, allowing it to capture spatial hierarchies of patterns with far fewer parameters than a fully connected design.

Recurrent Neural Networks (RNN) is a deep neural network trained on sequential or time series data to make sequential predictions or conclusions based on sequential inputs [41]. They maintain a hidden state that retains information from previous time steps, enabling the network to process sequences effectively [18]. RNNs are trained using **forward propagation** and **backpropagation through time (BPTT)**, which calculates errors across all time steps to adjust model parameters. Unlike traditional backpropagation in feedforward networks, BPTT accounts for shared parameters across the sequence, enabling the model to learn from temporal relationships within the data [41].

2.3.2 Lightweight DL

Lightweight deep learning indicates the procedures of compressing DNN models into more compact ones, which are suitable to be executed on edge devices due to their limited resources and computational capabilities while maintaining comparable performance to the original. Currently, the approaches of model compression include but are not limited to network **pruning**, **quantization**, **knowledge distillation**, **neural architecture search**, **low-rank factorization**, and **huffman coding**.

Pruning

Pruning is a common compression technique that aims to eliminate redundant parameters in DNNs. It can be used for minimizing the cost of computation by pruning parameters or filters from the convolutional layers. When it came to pruning filters, it could decrease the number of operations in the convolutional stage and thus improve the inference time [42]. Meanwhile, since the model parameters or filters could be pruned according to their importance [43, 44], it could thus strike a balance of execution speed and performance. Among the different types of pruning, the two main ones are weight pruning and filter pruning.

Weight pruning consists of setting certain weights to zero. Among all the recent research works, the most popular approach is by means of sparsity - the property of a model where most elements are zero (or near zero). Sparsity could be carried out by L1 regularization, a way of “punishing” the model by setting the unnecessary weights to zero if they do not meet a predefined threshold. While weight pruning is powerful, it faces unstructured or structured problems. Since unstructured sparsity requires additional overhead to record indices, this makes it less efficient on hardware. On the other hand, structured sparsity limits the sparsity patterns in the weights so that they can be described in low-overhead representations, such as strides or blocks. Although this reduces index storage overhead and simplifies processing, structured sparsity might result in a worse model because it limits the freedom of weight updates during the sparsification process [2].

Filter pruning is a technique that attempts to determine the importance of each filter and eliminate unnecessary ones. The most prevalent criterion is adopting the L1-norm or L2-norm based on the influence on the error function to verify unimportant filters. It can be done by ranking feature maps [45], or by search algorithms, such as PruningNet [46], which uses meta-learning to automatically determine which filters to prune.

Quantization

Quantization consists of representing weights, gradients, and activations - outputs from each layer - using fewer bits. This method is used to speed up both training and inference time for the model. However, it is challenging to maintain high accuracy while reducing bit precision. Recent research aims to solve this problem by **reducing quantization error** or **mimicking the results of networks with full precision** [2].

Knowledge Distillation

Instead of reducing the number of parameters from the DNNs, the method of knowledge distillation is to train a lightweight (student) model from the original (teacher) model. And because of that, knowledge distillation is not a compression technique but rather a methodology used to train a lightweight model.

One of the techniques used for knowledge distillation is the transfer of **"soft logits"** - a smoothed probability distribution that carries richer information - generated by the teacher model to **guide the optimization** of the student model. Another technique, aimed at reducing the difference between the student model and the teacher model, uses an **assistant teacher of intermediate size** to divide knowledge distillation into different phases (intermediate layer knowledge). We can also categorize knowledge distillation methods into two schemes: **offline distillation**, in which the teacher is trained until high accuracy is achieved and only then is the knowledge transferred, and **online distillation**, in which the teacher and student are trained simultaneously. Online distillation, although more complex, has a **more efficient training process and lower computational cost**, in addition to obtaining not only a compact student but also a more powerful teacher model. [2]

Neural Architecture Search (NAS)

Neural Architecture Search (NAS) represents a technique for automating the design of artificial neural networks, with its biggest challenge being the **extensive search space**. This approach addresses the traditionally time-consuming and labor-intensive process of manually designing neural network architectures, which typically requires specialized expertise. There are three main approaches. One is based on **Reinforcement Learning** (RL), in which the design process evolves using model accuracy as a reward. Another is based on **Evolutionary Algorithms** to optimize architectures, which requires considerable computational resources. And yet another is based on **Gradients**, which converts discrete choices into continuous variables, allowing for faster search, albeit with high memory consumption. To evaluate the architectures, accuracy is measured. However, training each model from scratch is unfeasible, so techniques such as **performance prediction**, **parameter sharing**, and **learning curve** prediction are used.

Data-free Compression

Data-free compression emerges as an alternative when it is **not possible to access the original training set** for privacy or security reasons. The main contributions are from Yin et al. [47], who introduced DeepInversion, a method that generates synthetic images by reversing a trained neural network (the "teacher") so that it produces inputs that look like the data it was trained on, even if there is no access to the real dataset. From Zhang et al. [48], that developed an approach for image super-resolution, training a generator capable of creating useful information from a pre-trained network. And Cai et al. [49] enabled mixed-precision quantization without access to the original data by optimizing a distilled dataset.

Low-Rank Factorization

Low-rank factorization is a mathematical approach for approximating a high-dimensional matrix with a low-dimensional one while maintaining as much information as possible from the original matrix. The purpose of low-rank factorization in ML is to decompose a dense weight matrix into the product of two lower-dimensional matrices with lower ranks, hence reducing the matrix's dimensionality while keeping its structure and significant properties. As a result, the model is represented more compactly, with fewer parameters and is more computationally efficient [3].

Huffman Coding

Huffman coding is a lossless compression method, which means that compressed data can be precisely rebuilt to its original form with no information loss. Huffman coding works by assigning binary codes to each symbol in the data set, with shorter codes for frequently appearing symbols and longer codes for less frequently occurring symbols. The rationale behind this method is that symbols that appear more frequently in the data will take up less space if they are represented by shorter codes [3].

2.3.3 TinyML

Now that we have talked about what Lightweight machine learning models are, we delve into **Tiny Machine Learning (TinyML)** [3] that refers to the deployment of these models on **ultra-low-power**, **resource-constrained devices**, such as microcontrollers, embedded systems, or IoT sensors. TinyML's key idea is to bring intelligence directly to the device, without relying on cloud processing.

TinyML is successfully applied in areas such as **healthcare**, **agriculture**, **industrial IoT**, and the **environment**, driven by the need to integrate intelligence into applications where it was previously unfeasible due to the high energy and resource consumption of conventional ML models.

Advantages of TinyML

- **Reduced Latency**
  TinyML models run on the device itself, so response time is much faster than when information needs to be sent to the cloud for processing. This is critical for applications that require real-time decision making. Deployment of models on TinyML systems significantly reduces latency, with a **range of 0 to 5 ms**, as compared to cloud-based machine learning models [3]. In addition, TinyML systems **maintain high accuracy**, with a slight reduction from 95% to 85% due to compression and optimization for devices with limited resources [3].

- **Offline Capability**
  TinyML models work without an internet connection, unlike cloud models, making them ideal for **areas with limited or no internet** access.

- **Improving Privacy and Security**
  TinyML **keeps data on the device** itself, avoiding sending it to the cloud, which protects user

privacy and complies with data protection regulations.

- **Low Energy Consumption**
  An important part of reducing energy consumption in TinyML is to decrease the amount of data to be transported and processed. The algorithms are designed to be **efficient**, helping to reduce device consumption. Other strategies include using **low-power components**, operating at low voltage or battery power, and adopting sleep or idle modes.

- **Reducing Cost**
  TinyML models save on the costs of sending data to the cloud, such as **bandwidth** and **storage**, in addition to having low energy consumption. They allow IoT devices to perform data analysis locally, **speeding up decision-making**, and to provide independent ML services.

Deployment of TinyML

Currently, models cannot be trained directly on embedded devices due to limited resources. They are typically trained in the cloud or on more powerful devices before being transferred. Deployment can be done by **hand coding** (low-level optimization, but time-consuming), **code generation** (optimized code, but with portability issues), or **ML interpreters**, which allow machine learning algorithms to be executed on Micro Controller Units within frameworks with predefined libraries and kernels.

A TinyML framework includes, in addition to the interpreter, libraries and tools for data processing, as well as a **Tiny Inference Engine**, which efficiently performs the calculations necessary for inference. These frameworks also offer development tools for data preparation, training, validation, and performance profiling, facilitating the creation and execution of models on low-power devices. [3]

Challenges Facing TinyML

TinyML faces several challenges. **Environmental adaptation** is limited since most models are trained offline and cannot adjust to changing conditions (concept drift), although solutions like TinyOL (TinyML with Online-Learning) enable real-time online learning on microcontrollers. **Memory limitations** force trade-offs between model size, accuracy, and energy use, addressed by techniques such as compression and quantization (e.g., AIMET). **Hardware and software heterogeneity** across devices, operating systems, and sensors complicates integration, while diverse data formats and noise hinder model generalization. **Accuracy drop** often occurs when deploying models on low-power devices due to memory and energy constraints, with reported losses of 1-2% in sensitive applications like healthcare. **Privacy risks** arise from sensors (cameras, microphones) capturing sensitive data, with mitigation strategies including on-device learning and TinyML as-a-Service (TinyMLaaS) to avoid cloud sharing. Finally, **reliability** and **robustness** are concerns, as TinyML devices may suffer from hardware errors, degradation, or interference in safety-sensitive domains. [3]

Chapter 3

State of Art

Around 1990, IDS prototypes used for the first time “inductive learning / sequential user patterns” (e.g., Time-based Inductive Machine (TIM) [50]). In the early 90s, systems like IDES (Intelligent Detection Expert System), NIDES (Next-generation Intrusion Detection Expert System) were developed. NIDES used a combination of statistical metrics and profiles [51]. And eventually, research into neural networks for intrusion detection started around 1992 [52] [53].

By incorporating machine learning, IDS products became more accurate, learning from past data to adapt to new and evolving threats. The following sections review research applications and empirical findings on ML and AI methods in IDS, focusing on system performance, comparative studies, and domain-specific implementations.

3.1 ML Methods Applied to IDS

While traditional ML methods are discussed in Section 2.2 of the Background chapter, this section examines empirical research on their application in IDS.

3.1.1 Comparative Performance Studies

Comparative studies have evaluated the performance of different ML algorithms on IDS benchmarks. Feature selection techniques, such as Boruta [54], have been shown to improve Random Forest efficiency in IDS. Naïve Bayes, when combined with **feature selection**, improves detection performance [23, 24]. One-Class SVM (OCSVM) combined with autoencoders improves detection in semi-supervised scenarios where labeled attack data is scarce [22]. Logistic regression combined with multinomial models demonstrates enhanced detection performance and reduced misclassification [28, 29].

3.1.2 Multi-Layer Perceptrons in IDS Research

MLPs have been extensively applied in IDS using labeled datasets. Studies have shown that MLPs can achieve **high detection accuracy** with reported rates up to 95.6% [55, 56] and effectively **distin-**

guish between different types of attacks [57], providing a **reliable** method for intrusion detection.

3.1.3 Reinforcement Learning Applications

Reinforcement Learning has shown promise in IDS for adaptive attack detection. Deep Q-Networks (DQN) extend Q-learning by using deep neural networks to approximate Q-values, learning complex patterns from raw network data and outperforming traditional Q-learning in detecting **sophisticated attacks** [58].

3.2 AI Methods and Deep Learning Approaches

While machine learning methods provide effective data-driven approaches, AI in IDS encompasses broader techniques including **expert** and **rule-based** systems that rely on **predefined rules or signatures** to detect known attacks, as well as **hybrid systems** that integrate **machine learning with rule-based logic** to enhance detection accuracy and reduce false positives. Deep learning models extend traditional ML by analyzing **complex**, **high-dimensional**, or **sequential data**. This section reviews research on CNN and RNN applications in IDS, as these architectures are discussed theoretically in Section 2.3.1 and 2.3.1 of the Background chapter.

3.2.1 Convolutional Neural Networks in IDS

CNN models have demonstrated superior performance compared to traditional ML methods in detecting diverse network intrusions. The capability to automatically learn and extract discriminative features directly from raw data makes CNNs particularly effective [59]. Vinayakumar et al. (2017) applied CNNs to analyze time-series network traffic data and demonstrated that the model **accurately detected anomalies**, **reduced false alarms**, and **improved overall IDS performance** [60]. Research in CNN-based IDS has shown that these architectures excel at capturing complex patterns in network traffic for **effective detection** of **sophisticated intrusions** [61, 62].

3.2.2 Recurrent Neural Networks in IDS

RNNs are well-suited for IDS because they can model the **temporal dynamics** of network traffic, capturing patterns that may **indicate** an **ongoing intrusion** [18]. Long Short-Term Memory (LSTM) networks, a variant of RNNs, have been emphasized in IDS research to enhance performance [63]. LSTMs improve upon standard RNNs by using memory cells that retain information over extended sequences, enabling the model to **capture long-term dependencies** in network traffic. Both RNNs and their variants are powerful tools for **sequential data analysis**, effectively learning temporal patterns critical for accurate intrusion detection [18].

3.3 IDS for IoT

The Internet of Things (IoT) represents a transformative shift in how networked devices communicate and coordinate, spanning from consumer smart homes and wearable devices to industrial sensors and connected vehicles. However, the extensive interconnection of these heterogeneous, resource-constrained devices introduces substantial security challenges, making robust Intrusion Detection Systems (IDS) essential for protecting the integrity, confidentiality, and availability of IoT networks [18]. Unlike traditional enterprise IDS deployed on well-resourced infrastructure, IoT environments demand lightweight detection methods that operate under strict computational, memory, and energy constraints. This section examines attack vectors specific to IoT networks, distinguishes between generic and IoT-specific IDS design requirements, and discusses evaluation criteria appropriate for resource-constrained deployments.

3.3.1 Types of Intrusions in IoT Networks

Intrusions in IoT environments can target the **physical infrastructure**, **communication protocols**, **software components**, or **cryptographic mechanisms**. Following the taxonomy of Thakkar and Lohiya [64], IoT intrusions are categorized into four primary types:

1. **Physical Intrusions** - Attacks directed at IoT hardware or physical devices that disrupt normal operations, damage components, or alter stored data.
2. **Network Intrusions** - Exploitation of weaknesses in data routing processes, enabling attackers to intercept, drop, or reroute network packets, potentially compromising multiple nodes within the network.
3. **Software Intrusions** - Malicious programs such as viruses or worms that exploit vulnerabilities in system hardware or software, leading to data theft, corruption, or deletion.
4. **Encryption Intrusions** - Attacks that undermine the confidentiality of encrypted communications by observing and decoding side-channel information transmitted through the network.

These intrusion categories collectively threaten the **integrity**, **confidentiality**, and **availability** (CIA triad) of IoT systems, as illustrated in Figure 3.1.

Flooding Attacks

Flooding Attacks: These represent a class of Denial-of-Service (DoS) attacks that exhaust device resources by overwhelming targets with malicious traffic. Common variants include:

- **SYN Flooding**: Attackers send numerous TCP SYN packets (often from spoofed IP addresses), leaving the server with unfinished half-open connections that exhaust connection tables and CPU resources [65].

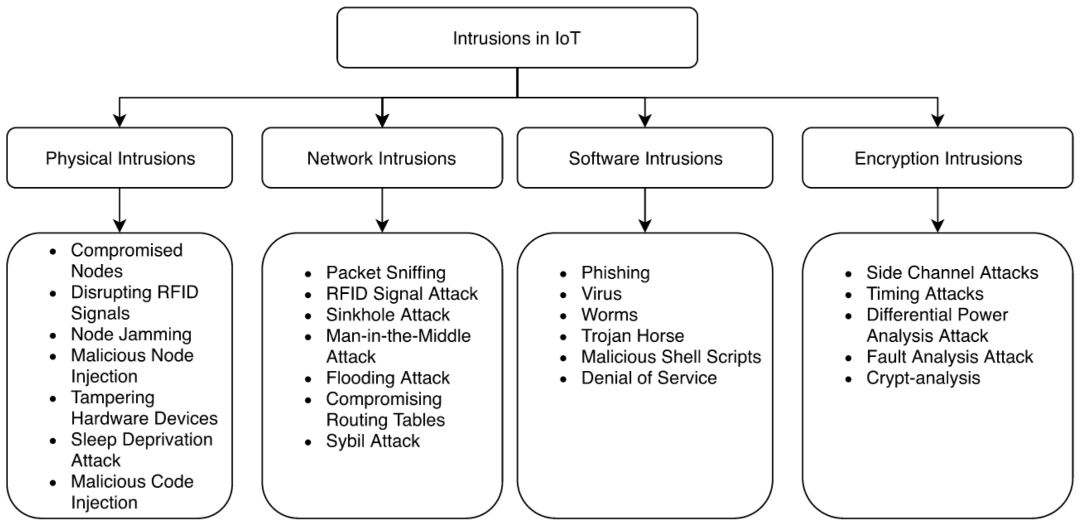

Figure 3.1: Taxonomy of intrusion types in IoT networks [64]

- **UDP Flooding**: Attackers transmit large volumes of UDP packets to consume bandwidth and computational resources, particularly effective against IoT devices with limited processing capability.
- **ACK Flooding**: Invalid ACK packets are sent to overwhelm the target, forcing it to expend resources on processing malformed acknowledgments [65].
- **HTTP Flooding**: Application-layer attacks sending legitimate-appearing HTTP requests to exhaust web service resources.

These flooding attacks are particularly dangerous in resource-constrained IoT environments, where even moderate attack volumes can deplete battery reserves, exhaust memory, or saturate communication channels.

3.3.2 Evaluation Criteria for IoT IDS

Table 3.1 compares evaluation criteria for generic and IoT-specific IDS, highlighting the distinct priorities and constraints of each deployment model.

3.3.3 Feature Selection for High-Dimensional IoT Data

Network traffic datasets often contain hundreds of features capturing protocol-level information, packet statistics, and behavioral indicators. However, high-dimensional feature spaces create substantial challenges for resource-constrained IoT devices: increased memory consumption for storing features, elevated computational burden during model inference, and amplified communication overhead when transmitting feature vectors. Feature selection techniques address these constraints by identifying and retaining only the most discriminative features while discarding redundant or irrelevant attributes, thereby reducing both model complexity and deployment resource requirements.

Table 3.1: Evaluation Criteria for Generic and IoT IDS [18]

Evaluation Factor	**Generic IDS**	**IoT IDS**
Detection Accuracy	Measures the system's ability to correctly identify intrusions and minimize false positives/negatives.	Equally important, but detection methods must be **lightweight** to **avoid overloading** limited IoT resources.
False Positive Rate	Low false positives reduce unnecessary alerts and prevent security team overload.	False positives can disrupt normal IoT operations and **waste energy**.
False Negative Rate	Avoiding missed attacks is critical to maintain security.	More critical in IoT, as missed detections can interrupt **real-time operations** (e.g., healthcare or industrial systems).
Real-time Performance	Quick response is desirable, but slight delays may be acceptable in non-critical systems.	**Extremely important**; IoT environments often require immediate threat detection (e.g., autonomous vehicles, industrial control).
Scalability	Must handle large enterprise networks using powerful infrastructure.	Must efficiently manage thousands of resource-limited IoT devices in distributed setups.
Computational Overhead	Can be high, especially for AI-based IDS, as enterprise devices have ample computing power.	Must remain low due to the limited CPU, memory, and energy of IoT devices.
Network Overhead	Moderate overhead is acceptable for monitoring and logging.	Must be **minimized**, as IoT networks often have low bandwidth.
Adaptability to New Attacks	Requires regular updates to detect evolving threats.	Needs lightweight, **adaptive models** capable of learning new threats with minimal **retraining**. Frequent updates may be impractical.
Energy Efficiency	Not a major concern in traditional IDS.	**Highly important**; IoT devices often run on batteries and cannot support energy-intensive monitoring.
Privacy & Data Sensitivity	Operates within secure enterprise infrastructure, so privacy is less of a concern.	**Critical** in healthcare, smart homes, and industrial IoT, where sensitive data must be protected.
Deployment Model	Typically centralized with a dedicated security team.	Often **decentralized**, using edge or fog computing to bring detection closer to devices.
Robustness Against Adversarial Attacks	Must handle sophisticated attacks like polymorphic malware.	More vulnerable to attacks like adversarial ML, sensor spoofing, and firmware exploits.
Integration with Security Frameworks	Works with firewalls, Security information and event management (SIEM) systems, and endpoint security tools.	Requires lightweight integration due to resource constraints; traditional firewalls may be unavailable.

Hybrid Approach: Correlation-Based Feature Selection and Principal Component Analysis Hassan et al. (2025) combined Correlation-Based Feature Selection (CFS) and Principal Component Analysis (PCA) for feature reduction on the CICIDS2017 dataset (79 features) [66]. CFS identifies features correlated with attack classes while removing redundancy, reducing features from 70 to 39-47 features (33-44% reduction) at different correlation thresholds. PCA performs linear dimensionality reduction while preserving variance, achieving 64-71% reduction (70 to 20-25 features) at corresponding variance thresholds. This hybrid approach, combined with optimized Fully Connected Neural Networks, effectively balances detection performance and computational efficiency.

The results demonstrate substantial benefits: F1-scores improved from 96.48% (baseline) to 98.43% (CFS at 90% correlation), model size reduced 28%, and parameters decreased from 7,192 to 5,208 [66]. Post-training quantization (32-bit to 8-16 bit) further compressed the model to 9 KB [66]. This multi-stage optimization approach demonstrates that high-performing intrusion detection systems can achieve low computational complexity suitable for resource-constrained vehicular ad-hoc networks (VANETs).

Deep Autoencoder-Based Dimensionality Reduction For extremely high-dimensional datasets, deep autoencoders offer nonlinear dimensionality reduction by learning compressed representations that preserve essential information while discarding noise. Kumar et al. (2025) applied autoencoders to the CSE-CIC-IDS2018 dataset containing 80 features across 16.2 million network traffic instances [67]:

The autoencoder compressed the feature space from 80 to 18 dimensions (77.5% reduction) while maintaining critical attack-detection capability. The dimensionality-reduced feature set was subsequently classified using an Extreme Learning Machine (ELM), achieving 98.52% detection accuracy [67]. This hybrid approach demonstrates that aggressive dimensionality reduction-when combined with lightweight classification methods-can achieve high-performance intrusion detection suitable for edge deployment.

Chapter 4

Frameworks for TinyML

TinyML relies on a variety of software platforms, hardware requirements, and libraries to make predictions. In this chapter, we will go over some of the most well-known frameworks.

4.1 Lite Runtime (LiteRT)

LiteRT is a deep learning framework that is open source and supports edge-aware learning inference [68]. Lite Runtime (LiteRT), formerly known as **TensorFlow Lite**, is Google's high-performance runtime for on-device AI. You can find ready-to-run LiteRT models for a wide range of ML/AI tasks, or **convert** and **run TensorFlow**, **PyTorch**, and **JAX models** to the **TFLite format** using the AI Edge conversion and optimization [69].

It provides tools to convert models from TensorFlow, PyTorch, and JAX models into the FlatBuffers format ($.tflite$), enabling the use of a wide range of state-of-the-art models on LiteRT.

4.2 ExecuTorch (PyTorch)

ExecuTorch is PyTorch's solution for efficient AI inference on edge devices - from mobile phones to embedded systems. Allows developers to **deploy PyTorch-trained models** directly on diverse platforms, from high-end mobile to constrained microcontrollers. It is a lightweight runtime with full hardware acceleration (CPU, GPU, NPU, DSP)[70].

4.3 Embedded Learning Library (ELL)

The Embedded Learning Library (ELL) is an **Open Source Library** for Embedded AI and Machine Learning from Microsoft. It enables the **design** and **deployment** of machine learning models on resource-constrained platforms and small single-board computers (e.g., Raspberry Pi, Arduino, micro: bit). Models run **locally** without cloud dependencies. ELL is aimed at makers, students, and developers, and its tools and code are freely available, **though the library is still evolving**.

4.4 Framework Comparison

For deploying machine learning models on resource-constrained hardware such as vehicular on-board units (OBUs), several TinyML frameworks have been proposed. These frameworks differ in their resource efficiency, inference performance, and optimization support.

TensorFlow Lite (LiteRT) is specifically designed for on-device inference with **minimal memory** and **computational overhead**. LiteRT uses **model conversion** and **optimization techniques** to reduce model size and runtime requirements, making it suitable for embedded scenarios where **memory** and **latency are critical** [71].

Compared to **ExecuTorch (PyTorch)**, LiteRT generally produces **smaller binary sizes** and **faster inference** on embedded devices, due to its aggressive optimizations and custom runtime tailored for limited hardware resources [72]. While ExecuTorch offers flexibility and ease of use, its default deployments often consume more CPU cycles and memory, which can be suboptimal for deeply resource-constrained environments [72].

Libraries such as the **Embedded Learning Library (ELL)** can generate efficient C++ code for embedded inference, yet they are generally **less mature** and **lack the extensive optimization** tooling found in LiteRT.

Taken together, these comparisons highlight that LiteRT offers a compelling combination of **low memory footprint, real-time inference performance, strong optimization support, and broad support for edge hardware targets**. These characteristics make LiteRT especially suitable for the thesis's objective of deploying lightweight deep learning intrusion detection models on resource-limited vehicular devices.

Chapter 5

Preliminary Experiment

5.1 Methodology

A **convolutional neural network** was trained in **Google Colab** (12 GB RAM, 2.20 GHz CPU with 2 cores [73]) using the **MNIST dataset** [74], a standard benchmark for handwritten digit classification consisting of 70,000 28×28 grayscale images (60,000 training, 10,000 testing). The model architecture comprised a 3×3 convolution with 12 filters, max-pooling for spatial reduction, a dense layer with 64 units, and a final classification layer, totaling **130,626 trainable parameters** predominantly in fully connected layers.

Training required approximately 5 min 28 sec. The model was converted to LiteRT using **dynamic range quantization** [75], statically converting weights to 8-bit integers while dynamically quantizing activations during inference, achieving a **91.43% size reduction** (1563 KB to 134 KB) in approximately 1.13 s.

Both the optimized LiteRT and original Keras models were deployed on a **Raspberry Pi 4** (4 GB RAM, 1.8 GHz CPU with 4 cores [76]).

Evaluation consisted of **1000 inference runs** over the full MNIST test set, capturing inference latency, throughput (images/s), and resource usage (CPU %, resident RAM bytes). Resource monitoring sampled every 10 ms; each run was summarized by modal values from 50-bin histograms, providing typical resource consumption while down-weighting transient spikes.

5.2 Conclusions

The comparative analysis of MNIST inference across Google Colab (Keras) and Raspberry Pi 4 (TensorFlow Lite and Keras) reveals critical insights into the effectiveness of model optimization for edge deployment. See Table 5.1 for a summary of results.

5.2.1 Model Quantization Efficacy

Both platforms achieve virtually identical accuracy ($\approx 98.5\%$), demonstrating that dynamic range quantization preserves the model's predictive capability while dramatically reducing computational burden. The LiteRT model achieves a **91.4% size reduction** (1563 KB to 134 KB), enabling deployment on severely resource-constrained devices without accuracy degradation.

5.2.2 Latency Trade-offs

On the Raspberry Pi 4, LiteRT demonstrates superior latency performance compared to Keras, achieving 0.209 ms per image versus 0.223 ms for the unoptimized Keras model, representing a **6.3% latency reduction**. This advantage stems from both the aggressive quantization applied to the model and the highly optimized LiteRT runtime, which minimizes computational overhead. While both implementations exhibit approximately 1.55× higher latency than the Colab environment (0.135 ms), this overhead is acceptable given the Pi's limited computational capacity.

5.2.3 Resource Efficiency on Edge Hardware

The most striking result is CPU utilization: LiteRT on the Pi consumes only **24.3% of the CPU load** required by Keras on the same hardware (88.90% vs 365.57%), demonstrating that quantization and the optimized LiteRT runtime dramatically improve computational efficiency. This efficiency advantage enables other processes to run concurrently without starvation, a critical requirement for embedded applications.

5.2.4 Memory Considerations

Quantization significantly reduces model size (91.4%), but does not necessarily reduce runtime memory usage. The observation that Keras consumes less RAM (728.09 MB) than LiteRT (999.82 MB) on Raspberry Pi likely reflects **differences in memory allocation strategies**. LiteRT preallocates a single contiguous memory arena before inference to accommodate all tensors, as documented in its arena allocator [77]. In contrast, Keras's eager execution mode, which executes operations immediately rather than deferring them to a graph [78], may employ more flexible allocation strategies.

The arena must hold not only model weights but also **all intermediate activation tensors**, which typically maintain full precision during inference and dominate memory consumption. Consequently, LiteRT requires 999.82 MB (24.4% of Pi's 4 GB RAM) despite its 134 KB quantized model file.

5.2.5 Practical Implications

This experiment validates LiteRT's suitability for edge-based intrusion detection on vehicular networks. The results demonstrate a clear optimization trade-off: while LiteRT consumes 37.3% more peak RAM than Keras (999.82 MB vs 728.09 MB), it delivers substantial benefits in deployment scenarios where computational resources are constrained.

Metric	**Colab (Keras)**	**Pi 4 (LiteRT)**	**Pi 4 (Keras)**	**Ratio Pi-Keras/Colab**	**Ratio Pi-LiteRT/Pi-Keras**
Hardware					
CPU	2.20 GHz (2 cores)	1.80 GHz (4 cores)	1.80 GHz (4 cores)	—	—
RAM	12 GB	4 GB	4 GB	—	—
Model					
Format	Keras (.keras)	LiteRT (.tflite)	Keras (.keras)	—	—
Size (KB)	1563	134	1563	—	0.086
Raw Performance					
Accuracy (%)	98.53	98.51	98.53	~ 1.000	~ 1.000
Latency (ms/image)	0.135	0.209	0.223	1.652	**0.937**
CPU Added Load					
Process CPU (%)	194.47	88.90	365.57	—	**0.243**
RAM Added Load					
Process RAM (MB)	1085.68	999.82	728.09	0.671	—
Share of total RAM (%)	8.85	24.40	17.80	—	**1.371**

Table 5.1: Comparison of model performance and resource utilization on Google Colab (Keras), Raspberry Pi 4 (LiteRT), and Raspberry Pi 4 (Keras). Raw metrics are from 1000 inference runs on the full MNIST test set. CPU and RAM added load metrics are modal values from samples taken every 0.01 seconds.

The **91.4% model size reduction enables deployment on storage-limited devices without significant accuracy loss** (98.51% vs 98.53%). More critically for embedded systems, **LiteRT's CPU efficiency is exceptional: consuming only 24.3% of Keras's CPU load** (88.90% vs 365.57%) frees computational resources for concurrent security monitoring tasks-essential for vehicular onboard units handling network traffic inspection alongside other vehicle functions. The **6.3% latency improvement (0.209 ms vs 0.223 ms) ensures real-time inference performance on constrained hardware**.

For V2V intrusion detection systems operating on resource-limited OBUs, LiteRT's trade-off profile is favorable: **manageable absolute RAM usage (24.4% of 4 GB) combined with dramatic CPU efficiency gains justifies the memory overhead**. This optimization enables deployment of multiple detection models or concurrent monitoring tasks without computational starvation - a critical requirement for vehicular cybersecurity applications where detection must not compromise vehicle safety or responsiveness.

Bibliography

[1] B. Sousa, N. Magaia, S. Silva, N. Thanh Hieu, and Y. Liang Guan. Vehicle-to-vehicle flooding datasets using mk5 on-board unit devices. *Scientific Data*, 11(1), Dec. 2024. URL `http://dx.doi.org/10.1038/s41597-024-04173-4`.

[2] C.-H. Wang, K.-Y. Huang, Y. Yao, J.-C. Chen, H.-H. Shuai, and W.-H. Cheng. Lightweight deep learning: An overview. *IEEE Consumer Electronics Magazine*, 13(4):51–64, July 2024. doi: 10.1109/MCE.2022.3181759.

[3] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid. A comprehensive survey on tinyml. *IEEE Access*, 11:96892–96922, 2023. doi: 10.1109/ACCESS.2023.3294111.

[4] Lightweight flow generator and packet analyzer - tranalyzer. `https://tranalyzer.com/`. Accessed: 2025-11-05.

[5] Wireshark • go deep. `https://www.wireshark.org/`. Accessed: 2025-11-05.

[6] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99–15, Chalmers University of Technology, Gothenburg, Sweden, 2000.

[7] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1), July 2019. doi: 10.1186/s42400-019-0038-7.

[8] Symantec. Internet security threat report 2017. Technical report, Symantec Corporation, April 2017. URL `https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf`. Accessed: 28-09-2025.

[9] H.-J. Liao, C.-H. Richard Lin, Y.-C. Lin, and K.-Y. Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, Jan. 2013. doi: 10.1016/j.jnca.2012.09.004.

[10] B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May 1994. doi: 10.1109/65.283931.

[11] P. Stavroulakis and M. Stamp, editors. *Handbook of Information and Communication Security*. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-04117-4.

[12] F. Sabahi and A. Movaghar. Intrusion detection: A survey. *2008 Third International Conference on Systems and Networks Communications*, page 23–26, 2008. doi: 10.1109/icsnc.2008.44.

[13] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, 36(1):42–57, Jan. 2013. doi: 10.1016/j.jnca.2012.05.003.

[14] G. Creech. *Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks*. PhD thesis, UNSW Sydney, 2014. URL `http://hdl.handle.net/1959.4/53218`.

[15] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network anomaly detection: Methods, systems and tools. *IEEE Communications Surveys & Tutorials*, 16(1):303–336, 2014. doi: 10.1109/surv.2013.052213.00046.

[16] C.-Y. Ho, Y.-C. Lai, I.-W. Chen, F.-Y. Wang, and W.-H. Tai. Statistical analysis of false positives and false negatives from real traffic with intrusion detection/prevention systems. *IEEE Communications Magazine*, 50(3):146–154, Mar. 2012. doi: 10.1109/mcom.2012.6163595.

[17] IBM. What is a decision tree? — ibm. `https://www.ibm.com/think/topics/decision-trees`, . Accessed: 2025-10-06.

[18] O. Hornyák. Intelligent intrusion detection systems – a comprehensive overview of applicable ai methods with a focus on iot security. *Infocommunications journal*, 17(Special Issue):61–76, 2025. doi: 10.36244/icj.2025.5.8.

[19] IBM. What is random forest? — ibm. `https://www.ibm.com/think/topics/random-forest`, . Accessed: 2025-10-06.

[20] W. Tao, F. Honghui, Z. HongJin, Y. CongZhe, Z. HongYan, and H. XianZhen. Intrusion detection system combined enhanced random forest with smote algorithm. Mar. 2021. doi: 10.21203/rs.3.rs-270201/v1.

[21] IBM. What is support vector machine? — ibm. `https://www.ibm.com/think/topics/support-vector-machine`, . Accessed: 2025-10-07.

[22] L. Mhamdi, D. McLernon, F. El-moussa, S. A. Raza Zaidi, M. Ghogho, and T. Tang. A deep learning approach combining autoencoder with one-class svm for ddos attack detection in sdns. In *2020 IEEE Eighth International Conference on Communications and Networking (ComNet)*, pages 1–6, 2020. doi: 10.1109/ComNet47917.2020.9306073.

[23] Z. Muda, W. Yassin, M. N. Sulaiman, and N. I. Udzir. Intrusion detection based on k-means clustering and naïve bayes classification. *2011 7th International Conference on Information Technology in Asia*, page 1–6, 2011. doi: 10.1109/cita.2011.5999520.

[24] D. H. Deshmukh, T. Ghorpade, and P. Padiya. Intrusion detection system by improved pre-processing methods and naïve bayes classifier using nsl-kdd 99 dataset. *2014 International Conference on Electronics and Communication Systems (ICECS)*, page 1–7, Feb. 2014. doi: 10.1109/ecs.2014.6892542.

[25] M. Panda and M. R. Patra. Network intrusion detection using naive bayes. *International journal of computer science and network security*, 7(12):258–263, 2007.

[26] IBM. What is the k-nearest neighbors algorithm? — ibm. `https://www.ibm.com/think/topics/knn`, . Accessed: 2025-10-07.

[27] IBM. What is logistic regression? — ibm. `https://www.ibm.com/think/topics/logistic-regression`, . Accessed: 2025-10-07.

[28] E. Besharati, M. Naderan, and E. Namjoo. Lr-hids: logistic regression host-based intrusion detection system for cloud environments. *Journal of Ambient Intelligence and Humanized Computing*, 10(9):3669–3692, Oct. 2018. doi: 10.1007/s12652-018-1093-8.

[29] Y. Wang. A multinomial logistic regression modeling approach for anomaly intrusion detection. *Computers & Security*, 24(8):662–674, Nov. 2005. doi: 10.1016/j.cose.2005.05.003.

[30] IBM. What is k-means clustering? — ibm. `https://www.ibm.com/think/topics/k-means-clustering`, . Accessed: 2025-10-07.

[31] Y. Y. Aung and M. Myat Min. Hybrid intrusion detection system using k-means and k-nearest neighbors algorithms. *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, page 34–38, 2018. doi: 10.1109/icis.2018.8466537.

[32] IBM. What is semi-supervised learning? — ibm. `https://www.ibm.com/think/topics/semi-supervised-learning`, . Accessed: 2025-10-07.

[33] S. J. Russell and P. Norvig, editors. *Artificial Intelligence A Modern Approach, Global Edition*. Pearson, 3rd edition, 2016. ISBN:9781292153964.

[34] T. M. Mitchell, editor. *Machine learning*. McGraw-Hill Science/Engineering/Math, 1997. ISBN:0070428077.

[35] I. Drori. *The Science of Deep Learning*. Cambridge University Press, 2022. ISBN:9781108835084.

[36] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 2002.

[37] B. J. Shiland. *Chapter 8-Introduction to Anatomy and Physiology*, page 204. Elsevier, 2nd edition, 2016.

[38] O. A. Montesinos López, A. Montesinos López, and J. Crossa. *Fundamentals of Artificial Neural Networks and Deep Learning*. Springer International Publishing, Cham, 2022. ISBN 978-3-030-89010-0. doi: 10.1007/978-3-030-89010-0_10.

[39] N. Lewis. Deep learning made easy with r. *A gentle introduction for data science. South Carolina: CreateSpace Independent Publishing Platform*, 2016.

[40] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[41] IBM. What is a recurrent neural network? — ibm. `https://www.ibm.com/think/topics/recurrent-neural-networks`, . Accessed: 2025-10-07.

[42] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[43] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[44] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5687–5695, 2017.

[45] M. Lin et al. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1526–1535, 2020. doi: 10.48550/arXiv.2002.10179.

[46] Z. Liu et al. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. doi: 10.48550/arXiv.1810.05270.

[47] H. Yin et al. Dreaming to distill: Data-free knowledge transfer via deepinversion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8172–8181, 2020. doi: 10.1109/CVPR42600.2020.00819.

[48] Y. Zhang, H. Chen, X. Chen, Y. Deng, C. Xu, and Y. Wang. Data-free knowledge distillation for image super-resolution. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7848–7857, 2021. doi: 10.1109/CVPR46437.2021.00774.

[49] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer. Zeroq: A novel zero-shot quantization framework. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13166–13175, 2020. doi: 10.1109/CVPR42600.2020.01318.

[50] H. S. Teng, K. Chen, and S. C. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 278–284, Oakland, California, May 1990. IEEE. doi: 10.1109/RISP.1990.63857.

[51] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. *2010 IEEE Symposium on Security and Privacy*, page 305–316, 2010. doi: 10.1109/sp.2010.25.

[52] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 240–250, Oackland, CA, May 1992. IEEE, IEEE Computer Society Press. ISBN 0-8186-2825-1. doi: 10.1109/RISP.1992.213257. URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=213257`.

[53] H. Debar and B. Dorizzi. An application of a recurrent network to an intrusion detection system. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN 1992)*, volume 2, pages 478–483, Baltimore, MD, USA, June 1992. IEEE, IEEE Computer Society Press. ISBN 0-7803-0559-0. doi: 10.1109/IJCNN.1992.226942. URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=226942`.

[54] S. Subbiah, K. S. M. Anbananthen, S. Thangaraj, S. Kannan, and D. Chelliah. Intrusion detection technique in wireless sensor network using grid search random forest with boruta feature selection algorithm. *Journal of Communications and Networks*, 24(2):264–273, Apr. 2022. doi: 10.23919/jcn.2022.000002.

[55] H. Ji, D. Kim, D. Shin, and D. Shin. A study on comparison of kdd cup 99 and nsl-kdd using artificial neural network. In J. J. Park, V. Loia, G. Yi, and Y. Sung, editors, *Advances in Computer Science and Ubiquitous Computing*, pages 452–457, Singapore, 2018. Springer Singapore. ISBN 978-981-10-7605-3.

[56] R. D. Ravipati and M. Abualkibash. Intrusion detection system classification using different machine learning algorithms on kdd-99 and nsl-kdd datasets - a review paper. *SSRN Electronic Journal*, 2019. doi: 10.2139/ssrn.3428211.

[57] A. Jassam Mohammed, M. Hameed Arif, and A. Adil Ali. A multilayer perceptron artificial neural network approach for improving the accuracy of intrusion detection systems. *IAES International Journal of Artificial Intelligence (IJ-AI)*, 9(4):609, Dec. 2020. doi: 10.11591/ijai.v9.i4.pp609-615.

[58] E. Suwannalai and C. Polprasert. Network intrusion detection systems using adversarial reinforcement learning with deep q-network. *2020 18th International Conference on ICT and Knowledge Engineering (ICT&KE)*, page 1–7, Nov. 2020. doi: 10.1109/ictke50349.2020.9289884.

[59] Y. Ding and Y. Zhai. Intrusion detection system for nsl-kdd dataset using convolutional neural networks. *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, page 81–85, Dec. 2018. doi: 10.1145/3297156.3297230.

[60] R. Vinayakumar, K. P. Soman, and P. Poornachandran. Applying convolutional neural network for network intrusion detection. *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, page 1222–1228, 2017. doi: 10.1109/icacci.2017.8126009.

[61] L. Mohammadpour, T. C. Ling, C. S. Liew, and A. Aryanfar. A survey of cnn-based network intrusion detection. *Applied Sciences*, 12(16):8162, Aug. 2022. doi: 10.3390/app12168162.

[62] M. Azizjon, A. Jumabek, and W. Kim. 1d cnn based network intrusion detection with normalization on imbalanced data. *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, page 218–224, Feb. 2020. doi: 10.1109/icaiic48513.2020.9064976.

[63] S. M. Sohi, J.-P. Seifert, and F. Ganji. Rnnids: Enhancing network intrusion detection systems through deep learning. *Computers & Security*, 102:102151, Mar. 2021. doi: 10.1016/j.cose.2020.102151.

[64] A. Thakkar and R. Lohiya. A review on machine learning and deep learning perspectives of ids for iot: Recent updates, security issues, and challenges. *Archives of Computational Methods in Engineering*, 28(4):3211–3243, Oct. 2020. doi: 10.1007/s11831-020-09496-0.

[65] D. Tymoshchuk, O. Yasniy, M. Mytnyk, N. Zagorodna, and V. Tymoshchuk. Detection and classification of ddos flooding attacks by machine learning method. *arXiv*, 2024. doi: 10.48550/ARXIV.2412.18990. URL https://arxiv.org/abs/2412.18990.

[66] F. Hassan, Z. S. Syed, A. A. Memon, S. S. Alqahtany, N. Ahmed, M. S. A. Reshan, Y. Asiri, and A. Shaikh. A hybrid approach for intrusion detection in vehicular networks using feature selection and dimensionality reduction with optimized deep learning. *PLOS ONE*, 20(2):e0312752, Feb. 2025. URL http://dx.doi.org/10.1371/journal.pone.0312752.

[67] A. Kumar, R. Radhakrishnan, M. Sumithra, P. Kaliyaperumal, B. Balusamy, and F. Benedetto. A scalable hybrid autoencoder–extreme learning machine framework for adaptive intrusion detection in high-dimensional networks. *Future Internet*, 17(5):221, May 2025. URL http://dx.doi.org/10.3390/fi17050221.

[68] N. Schizas, A. Karras, C. Karras, and S. Sioutas. Tinyml for ultra-low power ai and large scale iot deployments: A systematic review. *Future Internet*, 14(12):363, Dec. 2022. doi: 10.3390/fi14120363. URL http://dx.doi.org/10.3390/fi14120363.

[69] Lite rt overview — google ai edge — google ai for developers. https://ai.google.dev/edge/litert, . Accessed: 2025-12-02.

[70] Welcome to the executorch documentation. https://docs.pytorch.org/executorch/stable/index.html#wins-success-stories. Accessed: 2026-01-03.

[71] On-device inference with litert — google ai edge — google ai for developers. https://ai.google.dev/edge/litert/inference, . Accessed: 2026-01-03.

[72] P. D. Benchmarking tensorflow lite vs pytorch performance, 2025. URL http://bit.ly/3LwAuTL. Accessed: 2026-01-03.

[73] colab.google. https://colab.google/. Accessed: 2025-11-26.

[74] Mnist database - wikipedia. https://en.wikipedia.org/wiki/MNIST_database. Accessed: 2025-11-26.

[75] Post-training quantization — google ai edge — google ai for developers. https://ai.google.dev/edge/litert/models/post_training_quantization#dynamic_range_quantization. Accessed: 2025-12-02.

[76] Raspberry pi 4 model b specifications – raspberry pi. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. Accessed: 2025-11-26.

[77] Arena planner implementation. URL https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/arena_planner.h. TensorFlow Lite C++ API Reference, Accessed: 2026-01-10.

[78] Eager execution. URL https://www.tensorflow.org/api_docs/python/tf/compat/v1/enable_eager_execution. TensorFlow API Documentation, Accessed: 2026-01-10.