Journal of Autonomous Intelligence (2024) Volume 7 Issue 4
doi: 10.32629/jai.v7i4.1186

ORIGINAL RESEARCH ARTICLE

TinyML: Adopting tiny machine learning in smart cities

Norah N. Alajlan[1], Dina M. Ibrahim[1,2,*]

[1] *Department of Information Technology, College of Computer, Qassim University, Buraydah 51452, Saudi Arabia*

[2] *Department of Computers and Control Engineering, Faculty of Engineering, Tanta University, Tanta 31733, Egypt*

*** Corresponding author:** Dina M. Ibrahim, d.hussein@qu.edu.sa

ABSTRACT

Since Tiny machine learning (TinyML) is a quickly evolving subject, it is crucial that internet of things (IoT) devices be able to communicate with one another for the sake of stability and future development. TinyML is a rapidly growing subfield at the intersection of computer science, software engineering, and machine learning. Building deep learning (DL) networks that are a few hundred KBs in size has been the focus of recent research in this area. Deploying TinyML into small devices makes them smart. Reduced computation, power usage, and response time are just a few of the many advantages of TinyML. In this work, we provide an introduction to TinyML and demonstrate its benefits and architecture. Then, we investigate the meaning of quantization as a standard compression method for TinyML-related applications. There are two methods used to obtain the quantized weights of the deep learning models are quantization-aware training (QAT) and post-training quantization (PTQ), we described them in details. Next, TinyML-based solutions to improve the role of IoT devices in Smart Cities are highlighted as: lightweight training of deep learning models, inference of lightweight deep learning models in IoT devices, low power consumption, and inference of deep learning models in restricted resources of IoT devices. Finally, presenting some use cases for TinyML studies, with these studies applied to several cases in a variety of fields. To the best of the author's knowledge, few studies have investigated TinyML as it is an emerging field.

Keywords: TinyML, machine learning; deep learning, smart city; Internet of Things; quantization

ARTICLE INFO

Received: 28 August 2023
Accepted: 13 November 2023
Available online: 26 January 2024

COPYRIGHT

Copyright © 2024 by author(s).
Journal of Autonomous Intelligence is published by Frontier Scientific Publishing. This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).
https://creativecommons.org/licenses/by-nc/4.0/

1. Introduction

Tiny machine learning (TinyML) is a relatively recent discipline that has already produced a number of innovations and has sped up the development of IoT industries including autonomous vehicles, smart cities, and smart transportation. With the aid of the alternative paradigm known as TinyML, deep learning operations can be carried out locally on systems that typically consume less than one milliwatt of power. As a result, TinyML enables real-time data analysis and interpretation, which has enormous benefits for latency, privacy, and cost[1,2]. The main objective of TinyML is to increase the effectiveness of deep learning algorithms by using less computational power and data, which supports the enormous edge artificial intelligence (AI) industry and the Internet of Things[2]. ABI Research, a global digital industry consultancy firm, predicts that 2.5 billion TinyML-equipped devices would be sold worldwide by 2030. These gadgets are focused on ultra-power-efficient AI chipsets, low cost, low latency data transmission, and enhanced automation. While the training phase of these devices continues to rely on external resources, such as gateways, on-premises servers, or the cloud, the chipsets are characterized as intelligent IoT (AIoT) or embedded AI since they do AI inference almost entirely on the board.

As stated by the authors of the literature related to TinyML[3], The authors explore the design space for machine learning-aware architectures, frameworks, techniques, tools, and approaches that can perform on-device analytics for a wide range of sensing modalities (vision, audio, speech, motion, chemical, physical, textual, cognitive) at [an] mW (or below) power range setting, with a focus on battery-operated embedded edge devices, and with an eye toward deployment at scale, preferably in the Internet of Things (IoT) or wireless[4].

Hardware, software, and algorithms make up the three essential parts of TinyML. For a better learning experience, the hardware can include IoT devices that are based on analog computing, in-memory computing, or neuromorphic computing, with or without hardware accelerators. Due to their specs, microcontroller units (MCUs) are regarded as the best hardware platforms for TinyML[4]. A microcontroller is typically inexpensive, compact (less than 1 cm^3), and low-power[5]. A CPU, data and program memory (RAM and flash memory), and a number of input/output peripherals are all included in the microcontroller chip. Due to its inclusion of the majority of useful hardware characteristics, microcontrollers are used on a global scale. Their flash memory capacity goes from 32 KB to 2 MB, while their RAM storage spans from 8 KB to 320 KB. Their clock speed ranges from 8 MHz to around 500 MHz. Overall, TinyML leverages inexpensive hardware while effectively using electricity and delivering great performance[6]. Giants in the industry have recently expressed interest in TinyML's software. The TensorFlow Lite (TFLite) framework, for instance, was made available by Google and enables the use of neural network (NN) models on Internet of Things (IoT) devices[7]. Similar to how Microsoft released EdgeML[8], ARM[9], and published the Cortex microcontroller software interface standard neural net-work (CMSIS-NN), an open-source framework for Cortex-M processors that improves NN performance[10]. Additionally, a fresh program named X-Cube-AI[10] has been made available for STM 32-bit microcontrollers[11] to run deep learning models. For a TinyML system, deep learning algorithms should be compact (just a few KB). The size of the deep learning model is reduced using model compression techniques to enable deployment on IoT devices with limited resources[4].

2. Background on TinyML

TinyML is a dynamic and fast evolving subject that, in order to maintain stability and continuing advancement, requires interoperability across IoT devices. TinyML is an emerging topic that connects hardware, software, and machine-learning algorithms and is getting an incredible amount of traction. TinyML combines all three of these aspects of machine learning. Recent research in this area has fo-cused on the construction of deep learning networks that are only a few hundred KB in size. TinyML is implemented in low-powered devices in order to transform them into intelligent ones. TinyML offers a number of advantages, including a reduction in the amount of computation, power usage, and response time.

2.1. Benefits of TinyML

TinyML has a series of benefits once it is integrated with IoT devices in smart cities, with these in-cluding energy efficiency and latency. Other TinyML benefits are also described below:

- Energy efficiency: This is a significant advantage when using TinyML because IoT devices that use microcontroller units (MCUs) depend on batteries or even energy harvesting; as a result, they use less energy than other IoT devices that use more powerful processors and GPUs that demand a lot of power. As a result, IoT devices may be installed everywhere without having to be connected to the power grid, which makes way for cutting-edge cognitive nomadic applications. IoT devices can also be connected to bigger battery-powered devices due to their low power consumption, making them become connected smart entities, such as personal mobility devices like scooters or segways[12];
- Low cost: Various tasks are carried out by IoT devices in numerous IoT applications. They need great

computational processing power, vast storage, and a large memory, among other good criteria. Additionally, they use a lot of storage on the cloud to process data and store it, which results in expensive prices because they consume a lot of resources. TinyML uses low-cost microcontroller chips to analyze data locally and applies AI to the device itself for high-performance data processing[12]. Microcontrollers are less expensive than other options since they only employ a small amount of resources; their processors operate between 1 MHz and 400 MHz. The storage capacity can be between 32 KB and 2 MB, while the microcontroller memory can be between 2 KB and 512 KB. In comparison to other smart IoT devices used to process data locally using deep learning models, a microcontroller only costs a few dollars.

- Latency: TinyML does data processing locally since computations are done on the device. IoT devices therefore do not experience latency. In emergency situations, real-time local data processing in the devices enables quicker analysis and response times. Furthermore, the strain on the cloud is lessened[13].
- System reliability and data security: The transmission of raw data from IoT devices to the cloud re-quires communication channels. When data are transmitted to the cloud, they are susceptible to trans-mission errors, cyberattacks, such as surveillance or man-in-the-middle problems, and transmission errors. This could result in the data being compromised or lost during transmission. According to the "Cost of Data Breach" report from IBM for 2020, the average expense of a data breach is $3.86 million. Consequently, data must be processed locally to reduce cloud traffic. TinyML can avoid these problems by conducting data processing locally on the same device. This allows it to perform fewer transmissions containing aggregated or irrelevant data that could be compromised[12,13].

2.2. Architecture of TinyML

The architecture for TinyML (and hence its workflow) is divided into seven phases, with each phase connected to the next phase. **Figure 1** illustrates the stages which are described as follows:

1) First phase: Collection and pre-processing of data. Data, such as temperature, sound, signals, images, etc., are collected by sensors or any IoT device. Pre-processing is then performed to extract features which can be used as input to deep learning models for training and testing.

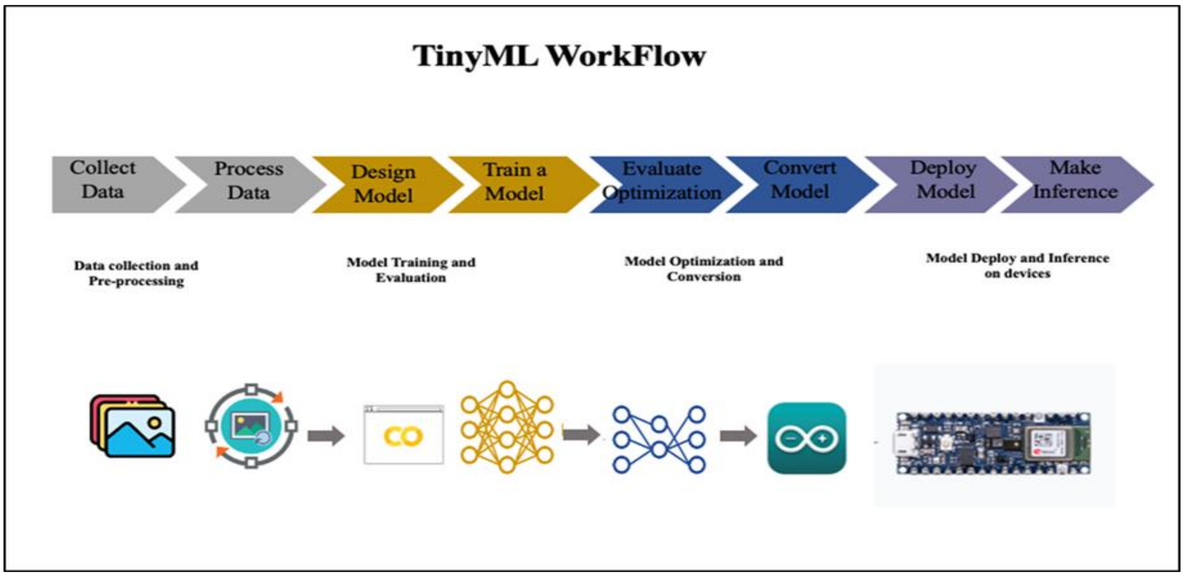
TinyML WorkFlow
Collect Data
Process Data
Design Model
Train a Model
Evaluate Optimization
Convert Model
Deploy Model
Make Inference
Data collection and Pre-processing
Model Training and Evaluation
Model Optimization and Conversion
Model Deploy and Inference on devices
CO

Figure 1. Workflow of TinyML architecture.

2) Second phase: Training and testing the deep learning model and evaluating the model's accuracy performance. Subsequently, the model is saved in .pb or .hd5 format.

3) Third phase: Optimizing and conversion use different compression methods such as quantization-aware training (QAT) and post-training quantization (PTQ) (dynamic range quantization [DRQ] or full integer

quantization [FIQ]) to quantize the models. The deep learning models are then converted to TFLite format (.tflite).

4) Fourth phase: The TFLite Interpreter tool is used to evaluate the run time and accuracy of quantized deep learning models after the optimization process is performed on these models.
5) Fifth phase: The model is converted using TensorFlow Lite (TFLite) Micro that converts deep learning models to C array. The model is then deployed to an independent platform that uses C++ language to connect with IoT devices.
6) Sixth phase: The model is run on IoT devices in real time, with action taken based on the data. At the same time, the inference time and the amount of flash memory and RAM consumption are evaluated.

3. Quantization as a standard compression method for TinyML-related applications

Quantization methods are an essential step in compressing deep learning models to meet the constraints of devices[14]. These methods reduce the number of bits in the weight of models, thus providing a significant decrease in the memory footprint. The architecture of deep learning models consists of weight, bias, and activation functions. When the neurons receive input data, the data undergo a multi-plication procedure involving their weight value; the result is thus passed to the next layer. The weight assigns significance to the input data, which in turn affects the output. The format of the weight determines the type of arithmetic operations needed for the neuron multiplication process. Thus, this strategy raises the overall computing device's performance, computational capacity, leading to other complications, such as usage of memory, rise in energy consumption, and a high level of latency[15].

The core idea of quantization is to replace the high-precision parameters of deep learning models with low-precision parameters which, by default, are 32-bit floating-point numbers, without changing the models' architecture, as illustrated in **Figure 2**[15,16]. Quantization is conducted using Equation (1) which represents the uniform mapping of real values to integer values:

$$\mathrm{Quant}(\mathrm{R}) = \mathrm{Int}\ (\mathrm{R}/\mathrm{S}) + \mathrm{Z} \tag{1}$$

where R is a collection of real number values, S denotes a scaling factor, and Z denotes an integer zero point. In general, Int converts the output result to an integer number by reducing to the closest integer value. S is scaling factor which is a positive integer that locates the size of the quantization step.

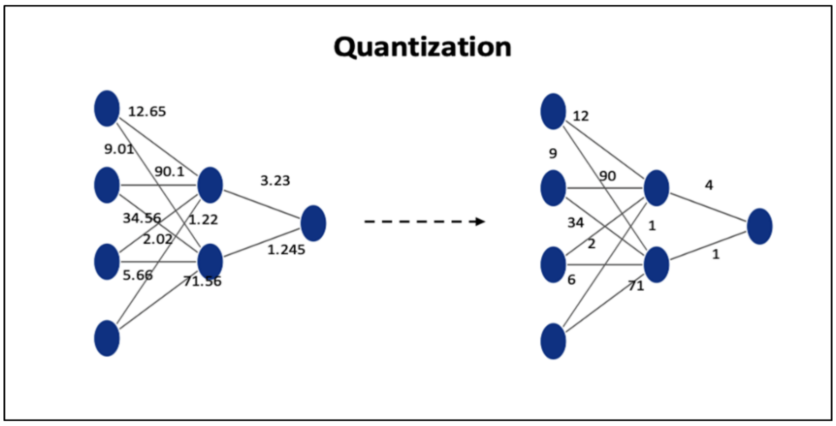

Figure 2. Quantization from floating-point numbers to int-8 integer.

Based on the factor of scaling, the set of real number values R is split into multiple stages, with each stage there is only one integer value mapped by the real number. Equation (2) calculates the scaling factor S in:

$$S = \frac{R_{max} - R_{min}}{(2^B - 1) - 0} \tag{2}$$

where R_{max} and R_{min} symbolize the clipping range, and B represents the quantization bit width[16]. The results of quantization can translate into several benefits. The firstly benefit of quantization is that smaller size models are obtained, with models with less volume occupying less storage space on devices. Furthermore, faster inference occurs, wherein these models with less volume require less time and bandwidth when deployed on devices. In addition, models with less volume use less memory, thus they use less RAM when they are run, freeing up memory for other uses by applications, with this translating into better performance and more stability. The second benefit of quantization is latency decrease, latency, also known as the duration of time required for the models to be executed. Quantization methods can reduce the amount of computation required to run inference using a model. The third benefit is accelerator compatibility through which various hardware acceleration devices, such as the Edge TPU, are capable of running inference at very high speeds when combined with models that have been appropriately optimized[17].

The two methods used in the current study to obtain the quantized weights of the deep learning models are quantization-aware training (QAT) and post-training quantization (PTQ), with their details described below.

3.1. Quantization-aware training (QAT)

Quantization-aware training (QAT) is an innovation produced by Skirmantas Kligys from the Google Mobile Vision team. This innovation is a type of quantization method that aims to compensate for the quantization error by training deep neural models using the quantized version during the forward pass. This should help to mitigate, to some extent, the drop in accuracy. The training of deep learning models still depends on non-quantized values or floating-point values. Thus, to obtain better results, on average, for the quantized model and to stabilize the learning phase, the deep learning models can be pre-trained using floating-point values to initialize the parameters to reasonable values. As shown in **Figure 3**, the input, bias, and weights for each layer are quantized before performing the layer's computation. The output of the layer is quantized immediately following the computation, before it is sent on to the subsequent layer.

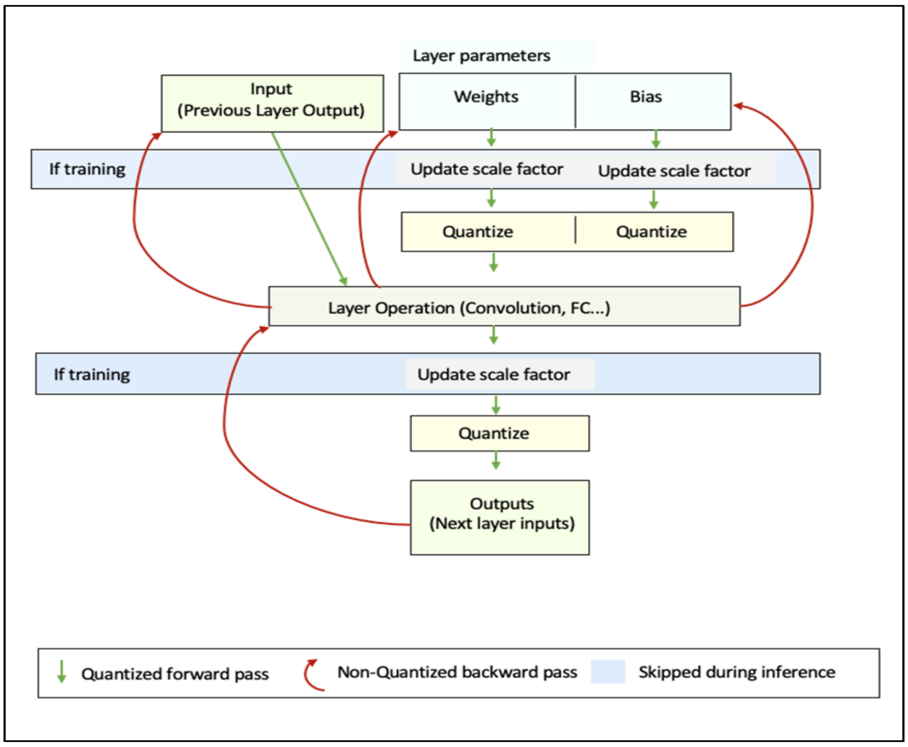

Figure 3. Quantization-aware training.

The quantization process is done through converting the floating-point number to a fixed-point number by defining the factor of the scale. This indicates that the value stored in the floating-point format can be expressed as an integer that has been multiplied by a factor of scaling. The factor of scale is either a negative or positive power of two and its value may be determined only by shifting either to the left or to the right. During the training phase, the range of values is reassessed separately to correct the factor of scale before performing the computation of the layers. In the inference phase, only the factor of scale is frozen[18]. The precision of smaller numbers will suffer as a result of the scaling factor being selected to represent the full range of values without overflow.

3.2. Post-training quantization (PTQ)

Post-training quantization (PTQ) is processed to quantize deep learning parameters after entirely training deep learning models. Once the training of deep learning models is completed, the parameters are frozen and then quantized. In other words, the weights of the parameters are quantized without any adjustments to the training parameters[19]. The quantization process causes an error to be introduced into each parameter's quantization, which in turn causes an error to be inserted into the activations. As the weight values decrease, this produces an increase in the quantization error, thus leading to a drop in the resulting accuracy. The disadvantage of PTQ is the buildup of quantized errors at the final result of the system which can cause the classifier to incorrectly predict the class of the input data, leading to an accuracy drop in comparison to the training model's accuracy. PTQ achieved a size reduction of approximately four times the deep learning model's size which basically comprised convolutional layers, accruing a faster executive of 10–50%[3]. In the study of Mohan et al.[19], they used the PTQ method to quantize the CNN model for inference in the constrained-resources device. The size of the CNN, as a training model, was 1.5 MB; after quantization, the CNN model's size using the PTQ was 138 KB. The inference speed was 30 fps with 99.83% accuracy[20]. Due to the reduction in the model's size, memory and computational requirements were also reduced which, in turn, reduced power consumption[21].

PTQ has three methods, namely, Float16 quantization, dynamic range quantization (DRQ) and full integer quantization (FIQ), with only the latter two, DRQ and FIQ, used in the current study. Due to these methods, the models were scaled four times smaller, while the 16-bit floating point was scaled twice as small:

3.2.1. Full integer quantization (FIQ):

This PTQ method quantizes the weights and activation output of a deep learning model from a 32-bit floating-point number to a full 8-bit fixed-point integer number. The FIQ method is valuable for increasing the speed of inference in lower-power IoT devices. It requires accelerators, and specifically integer-only accelerators, such as the Edge TPU[22].

In FIQ, all the weights and activations of the deep learning models are converted from 32-bit floating-point numbers to 8-bit integer numbers. Despite the advantages of this quantization method, it causes a reduction in accuracy, owing to the way that it quantizes the model. Opportunely, even though accuracy is reduced, it was within an acceptable range. Weights and activations in FIQ are quantized by scaling them over the range of the 8-bit integers. The FIQ method quantizes weights using symmetric quantization; for activations, asymmetric quantization is used. Symmetric quantization sets the values for parameter boundaries to an equivalent range (from −1 to 1) and convert them to a range of −127 to 127, as illustrated in **Figure 4**.

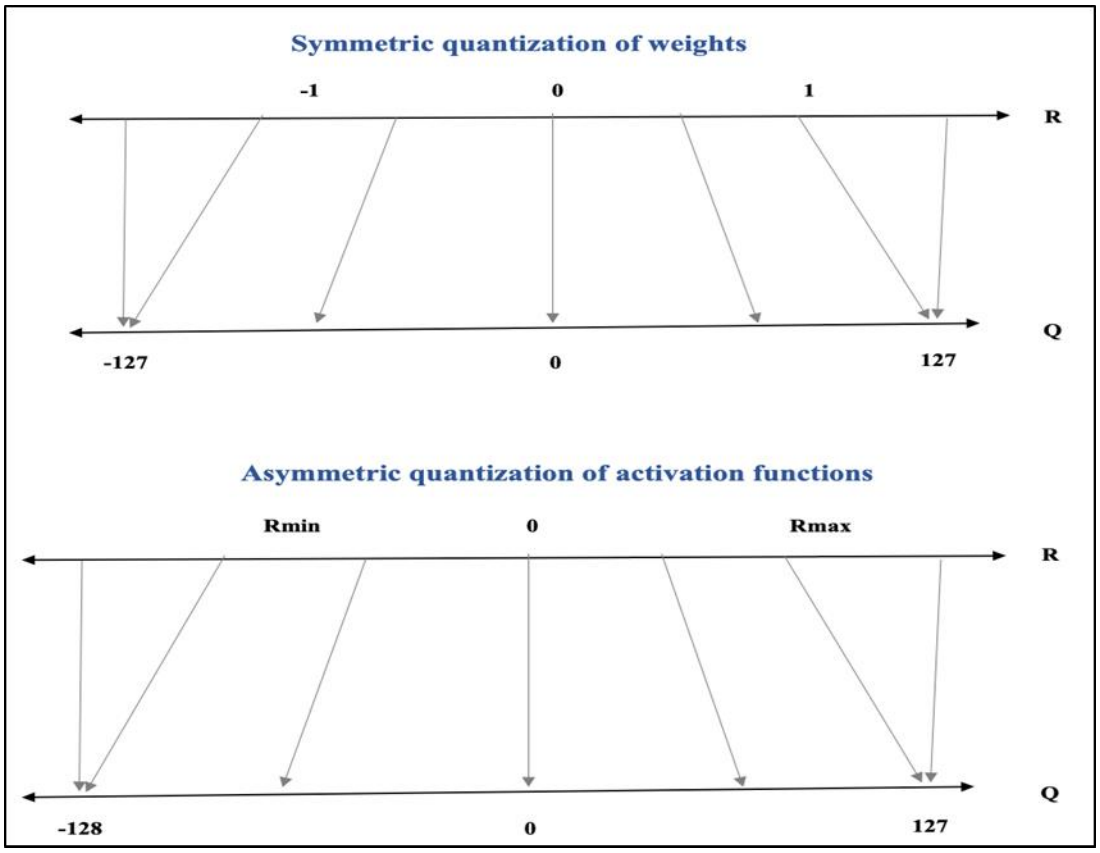
Symmetric quantization of weights
-1
0
1
R
-127
0
127
Q
Asymmetric quantization of activation functions
Rmin
0
Rmax
R
-128
0
127
Q

Figure 4. Quantization of weights (on top) and activations (below) which are symmetric and asymmetric.

The values of the parameter boundaries are called "clipping ranges," and their set-up procedures are called "calibration." If the values of the parameter boundaries are out of range, the values are clipped to the nearest value. Furthermore, the zero-point value for symmetric quantization is equal to 0 ($Z = 0$) as described in Equation (3):

$$\mathrm{Quant}(R) = \mathrm{Int}\,(R/S) \tag{3}$$

For this reason, symmetric quantization is more efficient and computationally less expensive. Moreover, because of this, weight values typically employ symmetric quantization. With the asymmetric quantization method implemented for values of the activation, the range of clipping is not equivalent ($R_{min} = -0.5$ and $R_{max} = 1.5$). At the same time, asymmetric quantization is utilized for activations and is implemented in the range of −128 to 127 which, in turn, provides better accuracy[23].

3.2.2. Dynamic range quantization (DRQ):

This type of PTQ entails quantizing all the weights and activation of deep learning models from a floating-point number to a fixed-point integer number after the models are trained. The DRQ method is used as the default for PTQ to reduce the size of models and to optimize latency in inference[17].

To increase the reduction of inference latency within the "dynamic range," operators quantize activations dynamically dependent on their domain, to 8-bit integers and implement the computations with 8-bit integer weights and activations. After the accumulation and the multiplication processes, the activation values are dequantized to 32-bit floating-point numbers. The DRQ method symmetrically quantizes weight values based on Equation (3), while it asymmetrically quantizes activation values according to Equation (1) above[23]. The dequantization process of activation from integer values to real values is shown below in Equation (4):

$$R = S\,(\mathrm{Quant}(R) - Z) \tag{4}$$

where Quant(R) could represent any positive integer. It can be concluded from Formula (4) that the original value of the dequantized real number cannot be recovered after dequantization. For this reason, DRQ may suffer a reduction in accuracy.

The main difference between FIQ and DRQ is that DRQ converts activation values to the integer format "on-the-fly" through the time of the inference. This considers the advantages of DRQ compared to the full integer as DRQ does not require any representative dataset for quantization. However, as DRQ during the "stand-by" period stores the values of activations as floating-point numbers, it is impossible to run Edge TPU as the quantized model custom hardware as it only supports integer arithmetic operations[23], as illustrated in **Figure 5**.

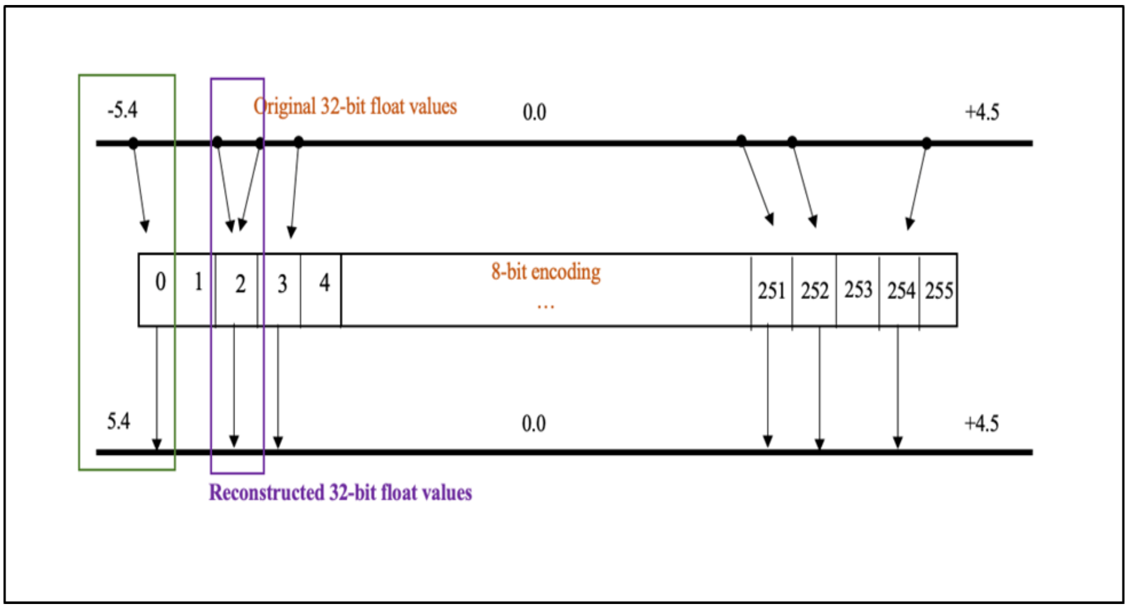
-5.4
Original 32-bit float values
0.0
+4.5
0
1
2
3
4
8-bit encoding
...
251
252
253
254
255
5.4
0.0
+4.5
Reconstructed 32-bit float values

Figure 5. DRQ process of quantization of 32-bit to 8-bit and dequantization.

4. TinyML-based solutions to improve the role of IoT devices in Smart Cities

Today, most smart city applications integrate deep learning with IoT devices and achieve remarkable performance in many fields, such as the smart environment, smart transportation, smart agriculture, and the smart home in smart cities. However, this leads to a number of challenges identified in the sur-vey and review studies. TinyML is a new technology that aims to integrate deep learning with IoT devices in various applications. It optimizes various deep learning models to introduce better performance and accuracy within the constrained resources of IoT devices. The TinyML system accommodates these requirements and provides a solution for integrating deep learning with IoT devices in smart cities. This is achieved by providing lightweight training of deep learning models, thus enabling inference of deep learning models in IoT devices with restricted resources and low power consumption, as further dis-cussed in the following subsections.

4.1. Lightweight training of deep learning models

The smart city applications have become complex, thus requiring complex deep learning models to achieve good performance. The size of the training of deep learning models is increasing and requiring powerful graphic cards, such as GPUs, and several hours of time in the effort to produce highly accurate results. This requires a high level of storage memory in IoT devices which, in turn, consumes power. IoT devices should be sufficiently powerful and need many resources to accurately perform certain tasks. TinyML has paved the way for IoT devices to be intelligent despite their constrained resources, by seeking to reduce the size of the training of deep learning models to enable deployment on IoT devices, such as a microcontroller. Using model compression methods, such as pruning and quantization (post-training quantization [PTQ], quantization-aware training [QAT]). This develops a lightweight training model to deploy in IoT devices, thus extending the battery life and saving costs of operating these devices which are fundamental to IOT use in smart cities[24,25].

4.2. Inference of lightweight deep learning models in IoT devices

Processing data in TinyML happens locally, on the device where the calculations are being done. Therefore, there is no latency experienced by IoT devices because data is processed locally in real time, allowing for instantaneous response and analysis, even in critical situations. This also reduces the cloud load[13,26]. In addition, IoT devices need gateways and cloud services for data intake, as well as for running deep learning models for data processing. Due to the large model size, the inference of deep learning in devices causes latency and response delays, and takes a long time. TinyML overcomes problems associated with response delays, consumption of network bandwidth, and storage problems through inference of the deep learning model and data processing on the device itself. Thus, bringing raw data to and processing raw data on the device lead to reduced latency and increased savings in terms of connection bandwidth[6,27].

4.3. Low power consumption

Low power consumption is a great benefit when adopting TinyML, as IoT devices working on microcontroller units (MCUs) rely on small batteries and consume a low amount of energy. Due to their powerful processors and GPUs, IoT devices generally use a large amount of resources including power. However, TinyML consumes 150 μW to 23.5 mW of energy enables IoT devices to be placed everywhere, thus being converted into connected smart entities, for example, scooters, Segway, and other personal mobility devices[12]. Thus, smart cities can now benefit from new behavioral wandering applications.

4.4. Inference of deep learning models in restricted resources of IoT devices

The significant goal of TinyML is to enable IoT devices with constrained resources to be intelligent. TinyML overcomes some challenges encountered by IoT devices in smart cities which can be categorized as follows: firstly, smart city applications use sensors and microcontrollers to detect raw data and to perform processing of deep learning on remote locations, such as the cloud, and for data storage. When cloud computing processes large deep learning models ranging from 16 GB to 32 GB with ultra-fast processing using a powerful computational system, such as TFLOPs, with hardware including GPUs, tensor processing units (TPUs), etc., this leads to latency of the data and consumption of network band-width. Secondly, in smart city applications, the inference of deep learning within devices with GPUs, and microprocessors enables deep learning models to be deployed to process data in real time without depending on computing power as would be the case with the cloud. For instance, mobile devices, such as laptops, tablets, and smartphones, have large storage and computation capacity, and are sufficiently powerful for deployment on devices using, neural processing unit (NPU) hardware, and 8 RAM of memory. Shifting smart city applications to mobile devices requires a high level of computational power with large storage memory, as well as requiring daily battery charging and consumption[28,29]. TinyML can process data on the device using deep learning: it is low cost and uses limited resources, with a microcontroller which has 2MB of RAM. Thus, TinyML enables innovation in many fields in smart cities. **Figure 6** presents the comparison of deploying deep learning on the cloud, on a mobile, and on a microcontroller.

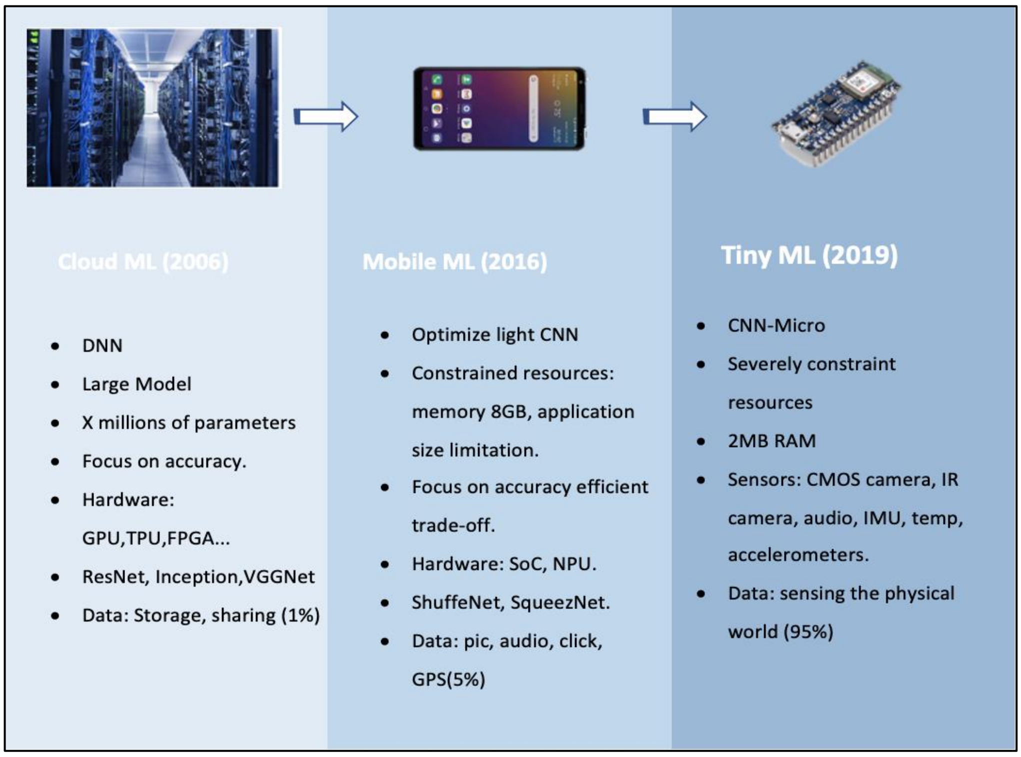
Cloud ML (2006)
DNN
Large Model
X millions of parameters
Focus on accuracy.
Hardware: GPU,TPU,FPGA...
ResNet, Inception,VGGNet
Data: Storage, sharing (1%)
Mobile ML (2016)
Optimize light CNN
Constrained resources: memory 8GB, application size limitation.
Focus on accuracy efficient trade-off.
Hardware: SoC, NPU.
ShuffeNet, SqueezNet.
Data: pic, audio, click, GPS(5%)
Tiny ML (2019)
CNN-Micro
Severely constraint resources
2MB RAM
Sensors: CMOS camera, IR camera, audio, IMU, temp, accelerometers.
Data: sensing the physical world (95%)

Figure 6. Comparison of deployment of deep learning on cloud, mobile, and microcontroller.

5. TinyML case studies in smart cities

Several scenarios from different fields are shown below as examples of how TinyML research has been put to use. To the best of the author's knowledge, few studies have investigated TinyML as it is an emerging field. Research efforts have centered on optimizing deep learning models for usage on devices boasting both high precision and high processing power. The subsequent paragraphs cover state-of-the-art TinyML studies that have used images in their experiments with sign language detection, handwritten recognition, medical face mask detection, autonomous mini vehicles, and the environment.

As shown in one use case[30], TinyML can be utilized in sign language detection. To aid the deaf-mute in communicating with others, a deep learning model was developed to identify the sign language alphabet on low-resolution edge devices. In the study of Chen et al.[25], they suggested a model for recognizing the alphabet of American Sign Language (ASL) and translating it into text and audio for use on portable, wearable Internet of Things gadgets in real time. The device used the smallest and least expensive microcontrollers available. The authors themselves built the third dataset using an OpenMV H7 camera and includes photos of the sign language alphabet. Two augmentation methods were applied to the suggested CNN model. Firstly, simple augmentation methods, such as rotation, flip, etc., were used; secondly, OpenCV made use of five different interpolation methods: INTER_NEAREST, INTER_LINEAR, IN-TER_AREA, INTER_CUBIC, and INTER_LANCZOS4. Success rates of 98.53 and 99.02 percent were recorded. To accommodate the limited memory of the OpenMV H7 board's microcontroller STM32H743VI, the authors used TFLite to reduce the model's 32-bit floating-point values to 8-bit integers. The model with five interpolation approaches outperformed the other after both were installed on the device, with an accuracy of 98.84% and a frame rate of 20 fps, respectively, compared to 95.24% for the model with simple augmentation techniques.

TinyML's low performance paves the path for the Internet of Things and makes it possible to quickly and simply train and implement a model that can identify handwriting on a microcontroller. In the study of Merenda et al.[26], they developed various models for TinyML application. The research used the X-CUBE-AI platform to test the STMicroelectronics NUCLEO-F746ZG board on the MNIST dataset. Using the TensorFlow library, the NN algorithm was developed with two hidden layers between the input and output, and then the CNN was constructed using the Keras library for the training phase. With ReLU and Sigmoid activation functions, the CNN raised the training accuracy from 97.25 percent to 99 percent. About 15 MB was needed for the CNN model, but only 7,172 KB was needed for the Keras model. The Keras model was converted using TFLite and TFLiteConverter and then progressively deployed on real embedded devices. In order to fit inside the memory constraints of embedded devices, these tools shrank the model down to 2.4 MB. Pruning and post-training quantization were used to improve the model's inference speed and decrease the energy it consumed, with only a little hit to accuracy. Energy and computing needs were reduced by post-training quantization. The prediction model achieved a perfect score, with a runtime of 330 milliseconds and a memory and storage footprint of just 135.68 and 668.97 bytes, respectively.

With the COVID-19 pandemic, widespread caution was imposed, with cough-related healthcare face mask recognition becoming a critical responsibility. TinyML could aid in the security and battery life of IoT-based smart health. In in the study of Mohan et al.[19], they investigated a real-time clinical face mask detection using a miniature convolutional neural network model. This was done without sending images to the server, keeping user privacy and security in mind, and the findings were sent back into the program. The model was trained and evaluated using four datasets and several augmentation methods. Notably, the third dataset was produced by the authors utilizing an OpenMV camera to find the categorization metrics. STMicroelectronics STM32H743VI, found inside the OpenMV Cam H7 housing, was used alongside a CNN model for the research. In addition, a modified SqueezeNet model was used to evaluate the suggested model's performance. All three models were quantized to minimize their size, but only the proposed model met the requirements for deployment on the device: it had a model size of 138 KB, an inference speed of 30 fps, and an accuracy of 99.83%.

Many different types of "smart" industries, "smart" environments, "smart" monitoring systems, etc. all make use of autonomous mini-vehicles. TinyML boosts their efficiency by facilitating the learning of complicated actions with minimal iteration and power usage. In their study, de Prado et al.[31] facilitated the execution of deep learning on low-power autonomous driving cars with the purpose of boosting performance (e.g., of actions/s) by learning complex challenges. As a result, the vehicle would be able to make decisions (picture categorization) in a high-noise environment while using little resources. The authors made use of a computer vision algorithm (CVA) to replace a CNN trained with a LetNet5 model, which predicted success exclusively in constant lighting conditions. After tweaking the LetNet5, researchers built a new network family they dubbed "vehicle neural networks" (VNNs). The researchers developed three datasets (named Dset-2.0, Dset-1.5, and Dset-1.0) comprising training sets of 1,000 photos and test sets of 300 images for each class. Following VNN training on Dset-All, the VNN model's weights and activations were post-quantized to 8-bit fixed-point integers to lessen the load on the system's memory and battery life. To test their VNNs, the authors used hardware and software from a variety of manufacturers and architectures, including the GAP8 (GAP8, a parallel ultra-low-power RISC-V SoC), STM32L4 (Cortex-M4), and NXP k64f (Cortex-M4). The VNN3, VNN4, and LeNet5 models employing GAP8 showed a 98.74% accuracy with the Dset-All (I1, I2, and I3) reinforced dataset.

There have been multiple attempts by writers to use deep learning in small devices for improved prediction accuracy. However, difficulties have arisen because there is no universal foundation for integrating TinyML into devices from different manufacturers. Gorospe et al.[32] developed a general environment for implementing the model of deep learning into different small devices. With the goal of enabling deep learning inference on various edge devices from various organizations like ARM and NXP, the environment consists of TFLite and Mbed OS software. In addition, the MobileNet collection (V1, V2, and V3) was used to compare the edge device's and PC's performance on a single instance of human detection. VWW was used to evaluate the model, with 115,287 photos used for training and 100 images used for testing (50 with people and 50 without). Both the STM32H747I-Disco (STM) and the OpenMV Cam H7 were used in the authors' three-stage experimentation. The first stage checked if the suggested environment could support deep learning device integration. As a result, MobileNet-V2, running on STM, was able to achieve an accuracy of 88% and a latency of just 220 ms. In the second stage, we wanted to assess and contrast the efficacy of various models on both edge devices and a PC. The comparison findings showed that the STM and PC implementations of the MobileNet-V2 model (as well as depth multiplier at 0.1 with resolution at 96 96) achieved the maximum degree of accuracy (88%) and model consumption (213,184 KB matrix size and 138,240 KB RAM). The purpose of this experiment was to use STM to examine MobileNet-V2's behavior before, during, and after quantization to determine the impacts of the application. The results demonstrated that the accuracy of the model might be improved by applying quantization after training.

In our recent paper[33], state-of-the-art driver drowsiness detection research using five deep learning models yielded great accuracy. MobileNet-V2, SqueezeNet, AlexNet, and MobileNet-V3 deep models identified driver sleepiness state with 0.9960, 0.9947, 0.9911, and 0.9832 accuracies, respectively. Deep learning models perform best with SSD pre-processing. The CNN deep model exceeded other TinyML-related research in model size with 0.05 MB. The redesigned SqueezeNet architecture was 0.141 MB less than the original. The pre-trained MobileNet-V2 and MobileNet-V3 models reached 1.55 MB and 1.165 MB, respectively, followed by the modified AlexNet model at 0.58 MB. DRQ reduced accuracy by 1%. QAT and FIQ followed DRQ in optimizing performance. After optimization, accuracy was not degraded by more than 1%. The experiments show that it can function successfully on resource-constrained IoT devices like the microcontroller. The OpenMV H7 board, STM32H743VI, and STM32 Nucleo-144 H743ZI2. The SparkEdge development board Apollo3 Blue, with 1 MB of Flash memory and 384 KB of RAM, can also run SqueezeNet, AlexNet, and CNN models. The CNN may also be used on an Arduino Nano Ple33 board with 256 KB flash and 32 KB RAM. **Table 1** summarizes these TinyML case studies.

Table 1. Case studies of TinyML.

Study	Model	Platform	Model size	Accuracy	Device name	Platform after inference	Accuracy after inference
Paul et al.[30]	CNN1 CNN2	TFLite	185KB	98.53% 99.02%	OpenMV H7 board STM32H743VI	TF-Converter	95.28% 98.84%
Merenda et al.[26]	NN	TFLite and TFLiteConver	15MB	97.25%	F746ZG	X-CUBE-AI tool	100%
	CNN		7172KB	99%	-	-	-
Mohan et al.[19]	CNN	OpenMV	1.5MB	99.83%	OpenMV H7 board STM32H743VI	-	99.83%
	SqueezeNet		8.0MB	98.50%			98.53%
	SqueezeNet2		3.8MB	98.93%			98.99%
Prado et al.[31]	LetNet5 VNN1 VNN2	PyTorch	-	99.53% 79.62% 81.27%	STM32 L476 board NXP k64f GAP8	X-Cube-AI ARM CMSIS-NN PULP-NN	- - -
Gorospe et al.[32]	MobileNet-V2	TensorFlow	Matrix size: 611,912	-	STM32H747I-Disco	TFLite and MbedOS	88%
Alajlan and Ibrahim[33]	SqueezeNet	TFLite	0.141 MB	-	-	-	99.6%

6. Conclusion

Within the scope of this article, we discuss TinyML in general terms and highlight some of its benefits and architecture. Then, we look into quantization as a common compression technique for TinyML-related software. Quantization-aware training (QAT) and post-training quantization (PTQ) are two strategies used to get the quantized weights of the deep learning models; we detailed them in depth. Next, we highlight TinyML-based solutions to enhance the function of IoT devices in Smart Cities, such as light-weight deep learning model training and inference, low power consumption, and inference within the constrained resources of IoT devices. Finally, we provide examples of how TinyML research has been put to use in the real world, covering a wide range of domains. Since TinyML is still a developing area, the author is only aware of a handful of research projects devoted to it.

Author contributions

Conceptualization, NNA and DMI; methodology, NNA and DMI; validation, NNA and DMI; formal analysis, NNA and DMI; investigation, NNA and DMI; resources, NNA and DMI; data curation, NNA and DMI; writing—original draft preparation, NNA; writing—review and editing, DMI; visualization, NNA and DMI; supervision, DMI. All authors have read and agreed to the published version of the manuscript.

Funding

The authors gratefully acknowledge Qassim University, represented by the Deanship of "Scientific Research, on the financial support for this research under the number (COC-2022-1-2-J-29832) during the academic year 1444 AH/2022 AD".

Acknowledgments

The authors gratefully acknowledge Qassim University, represented by the Deanship of "Scientific Research, on the financial support for this research under the number (COC-2022-1-2-J-29832) during the academic year 1444 AH/2022 AD".

Conflict of interest

The authors declare no conflict of interest.

References

1. Banbury CR, Reddi VJ, Lam M, et al. Benchmarking tinyml systems: Challenges and direction. arXiv 2020; arXiv:2003.04821. doi: 10.48550/arXiv.2003.04821
2. Lin J, Chen WM, Lin Y, et. al. Mcunet: Tiny deep learning on iot devices. Advances in Neural Information Processing Systems. 2020; 33: 11711-11722.
3. Warden P, Situnayake D. Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers. O'Reilly Media; 2019.
4. Ray PP. A review on TinyML: State-of-the-art and prospects. Journal of King Saud University-Computer and Information Sciences. 2022; 34(4): 1595-1623. doi: 10.1016/j.jksuci.2021.11.019
5. Estrebou CA, Fleming M, Saavedra MD, et al. Lightweight convolutional neural networks framework for really small tinyml devices. Communications in Computer and Information Science 2020; 1154: 29-39. doi: 10.1007/978-3-030-99170-8
6. Alajlan NN, Ibrahim DM. TinyML: Enabling of inference deep learning models on ultra-low-power IoT edge devices for AI applications. Micromachines. 2022; 13(6): 851. doi: 10.3390/mi13060851
7. TensorFlow Lite, TensorFlow, 2021. Available online: https://www.tensorflow.org/lite (accessed on 10 October 2023).
8. Dennis DK. Edgeml: machine learning for resource-constrained edge devices (2020). Available online: https://github.com/Microsoft/EdgeML (accessed on 10 October 2023).
9. Suda N, Loh D. Machine learning on arm cortex-m microcontrollers. Arm Ltd.: Cambridge, UK. 2019. Available online: https://www.arm.com/resources/guide/machine-learning-on-cortex-m (accessed on 10 October 2023).
10. X-CUBE-AI—AI Expansion Pack for STM32CubeMX—STMicroelectronics, 2021. Available online: https://www.st.com/en/embedded-software/x-cube-ai.html (accessed on 10 October 2023).
11. Sakr F, Bellotti F, Berta R, De Gloria A. Machine learning on mainstream microcontrollers. Sensors. 2020; 20(9): 2638. doi: 10.3390/s20092638
12. Sanchez-Iborra R, Skarmeta AF. Tinyml-enabled frugal smart objects: Challenges and opportunities. IEEE

Circuits and Systems Magazine. 2020; 20(3): 4-18. doi: 10.1109/MCAS.2020.3005467
13. Puthal D, Mohanty SP, Wilson S, Choppali U. Collaborative edge computing for smart villages [energy and security]. IEEE Consumer Electronics Magazine. 2021; 10(3): 68-71. doi: 10.1109/MCE.2021.3051813
14. Raza W, Osman A, Ferrini F, Natale FD. Energy-efficient inference on the edge exploiting TinyML capabilities for UAVs. Drones. 2021; 5(4): 127. doi: 10.3390/drones
15. Huang Q. Weight-quantized squeezenet for resource-constrained robot vacuums for indoor obstacle classification. AI. 2022; 3(1): 180-193. doi: 10.3390/ai3010011
16. Model optimization TensorFlow Lite. Available online: https://www.tensorflow.org/lite/performance/model_optimization (accessed on 10 October 2023).
17. TensorFlow Model Optimization. Available online: https://www.tensorflow.org/model_optimization (accessed on 10 October 2023).
18. Novac PE, Boukli Hacene G, Pegatoquet A, et al. Quantization and deployment of deep neural networks on microcontrollers. Sensors. 2021; 21(9): 2984. doi: 10.3390/s21092984
19. Mohan P, Paul AJ, Chirania A. A tiny CNN architecture for medical face mask detection for resource-constrained endpoints. In: Innovations in Electrical and Electronic Engineering: Proceedings of ICEEE 2021 2021 May 25 (pp. 657-670). Springer Singapore. doi: 10.1007/978-981-16-0749-3_52
20. Introducing the Model Optimization Toolkit for Tensor Flow TensorFlow, 2018. Available online: https://medium.com/tensorflow/introducing-the-model-optimization-toolkit-for-tensorflow-254aca1ba0a3 (accessed on 10 October 2023).
21. Krishnamoorthi R. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv 2018. arXiv:1806.08342.
22. Post Training Quantization. Available online: https://www.tensorflow.org/lite/performance/post_training_quantization (accessed on 10 October 2023).
23. Garifulla M, Shin J, Kim C, et al. A case study of quantizing convolutional neural networks for fast disease diagnosis on portable medical devices. Sensors. 2021; 22(1): 219. doi: 10.3390/s22010219
24. Signoretti G, Silva M, Andrade P, et al. An evolving tinyml compression algorithm for iot environments based on data eccentricity. Sensors. 2021; 21(12): 4153. doi: 10.3390/s21124153
25. Chen Y, Zheng B, Zhang Z, et al. Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions. ACM Computing Surveys (CSUR). 2020; 53(4): 1-37. doi: 10.1145/3398209
26. Merenda M, Porcaro C, Iero D. Edge machine learning for ai-enabled iot devices: A review. Sensors. 2020; 20(9): 2533. doi: 10.3390/s20092533
27. Wang F, Zhang M, Wang X, et al. Deep learning for edge computing applications: A state-of-the-art survey. IEEE Access. 2020; 8: 58322-58336. doi: 10.1109/ACCESS.2020.2982411
28. Soro S. TinyML for ubiquitous edge AI. arXiv 2021. arXiv:2102.01255.
29. David R, Duke J, Jain A, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. Proceedings of Machine Learning and Systems. 2021; 3: 800-811.
30. Paul AJ, Mohan P, Sehgal S. Rethinking generalization in american sign language prediction for edge devices with extremely low memory footprint. In: 2020 IEEE Recent Advances in Intelligent Computational Systems (RAICS); 3 December 2020; pp. 147-152. IEEE.
31. de Prado M, Rusci M, Capotondi A, et al. Robustifying the deployment of tinyml models for autonomous mini-vehicles. Sensors. 2021; 21(4): 1339. doi: 10.3390/s21041339
32. Gorospe J, Mulero R, Arbelaitz O, et al. A generalization performance study using deep learning networks in embedded systems. Sensors. 2021; 21(4): 1031. doi: 10.3390/s21041031
33. Alajlan NN, Ibrahim DM. DDD. TinyML: A TinyML-Based Driver Drowsiness Detection Model Using Deep Learning. Sensors. 2023; 23(12): 5696. doi: 10.3390/s23125696