Dynamic AI-IoT: Enabling Updatable AI Models in Ultra-Low-Power 5G IoT Devices

Mohammad AlSelek, Jose M. Alcaraz-Calero and Qi Wang

***Abstract*—This paper addresses the challenge of integrating dynamic AI capabilities into Ultra-Low-Power (ULP) IoT devices, a critical necessity in the rapidly evolving landscape of 5G and potential 6G technologies. We introduce the Dynamic AI-IoT architecture, a novel framework designed to eliminate the need for cumbersome firmware updates. This architecture leverages Narrowband IoT (NB-IoT) to facilitate smooth cloud interactions and incorporates tailored firmware extensions for enabling dynamic interactions with Tiny Machine Learning (TinyML) models. A sophisticated memory management mechanism, grounded in memory alignment and dynamic AI operations resolution, is introduced to efficiently handle AI tasks. Empirical experiments demonstrate the feasibility of implementing a Dynamic AI-IoT system using ULP IoT devices on a 5G testbed. The results show model updates taking less than one second and an average inference time of approximately 46 ms.**

***Index Terms*—Artificial Intelligence, Internet of Things, 5G, NB-IoT, TensorFlow, ESP32, Fipy, Pysense, Micropython.**

I. Introduction

THE combination of 5G, IoT and AI technologies creates limitless possibilities for novel and future-proof communication systems. IoT devices are key sources of data in 5G and beyond systems, while AI goes hand in hand with IoT for smart pervasive services. When coupled together, AI and IoT reshape the digital transformation mechanisms. Making ultra-low-power 5G-enabled IoT devices smart has a significant impact in terms of a wide range of use cases such as improving prediction and automating new processes. Moreover, dealing with low-power and ULP devices means that we can maintain the power consumption at a lower level, resulting in more sustainable end products and/or applications.

The 5G architecture embraces IoT, allowing diverse vertical IoT applications, such as Industrial IoT (IIoT), Smart Agriculture, and Smart Cities, to be instantiated across a shared physical network and cloudified 5G infrastructure [1]. Such infrastructure allows AI models to be dynamically loaded and executed by low-cost, low-power, 5G-enabled IoT devices, and thus the technological game changes dramatically.

However, many current ultra-low-power IoT devices lack sufficient support for training and executing AI models. When such support is available, it is often fixed, making these devices vulnerable to obsolescence and inflexibility over time. Specifically, we are referring to real ultra-low-powered tiny IoT devices (less than 250mA) with limited resources, unable to accommodate an embedded Linux-based system. Instead, they utilize a tiny mini-OS embedded in an 8Mb-32Mb memory size. Updating these devices necessitates re-flashing the firmware, creating challenges for many use cases. This requirement demands either in-situ re-flashing or remote re-flashing via Over-the-air (OTA) technology [2]. Such a requirement presents significant challenges in various IoT scenarios, exemplified by scenarios where a timely and smooth update process is crucial, like in industrial automation or healthcare applications.

In response to the above gaps in the state of the art, this research has focused on allowing the execution and dynamic loading/updating of AI models into low-power IoT devices. The proposed new enabling architecture, referred to as Dynamic AI-IoT, will be based on an ESP32 microprocessor using Micropython as a scripting language to provide dynamic AI capabilities of the AI models.

The proposed system offers several advantages and contributions, which include:

- First, Dynamic AI-IoT enables the execution of AI models into ULP IoT devices, thereby removing a major barrier in today's AI-empowered IoT use cases.
- Second, Dynamic AI-IoT enables the dynamic updating and re-configuration of AI models without the need to re-flashing the IoT devices, enabling further development into more advanced AI approaches such as "online learning" and "federated learning".
- Third, Dynamic AI-IoT enables the dynamic loading of different AI models to achieve a multi-proposed AI-empowered IoT device in 5G systems. Therefore, a single IoT device can be involved in multiple use cases depending on the loaded AI model.
- Forth, Dynamic AI-IoT enables connectivity with various networks such as Lora, LTE-M, or NB-IoT, among others. For example, we use a 5G network as a connection bridge between Mobile Edge Computing where the AI model is generated, and the ULP IoT devices where the transferred AI model is executed.
- Fifth, Dynamic AI-IoT has been prototyped to achieve a fully functional system. The system has been tested and empirically validated against two different AI use cases: hand gesture recognition and sine wave recognition.

The rest of the paper is laid out as follows. Section II presents the state-of-the-art. Section III outlines the proposed architecture of the dynamic AI-IoT system. Section IV explores the system's process and diagrams. Section V details the AI-IoT firmware development. Section VI showcases two

Mohammad AlSelek, Jose M. Alcaraz-Calero and Qi Wang are with the School of Computing, Engineering & Physical Sciences, University of the West of Scotland, Paisley PA1 2BE, United Kingdom.
(e-mail: mohammad.alselek@uws.ac.uk, jose.alcaraz-calero@uws.ac.uk, qi.wang@uws.ac.uk).

TABLE I: Comparing Related Works

	Programmability Support		AI Support						Technology	Connectivity for AI					Power Consumption
	Remote Flashing	Dynamic Re-programming	Training	Execution	Reloading	Multiple AI Models	Dynamic Model Configuration	Dynamic Memory Optimization	Micropython	Wifi	Bluetooth	LTE-M	NB-IoT	Lora/Sigfox	ULP
Shang et al. [3]	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
Zhang et al. [4]	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
Crocioni et al. [5]	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓
Dokic et al. [6]	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Hoang et al. [7]	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓
Spresense [8]	✗	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓
O'Cleirigh [9]	✗	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓
microAI [10]	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Yan [11]	✗	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓
OpenMV [12]	✗	✓	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓
Our Contribution	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

validation use cases. Section VII summarizes experimental outcomes. Section VIII discusses limitations and challenges, and, finally, Section IX concludes the paper and suggests directions for future research.

II. Related Work

Table I presents a comparison of the current state of Ultra-Low-Power (ULP) IoT devices and their AI capabilities, providing a basis for evaluating our contribution. The table enables readers to understand how our work contributes to the existing state of the art.

Recent efforts have been made to implement ML on IoT devices, marking advancements in both hardware and software domains [13]. On the software side, ML libraries like TensorFlow, Scikit-Learn, and PyTorch have been adapted for use on resource-constrained devices [14]. Simultaneously, hardware manufacturers are striving to improve chips by integrating advanced technologies. For instance, the incorporation of a dedicated co-processor, such as a Tensor Processing Unit (TPU), can effectively support the primary computing unit in ML tasks. While this approach enhances computational performance, it is less common due to its significant impact on price, power consumption, and processing platform complexity.

Furthermore, the concept of Edge Intelligence, which represents the convergence of Edge Computing and AI [15], has emerged as a pivotal paradigm for extending AI capabilities to the network's edge. This extension empowers real-time decision-making while simultaneously reducing latency [16]. In the context of these developments, Edge Impulse Studio, a cloud-based platform dedicated to embedded machine learning, has played a vital role in facilitating model training, evaluation, and subsequent deployment on IoT devices at the network's edge [17].

Most IoT devices are designed to use low power, whether they are in idle or active mode. The primary chip in their architecture, referred to as a Micro Controller Unit (MCU) such as ESP32 and STM32, should have the capability to integrate low-power units. Developing applications on MCUs has led to the creation of compatible technologies. For instance, the term TinyML is used instead of ML when implementing AI on MCUs. Some AI frameworks have been adjusted, such as TensorFlow Lite Micro (TFLM), which is a modified version of TensorFlow. Concerning the programming language, Micropython has been developed to provide access to alternative Python libraries for interaction with MCUs.

TFLM simplifies the deployment of TinyML models on hardware with limited resources by prioritizing portability and flexibility [18]. A straightforward approach for creating a TFLM model is to convert an already trained TensorFlow model. The optimized models, supporting various algorithms from the Neural Network (NN) class, can operate on diverse platforms like smartphones, embedded Linux systems, and MCUs. Specifically for MCUs, the optimized code is written in C++ and is compatible with 32-bit processors. It has been successfully applied to devices such as the Arduino Nano, as well as other architectures like the ESP32 and ARM Cortex-M series processors [14].

Currently, there is a growing focus on investments in Low-Power IoT and TinyML. In this context, researchers have made significant progress by considering factors such as chip architecture, programmability support, AI support, programming language, connectivity for AI, and power consumption. These factors are the main elements of our comparison, as illustrated in Table I. To simplify, we have categorized this table into three groups, each representing recent AI-empowered IoT devices. The first group pertains to devices equipped with high computational processors, the second highlights low-power IoT devices, and the third category involves low-power/ULP Micropython-enabled IoT devices. Despite their differences, all these IoT devices share the common feature of efficiently executing AI models.

The authors in [3] and [4] fall into the first group, demonstrating the capability to deploy Deep Learning (DL) inference models directly on IoT end-nodes. Their focus was on reducing the size and compressing high-accuracy DL models to suit the characteristics of the target IoT devices. These devices have resources that enable them to smoothly run large AI models, regardless of their complexity. For instance, in [3], a TensorFlow AI model was deployed on Raspberry Pi 3 Model B+, and in [4], a PyTorch AI model was trained and executed on Raspberry Pi 4 Model B. While these efforts have made it possible to run AI models on IoT devices, real-time processing of large AI models is power-intensive and can consume several Watts. For example, in an empirical test on an RPi 4 with 4 busy cores, it was found to consume around 6 Watts of power.

In [5], [6], [7], and [8], the authors proposed generating pre-trained TinyML models and adapting them to the constraints of MCUs. They explained the tools that typically take inference engines from well-known ML libraries such as TensorFlow, Scikit-Learn, or PyTorch and adapted their code for execution. The results demonstrate the feasibility of running AI models on Low-Power resource-constrained IoT devices.

However, using the "train-then-deploy" approach raises uncertainties about how to manage the complete separation

between the learning/training and runtime inference phases. Consequently, the second group of IoT devices resorts to using static AI models that cannot adapt to newly collected data without on-site data collection, analysis, and manual fine-tuning. If the edge has the necessary data to generate a new AI model, updating the AI model on an MCU requires manual intervention. Although the Spresense board used in [8] supports CircuitPython, providing some dynamic programming capabilities, this CircuitPython board does not support TensorFlow.

The third group aims to address the challenges identified in the second group. Researchers recognized the feasibility of achieving on-device TinyML inference and began exploring ways to dynamically interact with TinyML models. They realized that enabling Micropython in these devices could unlock various capabilities, particularly in supporting AI tasks like dynamic loading and execution of AI models.

Micropython implements a subset of Python functions and class libraries that directly replace Python libraries while considering the limitations of MCUs, such as RAM speed and size, process frequency, and the reduced number of cores. Micropython seamlessly handles TinyML functions, such as returning lists of data structures, iterating through lists, sorting, and filtering. These tasks can be instantly executed on the MCU. In contrast, using other languages requires users to delve deep, ensuring the code is correct, the firmware has been successfully built, and flashing it to the MCU has been executed smoothly.

There is currently no published paper detailing how to extend Micropython to incorporate AI. Nevertheless, some researchers have made progress on the technical aspects of this endeavor. For instance, in [9], [10], and [11], the integration of the TFLM library into Micropython has been implemented. They successfully compiled Micropython firmware, including a pre-trained TinyML model, which can be flashed into IoT Low-Power devices. These researchers drew inspiration from OpenMV projects [12], specifically Micropython-powered TFLM-integrated products [19]. OpenMV has been dedicated to developing machine vision modules for running machine vision algorithms on STM32.

While each of the existing studies offers distinct advantages, none has successfully addressed the challenges associated with the "train-then-deploy" design, resulting in static intelligence models. This static nature often impacts applications, making smart sensor platforms unreliable when deployed in the field. Additionally, the exchange of ML/TinyML models between the Edge and IoT devices remains unimplemented due to the requirement for a high-speed connection. Furthermore, none has developed dynamic AI model configurations, such as tensor arena size and AI operations resolving, leading to limited memory management.

Our contribution aims to provide a dynamic and robust solution to these challenges by leveraging 5G connectivity. This involves using 5G as a bridge between Edge and Low-Powered IoT devices, enabling the deployment of different TinyML models for various scenarios. The solution involves working with our implemented 5G infrastructure alongside our 5G-enabled, TFLM-integrated, and Micropython-powered IoT devices to facilitate dynamic exchanging and updating of AI models. Additionally, 5G offers an environment for seamlessly connecting a massive number of embedded devices by allowing scalability in data rates, power consumption, and mobility.

It is worth mentioning that none of the mentioned research endeavors can dynamically update AI models and optimize memory management with each update. Additionally, none of them have utilized cellular connectivity to enable Edge-to-IoT AI knowledge transfer. These advanced capabilities serve as the primary motivation for our contribution.

III. Proposed AI-IoT System for Dynamic AI Model Updates in Ultra-Low-Power 5G IoT Devices

5G, IoT, and AI can be integrated into end-to-end cloud-based scenarios, enabling rapid, distributed, and intelligent real-time decision-making. With IoT devices linked to the 5G edge, a single IoT device gains the intelligence to make predictions without relying on a central computing unit. Implementing AI technologies on IoT devices not only reduces latency but also enhances link capacity and network security [20].

The acceleration in the development of cellular IoT devices necessitates expandable and power-efficient communication technology. While 4G can connect around two thousand devices, 5G technology can support connectivity for up to one million devices over 1 Km^2. Managing such density without proper oversight could significantly drain device batteries. To address this, Narrowband IoT (NB-IoT), a cellular low-power wide-area (LPWA) connectivity standard, is used. NB-IoT allows IoT devices to send data directly to the cloud without an intermediary gateway, improving power consumption in user devices. The need for intelligence in IoT devices becomes crucial, especially in scenarios with high traffic from all connected nodes. Therefore, incorporating an AI layer into the 5G-IoT architecture ensures the capability to process big data with minimal latency, reliability, and continuous network service accessibility [21].

Our novel contribution aims to offer a Dynamic AI-IoT solution for application on ultra-low-power 5G-enabled IoT devices. This represents a significant step toward implementing federated learning for constrained-resource devices.

A. Dynamic AI-IoT System Architecture

Dynamic AI-IoT is an approach to automatically utilize AI capabilities in IoT systems connected to a 5G network. This design aims to develop an AI-enabled IoT component that can communicate with the AI Development Engine in 5G environments. The general workflow of the proposed architecture is presented in Fig. 1. 5G network offers a high data rate, wide bandwidth, acceptable latency, and a massive number of connected IoT devices. The AI Development Engine is capable of designing, training, compressing, and converting AI models. The AI-enabled IoT component then can exchange data and make an inference on an IoT device associated with dynamic updates of AI models.

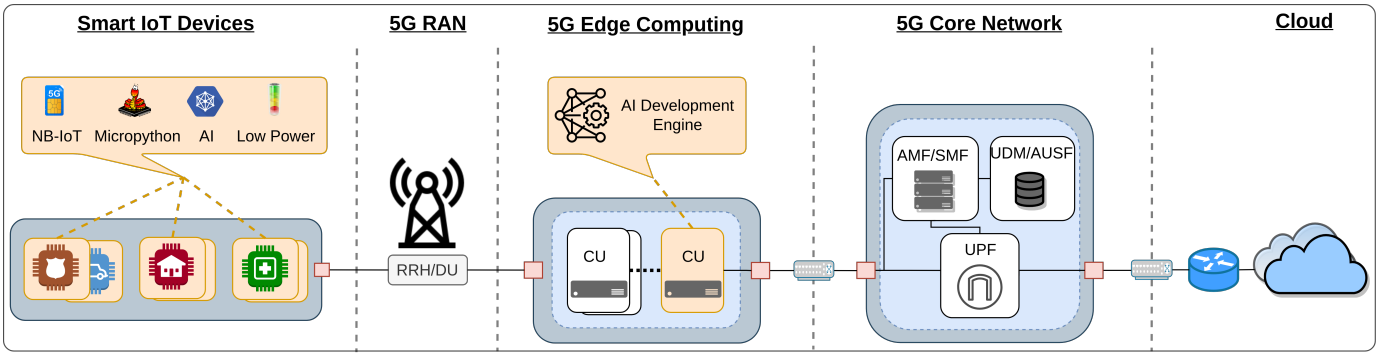


Fig. 1: Overview Architecture of a Dynamic AI-IoT in 5G infrastructure

In the context of this design, it is essential to recognize that the forthcoming 6G connectivity era holds significant potential benefits for our system. With 6G's projected ultra-high-speed, low-latency connectivity, our architecture can achieve even faster response times, support more complex AI models, and offer improved efficiency and reliability. Moreover, the expanded bandwidth and reduced latency characteristic of 6G networks can pave the way for new horizons in AI-powered IoT ecosystems.

In this design, a new IoT firmware has been implemented to support the Dynamic AI-IoT architecture by integrating Micropython, NB-IoT, AI capabilities and ULP unit. Flashing this firmware into IoT devices allows the tackling of the interactions with the AI Development Engine in a holistic manner. As depicted in Fig. 1, our proposed design shows that the AI Development Engine will be in the 5G edge computing. On the other hand, the IoT devices with the developed firmware can be connected to the 5G RAN, which oversees continuously receiving and sending the data to the 5G edge and then receiving and sending updates to IoT devices.

The proposed architecture in Fig. 1 shows how feasible it is to overcome the challenges raised in Section II. This is a logical change that should be put in place to support federated learning in smart IoT applications. For example, the inference computing processes, AI model configurations, and AI model memory management should be done on IoT devices, resulting in a significant reduction in the communication overhead with the connected 5G Edge.

In the previous section, the architecture overview provided a better understanding of how our contributions add value to 5G networks. In this section, the internal components of our proposed AI-IoT architecture are explored. As shown in Fig. 2, there are three layers: the IoT Layer, AI-IoT Layer, and AI Layer. These layers are explained in the following subsections. It's worth mentioning that this figure, for all the layers, does not cover all the functional blocks but includes key components used in our design.

*1) **AI Layer**:* The AI layer belongs to 5G edge computing, where networking, computing, and storage services are offered with high availability and low latency. A substantial number of mobile and IoT devices can benefit from these services. On the other hand, the AI models are generated on this layer by our AI Development Engine which runs into multiple development

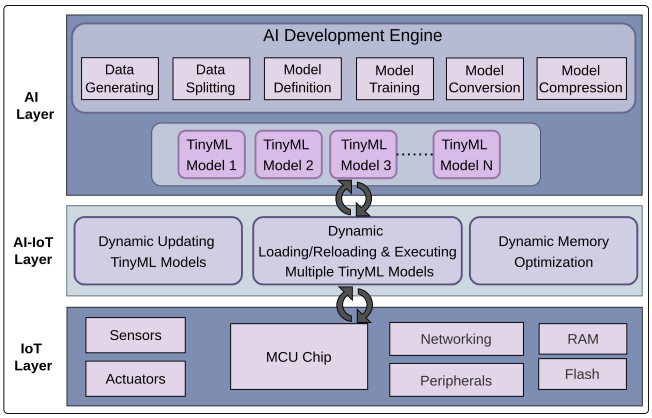


Fig. 2: Proposed Dynamic AI-IoT Architecture

phases to eventually give different versions of TinyML models available for different AI-IoT use cases.

Data Generating: Predictive models are based on the collection of data, which can be gathered from a variety of sources. The more error-free the data collection is, the more accurate the predictive models are.

Data Splitting: Datasets are commonly divided into training, validation, and test subsets [22]. The validation dataset assesses model accuracy, while the test data evaluates model predictions against actual values.

Model Definition: Model development begins with neural network design, focusing on layer architecture and connectivity. The neural network should be designed to effectively capture patterns from the training data.

Model Training: Training involves setting various parameters, such as input-output pairs, epochs, and batch size. Validation data is used to monitor model performance during training.

Model Compression: Efficient network algorithms like MobileNets and SqueezeNet are used to reduce model size. MobileNets uses depth-wise and point-wise convolutions to build lightweight deep neural networks, while SqueezeNet downs samples the data using special 1×1 convolution filters. Quantization [23] and pruning [24] techniques further enhance efficiency by reducing model weight size and computation [25].

Model Conversion: To make the model IoT-compatible, it

needs to be converted to a suitable format, such as a C array [26] or serialized file. This conversion facilitates integration into IoT applications.

In the realm of developing efficient AI models for IoT devices, TensorFlow variants like TensorFlow Lite (TFLite) and TFLM have emerged as optimized frameworks for running models on small, low-powered devices such as mobile phones and microcontrollers. These variants offer compact binary sizes and minimal dependency requirements, making them ideal choices for resource-constrained IoT environments where standard C or C++ libraries may not be readily available [27].

*2) **AI-IoT Layer**:* This layer is a logical bridge between the AI and IoT layers with three main tasks.

Firstly, dynamic updating of TinyML models to improve the accuracy in real-time. After making an on-device inference and analyzing the model performance, there will be an option to send back some useful information from the IoT device directly to the AI Development Engine in the AI layer. Then, the AI Development Engine will consider the changes to re-train the model and ensure that the IoT device has an updated model.

Secondly, dynamic loading, reloading and executing multiple TinyML models to prevent any delay caused by adapting the IoT device to new models. The IoT device will be able to load multiple TinyML models and choose one of them to be executed at a time. Re-loading an updated model is done automatically so the IoT device can make new predictions. The only limitation would be the memory capacity of the IoT device. Therefore, the more available memory the IoT device has, the more TinyML models can be loaded at the same time. This seems to be an obvious capability but instead, it has not been yet achieved in the literature and we are proving the suitability of this approach.

Thirdly, dynamic memory optimization to enable efficient memory usage of IoT devices in real-time and without re-flashing the firmware. Three main key components can be optimized.

1) Memory alignment: Ensuring the starting memory address of the model is aligned correctly.
2) Tensor arena: Any model reserves only a required amount of memory regarding its size.
3) Registration of the model's operations: Only the operations used in the model's design are assigned to take memory space for execution.

In the context of the AI-IoT Layer described in Subsection III-A2, Micropython emerges as the best language for IoT devices to execute the three critical tasks:

1) Dynamic Model Updates: Micropython simplifies real-time model updates based on on-device inference, allowing for improved accuracy.
2) Dynamic Model Loading: It supports the dynamic loading and execution of multiple TinyML models, preventing delays during model adaptation.
3) Dynamic Memory Optimization: Micropython enables efficient memory usage without the need for firmware re-flashing, optimizing memory alignment, tensor arena utilization, and model operation registration.

Micropython's origin in Python 3.5 and its compatibility with various microcontrollers make it an ideal choice for IoT solutions. Its integration with IoT development tools like Atom further streamlines the development process.

*3) **IoT Layer**:* The IoT layer is the target environment for executing TinyML models, offering end-to-end AI solutions. At its core, the MCU handles inferences and data processing, utilizing RAM and flash memory for storage. Networking and peripherals facilitate interactions with the environment, including other IoT devices, sensors, and actuators.

Furthermore, the Fipy platform, as depicted in Fig. 3 and powered by the Espressif ESP32 System-on-Chip (SoC), with its support for Micropython, an Ultra-Low-Power mode, and compatibility with five distinct network technologies, including NB-IoT cellular technology for direct cloud connectivity [28], has been selected as the foundation for our development. This choice is predicated on the platform's enhanced functionalities, rendering it particularly well-suited for the implementation of AI-IoT applications.

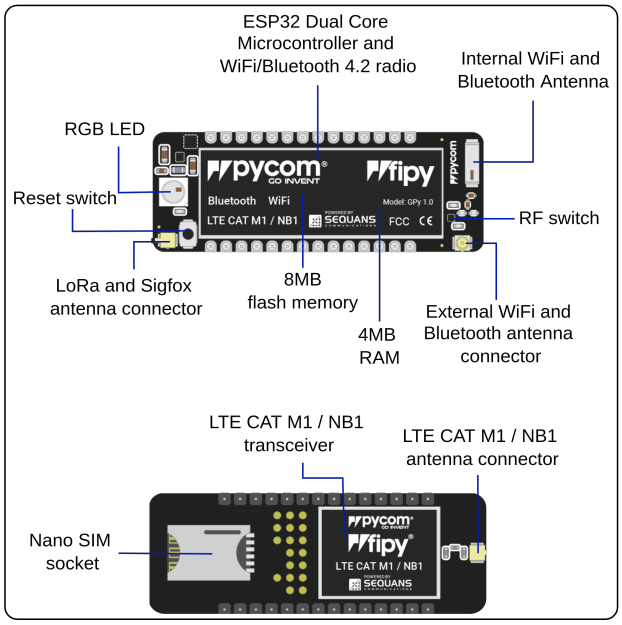


Fig. 3: Fipy Features

IV. Design of the Dynamic AI-IoT System

In this section, we delve into the intricate design aspects of an AI-IoT system that exhibits dynamic adaptability in response to changing conditions, facilitating real-time insights and actions. The subsequent subsections systematically dissect the various components of this system, unveiling its dynamic processes, interactive sequences, state transitions, and structural hierarchies.

A. Dynamic AI-IoT Process

In this section, we define the Dynamic AI-IoT Process, which is a framework for dynamically updating TinyML models on IoT devices efficiently and smoothly.

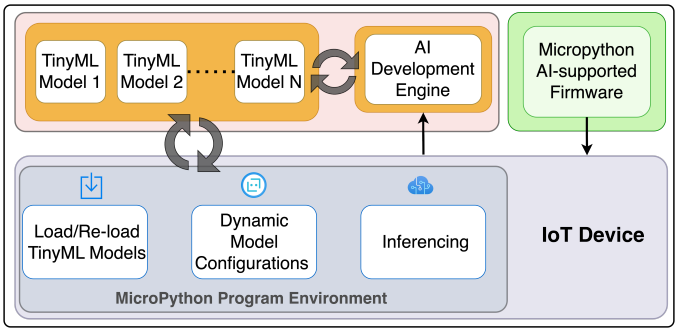


Fig. 4: Dynamic AI-IoT Process

Let $M = \{L, C, W\}$ represent an AI model, where L stands for layers, C denotes connections between layers, and W represents the weights of artificial neurons within each layer. We distinguish two types of AI model updates:

1) $M_t = \{L', C', W'\}$ represents a total model update where all model components are updated, denoted as L' for layers, C' for connections, and W' for weights.
2) $M_p = \{L, C, W'\}$ represents a partial update, focusing solely on weight updates (W') as a result of further training.

Our proposal supports both types of updates: M_t and M_p. The algorithmic representation of these updates is provided in Algorithm (1). It outlines a process for dynamically and efficiently updating TinyML models on an IoT device. The following list describes the specific steps involved in this process:

1) **Initialization (Step 1)**: The algorithm begins with the initialization of the IoT device, which is assumed to be equipped with an existing TinyML model and Micropython AI-supported Firmware.
2) **Firmware Flashing**: The algorithm proceeds by flashing the Micropython AI-supported Firmware onto the IoT device. This firmware contains the necessary infrastructure for handling TinyML models.
3) **Connecting to the AI Development Engine**: The IoT device establishes a connection to the AI Development Engine.
4) **Model Inference Loop (Step 2)**: With the model configured, the algorithm performs an inference, which means it uses the model to make predictions or perform specific tasks based on incoming data.
5) **Data Transmission (Step 3)**: After performing inference, the IoT device sends relevant data, which could include the results of the inference or other information, back to the AI Development Engine.
6) **Model Reception (Step 4)**: Every N times (100 by default), The IoT device requests a model update from the AI Development Engine. The IoT device receives either an updated version of the existing TinyML model or a completely new one.
7) **Model Loading (Step 5)**: The received TinyML model is loaded into the RAM of the MCU on the IoT device.
8) **Model Configuration (Step 6)**: The algorithm configures the loaded model as needed. This step involves setting up the model's parameters, input/output formats, and any other necessary settings.
9) **Edge Reception (Step 7)**: The AI Development Engine may generate a new TinyML model, possibly based on the data it has collected and processed.
10) **Model Generation (Step 8)**: The AI Development Engine may generate a new TinyML model after N numbers of received values based on the data it has collected and processed or will load a completely new model if it has been requested by the administrator.

Algorithm 1 Dynamic AI-IoT Model Updates

Require: IoT Device, TinyML Model, Micropython AI-supported Firmware
Ensure: Updating the IoT device with the new TinyML Model

```
IoT device: Initialize()                                  ▷ Step 1
IoT device: Flash(μPython Firmware)
IoT device: Connect(Edge), Inferences=0
while IoT device: True do                                 ▷ Step 2
    Output=Infer(Inputs, Model)
    Inferences=Inferences+1
    Send(Edge, Output)                                    ▷ Step 3
    if Inferences % 100 = 0 then
        Model = Receive_Last_Model(Edge)                  ▷ Step 4
        Load(Model.L, Model.C, Model.W)                   ▷ Step 5
        Configure(Model)                                  ▷ Step 6
    end if
end while
Edge: Samples=0
while Edge: True do
    Store(Output)                                         ▷ Step 7
    Samples = Samples +1
    if Samples % 100 == 0 then                            ▷ Step 8
        if Partial_Update() then
            LastModel=Train(LastModel,Output)
        end if
        if Full_Update() then
            LastModel=Load(Model.L,Model.C,Model.W)
        end if
    end if
end while
```

This algorithm enables an IoT device to continuously update its TinyML model, ensuring that it can adapt to changing data and requirements without the need for firmware re-flashing or significant device downtime.

B. Dynamic AI-IoT Sequence Diagram

The data flow in the system is illustrated by a sequence diagram in Fig. 5. The main components exchanging data in the system are the Firmware Development Engine, AI development Engine, Fipy as an IoT device, sensors and actuators. The data is propagated through the network from one component to another as follows:

1. The foundation for any IoT device to execute AI models and make predictions consists of two essential components:

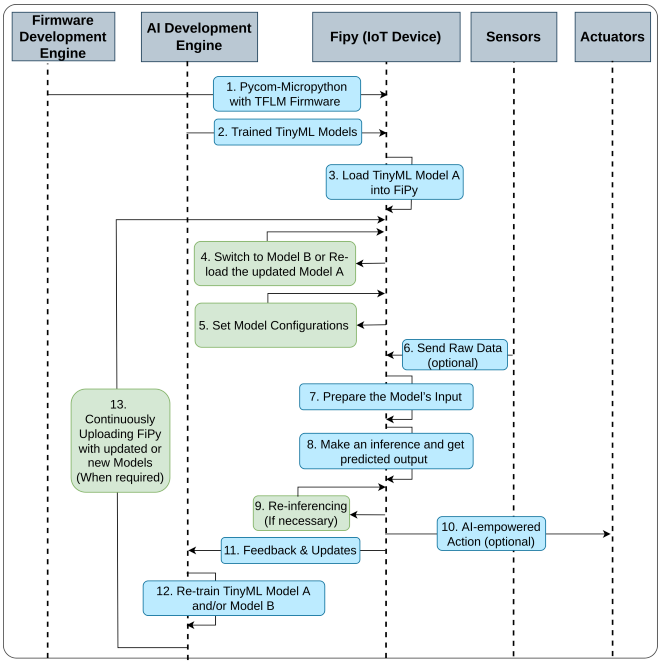


Fig. 5: Data Flow between the System Components

AI-supported firmware and an AI model. In our system, the Firmware Development Engine is where our customized Pycom Micropython firmware with TFLM extension has been developed and ensured to be executable on Fipy. Additionally, Fipy is equipped with the ability to update its firmware while still running, enabled by OTA update. The combination of OTA alongside NB-IoT provides Fipy with the capability to connect to 5G Edge through RAN and remotely update its firmware when updates in the capabilities of the AI IoT framework are required.

2. Meanwhile, the AI Development Engine generates AI models for various use cases, and only those TinyML models that can run on Fipy's MCU can be transferred. Transferring TinyML models is facilitated through RAN after enabling NB-IoT on Fipy. For simplicity, if Fipy is involved in two different use cases, two models (models A and B) are developed inside the AI Development Engine and then sent to Fipy.

3. To commence using the TFLM model, it needs to be loaded into Fipy's RAM, where the ESP32, the main chip of Fipy, can swiftly access the model's addresses. Loading the model into the RAM is one of the functions we developed to be executed from within Micropython. In Fig. 5, we illustrate the process of loading Model A into Fipy.

4. If Model A has been updated or there is a need to switch to Model B, the load function can reload the updated model or switch to a new model on demand. Importantly, this can be done without re-flashing the firmware; only the file of the new model needs to be transferred to the FiPy, providing a new level of flexibility.

5. Once the model is loaded, it should be configured dynamically. All the configuration functions were explained in the implementation section.

6. Sensors play a crucial role in enabling any IoT device to interact with the surrounding environment. In the case of Fipy, it is an extensible platform that allows connection to any sensor using widely supported buses. Several hats are available for Fipy, such as the Pysense shield, enabling end-users to sense the environment using five different sensors: accelerometer, light, pressure, temperature, and humidity. Once the data is collected, it can be processed to obtain useful information for AI-IoT applications.

7. The next step is to prepare the model's input. Two essential methods have been developed to obtain the input's size and type from a dynamically updated model. These methods make it easy for Micropython to generate the input. The AI model may have a single input that can be stored in a variable, or multiple inputs that need another method to store them, such as arrays.

8. Once the model is loaded, and the input is defined, it is possible to insert the input into the model and make an inference. The model's output, whether it is a single prediction or multiple ones, can be obtained and saved for later use.

9. The ability to remake the inference is available. Enabling this functionality is significantly useful, especially when the model's input is a real stream, and the IoT device should keep predicting, creating an effective prediction pipeline.

10. For each prediction made, different actions can be taken. The AI-driven action could be as simple as changing an onboard LED's colour, and it could be as sophisticated as turning on a production line in a factory.

11. A dynamic AI-IoT system requires continuous improvements. Therefore, performance data and updates related to the model are sent to the AI Development Engine.

12. The AI Development Engine collects all the data coming back from Fipy and uses it to regenerate more accurate AI models.

13. Fipy will be waiting to receive the updated model or even a new one. This model shapes the purpose of the Dynamic AI-IoT application, whether it is only for continuous improvement or starting a completely new use case.

C. TinyML Model's State Diagram

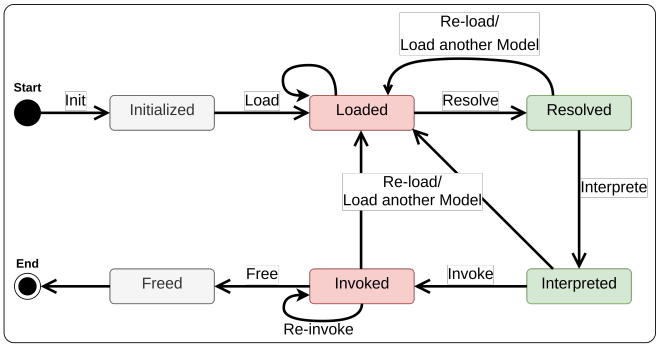


Fig. 6: State Diagram

The state diagram illustrated in Fig. 6 depicts how a TinyML model transitions from one state to another in a Dynamic AI-IoT application. For each state, there is a corresponding function designed to ensure seamless movement to the next state. These functions have been detailed in the implementation section. The process begins with initializing the model,

wherein sufficient memory space is reserved for the model to operate efficiently. Once reserved, the model enters the initialized state. Following this, the model is loaded into the IoT device's RAM. In this state, multiple models can be loaded, with consideration given to the latest one. The subsequent state involves configuring the model, and upon completion, the model transitions to the resolved state. After configuration, the interpret function ensures the model is ready for inference, moving it to the interpreted state. The invoked state is the subsequent logical state, occurring when the model has made predictions. This state can be revisited as needed, but only if the previous state was the interpreted state. The final state for any TinyML model is the freed state, where memory is cleared if there is no need to invoke the model any longer. In our Dynamic AI-IoT system, it is possible to load a new model or reload an updated model as needed. This capability is reflected in the state diagram, where the model can return to a loaded state from resolved, interpreted, or invoked states. This stands out as one of our key contributions.

D. Class Diagram

This section illustrates the design patterns applied in firmware development. Adapter and Façade designs have been combined to provide the user with an interface that grants access to all the implemented Micropython methods, whether from general Micropython modules, Pycom modules, or newly developed modules.

A façade pattern, as depicted in Fig 7, is a structural design pattern used as a wrapper to conceal the complexities of a large system by offering a simple interface to the client.

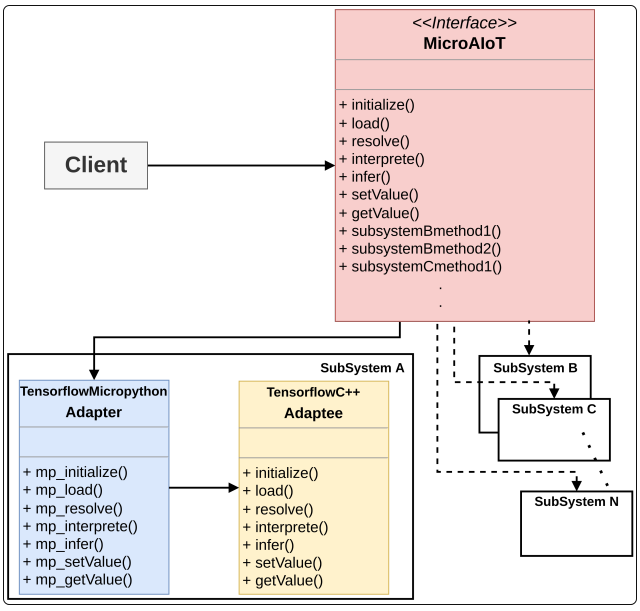


Fig. 7: Façade and Adapter Diagram

In our development, the user can invoke a Micropython function from any subsystem without needing to know the implementation details behind it. Subsystems are hidden from the client, representing, for this paper, a comprehensive development of the TensorFlow module or any other built-in module. For instance, the user might wish to initialize or load a new TensorFlow model. In such cases, the Micropython interface acts as the façade, simplifying the user experience by shielding them from the complexities associated with Micropython objects and their bindings in Subsystem A.

On the other hand, our TensorFlow module, Subsystem A in Fig. 7, employs a C++ to Micropython adapter for every class in the TensorFlow library. When an adapter receives a call, it translates the incoming Micropython data into C++ and then passes the call to the appropriate methods for wrapping a TensorFlow object. Here is the process:

1. The adapter acquires an interface compatible with one of the Micropython objects.

2. Using this interface, the Micropython object can safely invoke the adapter's methods.

3. Upon receiving a call, the adapter transfers the request to the C++ object but in a format that the C++ object expects.

This approach enables the combination of statically compiled C++ functionality with the dynamically available capabilities provided by Micropython, facilitating the reprogramming of the device without the need for re-flashing.

V. Dynamic AI-IoT System Development

This section aims to highlight how to fulfill our four innovations mentioned earlier in the introduction section. To achieve this end, dynamic intelligent capabilities have been added to an IoT device to act as an independent smart device, and dynamically be adapted to the possible changes.

A. AI-IoT Firmware Development

The proposed design in Fig. 8 presents how to enable AI on ESP32 core devices using Micropython to achieve dynamic AI-IoT solutions. In other words, this section describes how to compile Micropython firmware after successfully porting external TFLM C++ modules. We have two frameworks and a compiler. The compiler used is the Xtensa GCC compiler that can be configured to report any error during the building process. The first framework is ESP-IDF in which TFLM C++ modules will be integrated and ready to be compiled, see (1) in Fig. 8. At this stage, a compiled TFLM C++ library would be generated and added to the Micropython framework, see (2) and (3) in Fig. 8. After that, the TFLM C++ library will be linked and wrapped to Micropython objects to create a new TFLM Micropython library. The outcome of the second compilation is a binary file to be flashed into ESP32 core devices directly. The (4) and (5) steps in Fig. 8 show how to achieve this end. Next, TFLM C++ and Micropython libraries will be explained to discuss how the work has been done.

TensorFlow Lite Micro repository has the necessary functionalities to deal with a pre-trained TinyML model. From loading it properly to making an inference to get predictions. We have implemented a facade file with the key functions to be exposed to Micropython. The following subsections explain each of those functions.

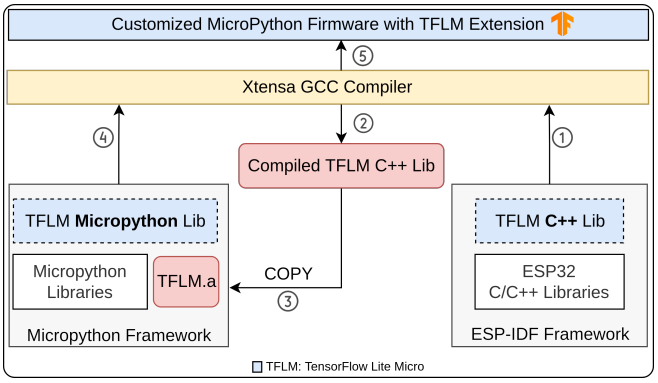


Fig. 8: Customized Micropython Firmware with TFLM Extension

1) ***Initialization Functionality****:* The first function is the *initialize* function, responsible for allocating a specific amount of memory known as the "tensor arena." This memory allocation reserves space for the model's input, output, and intermediate arrays. It's crucial to note that the memory addresses for these arrays must align to 16 bytes, and the required size varies depending on the size of the AI model. Currently, there is limited documentation in the community on this process, making it valuable to provide detailed guidance on the procedure:

1. Get the size required to load the model.

2. Adding 16 extra bytes to guarantee we have enough space after the alignment. (0x10 in hex);

3. Round down to a 16 bytes dividable address by AND-ing 0x0F (last 4 bits to zeros);

4. The result is aligned to 16 bytes.

2) ***Loading Functionality****:* The following function is the *loading* of the model, which can be done in two ways: either as a compressed file or a compressed char array. In our proposed AI-IoT architecture, we have outlined how the model is compressed and converted after training. Subsequently, it becomes essential to develop the logic for handling the loading of the AI primitive functions for model execution. The *resolve* function is introduced to dynamically register only the operations utilized by the loaded model.

Our approach to this design has not been previously covered in the existing literature, and it is specifically optimized for memory-constrained IoT devices. Currently, developers typically use the *AllOpsResolver* to register all available operations in TFLM, resulting in a memory-intensive implementation. However, since a given model only utilizes a subset of these operations, developers have turned to the *MicroMutableOpResolver* to register only the necessary operations. The drawback of this approach is that developers must manually know and register all operations beforehand.

Our implementation of the resolve function incorporates an introspection capability, allowing it to query the AI model to determine the required operations. It then registers only this subset, achieving a self-optimization of the memory footprint. This innovative approach enhances efficiency and addresses the limitations of the existing methods.

3) ***Interpreter Functionality****:* The next essential function is the *interpret* function. This function gathers the results of the preceding functions and combines them to create an object with complete access to the loaded model. For example, it retrieves pointers to the model's input and output tensors for later utilization, along with pointers to the model's input and output types and sizes. Once the input is ready, the interpreter is activated using the inference function to perform model inference and populate the predicted values in the output layer.

4) ***Inference Functionality****:* It executes the loaded model, utilizing the information available in the input and producing results accessible through the interpreter.

5) ***Generation of the Firmware****:* The TFLM project, along with our developed functions, can be integrated into the ESP-IDF framework and compiled using the Xtensa GCC compiler to generate the TFLM C++ library. However, directly flashing the compiled ESP-IDF as a binary file into Fipy, as done by other researchers, limits the AI capabilities of the device. This limitation arises because re-flashing the firmware is required every time a new model needs to be uploaded.

To overcome these challenges, the Micropython framework, implemented in native C, has been adjusted to incorporate the compiled TFLM C++ library. This modification enables the invocation of the previously described facade from C code, providing complete control over the model. The TFLM library can be imported, and its functions can be called within the Micropython scripting language. To achieve this, a Micropython object must be implemented, allowing the exposure of a Python library with all the functions described in the preceding subsections. These functions are, in turn, implemented using the native C language. Listing 1 provides an example of creating a Micropython object that calls the interpret function.

```
STATIC mp_obj_t interpreting()
{
interprete();
return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_0(interpreting_obj,
    interpreting);
```

Listing 1: Creating Micropython object

- Making the necessary bindings for new Micropython objects. This can be done in the same file or a separate file in the Micropython project. In both cases, a pointer to a new object should be added to the module's globals table, which in turn will be linked to the module globals dictionary, as shown in Listing 2.

```
STATIC const mp_map_elem_t
    microAIoT_module_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR_interprete),
            (mp_obj_t)&interpreting_obj },
};
STATIC MP_DEFINE_CONST_DICT(microAIoT_module_globals
    , microAIoT_module_globals_table);
const mp_obj_module_t microAIoT_module = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&
    microAIoT_module_globals,
};
```

Listing 2: Micropython Bindings Example

- Compiling Micropython project to get the final firmware with TFLM capabilities.

*6) **Dynamic Usage of the Library in Micropython**:* Our AI-IoT Micropython module can be accessed from a Micropython program, similar to any other built-in module. The usage of the microAIoT module in Micropython is illustrated in Listing 3.

```
import microAIoT
microAIoT.initialize(tensor_arena_size)
microAIoT.load(TFLM_model_byte_array)
microAIoT.resolve()
microAIoT.interprete()
microAIoT.setValue(inputNumber, inputValue)
microAIoT.infer()
microAIoT.getValue(outputNumber)
```

Listing 3: New Module Usage in Micropython program

VI. Validation Use Cases

Fig. 9 provides an overview of the different use cases used to validate our AI-IoT contribution. The following subsections elaborate on each of these use cases in detail.

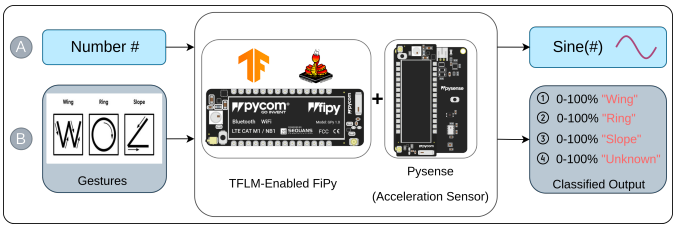


Fig. 9: Overview of Sine Wave Generation and Gesture Recognition Use Cases

A. Use Case A: Sine Wave Generation

This use case serves as a simple demonstration, illustrating the fundamental implementation of a fully dynamic AI-IoT application without introducing complex logic. This straightforward example highlights the basic utilization of TensorFlow Lite Micro. The 2.5 KB model is trained to replicate the mathematical sine function, capable of approximating the sine of a number within the range of 0 to 2π. Subsequently, the model is converted for inference on a microcontroller. The primary aim of the initial experiment is to verify the feasibility of running the TinyML model on Fipy and dynamically interacting with it through Micropython.

The neural model's structure employed in this use case is outlined in Fig. 10, comprising three fully connected hidden layers designed for inferring the value of the sine function.

Fig. 10: AI model structure used for Sine prediction

B. Use Case B: Gesture Recognition

Gestures, defined as meaningful movements used to interact with the environment, are recognized through a process known as gesture recognition. Embedded devices are increasingly incorporating emerging technologies to implement AI use cases involving gesture recognition. In the second experiment, a pre-trained TinyML model from TensorFlow is employed to recognize specific gestures. The model utilizes real-time data streaming from a 3-axis accelerometer as input, providing a classified set of recognized gestures as output. The source code for this project is available in the official TensorFlow repository, specifically the magic wand example.

For the implementation of this use case, Fipy is connected to Pysense, enabling the utilization of an integrated acceleration sensor. Data is streamed from the Pysense 3-axis accelerometer to feed the model's input. The magic wand TinyML model is trained to identify three gestures: a clockwise circle or "ring" gesture, a capital "W" or "wing" gesture, and a right-to-left "slope" gesture. When one of these gestures is detected, the onboard RGB LED will illuminate in green, blue, or red, corresponding to the performed gesture, serving as an example of AI application.

The model is a 20 KB Convolutional Neural Network (CNN) trained on a gesture dataset obtained from 10 individuals who performed four gestures (Ring, Wing, Slope, and an unknown gesture) fifteen times each. The model's input comprises raw accelerometer data in an array of 384 values, representing 128 times the real X, Y, and Z axes. The output provides probability scores for the three recognized gestures and an unknown gesture. The sum of these scores is 1, with a probability of 0.8 or higher indicating a confident prediction of a given gesture.

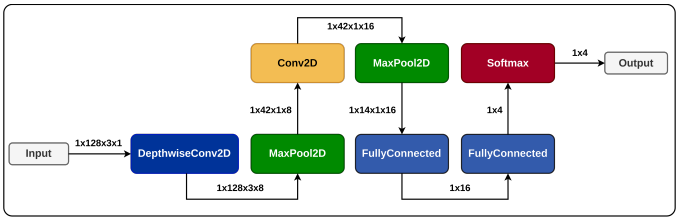
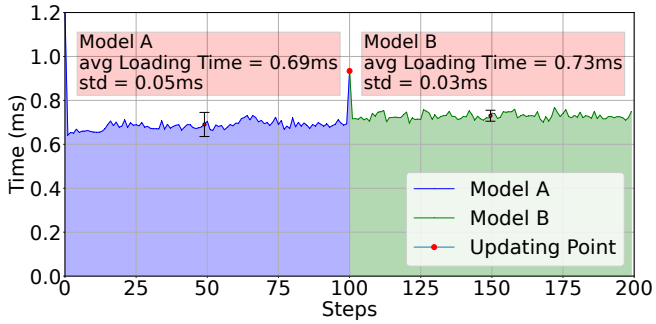


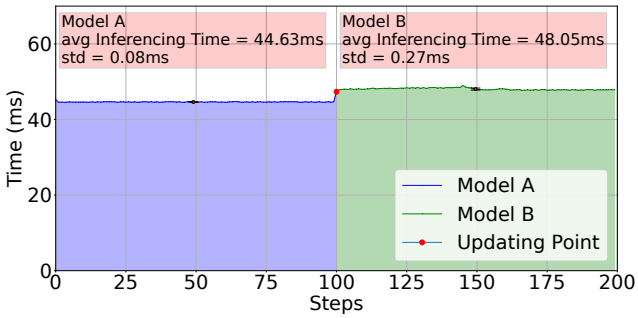
Fig. 11: AI model structure used for Gesture recognition

The neuronal network structure for training the CNN model in this application is depicted in Fig. 11. In contrast to the first use case, this one involves more operations in building the model, including FullyConnected, DepthwiseConv2D, MaxPool2D, and Conv2D operations. The Pysense's built-in accelerometer was configured at 50Hz, and data was downsampled to 25Hz, aligning with the model's training rate. The software accumulates sensor readings until sufficient data (128 readings) is obtained to perform a prediction.

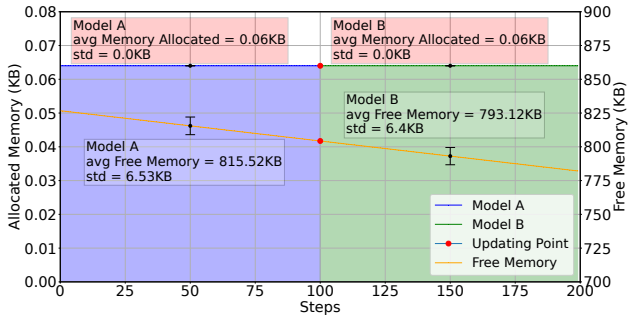
Dealing with accelerometer data can present variations and unexpected values. To enhance accuracy, the model is designed to consistently predict the same gesture multiple times, boosting confidence in predictions. In Micropython, the previous and current predictions are tracked. If a specific prediction persists for a defined number of consecutive times,



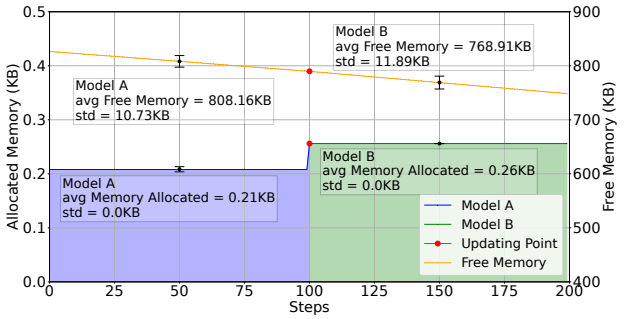
(a) Loading Time



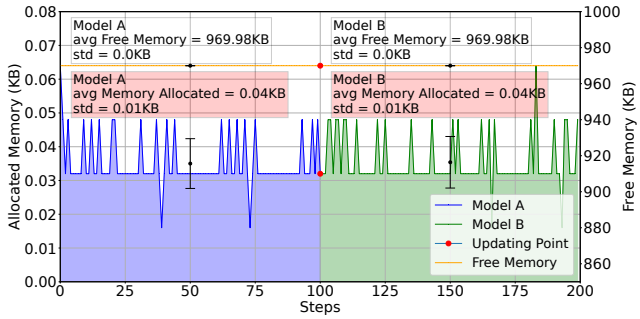
(b) Inferencing Time



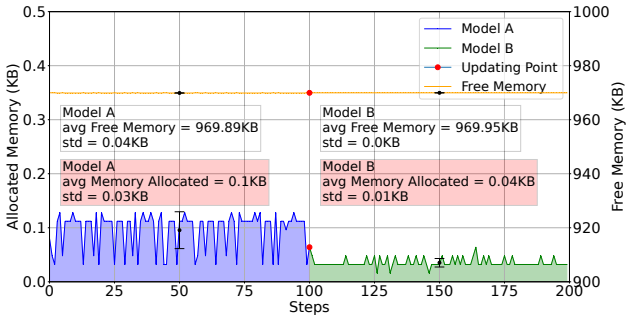
(c) Memory Usage for Loading without gc



(d) Memory Usage for Inferencing without gc



(e) Memory Usage for Loading with gc



(f) Memory Usage for Inferencing with gc

Fig. 12: Empirical Evaluation of Performance of the AI-IoT Prototyped Architecture

it is reported as the detected gesture; otherwise, it is identified as an unknown gesture.

The two previously described use cases are evaluated to demonstrate the feasibility of Dynamic AI-IoT implementation. The focus of the test is on loading a TinyML model into Fipy and executing inference. The experimentation is divided into three parts: execution time, memory usage, and power consumption. CPU usage is excluded from this test, as the CPU runs continuously, even in the absence of processing tasks, due to the "idle loop" code snippet. All experiments were conducted in a Micropython environment using Atom software with the Pymakr plugin.

VII. Results

A. Model Loading Results

To assess the loading time, we conducted an experiment involving the consecutive loading of the sine AI model 100 times. After this continuous reloading, a deliberate switch occurred to initiate the reloading of the gesture recognition model, which was then reloaded another 100 times. As depicted in Fig. 12(a), the average loading time for the sine model was 0.69 ± 0.05 milliseconds. On the other hand, the average loading time of the gesture recognition model was 0.73 ± 0.03 milliseconds.

B. Model Inference Results

For evaluating the inference time, we loaded the sine model and then called its inference function 100 times to improve the

accuracy of the inference time measurement. Following that, we intentionally switched to calling the inference function of the gesture recognition model 100 times. As illustrated in Fig. 12(b), the results revealed that the average inference time for the sine model was 44.63 ± 0.08 milliseconds, whereas the gesture recognition model, on average, had an inference time of 48.05 ± 0.27 milliseconds.

It is worth mentioning that these results allow the tiny IoT device to provide more than 20 times per second the interface of the values which is a very decent real-time performance. It validates the suitability of the proposed approach for close to real-time applications using AI IoT architectures.

C. Memory Usage in Model Loading

The average allocated memory for loading our two different models was measured without using Micropython's available memory management garbage collector (gc) utility. We loaded each of the AI models 100 times. Surprisingly, it was observed that both models consumed the same amount of memory. More interestingly, the repetitive loading of both similar and dissimilar models resulted in a significant reduction in available memory, as shown in Fig. 12(c). This is because there is no automatic unloading of the previous model when loading a new one, impacting the available memory for subsequent model loading. Notably, the standard deviation of the free memory is approximately 6.5 KB, reflecting fluctuations in the described mechanisms.

When we enable the Micropython garbage collector, as shown in 12(e), the results consistently follow a uniform distribution, with a standard deviation of 0 KB for free memory. This indicates that the garbage collector effectively clears 100% of the allocated memory. The garbage collector is employed just before and after loading our TinyML models.

D. Memory Usage in Inference

An analog experiment was conducted to assess memory consumption during the inference task. Without using a garbage collector, the average memory allocated for the sine model's inference is 0.21 KB ± 0.00, and for the gesture recognition model, it is 0.26 ± 0.00 KB, as depicted in Fig. 12(d). The significant change is evident in the fluctuation of free memory, which consistently decreases with each execution of the inference task. To be precise, there is a total decrease of 10.73 KB for each inference step with the sine AI model and 11.89 KB for the gesture recognition model. This continuous decrease is primarily attributed to the preparation of the input and output of the AI model, both directly handled by Micropython. Once ready, they are passed to the native C implementation, but this allocated memory is never freed. Consequently, in the next step, a new allocation in Micropython for the input and output is carried out.

The problem is resolved by enabling the Micropython garbage collector. This has a significant impact as it can free up all the memory allocated in the previous inference step, clearing 100% of the allocated memory. These outcomes are illustrated in Fig. 12(f).

This memory management technique ensures that the AI model can run for an extended period without experiencing memory shortages, validating the suitability of the proposed solution for the prolonged execution of AI models in IoT devices.

E. Model Update Time

In the context of dynamic TinyML deployment on IoT devices, it is essential to evaluate the time required to update and transmit TinyML models to these devices. To provide insights into this aspect, we measured the time it takes to transmit two different TinyML models to our IoT device, the FiPy. The first model, Sine Wave Generation, has a file size of 2.5 KB, while the second model, Gesture Recognition, has a larger size of 20 KB. These models represent typical use cases in the AI-IoT domain, with varying model sizes.

Our results reveal that transmitting the Sine Wave Generation model to the FiPy device requires approximately 0.068 seconds. In contrast, the larger Gesture Recognition model can be transmitted in roughly 0.545 seconds. These time frames account for the complete process of model transfer.

For updating the model, it is important to keep predictions running smoothly. Therefore, the system is designed to pause for less than 2 seconds during the update without requiring a FiPy re-flash or restart.

F. Power Consumption

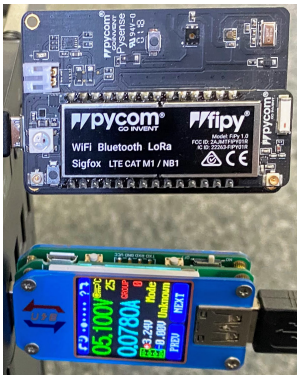
One of the primary deciding factors for choosing IoT devices like these over more powerful options such as the Raspberry Pi 4 is mainly associated with power consumption. Therefore, measuring power consumption provides insight into how long the IoT device can function on a given amount of energy or what power budget is needed for continuous AI IoT use cases. In this experiment, we utilized a USB Meter, illustrated in Fig. 13(a), to measure the device's current and voltage during a specific task. The power consumption in watts was calculated from these metrics.

The experiment proceeded as follows: first, the gesture recognition model was loaded 100 times, followed by 100 iterations of inference on the last loaded model. As shown in Fig. 13(b), the average power consumption for loading the gesture recognition model 100 times was 488.8 milliwatts, while inference consumed 485.89 milliwatts. Both exhibited similar and consistent trends, with standard deviations of just 27.44 milliwatts for loading and 31.95 milliwatts for inference.

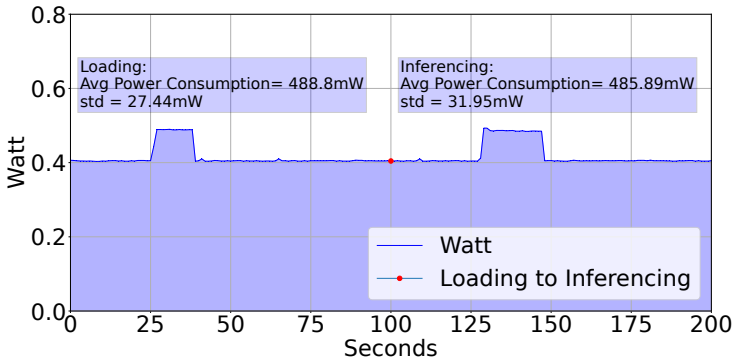
These consumption results are notably more efficient than other devices like the Raspberry Pi, which consumes around 6 watts when using all available cores and almost 3 watts in an idle state. This key finding makes this architecture ideal for prolonged deployments where battery life and lasting performance are critical elements for the success of the use case.

G. Overhead Time

To understand the cost of loading and executing AI models on IoT devices using Micropython, we measured the time



(a) Power measurements using USB Meter



(b) Power Consumption

Fig. 13: Power Consumption Experiment

required for these tasks without Micropython. Specifically, we used the original C++ implementation and compared the results. Table II demonstrates a noticeable difference for Model A (sine prediction) when loaded and executed in different languages. On the other hand, for Model B (gesture recognition), adding dynamic AI-IoT capabilities only increased the execution time by 6 ms. These values represent averages from 10 different executions.

It is important to note that as the size of the AI model increases, the difference in inference time between C++ and Micropython becomes smaller. This overhead adds value by allowing the uploading of multiple homogeneous and heterogeneous AI models in real-time without the need to update the device firmware. This capability is crucial for advanced use cases involving federated learning, online learning, and more.

TABLE II: Overhead time between C++ and Micropython

	C++	Micropython
Loading Time (Model A)	0.027ms	0.69ms
Inferencing Time (Model A)	0.226ms	44.63ms
Loading Time (Model B)	0.026ms	0.73ms
Inferencing Time (Model B)	42ms	48.05ms

VIII. Limitations and Challenges

The implementation of a Dynamic AI system on the FiPy device, while promising, presents a set of inherent limitations and associated challenges.

Resource Constraints and Scalability: FiPy devices are characterized by limited memory and processing capabilities, which can include empirically validated limits such as the 2.4 Mb maximum model size allowed. This constraint necessitates careful consideration when dealing with larger and more complex AI models, thereby impacting scalability.

Energy Efficiency and Latency Optimization: Achieving energy efficiency and optimizing latency, especially in real-time applications, presents a dual challenge. Frequent AI model updates must be balanced to prevent excessive power consumption without compromising responsiveness.

Interoperability: FiPy devices are designed to adapt to various connection technologies. Ensuring smooth transitions between them can be complex in heterogeneous IoT environments.

Handling Large AI Models: Transmitting AI models exceeding 500 KB in size is a significant challenge. Memory allocation constraints on the FiPy device come to the forefront, requiring careful management when dealing with more extensive models.

IX. Conclusion and Future Work

In this paper, we have introduced a groundbreaking IoT-based architecture designed to empower ultra-low-power and cost-effective IoT devices with Dynamic AI capabilities. Through the practical implementation of this architecture on a Micropython-enabled and AI-enhanced IoT device, we have successfully developed two distinct applications. Notably, our system demonstrated exceptional performance, with the ability to load a 20 KB TinyML model for sine prediction over 136 thousand times in less than a second, while executing the same model over 2 thousand times within the same timeframe.

Our results unequivocally illustrate the transformative potential of our Dynamic AI-IoT system. It equips connected IoT devices with autonomous AI functionalities, encompassing self-configuration of internal AI models, memory management optimization, and dynamic loading and execution of multiple AI models. Furthermore, the utilization of Micropython enhances code reusability for AI purposes. Significantly, our system offers a dynamic solution for remote IoT device updates, with minimal impact on system performance, achieving near real-time inference with power consumption below 0.5 Watts. This outcome holds considerable promise for a wide range of practical applications.

The system has demonstrated significant potential across various domains, including industrial IoT, agriculture, smart cities, healthcare, retail, and environmental monitoring. It offers real-time data analysis, predictive maintenance, efficiency

enhancements, safety improvements, and sustainability. The Dynamic AI-IoT system stands poised to bring transformative advancements, promoting efficiency and enhancing the quality of life in these diverse applications within 5G and beyond systems.

It is important to note that our current implementation follows a train-then-deploy approach. However, future work will explore the integration of the federated and transfer learning approaches, particularly suited for applications that do not necessitate full on-device training. Additionally, ongoing research will focus on incorporating dynamic switching among various connection technologies, such as Wi-Fi, LoRa, and Sigfox. This innovation will enable IoT devices to seamlessly transition between different use cases, thus expanding the operational scope and versatility of the Dynamic AI-IoT system.

ACKNOWLEDGEMENT

This work was funded in part by the European Commission Horizon 2020 5G-PPP Program under Grant Agreement Number H2020-ICT-2020-2/101017226 “6G BRAINS: Bringing Reinforcement learning Into Radio Light Network for Massive Connections” and under Grant Agreement Number H2020-SU-DS-2020/101020259 ”ARCADIAN-IoT: Autonomous Trust, Security and Privacy Management Framework for IoT.

REFERENCES

[1] X. Hou, Z. Ren, K. Yang, C. Chen, H. Zhang, and Y. Xiao, “Iiot-mec: A novel mobile edge computing framework for 5g-enabled iiot,” in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, 2019, pp. 1–7.

[2] B. Sudharsan, J. G. Breslin, M. Tahir, M. Intizar Ali, O. Rana, S. Dustdar, and R. Ranjan, “Ota-tinyml: Over the air deployment of tinyml models and execution on iot devices,” *IEEE Internet Computing*, vol. 26, no. 3, pp. 69–78, 2022.

[3] T. J. Sheng, M. S. Islam, N. Misran, M. H. Baharuddin, H. Arshad, M. R. Islam, M. E. H. Chowdhury, H. Rmili, and M. T. Islam, “An internet of things based smart waste management system using LoRa and tensorflow deep learning model,” *IEEE Access*, vol. 8, pp. 148 793–148 811, 2020.

[4] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, “DeepSlicing: Collaborative and adaptive CNN inference with low latency,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2175–2187, 2021.

[5] G. Crocioni, D. Pau, J.-M. Delorme, and G. Gruosso, “Li-ion batteries parameter estimation with tiny neural networks embedded on intelligent IoT microcontrollers,” *IEEE Access*, vol. 8, pp. 122 135–122 146, 2020.

[6] K. Dokic, M. Martinovic, and D. Mandusic, “Inference speed and quantisation of neural networks with TensorFlow lite for microcontrollers framework,” in *2020 5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*. IEEE, 2020, pp. 1–6.

[7] Hoang-The Pham, M.-A. Nguyen, and C.-C. Sun, “AIoT solution survey and comparison in machine learning on low-cost microcontroller,” in *2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. IEEE, 2019, pp. 1–2.

[8] “Get started with TensorFlow lite micro on sony’s spresense,” https://blog.tensorflow.org/2021/10/TF-Lite-Sony-Spresense.html, accessed: 2022-3-1.

[9] M. O’Cleirigh, “tensorflow-micropython-examples: A custom micropython firmware integrating tensorflow lite for microcontrollers and ulab to implement the tensorflow micro examples.”

[10] On-Device AI Co. , Ltd., “MicroAI: Integrate MicroPython and TensorFlow lite for microcontrollers on embedded linux.”

[11] M. Yan, “esp32_mpy: Master control of robot using esp32 chip with openmv and tensorflow-lite support.”

[12] “MicroPython libraries — MicroPython 1.15 documentation,” https://docs.openmv.io/library/index.html, accessed: 2022-2-28.

[13] R. Sanchez-Iborra and A. F. Skarmeta, “TinyML-enabled frugal smart objects: Challenges and opportunities,” *IEEE Circuits Syst. Mag.*, vol. 20, no. 3, pp. 4–18, 2020.

[14] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli, “TinyML platforms benchmarking,” 2021.

[15] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, “Edge intelligence: The confluence of edge computing and artificial intelligence,” *IEEE Internet of Things Journal*, 2020.

[16] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, “Edge intelligence: Paving the last mile of artificial intelligence with edge computing,” *Proceedings of the IEEE*, 2019.

[17] E. Impulse. Edge impulse studio. https://www.edgeimpulse.com/. Accessed: 2023-10-16.

[18] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, “TensorFlow lite micro: Embedded machine learning on TinyML systems,” 2020.

[19] “OpenMV cam H7,” https://openmv.io/products/openmv-cam-h7, accessed: 2022-2-28.

[20] K. Shafique, B. A. Khawaja, F. Sabir, S. Qazi, and M. Mustaqim, “Internet of things (IoT) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5G-IoT scenarios,” *IEEE Access*, vol. 8, pp. 23 022–23 040, 2020.

[21] N. Javaid, A. Sher, H. Nasir, and N. Guizani, “Intelligence in iot-based 5g networks: Opportunities and challenges,” *IEEE Communications Magazine*, vol. 56, no. 10, pp. 94–100, 2018.

[22] K. M. Kahloot and P. Ekler, “Algorithmic splitting: A method for dataset preparation,” *IEEE Access*, 2021.

[23] Gheorghe and M. Ivanovici, “Model-based weight quantization for convolutional neural network compression,” in *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, 2021.

[24] G. Tian, J. Chen, X. Zeng, and Y. Liu, “Pruning by training: A novel deep neural network compression framework for image processing,” *IEEE Signal Processing Letters*, 2021.

[25] S. Wiedemann, H. Kirchhoffer, S. Matlage, P. Haase, A. Marban, T. Marinč, D. Neumann, T. Nguyen, H. Schwarz, T. Wiegand, D. Marpe, and W. Samek, “Deepcabac: A universal compression algorithm for deep neural networks,” *IEEE Journal of Selected Topics in Signal Processing*, 2020.

[26] M. Saini and U. Satija, “On-device implementation for deep-learning-based cognitive activity prediction,” *IEEE Sensors Letters*, 2022.

[27] “TensorFlow lite for microcontrollers,” https://www.tensorflow.org/lite/microcontrollers, accessed: 2022-2-28.

[28] A. Taivalsaari, T. Mikkonen, and C. Pautasso, “Towards seamless IoT device-edge-cloud continuum:: Software architecture options of IoT devices revisited,” in *Communications in Computer and Information Science*. Cham: Springer International Publishing, 2022, pp. 82–98.