Deployment of Artificial Intelligence Models into Edge Devices: A Tutorial Brief

Marek Zylinski $^1$, Amir Nassibi $^2$, Ildar Rakhmatulin $^2$, Adil Malik $^2$, Christos Papavassiliou $^2$, and Danilo P. Mandic $^2$

$^1$Imperial College London
$^2$Affiliation not available

October 31, 2023

Abstract

Artificial intelligence (AI) on an edge device has enormous potential, including advanced signal filtering, event detection, optimization in communications and data compression, improving device performance, advanced on-chip process control, and enhancing energy efficiency.

In this tutorial, we provide a brief overview of AI deployment on edge devices, and we describe the process of building and deploying a neural network model on a digital edge device. The primary challenge when deploying an AI model in circuits is to fit the model within the constraints of the limited resources as the restricted memory capacity on IoT circuits and the finite computational power impose constraints on the utilization of deep neural networks on IoT. We address this issue by elucidating methods for optimizing neural network models.

Part of the tutorial also covers the deployments of deep neural network into logic circuits, as significantly enhanced computational speed can be attained by transitioning the AI paradigm from neural networks to learning automata algorithms. This shift involves a move from arithmetic-based calculations to logic-based approaches. This transformation facilitates the deployment of AI onto Field-Programmable Gate Arrays (FPGAs).

The last part of the tutorial covers the emerging topic of in-memory computation of the multiply-accumulate operation. Transferring computations to analog memories has the potential to improve speed and energy efficiency compared to digital architectures, potentially achieving improvements of several orders of magnitude.

It is our hope that this tutorial will assist researchers and engineers to integrate AI models on edge devices, facilitating rapid and reliable implementation.
Deployment of Artificial Intelligence Models on Edge Devices: A Tutorial Brief

Marek Żyliński, Amir Nassibi, Ildar Rakhmatulin, Adil Malik, Christos Papavassiliou and Danilo P. Mandic

Abstract—Artificial intelligence (AI) on an edge device has enormous potential, including advanced signal filtering, event detection, optimization in communications and data compression, improving device performance, advanced on-chip process control, and enhancing energy efficiency. In this tutorial, we provide a brief overview of AI deployment on edge devices, and we describe the process of building and deploying a neural network model on a digital edge device. The primary challenge when deploying an AI model in circuits is to fit the model within the constraints of the limited resources as the restricted memory capacity on IoT circuits and the finite computational power impose constraints on the utilization of deep neural networks on IoT. We address this issue by elucidating methods for optimizing neural network models. Part of the tutorial also covers the deployments of deep neural network into logic circuits, as significantly enhanced computational speed can be attained by transitioning the AI paradigm from neural networks to learning automata algorithms. This shift involves a move from arithmetic-based calculations to logic-based approaches. This transformation facilitates the deployment of AI onto Field-Programmable Gate Arrays (FPGAs). The last part of the tutorial covers the emerging topic of in-memory computation of the multiply-accumulate operation. Transferring computations to analog memories has the potential to improve speed and energy efficiency compared to digital architectures, potentially achieving improvements of several orders of magnitude. It is our hope that this tutorial will assist researchers and engineers to integrate AI models on edge devices, facilitating rapid and reliable implementation.

Index Terms—Artificial intelligence, Edge computing, Learning automata, In-memory computing, Neural networks.

I. INTRODUCTION

A RTIFICIAL INTELLIGENCE (AI) enables the analysis of extensive volumes of data, recognition of patterns, and informed decision making, and in this way drive unprecedented technological advancements. At present, it is viable to deploy AI models on small, low-cost microcontrollers; AI on edge devices can be utilized for various applications, including advanced signal filtering, event detection (e.g., monitoring machinery and fault detection [1]), optimization in communications and data compression [2], enhancement of device performance [3], advanced on-chip process control.

Current solutions for AI on edge devices provide comprehensive pipelines [4] that facilitate the generation of low-level C code from a trained deep neural network. Edge computing has advantages in terms of response latency, bandwidth occupancy, energy consumption, security, and expected privacy [5].

The primary challenge in deploying an AI model on circuits is to fit the model into the limited resources. Name by, the constrained memory capacity on IoT circuits, coupled with the restricted computational power, constrains the application of deep neural networks in IoT scenarios [6]. Model optimizations, such as weight matrix scarification, pruning and weights quantization, are necessary to fit within the constraints of limited memory.

On the other hand, AI can be deployed on logic FPGA using learning automata algorithms instead of neural network algorithms. By synergistically harnessing the capabilities of learning automata algorithms and FPGA-based implementations, substantial enhancements in both computational performance and energy efficiency within AI systems can be attained, surpassing those achievable through conventional digital computation methods.

Moreover, there exists further potential to enhance both speed and energy efficiency by using analog memory circuits and in-memory computation concept, where vector-matrix multiplication calculations are performed in analog domain.

In this tutorial brief, we examine the integration of AI into edge devices. The subsequent sections of this paper are organized as follows. Section II provides an overview of the deployment of AI models directly onto digital microcontrollers. Section III describes the utilization of logic circuits, specifically FPGA, for AI deployment, while Section IV presents an overview and the current state of the emerging in-memory computation.

II. AI DEPLOYMENT ON DIGITAL DEVICES

Technological advancements made it viable to deploy AI models on small, low-cost microcontrollers. For instance, Fang et al. [7] ingeniously employed peripheral AI within a system-on-a-chip design to achieve real-time emotion recognition from the electroencephalogram (EEG) signals. Wang et al. [8] introduced a motor brain-computer inference tailored explicitly for low-power edge computing scenarios on the ARM Cortex-M family of microcontrollers. Mezzina and Venuto [9], developed an embedded Convolutional Neural Network (CNN) with a distinct architecture, that incorporates two 1D convolutional layers, which are reinforced by an intermediate batch normalization step to counter data covariate shift. Mezzina and Venuto tested the model on STM32 microcontrollers. In another study, Park et al. [10] implemented a fault detection model based on a long short-term memory (LSTM) on an industrial robot manipulator.
Deployment of AI on an edge device can be divided into five steps, as illustrated in Figure 1. First, the problem to be addressed must be defined, followed by the acquisition of an appropriate dataset. Given the data-intensive nature of deep neural networks, a substantial dataset is essential for effectively training complex models. In the third step, the AI model is trained and the network architecture is adjusted to obtain optimal performance while avoiding overfitting of the model [11]. Common frameworks such as Keras, TensorFlow and PyTorch are written in Python. Therefore, in the fourth step, the model needs to be translated into an edge platform language (C for microcontrollers or Java for Android devices). During this phase, the model is optimized to fit the resources constraints, involving reduction of memory footprint and computation complexity. In the final step, the AI model is integrated into the edge project.

Typically, the problem to be addressed is known beforehand and datasets are available. However, datasets can often be noisy and incomplete. The data pre-processing step aims to reduce data flaws and increase the accuracy of the resulting model. Pre-processing includes tasks such as data transformation, integration, cleaning, and normalization [12]. During this stage, efforts should be directed towards addressing class imbalances within the dataset.

When it comes to constructing and training AI models, three frameworks have garnered significant popularity: TensorFlow, supported by Google; Keras, a high-level inference library for TensorFlow; and PyTorch, developed by Meta. All three frameworks are open source and offer a wide spectrum of flexibility, spanning tasks from data processing and analysis to model development and training. Nonetheless, the onus lies with researchers to meticulously select the ideal tool for their specific endeavor. This curation involves striking a delicate balance: harnessing the capabilities of a framework while maximizing its potential for enhancement. Ultimately, the choice of framework often hinges on the precise nature of the task at hand.

PyTorch operates with a dynamic approach, executing code in a more procedural manner, whereas TensorFlow requires designing the entire model before running it within a session [13]. Chirodea et al. [14] pointed out that the choice of library does impact the system during both training and execution, though not significantly enough to declare one superior to the other. In their study, PyTorch exhibited faster training and execution times compared to TensorFlow but yielded slightly lower accuracy. Both frameworks displayed remarkably similar training trajectories. Ultimately, the decision regarding which framework to use is frequently guided by the specific demands of the given task.

Achieving Cross-platform interoperability within diverse deep learning frameworks feasible through the utilization of the Open Neural Network Exchange (ONNX) format. Both TensorFlow and PyTorch provide support for ONNX, facilitating the conversion of models into this standardized format. Subsequently, the AI model needs to transition from a Python environment to a platform dependent one. For this purpose, Google introduced TensorFlow Lite as a streamlined iteration of the TensorFlow library, tailored explicitly for mobile and embedded devices (Arduino, Android, and Linux operation devices). The TensorFlow Lite encompasses tools that enable the conversion of models into a simplified and compact version, optimized for edge devices [15]. This framework boasts an array of optimization tools, including quantization and model reduction, which empower seamless deployment on devices with limited resources. Simultaneously, PyTorch Mobile provides analogous capabilities to TensorFlow Lite, yet is intricately integrated into the PyTorch ecosystem.

The architecture and development environment of edge devices vary significantly from that of high-end computing servers, necessitating rigorous optimization of AI models. Edge devices are constrained by limited memory capacity and computational resources, while also demanding energy efficiency to minimize power consumption. Two primary strategies for optimizing the size of model weights are as follows: (a) Pruning, to remove of connections with small weights, typically those falling below a predefined threshold; (b) Quantization, to reduce the precision of weight values, for example from 32-bits to 8-bits.
Han et al. [15] demonstrated that for AlexNet (a CNN model for image recognition [17]), pruning reduces the number of weights by a factor of 10, while quantization further achieves compression rates of between 27 and 31, times without any loss in accuracy. Additionally, Liu et al. [18] reported that by incorporating both weight and temporal sparsity, the energy efficiency increased by approximately 46 times over the unpruned speech recognition model. The final step in the AI model deployment on a digital device involves integrating the model with the native code project. TensorFlow offers inference APIs for Android, iOS, and Linux, simplifying model deployment. For microcontrollers, the model must be adapted for platform-specific implementations of neural network kernels function. This platform-specific optimization enhances performance. Gupta et al. [19] optimized TensorFlow Lite source code for RISC-V processors, achieving an 8-fold reduction in executed instruction count compared to the baseline implementation.

ARM has released an open-source library known as the Cortex Microcontroller Software Interface Standard Neural Network (CMSIS-NN) specifically designed for Cortex-M processors. This library is aimed at optimizing neural network performance [19]. STMicroelectronics has introduced the X-CUBE-AI framework [20], designed to facilitate the integration of machine learning algorithms into STM32 series microcontrollers. A noteworthy inclusion within X-CUBE-AI is the provision of quantization tools, facilitating the conversion of models from floating point precision to fixed point precision. This feature empowers developers to harness AI capabilities within STM32 microcontrollers without necessitating an extensive background in model deployment. The process of integrating AI models with microcontroller libraries has been automated. Sailesh et al. [21] introduced an open-source framework with a complete pipeline for the deployment of a quantized model on a Cortex-M microcontroller using the CMSIS-NN library. In simpler tasks, machine learning methods can provide similar accuracy as deep networks while consuming less memory and computation time. In our paper [22] we presented a framework for automatic deployment of Support Vector Machine and Naive Bayes classifiers, trained using SciKit-learn library, onto ARM microcontrollers compatible with CMSIS-DSP library.

III. LOGIC CIRCUITS

The deployment of neural networks on the logic circuits can significantly improve the performance metrics and energy efficiency of deployed algorithms. One method is to deploy learning automata algorithms instead of neural networks algorithms [23], based on the Tsetlin machine [24] principles. The Tsetlin machine simplifies the traditional learning automata by utilizing discrete-step action updates through Tsetlin automata, which are finite automata employing linear tactics. These Tsetlin automata modify actions based on rewards for reinforcement and penalties for reduced confidence in performing actions. This discrete approach with linear updates allows for the use of robust propositional logic in framing the learning problem. Moreover, it streamlines the learning process which leads to efficient on-chip learning [23], [25].

The deployment of learning automata algorithms based on Tsetlin machines offers potential for the creation of energy-efficient artificial intelligence hardware due to logic-based structure of Tsetlin machines [23]. Switching the AI paradigm from neural networks to learning automata algorithms and shifting from arithmetic-based calculations to logic-based methods can lead to a significant improvement in computational speed. This transition facilitates the utilization of AI on Field-Programmable Gate Arrays (FPGAs) [5].

The deployment of Tsetlin Machines on FPGAs has been discussed by G. Mao et al. [26]. Their study addresses the challenges associated with implementing custom machine learning (ML) accelerators using asynchronous hardware designs, which offer significant energy efficiency advantages. Traditional simulation-based training involves extensive clock cycle simulations per epoch, which is impractical with low-level commercial electronic design automation (EDA) tools. Asynchronous ML hardware further complicates this task. FPGA prototyping has potential for addressing automation challenges, but the adaptation of existing clocked FPGAs for asynchronous circuits is difficult due to the violation of typical timing constraints.

The paper focuses on automating the design of asynchronous circuits on FPGAs, specifically addressing the problem of mapping critical timing constraints while preserving delay-insensitivity properties across various algorithms compatible with FPGA synthesis tools. The study involves a novel ML algorithm for synthesizing circuits for the Tsetlin machine. The generated circuits include the asynchronous clause generator and majority voting components, achieved through FPGA-based prototyping, showcasing the effectiveness of this design flow for larger designs. The paper also explores the decomposition of the State Transition Graph (STG) of a full Tsetlin Automaton (TA).

The Tsetlin machine can also be utilized for regression tasks. The Regression Tsetlin Machine (RTM), a novel extension of the Tsetlin Machine (TM), has been introduced by K. Darshana Abeyrathna et al. [27]. In their study, the RTM is tailored for nonlinear regression tasks involving continuous input and output. The TM, which primarily relies on bitwise operators, has demonstrated competitive pattern classification accuracy in various benchmarks. The RTM addresses regression problems by converting continuous inputs into a binary representation through thresholding. The propositional formula employed by the TM is transformed to produce an aggregated continuous output. The RTM modifies the inner inference mechanism and clause polarities, generating individual continuous outputs instead of distinct categories. The learning process involves linear activation probabilities based on regression error magnitude.

The study compares the RTM with established regression techniques, presenting its performance on five datasets. The RTM showcases either superior or comparable results compared to Regression Trees, Random Forests, and Support Vector Regression on datasets involving predictions for Dengue incidences, heating load, real win rate, and house price per unit area. The RTM demonstrates robust performance on both artificial and real-world datasets, including outperforming existing
regression techniques and providing reasonable extrapolation beyond training data output boundaries. In addition to RTM, in another study by Granmo et al., the Convolutional Tsetlin Machine (CTM) has also been introduced. The authors have presented the CTM as a novel approach to address the computational complexity and lack of interpretability associated with Convolutional Neural Networks (CNNs). The CTM leverages easy-to-interpret propositional logic clauses from the Tsetlin Machine (TM) to solve complex pattern recognition tasks. By using clauses as convolution filters, the CTM categorizes images while retaining interpretability. Each patch in the image is augmented with coordinates to make the clauses location-aware. During the learning phase, the CTM contrasts clauses against randomly selected patches that trigger the clause to evaluate as true. This approach achieves competitive accuracy across various benchmarks, including peak test accuracy of 99.4% on MNIST, 96.31% on Kuzushiji-MNIST, 91.5% on Fashion-MNIST, and 100.0% on the 2D Noisy XOR Problem. The CTM’s performance is comparable to other established methods such as simple CNNs, BinaryConnect, Logistic Circuits, and an FPGA-accelerated Binary CNN.

To deploy automata on FPGA, High-Level Synthesis (HLS) can be employed, as discussed in this webinar (https://webinars.sw.siemens.com/en-US/machine-learning-how-hls-can-be-l1/). Angstadt et al. emphasized that the designs constructed using HLS may exhibit underwhelming performance and may require significant optimization. They presented a novel HLS end-to-end framework for accelerating Boolean string kernel functions using FPGAs, which is accessible at https://github.com/kevinaangstadt/automata-synth.

Rahimi et al. introduced an open-source framework for automata processing on FPGAs named Grapefruit. This framework is accessible at https://github.com/gr rahimi/APSIm. Grapefruit is equipped with an integrated compiler featuring numerous parameters for automata simulation, verification, minimization, transformation, and optimizations. Rahimi et al. used Grapefruit to test automata applications on a Xilinx Virtex UltraScale+ device.

By synergistically harnessing the capabilities of learning automata algorithms and FPGA-based implementations, substantial enhancements in both computational performance and energy efficiency within AI systems can be achieved. Learning automata algorithms, characterized by their adaptability and learning prowess, introduces a dynamic decision-making framework that empowers AI systems to recalibrate their actions in response to evolving feedback signals and dynamic environmental conditions. When conjoined with FPGA-based platforms, which offer programmable hardware architectures and inherent parallelism, these algorithms stand to benefit from hardware acceleration and optimized execution.

IV. ENHANCING AI ACCELERATION THROUGH IN-MEMORY COMPUTATION

The in-analog-memories computation can significantly improve the performance and energy efficiency of matrix-vector multiplication (MVM) operation, which is a prevalent operation in neural networks. It is particular useful in dense layers where n-neurons with m inputs are built upon MVM operations. The output of a dense layer is given by:

\[ y = f_{\text{activation}} \left( \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \right) + \text{bias} \]

The computational complexity (as shown in Table 1) of a dense layer is tied to the dimensions of the input vector and number of neurons. In addition, within the von Neuman architecture, where computation units and memory are distinct entities, transmitting substantial data quantities can augment computation time due to added communication latency.

<table>
<thead>
<tr>
<th>Dense layer</th>
<th>Computational complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D CNN</td>
<td>(O(n_f n_k N_h \text{OutputSize}))</td>
</tr>
<tr>
<td>LSTM</td>
<td>(O(n_f n_k (4n_i + 4n_h + 3)))</td>
</tr>
</tbody>
</table>

TABLE 1

COMPUTATIONAL COMPLEXITY OF DIFFERENT NEURAL NETWORK LAYERS: \(n_f\) - NUMBER OF MULTIPLICATION OPERATIONS AS A FUNCTION OF LAYER PARAMETERS: \(N_k\) - NUMBER OF NEURONS IN LAYER, \(N_h\) - SIZE OF INPUT VECTOR, \(N_f\) - NUMBER OF FILTERS, \(N_k\) - FILTER’S KERNEL SIZE, \(N_i\) - SIZE OF INPUT SEQUENCE, \(N_h\) - NUMBER OF HIDDEN UNITS.

The computation time of MVM can experience a slight reduction through the adoption of an emerging in-memory computing strategy employing analog memory. In this approach, matrix weights are stored in memory conductance values \((Gij)\), the input vector \((x)\) is converted into voltages using Digital-to-Analog Converters (DAC), and the results of vector-matrix multiplication \((y)\) values are represented as currents \((I)\) at the output:

\[ I_k = \sum_{j=1}^{n} V_j G_{kj} \]

The current is measured using Analog-to-Digital Converters (ADC). In-memory computing of MVM is performed in a single operation, thereby rendering its computational complexity as \(O(1)\). This in-memory computation strategy is particularly suitable for applications where the matrix remains fixed, and only the input vector changes.

Since the MVM emerges as the performance bottleneck in such cases, in-memory computing improves energy efficiency, as the number of operations per Joule and performance per chip area. Potentially analog memories can achieve up to three orders of magnitude less energy compared to NVIDIA V100 GPU (potentially 100 TOPs/s/W compared to 100 GOPs/s/W).

There are several types of analog memories:

- Resistive RAM (RRAM) (Wong et al.), which uses modulation of conductive filaments between electrodes. RRAM is constructed based on insulators and various metal-oxide materials, that change their resistance under an electric field.
- Magnetic RAM (MRAM) (Matsukura et al.), which uses ferromagnetic switching between parallel or antiparallel spin polarization.
- Phase-Change Memory (PCM) (Burret et al.) is based on thermally driven reversible transitions between amorphous and crystalline states of a chalcogenide layer, leading to low and high cell conductance.

The RRAM has several advantages, including excellent stability, nanosecond read speed, prolonged data retention, and
Moving computation to the analog domain has some disadvantages. Analog signals are susceptible to noise interference, and in analog memories, telegraphic noise has been observed [42]. Furthermore, the introduction of additional DACs and ADCs constrains computation resolution to a few bits [43]. However, for neural networks, particularly in classification tasks, this resolution might suffice to attain acceptable network accuracy [44].

In addition, analog memories are susceptible to conductance drift, which entails changes in cell resistance over time. Furthermore, resistance is influenced by operational temperature variations, and device-to-device parameter variability has also been observed, stemming from chip nonidealities. To address these challenges, in-memory training algorithms have been introduced, such as chip-in-the-loop progressive model fine-tuning [45]. In this technique, training error is directly measured on the chip [46], and matrix weights are adjusted through reprogramming to mitigate the effects of imperfections.

To comprehensively investigate and explore the capabilities of in-memory computing devices within the realm of artificial intelligence, the IBM Analog Hardware Acceleration Kit was introduce [47]. This open-source python toolkit facilitates the simulation of analog device performance, including device-to-device systematic variations, forward pass output-referred noise and device fluctuations. The toolkit is seamlessly integrated with PyTorch, and offers a range of primitives for analog neural network modules, including dense, convolution and LSTM layers. This toolkit is available on GitHub at the following repository: https://github.com/IBM/aihwkit. The IBM Analog Hardware Acceleration Kit allows for the execution of high-level use case experiments and facilitates cloud-based execution through the AIHW Composer platform.

IBM is also actively involved in the development of hardware for in-memory computing. They have introduced the IBM Artificial Intelligence Unit [48] which is a specialized AI hardware accelerator. Additionally, their latest advancements include a 64-core mixed-signal in-memory chip [49], which can achieve an impressive throughput of 63.1 Tera operations per second and an energy efficiency of 9.76 Tera operations per second per Watt, particularly in the low-precision mode.

To further enhance the accuracy of in-memory computation, a mixed-precision architecture can be employed. This approach combines fast but low-precision analog memory with a high-precision von Neumann machine. The von Neumann machine applies a backward method to iteratively enhance the performance of machine learning models, especially deep neural networks, by dramatically improving computation speed and energy efficiency.

V. Conclusion

The deployment of AI models onto edge systems is a prerequisite for AI to play an indispensable role in shaping the future of technology and to integrate AI rapidly and reliably into their larger systems. The end-to-end frameworks like TensorFlow Lite and PyTorch Mobile provide comprehensive pipelines for generating edge device code directly from trained deep neural networks, and brings remarkable advancements in computation time. Transitioning from neural networks to learning automata algorithms and shifting from arithmetic-based calculations to logic-based approaches offers significant gains in computation speed. This transformation allows for the deployment of AI onto FPGAs.

Furthermore, the utilization of analog memory circuits presents the potential for even greater enhancements in speed and energy efficiency when compared to digital architectures. Within these circuits, memories execute the critical multiply-accumulate operation that profoundly impacts the performance of neural network models. Transferring computations to analog memories has the potential to improve speed and energy efficiency compared to digital architectures, potentially achieving improvements of several orders of magnitude.

References
