Bonseyes AI Pipeline—Bringing AI to You: End-to-end integration of data, algorithms, and deployment tools

MIGUEL DE PRADO, Haute Ecole Arc Ingenierie; HES-SO/Integrated Systems Lab, ETH Zurich, Switzerland
JING SU, School of Computer Science and Statistics, Trinity College Dublin, Ireland
RABIA SAEED, Haute Ecole Arc Ingenierie; HES-SO, Switzerland
LORENZO KELLER, Nviso, Switzerland
NOELIA VALLEZ, Universidad de Castilla—La Mancha, Spain
ANDREW ANDERSON and DAVID GREGG, School of Computer Science and Statistics, Trinity College Dublin, Ireland
LUCA BENINI, Integrated Systems Lab, ETH Zurich, Switzerland
TIM LLEWELLYNN, Nviso, Switzerland
NABIL OUERHANI, Haute Ecole Arc Ingenierie; HES-SO, Switzerland
ROZENN DAHYOT, School of Computer Science & Statistics, Trinity College Dublin, Ireland
NURIA PAZOS, Haute Ecole Arc Ingenierie; HES-SO, Switzerland

Next generation of embedded Information and Communication Technology (ICT) systems are interconnected and collaborative systems able to perform autonomous tasks. The remarkable expansion of the embedded ICT market, together with the rise and breakthroughs of Artificial Intelligence (AI), have put the focus on the *Edge* as it stands as one of the keys for the next technological revolution: the seamless integration of AI in our daily life. However, training and deployment of custom AI solutions on embedded devices require a fine-grained integration of data, algorithms, and tools to achieve high accuracy and overcome functional and non-functional requirements. Such integration requires a high level of expertise that becomes a real bottleneck for small and medium enterprises wanting to deploy AI solutions on the *Edge*, which, ultimately, slows down the adoption of AI on applications in our daily life.

In this work, we present a modular AI pipeline as an integrating framework to bring data, algorithms, and deployment tools together. By removing the integration barriers and lowering the required expertise, we can interconnect the different stages of particular tools and provide a modular end-to-end development of

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732204 (Bonseyes). This work is supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 16.0159.
Authors' addresses: M. de Prado, Haute Ecole Arc Ingenierie; HES-SO /, Integrated Systems Lab, ETH Zurich, Switzerland; email: miguel.deprado@he-arc.ch; J. Su, A. Anderson, D. Gregg, and R. Dahyot, School of Computer Science and Statistics, Trinity College Dublin, Ireland; emails: jing.su@tcd.ie, {andersan, david.gregg}@cs.tcd.ie, Rozenn.Dahyot@tcd.ie; R. Saeed, N. Ouerhani, and N. Pazos, Haute Ecole Arc Ingenierie; HES-SO, Switzerland; emails: {rabia.saeed, nabil.ouerhani, nuria.pazos}@he-arc.ch; L. Keller and T. Llewellynn, Nviso, Switzerland; emails: lorenzo.keller@nviso.ai, tim.llewellynn@nviso.ch; N. Vallez, Universidad de Castilla—La Mancha, Spain; email: Noelia.Vallez@uclm.es; L. Benini, Integrated Systems Lab, ETH Zurich, Switzerland; email: lbenini@iis.ee.ethz.ch.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2577-6207/2020/08-ART26 $15.00
https://doi.org/10.1145/3403572

AI products for embedded devices. Our AI pipeline consists of four modular main steps: *(i)* data ingestion, *(ii)* model training, *(iii)* deployment optimization, and *(iv)* the IoT hub integration. To show the effectiveness of our pipeline, we provide examples of different AI applications during each of the steps. Besides, we integrate our deployment framework, Low-Power Deep Neural Network (LPDNN), into the AI pipeline and present its lightweight architecture and deployment capabilities for embedded devices. Finally, we demonstrate the results of the AI pipeline by showing the deployment of several AI applications such as keyword spotting, image classification, and object detection on a set of well-known embedded platforms, where LPDNN consistently outperforms all other popular deployment frameworks.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: AI pipeline, deep learning, keyword spotting, fragmentation

ACM Reference format:
Miguel De Prado, Jing Su, Rabia Saeed, Lorenzo Keller, Noelia Vallez, Andrew Anderson, David Gregg, Luca Benini, Tim Llewellynn, Nabil Ouerhani, Rozenn Dahyot, and Nuria Pazos. 2020. Bonseyes AI Pipeline—Bringing AI to You: End-to-end integration of data, algorithms, and deployment tools. *ACM Trans. Internet Things* 1, 4, Article 26 (August 2020), 25 pages.
https://doi.org/10.1145/3403572

1 INTRODUCTION

Embedded Information and Communication Technology (ICT) systems are experiencing a technological revolution [8]. Embedded ICT systems are interconnected and collaborative systems able to perform smart and autonomous tasks and will soon pervade our daily life. The rapid spread of the embedded ICT market and the remarkable breakthroughs in Artificial Intelligence (AI) are leading to a new form of distributed computing systems where edge devices stand as one of the keys for the spread of AI in our daily life [73].

The rapid adoption of Deep Learning techniques has prompted the emergence of several frameworks for the training and deployment of neural networks. Frameworks such as Caffe [6], TensorFlow [48], and PyTorch [42] have become the most popular training environments by providing great flexibility and low complexity to design and train neural networks. These frameworks also support the deployment of the trained networks, though such deployment is cloud-oriented, which makes it impractical for resource-constrained devices operating on the *Edge*. Hence, a second generation of frameworks has come forth to cover such constrained requirements, e.g., TF Lite [49], ArmCL [4], NCNN [37], TensoRT [50], and Core ML [21]. Such deployment frameworks offer inference engines with meaningful compression and runtime optimizations to reduce the computational and memory requirements that neural networks demand and, thus, meet the specifications for mobile and embedded applications.

Developing embedded AI solutions for Computer Vision [63] or Speech Recognition [60] implies a significant engineering and organizational effort that requires large investments and development. Although the above-mentioned Deep Learning frameworks are powerful tools, a successful development of custom AI solutions on the Edge requires a fine-grained integration of data, algorithms, and tools to train and deploy neural networks efficiently. Such integration requires a high level of expertise to overcome both software and hardware requirements as well as the fragmentation of AI tools [11], e.g., massive and distributed computation for training while reduced and constraint resources for inference, hardship in portability between frameworks, and so on. All previous examples represent a real bottleneck for small and medium enterprises who would like to deploy AI solutions on the resource-constrained devices. Only large companies, e.g., Google [25] and Apple [16], can build end-to-end systems for the optimization of Deep Learning applications and are taking the lead of the AI industry [31]; see Figure 1.

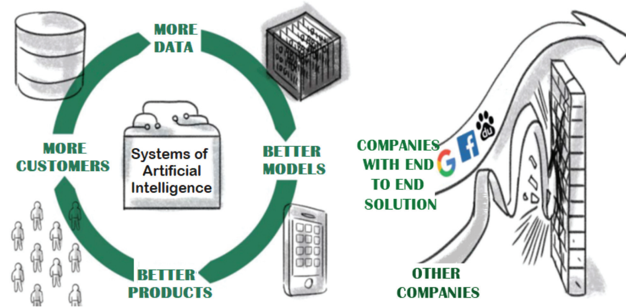


Fig. 1. AI pipeline and data wall. Companies with end-to-end solutions outperform the ones unable to follow the complete cyclic process.

As an alternative to monolithic and closed solutions, we form part of the Bonseyes project [59], a European collaboration to facilitate Deep Learning to any stakeholder and reduce development time. We propose an AI pipeline as a way to overcome the aforementioned technological barriers where data, algorithms, and deployment tools are brought together to produce an end-to-end system. We focus on the development of AI solutions for embedded devices, e.g., CPU Arm Cortex-A, embedded GPU, or DSPs, that feature sufficient computational power to run such solutions but need to be efficiently tailored to employ them in real-time applications. Our proposed AI pipeline provides key benefits such as the reusability of custom and commercial tools thanks to its dockerized API, and the flexibility to add new steps to the workflow. Thus, the contributions of our work are the following:

— We present a modular AI pipeline as an integrating framework to bring data, algorithms, and deployment tools together. By removing the integration barriers and lowering the required expertise, we open up an opportunity for many stakeholders to take up AI custom solutions for embedded devices.
— Our AI pipeline encourages the reusability of particular tools by interconnecting them on different stages to provide a modular end-to-end development. The AI pipeline consists of four main steps: *(i)* data ingestion, *(ii)* model training, *(iii)* deployment optimization, and *(iv)* IoT hub integration, which we illustrate giving different examples of AI applications.
— We integrate deployment framework (Low Power Deep Neural Network (LPDNN)) into the AI pipeline and present its lightweight architecture and deployment capabilities for embedded devices. Further, we show the deployment of several AI applications such as keyword spotting, image classification, and object detection on a set of embedded heterogeneous platforms. Finally, we evaluate LPDNN against a range of popular deployment frameworks where LPDNN consistently outperforms all other frameworks.

The article is organized as follows: in Section 2, we detail the state-of-the-art. In Section 3, the Bonseyes AI pipeline architecture is introduced. Section 4 describes the data ingestion process. Section 5 presents the training of neural networks. In Section 6, the deployment optimization is detailed. Section 7 introduces IoT hub integration and, finally, in Sections 8 and 9, we present the results and conclusions.

2 RELATED WORK

In this section, we introduce the state-of-the-art of AI pipelines and frameworks. We take a bottom-up approach starting from those works that propose single services, e.g., training or deployment, to then step up to those works presenting end-to-end solutions.

2.1 Deep Learning Platforms for High-Performance Computing (HPC)

Open-source Deep Learning frameworks have developed fast in recent years. These frameworks bring much convenience to AI project development as they provide the tools to design and train neural networks with available libraries and built-in optimizers, instead of coding from scratch. Here, we present a list of popular frameworks oriented for high-performance computing (HPC)-based AI applications.

- Google's TensorFlow (TF) is a powerful framework that provides APIs in multiple languages. TF is built for numerical computation using dataflow graphs in which nodes represent mathematical operations, and graph edges represent multi-dimensional data arrays [48].
- Caffe was developed by Berkeley AI Research (BAIR) and by community contributors [6]. As a pure C++ library, Caffe features expressive architecture, modularity design, and speed of training and inference. Command line, Python, and Matlab interfaces are provided.
- PyTorch is a scientific framework based on Torch, a C-based library with its scripting language LuaJIT [52]. PyTorch can be used as a native library and use popular Python libraries [42]. PyTorch employs dynamic computation graphs, which offers great flexibility.
- Microsoft Cognitive Toolkit (CNTK) is an open-source toolkit for commercial-grade distributed deep learning [19]. CNTK can be used as a library in Python, C#, or C++ programs.
- Apache MXNet is an open-source framework suited for flexible research prototyping and production [36]. It features hybrid front-end, distributed training, and rich language bindings.
- Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano [30]. Keras is designed for easy and fast prototyping.
- ONNX is an open ecosystem that provides an open source format for AI models [2, 3]. ONNX greatly improves interoperability between different deep learning frameworks and is becoming a standard for model exchange.

Our AI pipeline currently supports Caffe and PyTorch off-the-shelf and is compatible with any of the other frameworks by packaging and integrating them in the second step of the pipeline. Thus, we provide flexibility for the user to select the most suitable training framework.

2.2 Edge-Oriented Deep Learning Frameworks

HPC-oriented Deep Learning frameworks facilitate neural network training and deployment. However, they are not efficient for deployment on resource-constrained devices. To address the need of deployment optimization, a new set of frameworks has appeared to boost the runtime performance of trained models.

- TensorFlow Lite is a framework tailored for on-device inference [49]. TF Lite does not support the training of neural network directly; users need to convert a trained TF model into the TF Lite format. Thereafter, on-device inference can be carried out through a TF Lite Interpreter.
- Caffe2 is a lightweight deep learning framework focusing the deployment of AI on mobile devices [18]. Built on the original Caffe, it comes with C++ and Python API's providing speed and portability. Caffe2 is now a part of PyTorch [17].
- Android Neural Networks API (NNAPI) is a C API designed for deployment on Android devices [15]. NNAPI works as a base layer of functionality, which is directly used by an Android app. NNAPI supports and optimizes pre-trained models from TF or Caffe2.

- Arm Compute Library (CL) is a framework that collects low-level functions optimized for Arm CPUs and GPUs [4]. This library is built specially to accelerate image processing, computer vision, and machine learning tasks.
- Intel OpenVINO toolkit is a comprehensive toolkit for quickly developing computer vision applications [29]. It is designed to maximizing CNNs performance on Intel hardware.
- Tencent NCNN is a high-performance neural network inference framework optimized for mobile platforms [37]. Currently, a selection of Tencent apps integrate NCNN and make it a popular tool for phone app developers. Most commonly used CNN networks are supported.
- Alibaba Mobile Neural Network (MNN) is a lightweight inference engine [34]. MNN is built to accelerate inference on mobile and embedded devices on iOS and Android. Tensorflow, Caffe, and ONNX architectures are supported.
- Nvidia TensorRT is built on CUDA and it provides capabilities to optimize a trained model for a data center, embedded devices, or autonomous driving platforms [50]. TensorRT is widely compatible with neural network models trained in major frameworks.
- Tengine by Open AI Lab is a lite, high-performance, and modular inference engine for embedded devices [47]. Most Convolutional network operators are supported. Caffe, ONNX, Tensorflow, and MXNet models can be loaded directly by Tengine.
- Core ML is Apple's tool to create or convert machine learning models for iOS apps [21]. Core ML APIs enable on-device prediction with user data as well as on-device training. Besides, it supports models from Caffe, Keras, and conversions from TensorFlow and MXNet [20].

Our Bonseyes AI pipeline relies on LPDNN for the deployment on emdebbed devices. LPDNN provides optimized, portable, and light implementations for AI solutions across heterogeneous platforms. Besides, it can integrate third-party libraries or inference engines into its architecture, which makes it very flexible for custom platforms.

2.3 End-to-End AI Pipelines

In Sections 2.1 and 2.2, we have reviewed the frameworks for HPC- and edge-oriented AI-applications. However, these frameworks are oriented for developers and not as off-the-shelf products for end users. In this section, we introduce the advances of *AI as a Service* (or *AI pipeline*).

The *leaders* in providing AI services are Google, Amazon, and Microsoft. Google AI Platform [23] and AI Tools [24] offer a great convenience to build AI pipeline with TensorFlow as a backend. These platforms cover every step from data ingestion to model deployment, and they empower customer's AI applications for production. Amazon SageMaker [13, 14] is an end-to-end AI pipeline that features easy deployment of machine learning models on AWS at any scale. Microsoft Azure [32] is an enterprise-grade service designed to accelerate thevmachine learning cycle. Azure highlights a machine learning interpretability toolkit to explain model outputs during training and inference phases [35]. Moreover, Azure has wide compatibility with open source frameworks. Overall, these AI pipelines provide a fast solution to build up an AI application out of training models, but their deployment is mainly based on the cloud.

Next, we look into AI pipelines that provide edge-oriented deployment. Microsoft Azure IoT Edge [33] enables direct deployment of business logic on edge devices via Docker containers. However, this service is clearly in public preview as many tools are under development and limited to specific options. Amazon AWS IoT Greengrass [26] provides local inference on edge devices while depending on its cloud for management. They provide support for a range of edge devices, but the service is only available in a few regions [27] and under the amazon format.

Google AI Platform [23] also aims to edge deployment via TF Lite, which makes the pipeline very competitive and complete. On the other hand, the pipeline lacks flexibility to use or integrate

external tool and constrains the user to employ theirs. This fact brings incompatibility issues with user custom systems and drops in performance as we explain later in Section 8.2.3.

Apart from enterprise-level general purpose AI services, we also look into an edge-oriented AI service for vision tasks. Eugene is created as a suite of machine intelligence services toward IoT applications [75]. With components of data labeling, model training, deployment optimization, and IoT integration, Eugene is an excellent example of AI service. The authors show how to tailor deep neural networks to gain efficiency and introduce a scheduling algorithm to select the best network depth at runtime. However, Eugene's IoT service pipeline is limited in a few aspects. First, the scheduler is designed to tune the depth of ResNet, which raises questions about how effective this design works for other network architectures. Network reduction methods such as pruning may work in Eugene's pipeline but with significant refactoring. Besides, very little information is available on the supported hardware platforms and the deployment inference engine. Neither did the authors mention system integration or support issues, which can be crucial for the successful deployment of AI service. Bonseyes comes in to address all these drawbacks. A detailed explanation of our proposed AI pipeline is provided in the following sections.

3 BONSEYES AI PIPELINE ARCHITECTURE

We introduce the fragmentation and obsolescence of tools and propose the pipeline as a solution to create end-to-end environments for deep learning solutions.

3.1 Fragmentation and Obsolescence of Tools

The rapid evolution of technology in the field of machine learning requires companies to update their environments continuously. Being able to upgrade to the latest algorithm or technology is critical to maintaining the leadership in their field. Besides, upgrading existing data processing tools is a very demanding task. Differences in library versions, runtime environments, and formats need to be handled. These challenges can consume a significant amount of time and introduce hard-to-solve bugs. Moreover, companies are often interested in acquiring the off-the-shelf code or complete pipelines to reduce their costs and to acquire advanced technology that they would not be able to develop in-house. Integrating externally generated code is an even more challenging task, as it is very likely that what is acquired is not compatible with the existing pipeline.

3.2 Pipeline as a Solution

One of the main objectives of the Bonseyes AI pipeline framework is to alleviate this problem and link fragmented environments by defining a way to split the AI pipeline in reusable components, define interfaces for interoperability, and provide a documented reference implementation of the interfaces to accelerate development. The main goals of the pipeline are twofold; on one side, to isolate the different parts of the pipeline along with their dependencies, and on the other hand, to insert the glue code that combines them explicitly. The Bonseyes AI pipeline framework is structured around three concepts:

- **Tool:** A software component that performs a specific function in the pipeline. An example of a tool is a software that is capable of training a model from a training dataset.
- **Artifact:** The product of the execution of a tool, e.g., models, datasets. It can be an output of the pipeline or an intermediate result that is processed by other tools.
- **Workflow:** A declarative pipeline description that lists the tools that need to be used and the artifacts that need to be created. For example, a workflow may import a dataset and use it to train a model.

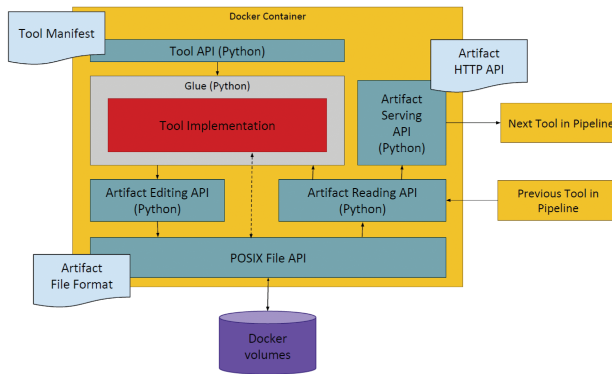


Fig. 2. Docker container including all submodules and interfaces.

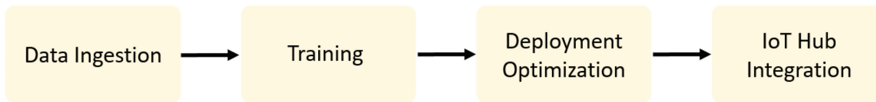


Fig. 3. Bonseyes AI Pipeline: Data ingestion, Training, Deployment Optimization, and IoT Hub Integration.

The AI pipeline relies on Docker containers [9] to package all software dependencies needed to run a tool; see Figure 2. Besides, a high-level HTTP API is defined to control the workflows and tools. The AI pipeline provides a collection of standard formats that define on-disk serialization and HTTP REST API for them. In addition, the user can define additional formats with different interfaces to adapt to new designs.

3.3 End-to-End AI Pipeline

Based on the previous concepts, we propose an end-to-end AI pipeline to develop and deploy Deep Neural Network solutions on embedded devices. We propose a modular AI pipeline architecture which contains four modular main tasks: *(i)* Data ingestion, *(ii)* Model training, *(iii)* Deployment on constraint environment, and *(iv)* IoT hub integration; see Figure 3. The four main tasks may be decomposed into several steps that can be later accomplished by a set of tools. The number of tools for each step is variable since it is possible to have several of them with the same purpose but using different frameworks or dealing with data from various AI challenges such as image classification, object detection, or keyword spotting (KWS). Moreover, some subtasks may also be optional. For example, Data Partitioning may not be required if the raw data was already partitioned.

Fragmentation quickly occurs between these tasks as each one needs and employs specific formats to operate. Therein, artifacts come into play as they represent the way by which data can be stored and exchanged between tools. Artifacts must follow a definition to standardize them for each problem type. Hence, tools define their inputs and outputs according to these artifact definitions. This fact makes tools with the same input and output definitions to be easily interchangeable and leads to modularized and reusable pipelines.

The end-to-end pipeline can be executed following a workflow definition from collecting the data until IoT hub integration. A workflow specifies the steps that are required to obtain the final result. This step involves describing which tools are used and in which order, and how the output artifacts of one tool are used to feed another one.

In the Bonseyes framework, four different DNN-based AI applications and their corresponding artifact definitions are currently available for the most common AI challenges, although it is

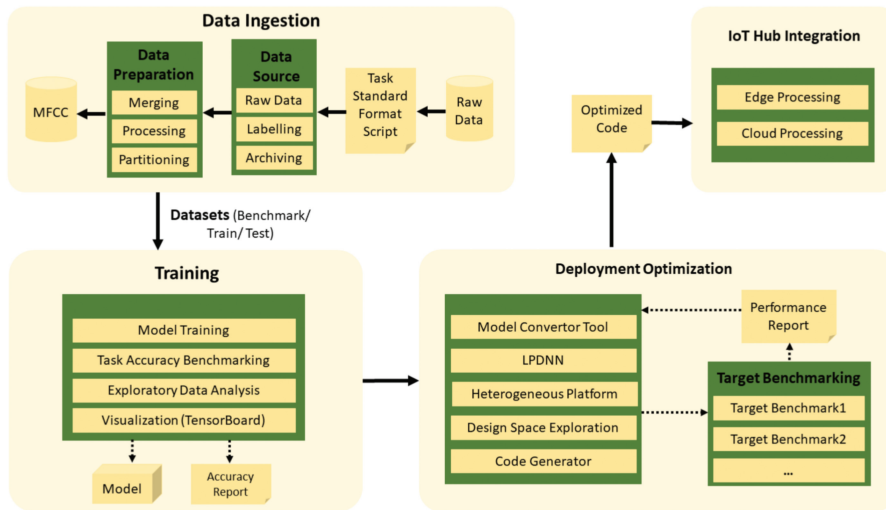


Fig. 4. Bonseyes AI Pipeline Architecture and functional units.

possible to add more if needed. These are image classification, face recognition, keyword spotting, and object detection. In the following sections, we introduce and explain the four main steps of the Bonseyes AI pipeline showing different AI applications on each step.

4 AI DATA INGESTION (1/4)

Data Ingestion is the first step in the AI pipeline; see Figure 4. It entails the complete process from acquiring the raw data to have it prepared for model training. The first step of this process involves parsing the raw data and representing it in a standardized format to be used in the next steps according to the problem definition, i.e., classification, KWS, and the like. Thus, audio or images are stored together with their annotations. This step is crucial since data is an essential part of the AI pipeline and is shared across multiple stages. However, standardizing it is a tedious task due to the variety of file formats and the lack of a consistent annotation representation. This is the reason why the Bonseyes AI pipeline API provides a set of mechanisms to reduce the efforts when importing new data. Collecting it from the internet or a local hard drive is already supported by specifying only where the resource is located. Moreover, API also manages how the resulting dataset artifact is stored.

After importing the data into the pipeline, a processing step may be carried out to prepare it for model training. This process includes operations such as image resizing, normalization, or face alignment, among others. This process can be accomplished by more than one step, which may vary according to the data, the model to be trained, or even the application. Finally, the dataset may also require to be partitioned into training, validation, and test sets depending on the needs. This partitioning is not necessary if it was already done in the raw data, but a large number of public datasets are stored in a single compressed file that requires further processing.

To illustrate this step of the AI pipeline further, we show an example of data ingestion for a KWS application. Automatic Speech Recognition (ASR) is a classical AI challenge where data, algorithms, and hardware must be brought together to perform well. Speech source with regional accents has high phonetic variance even in the same word, and expressions in dialects may exceed the standard transcription dictionary. These scenarios suppose a great challenge as retraining a recognizer system with a large amount of real-user data becomes necessary before

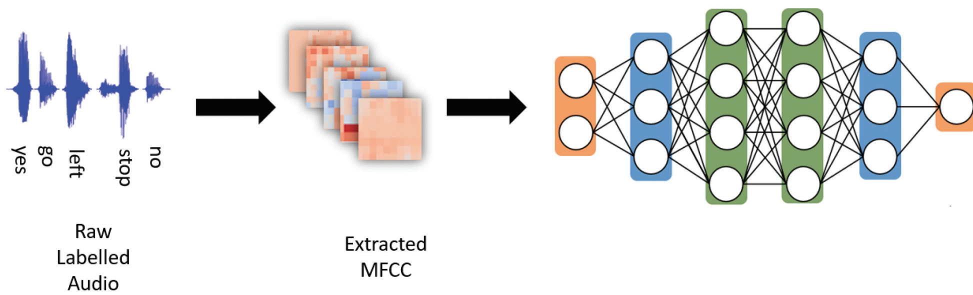


Fig. 5. Keyword Spotting (KWS). Process of recognizing predefined words from a speech signal. The process involves a feature extraction step before the signal is fed into a DNN model. The model then compares the input to the predefined words and detects their presence [76].

deploying the system. Therefore, only stakeholders that can acquire custom data and train on it can overcome such a challenge. Keyword Spotting, a particular case of ASR, is the process of recognizing predefined words from a speech signal, which, in many cases, serves as a "wake-up" signal to initiate a larger service; see Figure 5.

We have set the *AI Data Ingestion* step to download the complete Google Speech Commands dataset [1] containing WAV files. The raw data is downloaded from the provider, parsed, and standardized into HDF5 format files to comply with reusability and compatibility. Because it is a labeled dataset, we skip labeling in the pipeline. Finally, data are partitioned into training, validation, and benchmarking sets. Regarding the training set, the import tool stores each WAV file together with sample ID and class label in one data tensor. This single data tensor is used as input to the next tool. Validation dataset and test dataset can be imported in the same way.

However, raw audio samples are difficult to process. A human's voice is generated from vocal cord vibrations, which varies with different sounds. Spectral features of speech signals are more representative than sound waveform in speech recognition. A human's cochlea is more capable of discerning low-frequency signals than high-frequency signals. To model cochlea characteristics, we apply the Mel scale over the power spectrum, and it shows higher resolution on low-frequency bands. After discrete cosine transform of Mel log powers, we get Mel-frequency spectral coefficients (MFCCs). MFCC is a widely used audio feature format for speech recognition [74].

Given the flexibility and modularity of the AI pipeline, we have integrated the *MFCC generation* extraction process into the *Data ingestion step* as pre-processing option. This process employs an audio feature generation tool, which produces MFCC features of each audio sample and saves them together with the keyword labels in an HDF5 file. The audio feature generation has been accomplished by leveraging the Librosa library [71]. Since audio files in the Google Speech Command dataset are recorded in 16kHz sampling rate, a moving frame of 128ms length and 32ms stride generates 32 temporal windows in one second. We apply 40 frequency bands per frame, and the output MFCC features of one-second long audio sample are in a 40×32 tensor. The generated MFCC features (training set and test set) can also be reused for training and benchmarking tools of new models.

5 TRAINING (2/4)

Training and accuracy benchmarking follows the data ingestion step of the AI pipeline. The training step usually consists of a tool that requires a training dataset (and usually a validation dataset) and produces a model. Since data are standardized, the same training tool can be used with different datasets from the same problem type. On the other hand, the benchmarking tool requires a

test dataset and a trained model to produce an accuracy report as output. Visualization tools are often used at this point to help the training process.

In general, an AI pipeline may require a different training framework, another versions of the same one or even several configurations to solve different AI challenges. These differences often lead to software configuration problems when a user tries to solve several AI challenges in the same machine. In this respect, the main benefit of using the Bonseyes AI pipeline for training is the encapsulation of all the needed software and dependencies inside the Docker container that runs the tool. Thus, it is possible to have tools with different deep learning frameworks or different configurations without interfering with each other since they run in an isolated environment. Nonetheless, training procedures are defined in separate training tools to facilitate tools reusability and modularity. Bonseyes includes off-the-shelf Caffe and PyTorch frameworks, but any other framework could also be included.

The flexibility of the dockerized training pipeline allows us to create additional tools that perform model optimizations during training, such as quantization or sparsification. In this case, the new model can be trained from scratch using these optimizations or using a pre-trained model with a new training dataset to optimize and adapt the final model.

To describe this step of the AI pipeline further, we detail an example of training for a KWS application on Caffe. Two different KWS neural network architectures have been created to cross-compare accuracy, memory usage, and inference time—Convolutional Neural Network (CNN) and Depth-wise Separable CNN (DS-CNN). Since the Long Short-Term Memory (LSTM) -based models do not show a significant advantage of accuracy over DS-CNN, memory footprint, and inference latency [76], we only develop two types of CNN models in this article. In the following sections, we introduce the training configuration that we have followed, the networks architectures that we have developed and a final optimization step: a Neural Architecture Search.

5.1 Training Configurations

All training tools generate both the training model and the solver definition files automatically. We have trained the CNN and DS-CNN models using Bonseyes-Caffe [5]. These tools import the output generated in the MFCC generation step using the training dataset where the extracted MFCC features and labels are packed all together into an HDF5 file. Training is carried out with a multinomial logistic loss and Adam optimizer [66] over a batch of 100 MFCC samples (since our input sample size is 40×32, we opt to use a relatively big batch size). The batch size and number of iterations are specified in the workflow files that control the execution of the tools. Each model is trained for 40K iterations following a multi-step training strategy. The initial learning rate is 5×10^{-3}. With every step of 10K iterations, the learning rate drops to 30% of the previous step.

Further, a benchmarking tool has been built to validate the trained models. This tool takes two inputs: the MFCC features generated from the test dataset (HDF5 file) and the trained model. The inference is performed, and the predicted classes are compared with the provided ground truth, and the results are stored in a JSON file.

5.2 Network Architectures

To build a KWS model with a small footprint, we have started off by modeling from a 6-layer convolutional neural network. The CNN architecture contains six convolution layers, one average pooling layer, and one output layer. More specifically, each convolution layer is followed by one batch normalization layer [64], one scale layer, and one ReLU layer [72]. Output features of the pooling layer are flattened to one dimension before connecting with the output layer. CNN model architecture is explained per layer in Table 1. We can see that the first convolution layer uses a non-square kernel of size 4×10. This kernel maps on 4 rows and 10 columns of an MFCC input

Table 1. Initial Architectures of CNN and DS-CNN Networks

Model	$conv1^*$	$conv2^*$	conv3	conv4	conv5	conv6	TOP-1	MFP_{ops}	Size (KB)
CNN	4×10, 100	3×3, 100	3×3, 100	3×3, 100	3×3, 100	3×3, 100	94.2%	581.1	1,832
DS-CNN	4×10, 100	3×3, 100	3×3, 100	3×3, 100	3×3, 100	3×3, 100	90.6%	69.9	1,017

Filter shape is defined as $k_h \times k_w$, M where k_h and k_w are kernel height and kernel width, and M is the number of output channels. * *conv1* has stride shape 1×2 and *conv2* has stride shape 2×2. All other convolutional layers have stride shape 1×1.

image, which, in turn, refers to 4 frequency bands and 10 sampling windows. A 4×10 kernel has an advantage of capturing power variation in a longer period and narrower frequency bands. This setting complies with Ref. [76]. In the following convolution layers (Conv2 ~ Conv6), 3×3 square kernels are applied.

DS-CNN was introduced by Ref. [62], and we also apply it for KWS. DS-CNN improves the execution of standard CNN as it reduces the number of multiplication operations by dividing a standard convolution into two parts: depth-wise and point-wise convolution. In this study, we substitute the standard convolutional layer from the CNN model by depthwise separable convolutions. Thus, a DS-CNN model has one basic convolution layer (Conv), five depthwise separable convolution layers (DS_Conv), one average pooling layer, and one output layer. Both parts of the DS_Conv layer, depthwise convolution, and pointwise convolution are followed by one normalization layer, one scale layer, and one ReLU layer. The first convolution layer is Conv instead of DS_Conv. This setting follows the original MobileNet [62], and it helps to extract a 2D structure from the input.

5.3 Neural Architecture Search

Although the manually designed CNN network achieves 94.2% prediction accuracy (Table 1), it is not guaranteed to be the best choice for deployment. Applications on embedded devices are sensitive to energy consumption, and a KWS model is required to respond with low latency. The exact inference latency needs to be gauged on the hardware test, but we can hold an assumption that a model with a smaller number of floating point operations (FP_{ops}) will be faster and be more energy efficient. Joint optimization of model accuracy and FP_{ops} is challenging because these two metrics cannot be implemented in one loss function. A solution of model selection comes from Neural Architecture Search (NAS). NAS is a process of automating network architecture engineering [58]. Alternatively speaking, NAS offers a method to automatically explore high-dimensional network hyperparameter space and populate network candidates with good prediction accuracy.

Elsken et al. categorized the tasks of NAS in three folds, including search space, search strategy, and performance estimation strategy [58]. In this work, we have applied the popular search strategy Tree-structured Parzen Estimator (TPE) [54]. The performance estimation strategy that we have followed is a multinomial logistic loss function, and we have used Microsoft NNI library [55] for the NAS experiment. The main challenge comes from search space setup as the number of possibilities grows exponentially. We first explore optimization parameters (including learning rate, batch size, weight decay strategy, and training iterations), and then we freeze a set of optimization settings to explore the target network parameters: kernel height k_h, kernel width k_w, and output channel number M of each convolution layer. When the network parameter search space is explored, a selection of CNN architectures is available. These architectures are populated in a two-dimensional space of model accuracy and FP_{ops}. We use Pareto frontier [70] to select candidate architectures. If one candidate architecture is on Pareto frontier, it means no other candidate can be more accurate without paying a higher computational cost, and vice versa. We've created

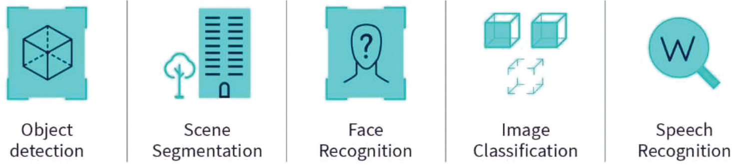


Fig. 6. AI application types.

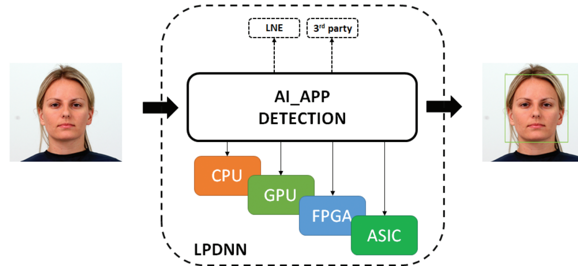


Fig. 7. Low-power Deep Neural Network (LPDNN) framework. LPDNN enables the deployment and optimization of AI applications on heterogeneous embedded platforms such as CPU, GPU, and ASIC.

an integrated solution of neural architecture search and Pareto frontier selection for the aim of performance-oriented model selection. More details can be found in Ref. [53].

6 DEPLOYMENT OPTIMIZATION (3/4)

After training a Deep Neural Network (DNN), the next step in the Bonseyes AI pipeline is the deployment of such DNN on embedded devices. The support and optimization for the deployment of DNNs rely on LPDNN. LPDNN is an enabling deployment framework that provides the tools and capabilities to generate portable and efficient implementations of DNNs for constrained and autonomous applications such as Healthcare Auxiliary, Consumer Emotional Agent, and Automotive Safety and Assistant. The main goal of LPDNN is to provide a set of AI applications, e.g., object detection, image classification, speech recognition (see Figure 6), which can be deployed and optimized across heterogeneous platforms, e.g., CPU, GPU, FPGA, DSP, ASIC (see Figure 7). In this work, we integrate LPDNN into the AI pipeline and present its lightweight architecture and deployment capabilities for embedded devices. Further, we show the deployment of KWS, image classification, and object detection applications on a set of embedded platforms while comparing to other deployment frameworks.

6.1 LPDNN Architecture

One of the main issues of AI systems is the hardship to replicate results across different systems [12]. To solve that issue, LPDNN features a full development flow for AI solutions on embedded devices by providing platform support, sample models, optimization tools, integration of external libraries, and benchmarking at several levels of abstraction; see Figure 8. LPDNN's full development flow makes the AI solution very reliable and easy to replicate across systems. Next, we explain LPDNN architecture by describing the concept of AI applications and the LPDNN Inference Engine.

6.1.1 AI Applications. AI applications are the result of LPDNN's optimization process and the higher level of abstraction for the deployment of DNN on a target platform. They contain all

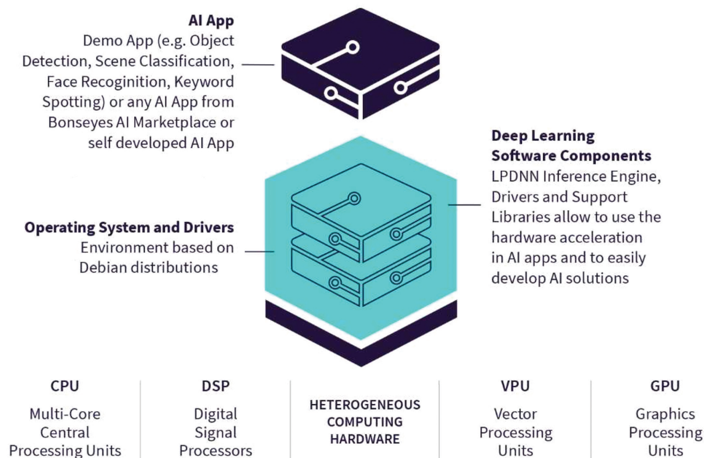


Fig. 8. LPDNN full stack. LPDNN provides a complete development flow for AI solutions for embedded devices by providing platform support, sample models, optimization tools, and integration of external libraries.

the necessary elements or modules for the execution of DNN. The minimum number of modules that an AI application may contain are two: pre-processing and inference engine modules. More modules can be included to extend the capabilities of the AI application, e.g., connection of several neural networks in a chain fashion.

Furthermore, AI applications contain a hierarchical but flexible architecture that allows new modules to be integrated within the LPDNN framework through an extendable and straightforward API. For instance, LPDNN supports the integration of 3rd-party self-contained inference engines for AI applications. The AI application could select as a backend LPDNN Inference Engine (LNE) or any other external inference engine integrated into LPDNN, e.g., Renesas e-AI [46], TI-DL [51]. The inclusion of external engines also benefits LPDNN as certain embedded platforms provide their own specific and optimized framework to deploy DNNs on them.

6.1.2 LPDNN Inference Engine. In the heart of LPDNN lies the Inference Engine (LNE), initially introduced in Ref. [56], which is a code generator developed within the Bonseyes project to accelerate the deployment of neural networks on resource-constrained environments [56]. LNE can generate code for the range of DNN models and across a span of heterogeneous platforms. LNE supports a wide range of neural network models as it provides direct compatibility with Caffe [6]. In addition, LNE supports ONNX format [3], which allows models trained on any framework to be incorporated into LPDNN providing they can export to ONNX, e.g., PyTorch or TensorFlow. The network model (Caffe, ONNX) is converted to an internal computation graph in a unified format. At this point, several steps such as graph analysis for network compression and memory allocation optimization are performed. LNE provides a plugin-based architecture where a dependency-free inference core is complemented and built together with a set of plugins (acceleration libraries) to produce optimized code for AI applications given a target platform; see Figure 9. Thus, each layer is assigned an implementation among the available computing libraries. Finally, layout conversions are performed in the code generation process to assure the compatibility of the network execution. More details of LNE are provided in Section 6.2.

6.1.3 Heterogeneous Computing Support. One of the main factors for LPDNN's adoption is performance portability across the wide span of hardware platforms. The plugin-based architecture maintains a small and portable core while supporting a wide range of heterogeneous platforms,

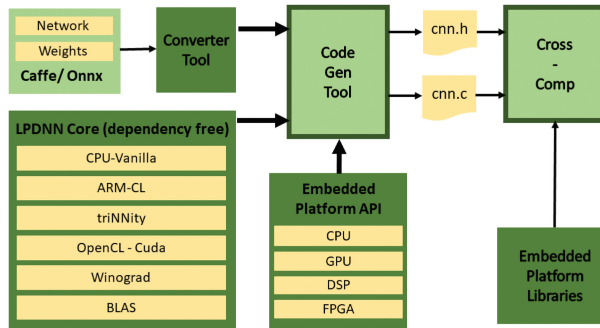


Fig. 9. LPDNN Inference Engine (LNE) [56]. Plugins can be included for specific layers, which allow a broad design space exploration suited for the target platform and performance specifications.

including CPU, GPU, DSP, and FPGA. One of the objectives of Bonseyes is to provide full support for reference platforms by providing:

— Board Support Package (BSP) containing OS images, drivers, and toolchains for several heterogeneous platforms.
— A dockerized [9] and stable environment, which increases the reliability by encouraging the replication of results across platforms and environments.
— Optimization tools and computing libraries for a variety of computing embedded platforms that can be used by LNE to accelerate the execution of neural networks.

6.2 Inference Optimizations

LPDNN contains several optimization tools and methods to generate efficient and light implementations for resource-constrained devices.

6.2.1 Network Compression. LNE supports folding of batch normalization and scale layers into the previous convolution or fully connected layer [22] at compilation time. This optimization provides a reduction in memory size, as the weights of the folded layers are merged, and an acceleration during the inference as the execution of the folded layers are skipped. In addition, some plugins in LNE support fusion of activation layers into the previous convolution at runtime. Fusing activation layers halves the number of memory accesses for a data tensor passing through the combination of convolution + activation layer.

6.2.2 Memory Optimization. LNE analyzes the computation graph for memory usage and optimizes the overall allocation by sharing the same memory between layers that are not active concurrently (similar to temporary-variables allocation techniques used in compilers). Besides, LNE enables, when possible, in-place computation: layers share the same memory for input and output tensors.

6.2.3 Optimized Plugins. Acceleration libraries can be included as plugins in LNE for specific layers to accelerate the inference by calling optimized primitives of the library, e.g., BLAS, ARM-CL [4], NNPACK [38] cuDNN. Besides, several libraries can be combined or tuned to boost the performance of the neural network execution. Certain computing processors may provide better performance for specific tasks depending on the model architecture, data type, and layout. Thanks to LNE's flexibility, it is possible to select in what type of processor to deploy each layer and what backend to use; see Figure 10. Therefore, we find a design exploration problem—named Network Deployment Exploration—when we aim to optimize metrics such as accuracy, latency, or memory.

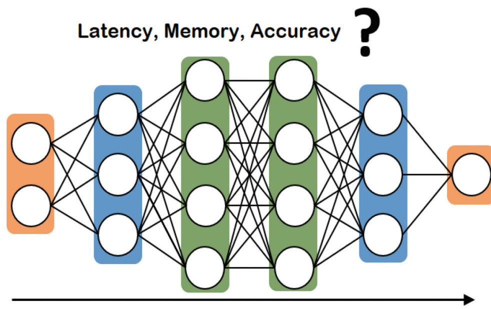


Fig. 10. Design space. Optimization can be achieved by deploying each layer on different processing systems based on their capabilities regarding latency, memory, or accuracy. Colors match Figure 7.

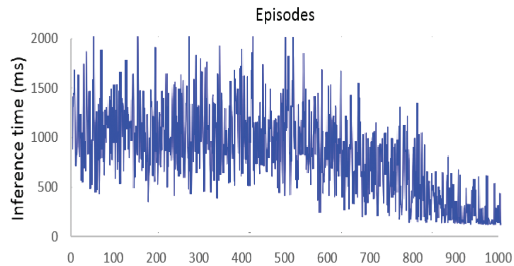


Fig. 11. RL optimization [57]. In the first stage (500 episodes), the agent searches through the design space to learn about the environment. In the second stage, the agent starts slowly selecting those implementations that yield a faster inference.

6.2.4 Network Deployment Exploration. To solve this space exploration problem, we propose an automatic exploration framework called QS-DNN, which has been previously introduced in Ref. [57]. QS-DNN implements a learning-based approach, based on Reinforcement Learning [68], where an agent explores through the design space, e.g., network deployment across heterogeneous computing processors, and empirically finds an optimal implementation by selecting optimal layer configuration as well as cross-layer optimizations. The network deployment space can be defined as a set of states $\mathcal{S}$, i.e., layer representations. The agent explores such space by employing a set of actions $\mathcal{A}$, i.e., layer implementations, with the final aim of learning a combination of primitives (from LNE's acceleration libraries), that speeds up the performance of the DNN on a given platform; see Figure 11. QS-DNN has been integrated into LPDNN and is tightly coupled with LNE.

6.2.5 Network Quantization. Neural networks can be further compressed and optimized through approximation [65]. A quantization exploration tool has been integrated within LPDNN, which analyzes the sensitivity of each layer to reduced-numerical precision, e.g., int8 [43]. The tool yields a set of quantization parameters (scale values) that are applied to the weight and output tensors of each layer to minimize the loss in accuracy when using quantized methods. Thanks to LPDNN's benchmark architecture, it is possible to obtain latency measure per layer as well as the accuracy of the network for specific quantization parameters.

6.3 Deployment of AI Applications

To demonstrate LPDNN's optimizations and capabilities, we compare LPDNN with several deployment frameworks for KWS, image classification, and object detection applications on a set of embedded platforms. The reader is referred to Section 8.2.3 for the analytical results of the following comparison scenarios.

6.3.1 LPDNN vs. Caffe (KWS). KWS applications are often used as a "wake-up" signal for larger systems due to their low energy consumption, which makes them affordable for always-on modes. Based on this motivation, we focus on deploying the KWS models on a single core of a multi-core CPU platform while leaving available other cores for more consuming tasks or powered-off if not needed. We have chosen the Nvidia Jetson Nano [39], which features a quad-core ARM Cortex A-57. Since the KWS models have been trained on Caffe—see Section 5—we compare the deployment of LPDNN against Caffe for such models.

6.3.2 LPDNN vs. PyTorch (Object Detection). Object detection applications must detect, identify, and localize multiple subjects in the input image. In this work, we show a body-pose estimation model, a case of object detection, where the model has to identify and estimate the skeleton of the people. We take two pre-trained resnet-based models, which have been initially trained by the work of Ref. [67] on PyTorch. We export the model to ONNX and import it in LPDNN where we can leverage LPDNN's optimization and, thus, compare PyTorch deployment with LPDNN's. As body-pose estimation is a very computationally intensive task, we evaluate a heterogeneous implementation for which we propose a last generation automotive platform, the Nvidia Jetson Xavier, which features 8-core ARM v8.2 and a 512-core Volta GPU [40].

6.3.3 Comparison with Embedded Deployment Frameworks (Image Classification). We further compare LPDNN with state-of-the-art deployment frameworks, which provide inference engines for resource-constrained devices. Image classification is a classical AI application where an image needs to be classified according to its content. We show the deployment of a representative range of models for the ImageNet challenge [28] on two embedded platforms: The Raspberry 3b+ [44] and Raspberry 4b [45] featuring a quad-core ARM Cortex-A53 and Cortex-A72, respectively. We especially evaluate several network topologies for resource-constrained devices, e.g., Mobilenets, Squeezenet, but also reasonably large networks like Resnet50, which allows us to show how the inference engines adapt to the requirements of each network on the selected target platform. We evaluate the following deployment frameworks: *(i) Caffe-SSD* [7], *(ii) ArmCL-DEV20191107* [4], *(iii) MNN-0.2.1.5* [34], *(iv) NCNN-20191113* [37], *(v) Tengine-DEV20190906* [47], *(vi) TFLite-2.0.0* [49], and *(vii) LPDNN-20191101.*

7 IOT HUB INTEGRATION (4/4)

IoT hub integration is the last step of the AI pipeline. The rise and spread of highly distributed embedded systems bring about scalability issues in the deployment and integration of such systems. Generally, AI applications run on systems that are part of a broader application and service ecosystem that support a value chain. The inclusion of AI enabled systems into an IoT ecosystem is of particular interest when it comes to resource-constrained systems to alleviate computing and memory demands. The heterogeneous nature of IoT and embedded devices not only in terms of HW capabilities, i.e., computation, memory, and power consumption, but also in terms of SW support supposes a great challenge to build a global ecosystem. Further, several other problems, such as security and privacy, need to be addressed in a distributed platform where information is continuously shared.

7.1 IoT Scenarios

The IoT hub integration in the AI pipeline backs two main scenarios that are relevant for low-power and autonomous environments, e.g., command voice in medical health care, pedestrian detection in automotive, and so on—

- **Edge-processing:** Data are processed on the embedded device, and results are retrieved and stored in the cloud for further processing and exploitation; see Figure 12(a).
- **Cloud-processing:** Part of the computation steps in the embedded system is delegated to server- or cloud-based AI engines. This scenario can be of great interest for constrained systems when their resources are not able to offer enough computational power to execute AI algorithms; see Figure 12(b).

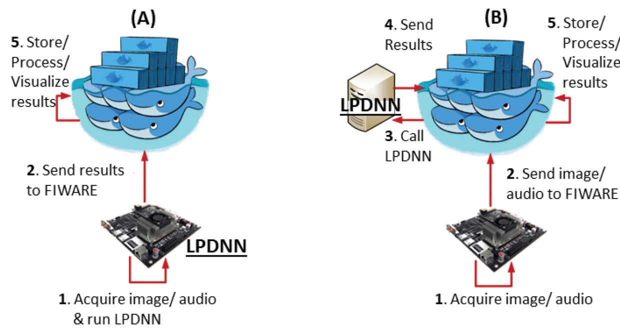


Fig. 12. Bonseyes IoT tools. Bonseyes relies on the FIWARE platform [10] for the implementation of the integration pipeline. It backs two main scenarios—(A) Edge-processing and (B) Cloud-processing.

7.2 Bonseyes IoT Tools

Bonseyes relies on the FIWARE platform [10] for the implementation of the IoT hub integration. Both *edge-processing* and *cloud-processing* scenarios are supported by the use of a set of FIWARE Generic Enablers not only to exchange data between different enablers but also to manage embedded systems as IoT agents. Also, the second scenario requires the use of Kurento Media Server to seamlessly transfer media contents from the embedded platforms to the cloud computing infrastructure.

7.2.1 FIWARE. It is an open-source community, which provides a rich set of APIs to facilitate the connection to IoT devices, user interaction, and process of data. FIWARE offers a rich library of components, called Generic Enablers (GE), which provide reference implementations that allow users to develop new applications. GEs provide the general-purpose functions such as data context management, IoT service enablement, advance web-based UI, security, interface to networks, the architecture of application/services ecosystem, and cloud hosting.

7.2.2 Kurento Media Server. Kurento is a Stream-Oriented GE providing a media server and a set of APIs to help the development of web-based media applications for browsers or smartphones. It offers ready-made bricks of media processing algorithms such as computer vision, augmented reality, and speech analysis.

In this work, we have focused on the *edge processing* scenario, since the selected embedded platforms in Section 6 are fairly able to process data directly on the *Edge*. Hence, we have created a dedicated media module in Kurento, which calls LPDNN's AI application and stores the results.

8 RESULTS AND DISCUSSION

In this section, we introduce the results of the AI pipeline and detail the outcome of the *Training* and *Deployment* steps. We give examples of different AI applications while we analyze and demonstrate the effectiveness and benefits of the Bonseyes AI pipeline for embedded systems.

8.1 Training

Table 2 shows benchmark scores of CNN models and DS-CNN models trained with Bonseyes-Caffe [5] for the KWS application shown in Section 5. The test set contains 2,567 audio samples, which are recorded from totally different speakers of the training samples. After 40K iterations training, the CNN model marks 94.23% accuracy on the test set, and the model size is 1.8 MB. DS-CNN model marks 90.65% accuracy, and the model size is 1 MB. With current hyper-parameter settings,

Table 2. KWS Trained Models: Benchmark of the Models on Test Set

Model	Acc	Sparsity	Size (KB)	Model	Acc	Sparsity	Size (KB)
CNN	94.23%	0%	1832	DS-CNN	90.65%	0%	1017
CNN + Q	94.04%	0%	918	DS-CNN + Q	90.62%	0%	511
CNN + S	93.69%	39.6%	1832	DS-CNN + S	89.96%	27.9%	1017
CNN + Q + S	94.27%	39.8%	918	DS-CNN + Q + S	90.19%	27.7%	511

Q: Quantization (16-bit), S: Sparsity.

DS-CNN is 4% less accurate than CNN, but its model size is about half of CNN. According to Zhang et al. [76], DS-CNN has the potential to be more accurate, and we will refine this model.

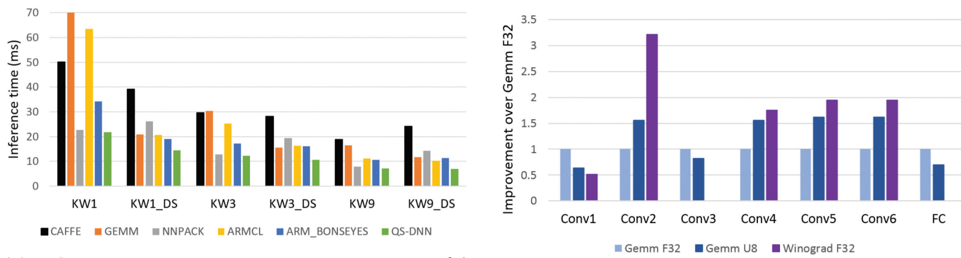
In this work, we have decided to build small footprint KWS models to ease the deployment on embedded devices. As introduced in Section 5, we can apply quantization (Q) and sparsification (S) functions to obtain a further compression in the models. On both CNN and DS-CNN, Q and S have a minor disadvantage (<0.7% loss) on test accuracy. Moreover, 16-bit fixed-point quantization can save half memory space and reduced bandwidth requirements at runtime. An S model may also leverage the amount of zeroes in its matrices and obtain benefits in memory and computation when it is deployed. Finally, we also observe that a Q+S model is more accurate than an S model as quantization may act as a regularizer and slightly increase the accuracy.

Manually designed CNN and DS-CNN models achieved over 90% prediction accuracy on KWS (Table 1). However, these models contain much redundancy and suppose a challenge for performance on embedded devices. We propose the Neural Architecture Search method to explore KWS models with reduced model size and FP_{ops}. Twelve CNN models are spotted through NAS and Pareto-optimal selection [53] and three models are presented in Table 4. In *kws1*, we can see that kernel sizes of *conv2* to *conv6* are no longer fixed at 3×3 but vary from 1×1 to 5×5. Output channels are all below 50. All these modifications reduce MFP_{ops} from 581.1 to 223.4 and improve TOP-1 accuracy from 94.2% to 95.1%. Further model size reduction is found in *kws3* and *kws9* by the price of a minor drop in accuracy. Another observation is that 4×10 kernels of the first convolution layer are no longer needed for an accurate KWS model. These rectangular kernels were designed to cover a longer temporal sequence than frequency bands from MFCC features [76]. As MFCC features were generated with 128ms frame length in this study, much longer than 40ms in literature, CNN with only square kernels is capable of delivering accurate KWS models.

Overall, we can see that NAS discovers obsolete network sub-structures introduced by manual network design. We leverage the advantages of DS-CNN and adapt CNN architectures in Table 4 to DS-CNN version (Table 5). Three new DS-CNN architectures are coined and are trained in 300K iterations. Each DS-CNN model achieves higher prediction accuracy than the seed DS-CNN model. In the meantime, the MFP_{ops} are only 11.9, 9.7, and 7.0, respectively. *ds_kws9* reports the minimum computational load in this study.

8.2 Deployment Optimization

Following the description of Section 6.3, we employ LPDNN's tools to optimize the deployment of AI applications on the embedded platforms, and we compare it with several well-known deployment frameworks. To ensure a fair comparison across the frameworks, we have enabled all the optimizations as provided by each vendor of the framework. All benchmarks have been performed identically: calculating the average of 10 inferences after an initial (discarded) warm-up run. In addition, we have set the platforms in performance mode, which sets the clocks to the highest frequency. To ensure that the platform does not overheat, triggering thermal throttling, we have monitored the platforms and sampled the OS registers each second.



(a) Inference time using FP32 operations. Caffe's time is given in black. Other colors represent the deployment of LPDNN employing a single library. On green, QS-DNN's solution after having learned an optimized combination of primitives.

(b) Quantization analysis for KWS1. Speedup of GEMM int8 primitives over GEMM F32 and comparison with Winograd F32 for each layer of the network.

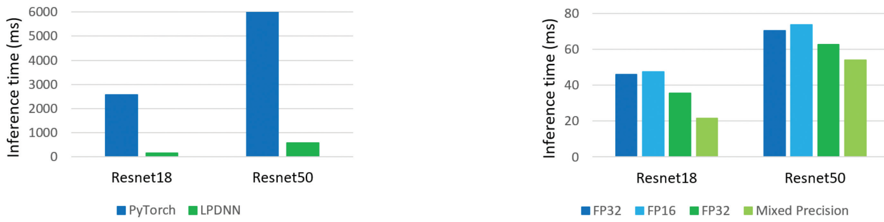
Fig. 13. LPDNN vs. Caffe (KWS). Inference time using single-thread operations on the Nvidia Jetson Nano (CPU) (the lower, the better).

8.2.1 LPDNN vs. Caffe (KWS). We compare the deployment of the trained KWS models from Tables 4 and 5 between Caffe and LPDNN on the Nvidia Jetson Nano platform. Caffe has been installed natively on the platform using Openblas as the backend. On the LPDNN side, we employ LNE coupled with an RL-based search (QS-DNN) to find an optimized solution for the deployment on the target platform. Figure 13(a) shows the results of the deployment when processing a one-second audio input using a single-thread and 32-bit floating-point operations on the CPU.

Overall, we can see that while Caffe takes, from largest to smallest, between 50 ms and 24 ms to process a single KWS network while LPDNN employs between 21 ms and 7 ms. Caffe-Openblas featuring general-matrix multiplication (GEMM) only outperforms LPDNN-GEMM on KWS1. Nonetheless, QS-DNN's capability to learn an optimized combination of libraries makes LPDNN significantly outperform Caffe on every network, being up to x3.5 faster. Regarding LPDNN's acceleration libraries, it is noted that no single library outperforms all other libraries across all networks. QS-DNN, however, always outperforms all individual libraries across all networks. Hence, it is possible to prove that QS-DNN adapts to each specific use case and supposes a powerful tool for LPDNN's optimized deployment.

Networks can be further compressed through quantization by employing primitives featuring reduced-numerical precision, as explained in Section 6.2.5. Figure 13(b) illustrates an analysis of KWS1's layers using int8 primitives from armCL. We observe that GEMM int8 generally—but not always—outperforms its FP32 counterpart. The overall improvement of having KWS1 full int8 accounts for 52% over GEMM FP32, as Conv4–Conv6 are the most computationally intensive layers, and 1/4 of the memory size with only 1% drop in accuracy. However, this improvement is shadowed by efficient convolution primitives such as Winograd [69], whose F32 implementation outperforms GEMM FP32 by 88%. Based on these results, we state that the use of quantization on Jetson Nano devices provides a tradeoff between latency and memory consumption for the KWS application.

8.2.2 LPDNN vs. PyTorch (Object Detection). We compare the deployment of the (resnet-based) body-pose estimation models presented in Section 6.3.2 between PyTorch and LPDNN on the Nvidia Xavier platform. We have installed PyTorch natively on the platform using the latest release provided by Nvidia [41]. We perform a first experiment deploying the models on the Arm CPU of the platform and employing single-thread and 32-bit floating-point operations. PyTorch uses the ATen library based on c++11, while LPDNN employs QS-DNN coupled to LNE to find an optimized



(a) CPU deployment using single-thread FP32.

(b) GPU deployment using FP32 and FP16.

Fig. 14. LPDNN vs. PyTorch (Object detection). Inference time of the (resnet-based) body-pose estimation models on the Nvidia Jetson Xavier.

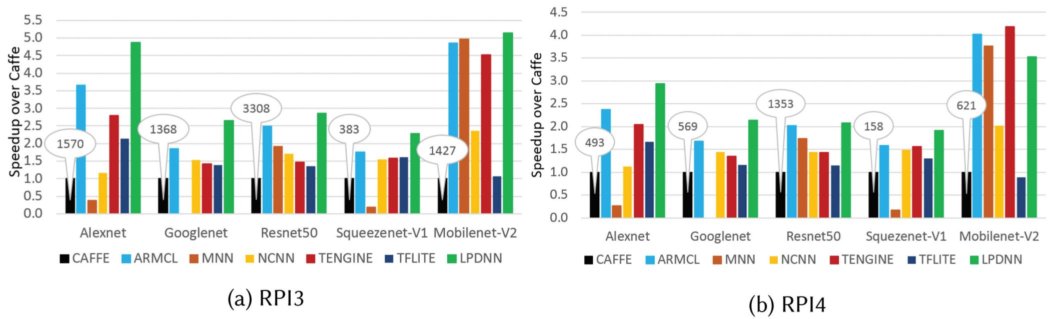
combination of primitives. Figure 14(a) presents the deployment results, and we can observe that LPDNN amply outperforms PyTorch on the CPU, being up to x15 faster for the resnet18-based model.

As PyTorch is mainly a training framework, we assume that CPU has not been largely optimized in favor of GPU deployment. Hence, we perform a second experiment on the GPU employing the same backend, CUDA-10, in both frameworks. As it can be in Figure 14(b), LPDNN outperforms PyTorch on both networks performing up to 28% faster. Further, we analyze the performance of half-precision to speed up the inference and reduce memory footprint. PyTorch employing FP16 out-of-the-box turns out to be slower than FP32. This might be due to a direct conversion from FP32 to FP16, and, as Ref. [61] suggests, this conversion needs to be carefully carried out to keep performance up. In LPDNN, by contrast, we set QS-DNN to automatically learn what data type performs better and give an optimized combination of primitives for LNE. Thus, we achieve up to 65% improvement on Resnet18 when leveraging mixed precision.

8.2.3 Comparison with Embedded Deployment Frameworks (Image Classification). To demonstrate the capabilities of LPDNN further, we compare it with a range of embedded deployment frameworks on the RPI3b+ and RPI4b[1] for the ImageNet challenge as stated in Section 6.3.3. We have built all the deployment frameworks natively and enabled all the optimizations as provided by each vendor. To further guarantee a fair comparison, we have chosen five representative networks that are used across all deployment frameworks: *Alexnet*, *Resnet50-V1*, *Googlenet-V1*, *Squeezenet-V1.1*, and *Mobilenet-V2-1.0-224*. The reference networks are taken from Caffe repository, which we assume as the reference framework, and are imported into each of the tested frameworks directly if supported by the framework, or via an official conversion tool (provided by the framework[2]). All inferences are performed using a single-thread and 32-bit floating-point operation on the CPU.

Figure 15 presents the results of the reference networks across the range of deployment frameworks on the RPI3 and RPI4. We show the relative speedup of each framework with respect to Caffe (reference) for which we display the absolute time in milliseconds. We can easily observe two general trends. *(i)* Certain frameworks perform very well on a single network but drastically drop performance on other networks, e.g., MNN, Tengine. *(ii)* The deployment of some networks has been remarkably optimized, e.g., several frameworks achieve over 4x improvement over the reference in Mobilenet-V2, while generally, no framework accomplishes a comparable speed-up in other network topologies, e.g., Googlenet, Squeezenet.

[1]Both platforms have been flashed with 64-bit Debian OS images.
[2]With exception of TF Lite. We convert Caffe to TF via MMDNN and ONNX. From TF to TF Lite, we use the official converter.



(a) RPI3 (b) RPI4

Fig. 15. Comparison with embedded deployment frameworks (Image classification). Inference results of the range of deployment frameworks for the reference networks on the RPI3 and RPI4. The bars represent the relative speedup over Caffe, which displays the absolute time in milliseconds.

Table 3. Inference Comparison in Milliseconds between TF Lite and LPDNN Taking TF Original Networks

DNN	RPI3		RPI4	
	LPDNN	TF Lite	LPDNN	TF Lite
Mobilenet-V2 (from TF Lite)	217	246	105	119
Googlenet (from TF)	429	839	216	430
Resnet50 (from TF)	1,172	2,024	667	981

These two trends confirm the importance of selecting several network topologies to have a sound estimation of the frameworks and how they can adapt to each structure. ArmCL and LPDNN are the frameworks that provide the most stable performance improvements. From these two, LPDNN obtains the highest speedups and outperforms all other frameworks across networks and target platforms, performing over 2× better than the average and 5× better than the worst performing framework. LPDNN's high performance can be explained by the abundant number of optimized primitives that LPDNN contains and its ability to learn a combination of primitives for each network. The stability across the two platforms proves that LPDNN is robust and can adapt to different architectures while retaining high performance.

TF Lite exception. We have benchmarked all frameworks, taking the networks from Caffe as reference. TF Lite is the only framework that neither supports nor provides an official conversion tool for Caffe networks and, hence, we have employed MMDNN and ONNX for the conversion to TF. From TF to TF Lite, we use the official converter. We argue that this conversion, although providing correct output classifications results, might be the cause of the low performance of TF Lite. Therefore, we offer a new benchmark comparing TF Lite with LPDNN, taking three networks from TF repositories instead. We take Mobilenet-V2 from TF Lite repository and Googlenet-V1 and Resnet50-V1 from the original TF, as they are not available in TF Lite directly. We convert all of them from TF to LPDNN via ONNX and the last two from TF to TF Lite via TF Lite official converter. Table 3 depicts the results of such benchmarks on the RPI3 and RPI4.

We can remark that the native TF Lite network, Mobilenet-V2, performs notably well, achieving almost the performance of LPDNN. However, we can already see that the original networks from standard TF, converted to TF Lite, drop in performance, being up to 2.5× slower than LPDNN. This point denotes the lack of proper support in TensorFlow for other formats, e.g., Caffe, ONNX, TF, as it appears that TF Lite only performs well when the networks have been written in a spe-

cific format, i.e., TF Lite format, or contains a specific architecture, e.g., Mobilenet. This fact supposes a constraint for users wanting to employ custom models, as they would either have poor performance when executing non TF Lite networks or have TF framework a fixed dependency. Nevertheless, LPDNN also outperforms TF Lite using TF original models. We can thus prove LPDNN's flexibility and support for other network's formats and its domain over the range of embedded-oriented deployment frameworks.

9 CONCLUSION AND FUTURE WORK

Nowadays, training and deployment of custom AI solutions on embedded and IoT devices poses many issues as it requires a fine-grained integration of data, algorithms, and tools. These barriers prevent the massive spread of AI applications in our daily life as only end-to-end systems can overcome these hurdles and achieve accurate and fast solutions. In this work, we present a modular end-to-end AI pipeline architecture, which brings data, algorithms, and deployment tools together to facilitate the production and porting of AI solution for embedded devices. We ease the integration and lower the required expertise by providing key benefits such as the reusability of tools thanks to a dockerized API, and the flexibility to add new steps to the workflow. Thus, we propose a pipeline with four main steps: (i) data ingestion, (ii) model training, (iii) deployment optimization, and (iv) the IoT hub integration.

We have demonstrated the effectiveness of the AI pipeline by providing several examples of AI applications in each of the steps and show the significance of a tight integration of the pipeline steps toward having an efficient and competitive implementation. Thus, we are able to create a data ingestion step for the Google speech commands dataset seamlessly and train two families of CNN and DS-CNN networks achieving up to 95.1% and 92.6%. Further, we have presented the lightweight architecture and deployment capabilities of our deployment framework, LPDNN, and demonstrate that it outperforms all other popular deployment frameworks on a set of AI applications and across a range of embedded platforms.

As future work, we envision to fully optimize and deploy trained models on very low-power devices that can be employed for applications such as healthcare sensing, wearable systems, or a car-driving assistant.

APPENDIX

A NEURAL NETWORK ARCHITECTURES

Table 4. Pareto Optimal CNN Architectures for KWS [53]

Model	conv1	conv2	conv3	conv4	conv5	conv6	TOP-1	M FP_{ops}	Size (KB)
seed	4 × 10, 100	3 × 3, 100	3 × 3, 100	3 × 3, 100	3 × 3, 100	3 × 3, 100	94.2%	581.1	1832
kws1	3 × 3, 40	3 × 3, 30	1 × 1, 30	5 × 5, 50	5 × 5, 50	5 × 5, 50	95.1%	223.4	707.0
kws3	5 × 5, 50	1 × 1, 30	5 × 5, 40	3 × 3, 20	5 × 5, 30	3 × 3, 50	94.1%	87.6	282.1
kws9	5 × 5, 50	1 × 1, 20	1 × 1, 50	3 × 3, 20	5 × 5, 20	3 × 3, 40	93.4%	37.7	125.3

Table 5. Optimized DS-CNN Architectures Based on CNN Models

Model	conv1	conv2	conv3	conv4	conv5	conv6	TOP-1	M FP_{ops}	Size (KB)
seed	4×10, 100	3×3, 100	3×3, 100	3×3, 100	3×3, 100	3×3, 100	90.6%	69.9	1017
ds_kws1	3×3, 40	3×3, 30	1×1, 30	5×5, 50	5×5, 50	5×5, 50	92.6%	11.9	61.5
ds_kws3	5×5, 50	1×1, 30	5×5, 40	3×3, 20	5×5, 30	3×3, 50	91.2%	9.7	48.4
ds_kws9	5×5, 50	1×1, 20	1×1, 50	3×3, 20	5×5, 20	3×3, 40	91.3%	7.0	39.0

ACKNOWLEDGMENTS

The opinions expressed and arguments employed herein do not necessarily reflect the official views of these funding bodies.

REFERENCES

[1] 2017. Google Speech Commands Dataset. Retrieved from https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html.
[2] 2017. The ONNX Project. Retrieved from https://github.com/onnx/onnx.
[3] 2017. Open Neural Network Exchange (ONNX). Retrieved from https://onnx.ai/.
[4] 2018. Arm Compute Library. Retrieved from https://developer.arm.com/ip-products/processors/machine-learning/compute-library.
[5] 2018. Bonseyes Official Caffe 1.0 Version. Retrieved from https://github.com/bonseyes/caffe-jacinto.
[6] 2018. Caffe. Retrieved from http://caffe.berkeleyvision.org/.
[7] 2018. Caffe-SSD. Retrieved from https://github.com/weiliu89/caffe.
[8] 2018. Discover the Power of Artificial Intelligence to Drive ICT Innovation. Retrieved from https://news.itu.int/discover-the-power-of-artificial-intelligence-to-drive-ict-innovation-in-the-first-issue-of-the-itu-journal/.
[9] 2018. Docker. Retrieved from http://www.docker.com.
[10] 2018. FI-ware Project. Retrieved from https://www.fiware.org/.
[11] 2018. Machine Learning Fragmentation Is Slowing Us Down: There Is a Solution. Retrieved from https://www.cmswire.com/digital-experience/machine-learning-fragmentation-is-slowing-us-down-there-is-a-solution/.
[12] 2018. Scientists Can't Replicate AI Studies. That's Bad News. Retrieved from https://futurism.com/scientists-cant-replicate-ai-studies.
[13] 2019. Amazon Machine Learning on AWS. Retrieved from https://aws.amazon.com/machine-learning.
[14] 2019. Amazon SageMaker. Retrieved from https://aws.amazon.com/sagemaker/.
[15] 2019. Android Neural Networks API (NNAPI). Retrieved from https://developer.android.com/ndk/guides/neuralnetworks.
[16] 2019. Apple AI. Retrieved from https://www.zdnet.com/article/apple-says-artificial-intelligence-and-machine-learning-critical-area-as-it-promotes-ai-chief/.
[17] 2019. CAFFE2. Retrieved from https://caffe2.ai/.
[18] 2019. CAFFE2FB. Retrieved from https://research.fb.com/downloads/caffe2/.
[19] 2019. CNTK: The Microsoft Cognitive Toolkit. Retrieved from https://docs.microsoft.com/en-us/cognitive-toolkit.
[20] 2019. Converting Trained Models to Core ML. Retrieved from https://developer.apple.com/documentation/coreml/converting_trained_models_to_core_ml
[21] 2019. Core ML: Integrate Machine Learning Models Into Your App. Retrieved from https://developer.apple.com/documentation/coreml.
[22] 2019. Folding of Bnorm Into Convolution. Retrieved from https://tehnokv.com/posts/fusing-batchnorm-and-conv/.
[23] 2019. Google AI Platform. Retrieved from https://cloud.google.com/ai-platform/.
[24] 2019. Google AI Tools. Retrieved from https://ai.google/tools/.
[25] 2019. Google ML. Retrieved from https://cloud.google.com/products/machine-learning/.
[26] 2019. Greengrass. Retrieved from https://aws.amazon.com/greengrass/.
[27] 2019. Greengrass Region. Retrieved from https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/.
[28] 2019. ImageNet. Retrieved from http://www.image-net.org.
[29] 2019. Intel OpenVINO Toolkit. Retrieved from https://docs.openvinotoolkit.org.
[30] 2019. Keras: The Python Deep Learning Library. Retrieved from https://keras.io.

[31] 2019. Lead in AI. Retrieved from https://www.forbes.com/sites/danielaraya/2019/01/01/who-will-lead-in-the-age-of-artificial-intelligence/#4f3a15aa6f95.
[32] 2019. Microsoft Azure. Retrieved from https://azure.microsoft.com/en-us/services/machine-learning/.
[33] 2019. Microsoft Azure IoT Edge. Retrieved from https://azure.microsoft.com/en-in/services/iot-edge/.
[34] 2019. Mobile Neural Network (MNN): A Lightweight Deep Neural Network Inference Engine. Retrieved from https://github.com/alibaba/MNN.
[35] 2019. Model Interpretability in Azure Machine Learning Service. Retrieved from https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-machine-learning-interpretability.
[36] 2019. MXNet: A Flexible and Efficient Library for Deep Learning. Retrieved from https://mxnet.apache.org.
[37] 2019. NCNN: A High-Performance Neural Network Inference Framework Optimized for the Mobile Platform.Retrieved from https://github.com/Tencent/ncnn.
[38] 2019. NNPACK. Retrieved from https://github.com/Maratyszcza/NNPACK.
[39] 2019. Nvidia Jetson Nano. Retrieved from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.
[40] 2019. Nvidia Jetson Xavier. Retrieved from https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.
[41] 2019. Nvidia Realeased of PyTorch. Retrieved from https://docs.nvidia.com/deeplearning/frameworks/pytorch-release-notes/overview.html#overview.
[42] 2019. PyTorch: From Research to Production. Retrieved from https://pytorch.org.
[43] 2019. Quantization Analysis Tool. Retrieved from https://github.com/BUG1989/caffe-int8-convert-tools.
[44] 2019. Raspberry Pi3. Retrieved from https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.
[45] 2019. Raspberry Pi4. Retrieved from https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.
[46] 2019. Renesas e-AI. Retrieved from https://www.renesas.com/eu/en/solutions/key-technology/e-ai.html.
[47] 2019. Tengine: A Lite, High-performance, and Modular Inference Engine for Embedded Device. Retrieved from https://github.com/OAID/Tengine.
[48] 2019. TensorFlow: An End-to-end Open Source Machine Learning Platform. Retrieved from https://www.tensorflow.org.
[49] 2019. TensorFlow Lite: Deploy Machine Learning Models on Mobile and IoT Devices. Retrieved from https://www.tensorflow.org/lite.
[50] 2019. TensorRT. Retrieved from https://developer.nvidia.com/tensorrt.
[51] 2019. TI-DL. Retrieved from https://training.ti.com/texas-instruments-deep-learning-tidl-overview.
[52] 2019. Torch: A Scientific Computing Framework for LuaJIT. Retrieved from http://torch.ch.
[53] Andrew Anderson, Jing Su, Rozenn Dahyot, and David Gregg. 2019. Performance-oriented neural architecture search. In *Proceedings of the 2019 International Conference on High Performance Computing and Simulation*. IEEE.
[54] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*. 2546–2554.
[55] Microsoft NNI contributors. 2019. An Open Source AutoML Toolkit for Neural Architecture Search and Hyper-parameter Tuning. Retrieved May 27, 2019 from https://github.com/Microsoft/nni.
[56] Miguel de Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. 2018. QUENN: QUantization engine for low-power neural networks. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 36–44.
[57] Miguel de Prado, Nuria Pazos, and Luca Benini. 2018. Learning to infer: RL-based search for DNN primitive selection on heterogeneous embedded systems. *arXiv preprint arXiv:1811.07315* (2018).
[58] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2018. Neural architecture search: A survey. *arXiv e-prints*, Article arXiv:1808.05377 (Aug 2018), arXiv:1808.05377 pages. arxiv:stat.ML/1808.05377
[59] T. Llewellynn et al. 2017. BONSEYES: Platform for open development of systems of artificial intelligence. In *Proceedings of the Computing Frontiers Conference*. ACM, 299–304.
[60] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 6645–6649.
[61] Nhut-Minh Ho and Weng-Fai Wong. 2017. Exploiting half precision arithmetic in Nvidia GPUs. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[62] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR* abs/1704.04861 (2017).
[63] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. 2015. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716* (2015).

[64] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning—Volume 37 (ICML'15)*. JMLR.org, 448–456. http://dl.acm.org/citation.cfm?id=3045118.3045167.

[65] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).

[66] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*.

[67] Sven Kreiss, Lorenzo Bertoni, and Alexandre Alahi. 2019. PifPaf: Composite fields for human pose estimation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[68] Yuxi Li. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274* (2017).

[69] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse Beu, Matthew Mattina, and Robert Mullins. 2019. Efficient winograd or cook-toom convolution kernel implementation on widely used mobile CPUs. *arXiv preprint arXiv:1903.01521* (2019).

[70] Vijay K. Mathur. 1991. How well do we know pareto optimality? *The Journal of Economic Education* 22, 2 (1991), 172–178. http://www.jstor.org/stable/1182422.

[71] B. McFee, C. Raffel, D. Liang, D. P.W. Ellis, M. McVicar, E. Battenberg, and O. Nieto. 2015. librosa: Audio and music signal analysis in Python. In *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra (Eds.). 18–25.

[72] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*. Omnipress, 807–814. http://dl.acm.org/citation.cfm?id=3104322.3104425.

[73] Muhammad Shafique, Theocharis Theocharides, Christos-Savvas Bouganis, Muhammad Abdullah Hanif, Faiq Khalid, Rehan Hafız, and Semeen Rehman. 2018. An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era. In *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 827–832.

[74] G. Tzanetakis and P. Cook. 2002. Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing* 10, 5 (July 2002), 293–302. DOI: https://doi.org/10.1109/TSA.2002.800560

[75] S. Yao, Y. Hao, Y. Zhao, A. Piao, H. Shao, D. Liu, S. Liu, S. Hu, D. Weerakoon, K. Jayarajah, A. Misra, and T. Abdelzaher. 2019. Eugene: Towards deep intelligence as a service. In *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1630–1640. DOI: https://doi.org/10.1109/ICDCS.2019.00162

[76] Y. Zhang, N. Suda, L. Lai, and V. Chandra. 2018. Hello edge: Keyword spotting on microcontrollers. *ArXiv e-prints* (Feb. 2018). arxiv:cs.SD/1711.07128.

Received June 2019; revised May 2020; accepted May 2020