AI Pipeline - bringing AI to you. End-to-end integration of data, algorithms and deployment tools

Jing Su
Trinity College Dublin
19 PUBLICATIONS **64** CITATIONS
SEE PROFILE

Rozenn Dahyot
National University of Ireland, Maynooth
126 PUBLICATIONS **1,600** CITATIONS
SEE PROFILE

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 Laboratory, ETH Zurich
Switzerland
miguel.deprado@he-arc.ch

Jing Su
School of Computer Science & Statistics
Trinity College Dublin
Ireland
jing.su@tcd.ie

Rozenn Dahyot
School of Computer Science & Statistics
Trinity College Dublin
Ireland
Rozenn.Dahyot@tcd.ie

Rabia Saeed
Haute Ecole Arc Ingenierie; HES-SO
Switzerland
rabia.saeed@he-arc.ch

Lorenzo Keller
Nviso
Switzerland
lorenzo.keller@nviso.ai

Noelia Vallez
Universidad de Castilla - La Mancha
Spain
Noelia.Vallez@uclm.es

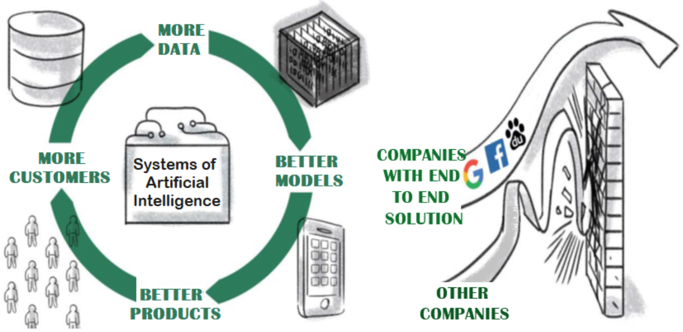

MORE DATA
BETTER MODELS
BETTER PRODUCTS
MORE CUSTOMERS
Systems of Artificial Intelligence
COMPANIES WITH END TO END SOLUTION
OTHER COMPANIES

Figure 1: Companies with end to end solutions outperform the ones unable to follow the complete cyclic process

ABSTRACT

Next generation of embedded Information and Communication Technology (ICT) systems are interconnected collaborative intelligent systems able to perform autonomous tasks. Training and deployment of such systems on Edge devices however require a fine-grained integration of data and tools to achieve high accuracy and overcome functional and non-functional requirements.

In this work, we present a modular AI pipeline as an integrating framework to bring data, algorithms and deployment tools together. By these means, we are able to interconnect the different entities or stages of particular systems and provide an end-to-end development of AI products. We demonstrate the effectiveness of the AI pipeline by solving an Automatic Speech Recognition challenge and we show that all the steps leading to an end-to-end development for Key-word Spotting tasks: importing, partitioning and pre-processing of speech data, training of different neural network architectures and their deployment on heterogeneous embedded platforms.

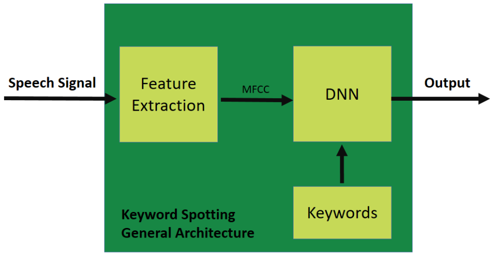


Figure 2: Keyword Spotting (KWS), a special case of ASR, is the process of recognizing predefined words from a speech signal. The process involves converting speech signals into a feature vector that can be used as an input to a model. The model then compares the input to the predefined words and detects their presence.[Zhang et al. 2018]

KEYWORDS

AI pipeline, Key-word Spotting, fragmentation

INTRODUCTION

Artificial Intelligence (AI) is rapidly spreading to embedded and Internet-of-Thing (IoT) devices in a new form of distributed computing systems. Fragmentation of AI knowledge and tools is becoming an issue difficult to overcome for most stakeholders. Only large companies, e.g. Google, Apple, are able to build end-to-end systems where data, algorithms, tools and dedicated hardware are built together and optimized for Deep Learning applications. Such companies are taking the lead of the AI industry partly due to having privileged access to data, see Fig. 1.

Automatic Speech Recognition (ASR) is a classical AI challenge where data, algorithms and hardware must be brought together to perform well. Upon the introduction of Deep Neural Network (DNN) models, speech recognition accuracy has been greatly improved and the word error rate can be as low as 5.5% [Saon et al. 2017]. However, a top score in standard benchmark datasets, e.g., 2000 HUB5 [hub 2002], does not guarantee successful ASR deployment in real-world scenarios. Speech source with regional accents has high phonetic variance even in the same word, and expressions in dialects may exceed standard transcription dictionary. These scenarios suppose a great challenge as retraining a recognizer system with large amount of real-user data becomes necessary before the system is deployed. Therefore, only stakeholders that are able to acquire large amounts of data and knowledge, state-of-the-art algorithms and highly optimized tools for deployment are able to overcome such a problem.

Opposed to monolithic and closed solutions, Bonseyes [et al. 2017] comes in as a European collaboration to facilitate Deep Learning to any stakeholder and to reduce development time and cost of ownership. Developing AI solutions such as ASR, is a significant engineering and organizational effort that requires large investments as there are significant technological barriers (e.g. ease of usage, fragmentation, reliability). In this context, Bonseyes proposes an AI pipeline framework as a way to overcome these barriers and to provide key benefits such as scalability, reusability, reproducibility and flexibility. We demonstrate the effectiveness of the AI pipeline by applying it to Key-word Spotting (KWS), a special case of ASR challenge, see Fig. 2.

BACKGROUND

We propose an end-to-end AI pipeline to solve four main tasks: data collection, model training, deployment on constraint environment e.g. speech recognition in a car, and IoT integration. More tasks can be added based on the needs of the application. Fragmentation easily occurs between these tasks as each task needs and employ specific formats to operate. The AI pipeline allows to link

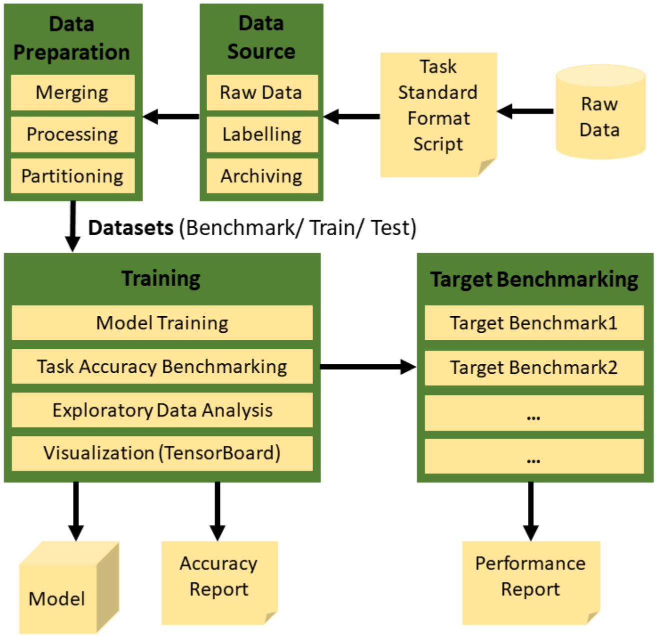
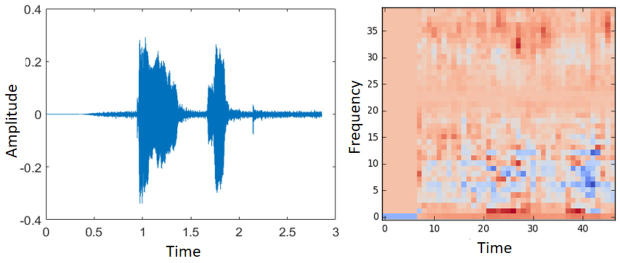


Figure 3: AI data collection and training pipeline



(a) Speech Signal

(b) MFCC Features

Figure 4: (a) Input Signal is converted to (b) MFCC feature

fragmented environments by covering three main concepts: tools, artifacts and workflows. Artifacts are the data elements that tools generate and consume. Workflows describe the tool that composed the pipelines and how to chain them to achieve a given goal. The AI pipeline relies on Docker containers [doc 2018] to package all software dependencies needed to run a tool. In addition, a high-level HTTP API is defined to control the workflows and tools.

KEYWORD SPOTTING PIPELINE (KWS)

Our KWS pipeline contains 7 functional units:

- Download and split Google speech commands dataset
- Import training and tests dataset
- Generate MFCC of training and test samples
- Train DNN models
- Benchmark DNN models
- Deployment optimization
- IoT integration

Next, we introduce and explain the main components of the AI pipeline for KWS.

AI DATASET COLLECTION AND PREPARATION

This includes all steps needed to obtain and pre-process the data as needed for the training and benchmarking phases (Fig. 3). The tool downloads the complete Google Speech Commands dataset [Spe 2017] containing WAV files. The raw data is downloaded from the provider, parsed and standardized into HDF5 format files to comply with reusability and compatibility. Finally, data is partitioned into training, validation and benchmarking sets.

In case of training set, the import tool stores each WAV file together with sample ID and class label in one data tensor. This data tensor follows Bonseyes data format. Keyword labels are read from corresponding folder names. This single data tensor is used as input to the next tool. Validation set and test set can be imported in the same way.

MFCC GENERATION AND KWS

MFCC is a well-known audio feature for music and speech processing [Tzanetakis and Cook 2002]. In KWS pipeline, the feature generation tool produces MFCC features of each audio sample and saves MFCC features together with keyword labels and sample IDs in a HDF5 file. The Librosa library [McFee et al. 2015] is adopted in this tool. Each output sample of MFCC contains 40 frequency bands in the spectrogram as well as 44 temporal windows per second. The generated MFCC features (training set and test set) can also be reused for training and benchmarking tools of new models. Fig. 4 illustrates an audio signal and its MFCC features.

Layer	Filter Shape	Stride	Padding
Conv1	4 X 10 X 100	1 X 2	2 X 5
Conv2	3 X 3 X 100	2 X 2	1 X 1
Conv3~6	3 X 3 X 100	1 X 1	1 X 1
Pooling	21 X 12 X 1		

Table 1: CNN model architecture in which Filter Shape follows (H x W x C), Stride follows (H x W) and Padding follows (H x W). H: Height, W: Width, C: Channel.

Layer	Filter Shape	Stride	Padding
Conv1	4 X 10 X 100 X 1	1 X 2	2 X 5
DS_Conv1 (D)	3 X 3 X 100 X 100	2 X 2	1 X 1
DS_Conv1 (P)	1 X 1 X 1 X 1	1 X 1	
DS_Conv2~5 (D)	3 X 3 X 100 X 100	1 X 1	1 X 1
DS_Conv2~5 (P)	1 X 1 X 1 X 1	1 X 1	
Pooling	3 X 3 X 1 X 1		

Table 2: DS_CNN model architecture in which Filter Shape follows (H x W x C x G), Stride follows (H x W) and Padding follows (H x W). H: Height, W: Width, C: Channel, G: Group, D: Depthwise, P: Pointwise.

TRAINING AND BENCHMARKING

Three different KWS neural network architectures have been implemented to cross-compare accuracy, memory usage and inference time: Convolutional Neural Network (CNN), Depth-wise Separable CNN (DS-CNN) and Long short-term memory (LSTM) based network. The training procedures are defined in separate training tools to facilitate tools reusability and modularity. All training tools generate automatically both the training model and the solver definition files. 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. Batch size and number of iterations can be specified in the workflow files that control the execution of the tools.

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.

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. Inference is performed and the predicted classes are compared with the provided ground truth and the results are stored in a JSON file.

Network Architectures

In order to build a model with small footprint, we choose a CNN architecture with six convolution layers, one average pooling layer and one output layer. More specifically, each convolution layer is followed by one batch normalization layer [Ioffe and Szegedy 2015], one scale layer and one ReLU layer [Nair and Hinton 2010]. 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 x 10. This kernel maps on 4 rows and 10 columns of a MFCC input image, which in turn refers to 4 frequency bands and 10 sampling windows. A 4 x 10 kernel has advantage in capturing power variation in a longer time period and narrower frequency bands. This setting complies with [Zhang et al. 2018]. In the following convolution layers (Conv2 ∼ Conv6) 3 x 3 square kernels are applied.

Depthwise Separable CNN (DS_CNN) is introduced by [Howard et al. 2017], and it is also applied for KWS. In this study, 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. A DS_Conv layer is composed of a depthwise convolution layer and a pointwise convolution layer, each is followed by one normalization layer, one scale layer and one ReLU layer. Table 2 explains parameter settings of the DS_CNN model.

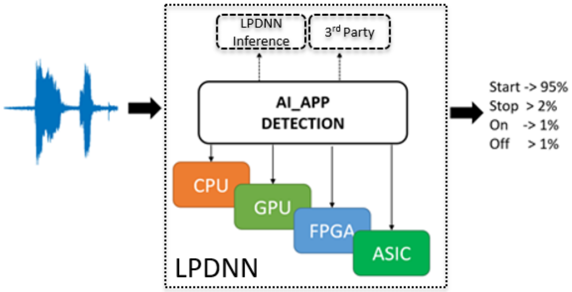


Figure 5: AI application can be executed by LPDNN or a 3rd party framework. Application can be deployed on one or several processing systems.

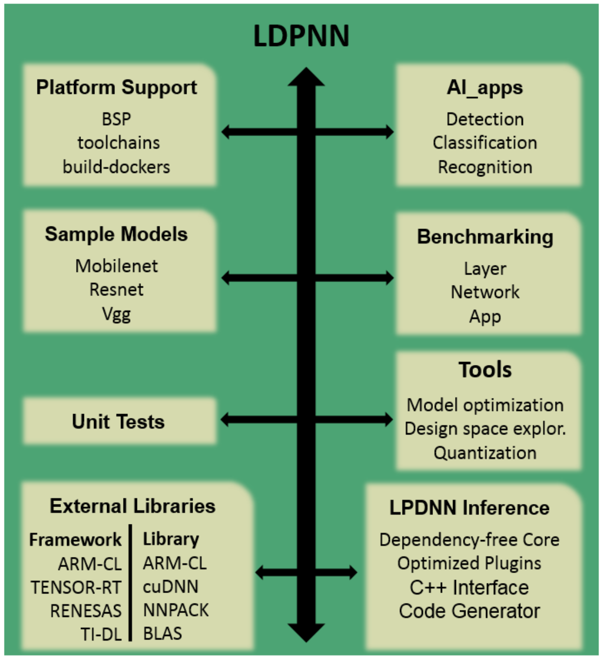


Figure 6: LPDNN. External libraries can be used to build optimized plugins using LPDNN Inference or can be used as self-contained frameworks for AI applications.

Training Configurations

CNN and DS_CNN models are trained using Bonseyes-Caffe. Training is carried out with multinomial logistic loss and Adam optimizer [Kingma and Ba 2015] over a batch of 100 MFCC samples (since input sample size is 40 x 44, we opt to use a relatively big batch size). 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, learning rate drops to 30% of previous step.

DEPLOYMENT OPTIMIZATION

LPDNN

LPDNN is an inference engine optimizer developed within the Bonseyes project to accelerate the deployment of neural networks on embedded devices [de Prado et al. 2018a]. In this work, we integrate LPDNN and its optimization capabilities into the AI pipeline to optimize the deployment of the trained KWS models for a specific embedded target platform, see Fig. 6. The overall goal of LPDNN is to provide a set of AI applications, e.g. detection, classification, recognition, which can be ported and optimized across heterogeneous platforms, e.g. CPU, GPU, FGPA, VPU, see Fig. 5. LPDNN features a complete development flow for users aiming to take up AI solutions on their systems:

- Reference platforms support and Board Support Package (BSP) containing toolchains.
- A wide set of sample models e.g. Mobilenets, GoogleNet, VGG16, Resnets.
- Optimization tools such as model compression and design space exploration.
- Integration of 3rd-party libraries.
- Benchmarking tool at several levels of abstraction, e.g. layer, network and application.

In the heart of LPDNN, there is an inference module (LPDNN Inference) which generates code for the range of DNN models and across the span of heterogeneous platforms. LPDNN Inference provides compatibility with Caffe [Caf 2018] and ONNX [onn 2017] while improving the execution of trained deep networks on resource-constrained environments. LPDNN Inference provides a plugin-based architecture where a dependency-free inference core is complemented and built together with a set of plugins to produce optimized code for AI APPs, see Fig. 7. In addition, LPDNN supports the integration of 3rd-party libraries at two different levels:

- Fine grain: developing a plugin at LPDNN Inference level for specific layers to accelerate the inference by calling optimized primitives of the library. Besides, several libraries can be combined or tuned to boost the performance of the network execution, e.g. BLAS, ARM-CL, cuDNN, etc.
- Coarse grain: as a self-contained framework at LPDNN level for AI APPs, e.g. TensorRT, TI-DL, NCNN, etc. External frameworks are included in LPDNN as certain embedded platforms provide their own specific and optimized framework to deploy DNNs on them.

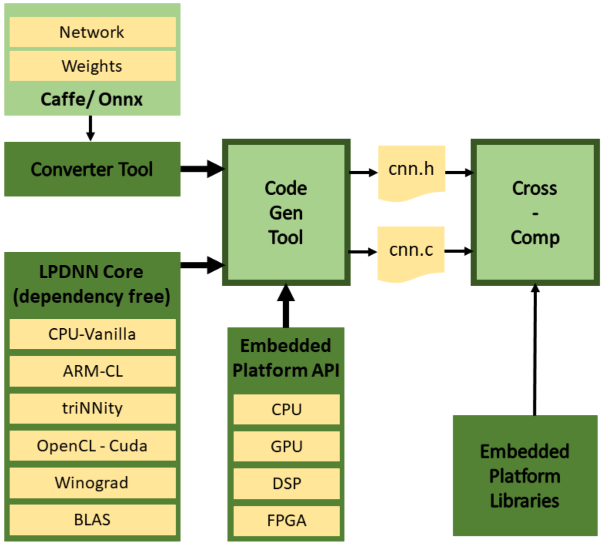


Figure 7: LPDNN Inference. Plugins can be included for specific layers which allows a broad design space exploration suited for the target platform and performance specifications.

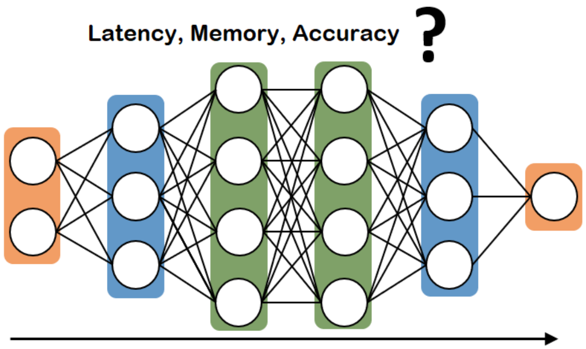


Figure 8: Optimization can be achieved by deploying one or several layers on different processing systems based on their capabilities regarding latency, memory or accuracy. Colours match Fig. 5.

When building an AI application, LPDNN can redirect the flow to either Inference or to an external framework. Being able to integrate 3rd-party libraries and frameworks broadens the scope of LPDNN as by default, LPDNN can be at least as good as any other included framework.

In this work, LPDNN provides inference acceleration for the KWS architectures: CNN, DS-CNN and LSTM. CNNs can be accelerated by calling BLAS (CPU) or Cuda (GPU) routines which implement highly optimized matrix multiplication operations. DS-CNN improves the execution of standard CNN as it reduces the number of multiplication operations by dividing a normal convolution into two parts: depth-wise and point-wise convolution. LPDNN makes use of Arm-CL optimized routines to parallelize kernel execution of depth-wise convolutions [arm 2018]. LSTM (Long-Short Term Memory) is a variant of Recurrent Neural Networks (RNN). These networks implement a memory state so as to predict results based on current and previous inputs. LPDNN supports time-series inputs and LSTM layers with multi-gated units, see Fig. 9.

Heterogeneous computing

A main factor for LPDNN 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 including CPUs, GPUs, DSPs, VPUs and Accelerators. Certain computing processors may provide better performance for specific tasks depending on the model architecture, data type and layout which brings a design exploration problem, see Fig. 8.

Network Space Exploration

We propose a learning-based approach called QS-DNN [de Prado et al. 2018b] where an agent explores through the design space, e.g. network deployment, across heterogeneous platforms to find an optimal implementation by selecting optimal per-layer configuration as well as cross-layer optimizations. The agent's aim is to learn an optimized path through a set states $\mathcal{S}$ i.e. layer representations, employing a set of actions $\mathcal{A}$ i.e. layer implementations. Neural networks can be further compressed through approximation. A quantization engine is integrated within LPDNN which analyzes the sensitivity of each layer to quantization and performs a Neural Network Design Exploration.

IoT INTEGRATION

Generally, AI enabled systems are part of a broader application and service ecosystem that supports a value chain. The integration of AI enabled systems into an IoT ecosystem is of particular interest when it comes to resource-constrained systems. This type of integration backs two major scenarios which are relevant for Automatic Speech Recognition in low-power and autonomous environments e.g. Automotive, Medical Health Care and Consumer Electronics.

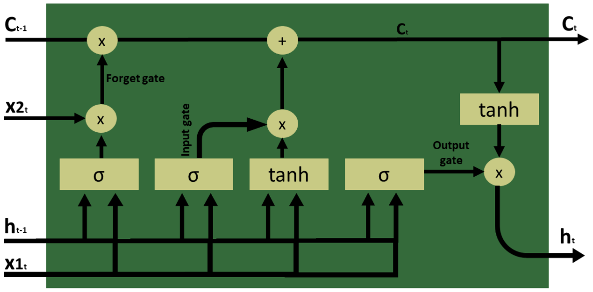


Figure 9: Memory of LSTM comes from the feedback mechanism. The output of each time step h_t along with the cell state C_t are used as an input in the next time step. A single LSTM cell (shown above) consists of 3 gates (input, output and forget gate), used to exploit the system's memory [Hochreiter and Schmidhuber 1997]. Apart from the conventional data input $X1_t$, an indicator $X2_t$ to specify the start of a new time series is also required.

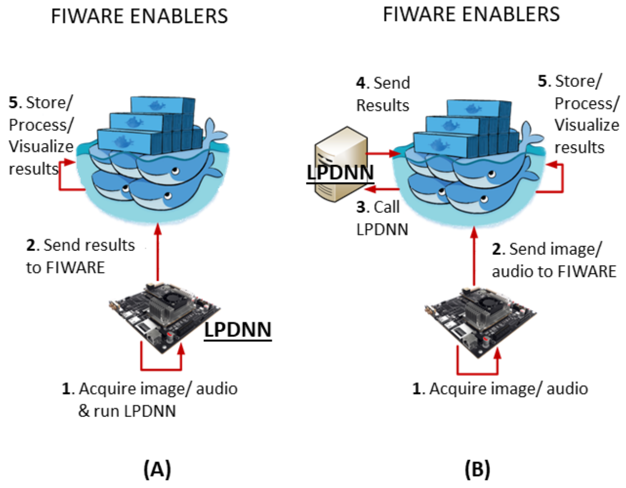

FIWARE ENABLERS
5. Store/ Process/ Visualize results
2. Send results to FIWARE
LPDNN
1. Acquire image/ audio & run LPDNN
(A)
FIWARE ENABLERS
4. Send Results
5. Store/ Process/ Visualize results
LPDNN
3. Call LPDNN
2. Send image/ audio to FIWARE
1. Acquire image/ audio
(B)

Figure 10: Bonseyes relies on the FIWARE platform [fiw 2018] for the implementation of the integration pipeline.

Edge-processing

Speech data are processed on the embedded AI device and results are retrieved and stored in the cloud for further processing and exploitation, see Fig. 10-A. Since the amount of data that should be dealt with can be large, the use of IoT architectures as an integration mechanism is highly relevant.

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 Fig. 10-B. IoT architectures offer an elegant solution for this integration to many distributed and diversified AI systems which can notably benefit from such integration.

Bonseyes relies on the FIWARE platform [fiw 2018] for the implementation of the integration pipeline. For the integration of both scenarios, we use a set of FIWARE Generic Enablers not only to exchange data between different enablers but also to manage embedded AI systems as IoT agents. In addition, the second scenario requires the use of Kurento Media Server in order to seamlessly transfer media contents from the embedded AI platforms to the cloud computing infrastructure.

RESULTS AND DISCUSSION

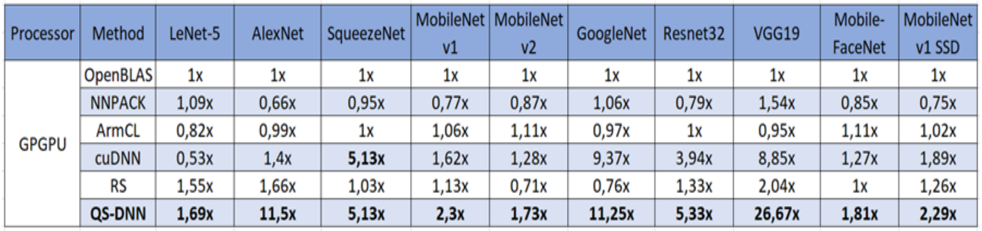
Table 4 shows benchmark scores of CNN models and DS_CNN models trained with Bonseyes-Caffe [bon 2018] for KWS. The test set contains 2567 audio samples which are recorded from totally different speakers of the training samples. After 40K iterations training (batch size 100), CNN model marks 94.23% accuracy on 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, DS_CNN is 4% less accurate than CNN, but its model size is about half of CNN. According to Zhang et al.[Zhang et al. 2018], DS_CNN has potential to be more accurate and we will refine this model. The encouraging aspects come from quantization (Q) and sparsification (S) functions. On both CNN and DS_CNN, Q and S have minor disadvantage ($< 0.7\%$ loss) on test accuracy. However, 16-bit quantization can save half memory space (here we use 32-bit space to save a quantized model). An S model can be faster than original version. A Q+S model is more accurate than an S model. In this work, we have decided to build a small footprint KWS model in the pipeline to ease the deployment on edge devices and easy adoption of customer-owned data.

LPDNN brings further optimization to trained models by generating specific and optimized code for the platform where the model is to be deployed. By searching through the Network Design Space when the model is deployed on a platform, we can find a suitable set of primitives that optimize inference on the platform. Table 3 shows the improvement in performance of QS-DNN (LPDNN

Model	Acc	Sparsity	Size
CNN	94.23%	0%	1832 KB
CNN + Q	94.04%	0%	1835 KB
CNN + S	93.69%	39.6%	1832 KB
CNN + Q + S	94.27%	39.8%	1835 KB
DS_CNN	90.65%	0%	1017 KB
DS_CNN + Q	90.62%	0%	1021 KB
DS_CNN + S	89.96%	27.9%	1017 KB
DS_CNN + Q + S	90.19%	27.7%	1021 KB

Table 4: Benchmark of the trained CNN models. *Q*: Quantization (16-bit), *S*: Sparsification, Acc: Accuracy on test set.

optimizer) when a RL-based agent is set to learn an optimal implementation by selecting a suitable combination of primitives per-layer:



Processor	Method	LeNet-5	AlexNet	SqueezeNet	MobileNet v1	MobileNet v2	GoogleNet	Resnet32	VGG19	Mobile-FaceNet	MobileNet v1 SSD
GPGPU	OpenBLAS	1x	1x	1x	1x	1x	1x	1x	1x	1x	1x
	NNPACK	1,09x	0,66x	0,95x	0,77x	0,87x	1,06x	0,79x	1,54x	0,85x	0,75x
	ArmCL	0,82x	0,99x	1x	1,06x	1,11x	0,97x	1x	0,95x	1,11x	1,02x
	cuDNN	0,53x	1,4x	**5,13x**	1,62x	1,28x	9,37x	3,94x	8,85x	1,27x	1,89x
	RS	1,55x	1,66x	1,03x	1,13x	0,71x	0,76x	1,33x	2,04x	1x	1,26x
	QS-DNN	**1,69x**	**11,5x**	**5,13x**	**2,3x**	**1,73x**	**11,25x**	**5,33x**	**26,67x**	**1,81x**	**2,29x**

Table 3: Inference time speedup of a GPGPU-based platform respect to OpenBLAS. Results correspond to most performing libraries employing their fastest primitive for single-thread and 32-bit FLT operations. QS-DNN shows the improvement of the search and clearly outperforms Random Search (RS) [de Prado et al. 2018b].

CONCLUSION AND FUTURE WORK

We have presented an AI pipeline to solve the Automatic Speech Recognition challenge. We have integrated a Key-word Spotting (KWS) system into the AI pipeline which addresses the current fragmentation of data, algorithms, hardware platforms and IoT services for building systems of Artificial Intelligence. The AI pipeline allows to link consecutive tasks by packaging the tools and dependencies and providing interconnectability between them. We have demonstrated the effectiveness of the pipeline by showing an end-to-end system to collect data, train and benchmark 3 different neural networks achieving up to 94% accuracy for KWS tasks.

We aim to extend the current status of the work for KWS by finalizing training of LSTM-based model and the last two steps of the pipeline: deployment optimization and IoT integration for KWS models. Thereon, we envision to fully optimize and deploy the trained models on low-power or IoT devices that can be employed for applications such as: healthcare sensing, wearable systems or car-driving assistant.

ACKNOWLEDGMENTS

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

REFERENCES

2002. 2000 HUB5 English Evaluation Speech. (2002). https://catalog.ldc.upenn.edu/LDC2002S09

2017. Google Speech Commands dataset. (2017). https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html

2017. ONNX. (2017). https://onnx.ai/

2018. Arm Compute Library. (2018). https://developer.arm.comtechnologies/compute-library

2018. Bonseyes Official Caffe 1.0 Version. (2018). https://github.com/bonseyes/caffe-jacinto

2018. Caffe. (2018). http://caffe.berkeleyvision.org/

2018. Docker. (2018). http://www.docker.com

2018. FI-ware Project. (2018). https://www.fiware.org/

Miguel de Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. 2018a. QUENN: QUantization engine for low-power neural networks. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 36–44.

Miguel de Prado, Nuria Pazos, and Luca Benini. 2018b. Learning to infer: RL-based search for DNN primitive selection on Heterogeneous Embedded Systems. *arXiv preprint arXiv:1811.07315* (2018).

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.

S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural computation, 9(8)* (1997).

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).

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

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*.

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.

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, USA, 807–814. http://dl.acm.org/citation.cfm?id=3104322.3104425

G. Saon, G. Kurata, T. Sercu, K. Audhkhasi, S. Thomas, D. Dimitriadis, X. Cui, B. Ramabhadran, M. Picheny, L.-L. Lim, B. Roomi, and P. Hall. 2017. English Conversational Telephone Speech Recognition by Humans and Machines. *ArXiv e-prints* (March 2017). arXiv:cs.CL/1703.02136

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. https://doi.org/10.1109/TSA.2002.800560

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