ALOHA: an architectural-aware framework for deep learning at the edge

P. Meloni D. Loi
G. Deriu
University of Cagliari, Italy
paolo.meloni@diee.unica.it

A. D. Pimentel
D. Sapra
University of Amsterdam, The Netherlands

B. Moser
N. Shepeleva
SCCH, Austria

F. Conti L. Benini
ETH Zurich, Switzerland

O. Ripolles
D. Solans
CA Technologies, Spain

M. Pintor
B. Biggio
Pluribus One, Italy

T. Stefanov
S. Minakova
Leiden University, The Netherlands

N. Fragoulis
I. Theodorakopoulos
Irida Labs, Greece

M. Masin
IBM Research, Israel

F. Palumbo
University of Sassari, Italy

ABSTRACT

Novel Deep Learning (DL) algorithms show ever-increasing accuracy and precision in multiple application domains. However, a step further is needed towards the ubiquitous adoption of this kind of instrument. First, effort and skills required to develop new DL models, or to adapt existing ones to new use-cases, are hardly available for small- and medium-sized businesses. Second, DL inference must be brought at the edge, to overcome limitations posed by the classically-used cloud computing paradigm. This requires implementation on low-energy computing nodes, often heterogenous and parallel, that are usually more complex to program and to manage. This work describes the ALOHA framework, that proposes a solution to these issue by means of an integrated tool flow that automates most phases of the development process. The framework introduces architecture-awareness, considering the target inference platform very early, already during algorithm selection, and driving the optimal porting of the resulting embedded application. Moreover it considers security, power efficiency and adaptiveness as main objectives during the whole development process.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**;

KEYWORDS

Deep Learning, Convolutional Neural Networks, Computer aided design

ACM Reference Format:
P. Meloni D. Loi, G. Deriu, A. D. Pimentel, D. Sapra, B. Moser, N. Shepeleva, F. Conti L. Benini, O. Ripolles, D. Solans, M. Pintor, B. Biggio, T. Stefanov, S. Minakova, N. Fragoulis, I. Theodorakopoulos, M. Masin, and F. Palumbo. 2018. ALOHA: an architectural-aware framework for deep learning at the edge. In *Proceedings of* . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Learning (DL) algorithms, often called Deep Neural Networks (DNN), currently represent the state-of-the-art approach in machine learning and artificial intelligence to complex problems such as image recognition, object identification, speech recognition, video content analysis, and machine translation [9].

Novel algorithm configurations continuously improve the precision of DL systems, often at the price of significant requirements in terms of processing power. Nevertheless, the edge computing paradigm pushes towards the deployment of DL inference tasks on embedded devices, to overcome limitations of cloud-based computing. When DL is moved at the edge, severe performance requirements must coexist with tight constraints in terms of power and energy consumption.

A promising solution to this problem relies on the use of parallel heterogeneous processing architectures. In this landscape, the implementation of modern DL systems becomes a very error prone and effort hungry activity that limits the adoption of DL instruments for small and medium software development companies, for two main reasons. First, the configuration of the specific DL algorithm usable to solve a problem is often chosen using a manual trial-and-error approach that relies on multiple training and evaluation iterations to compare different candidate configurations. With this approach, good algorithm configurations are hard to find in reasonable time, even for experts. Second, the programming of embedded heterogeneous systems to implement the inference requires advanced skills in parallel computing, and must be carefully tuned to the specific target heterogeneous architecture, in order to optimally exploit the underlying system in terms of performance and power.

Thus, there is a growing need for computer-aided design tools capable of assisting software developers in implementing DL algorithms on heterogeneous low-energy processing platforms in the embedded system industry. In this work we present a novel software development toolflow, composed of interacting utilities automating:

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(1) the selection of an optimal algorithm configuration able to meet the requirements of a specific application (use-case),
(2) the optimization of its partitioning and mapping on a heterogeneous low-energy target processing platform,
(3) the optimization of power and energy savings during its deployment.

The approach will been practically validated on two main reference platforms, NEURAghe [15] and Orlando [4], showing that it can actually support state-of-the-art computing technologies. In this paper we show a first set of experimental results highlighting the advantages of the proposed method.

2 RELATED WORK

Researchers communities and vendors are targeting different aspects of DL, from algorithm design to implementation on computing architectures. Recently, significant effort has been dedicated to the development of open-source deep learning toolkits aiming to improve the efficiency in building new neural network models and in training and testing them on production-scale data, including Theano [19], Caffe [8], TensorFlow [5] and CNTK [3]. All the mentioned tools typically target Graphic Processing UNits (GPUs) as their primary target platform, often using libraries such as NVIDIA CUDA and cuDNN [7] under the hood, and desktop Central Processing Units (CPUs) as a second target. Despite continuous advancements in GPU-related technology, GPUs are far from being the lonely actors in this field. A wide landscape of novel very power- and performance-efficient processing architectures are emerging on the market and in literature, often endowed with accelerators and specialized hardware for speeding-up the most computation-intensive tasks and/or reduce power consumption, such as convolution layers in CNN. Successful examples are the Tensor Processing Unit from Google [6] and the NVIDIA Deep Learning Accelerator. Other approaches rely on specialized hardware or consider FPGAs as target implementation technology, thanks to their flexibility in terms of logic and IOs.

Although the majority of architectures comes with a dedicated middleware layers implementing computing primitives (cuDNN, AuvizDNN, Tensor Processor Unit SDK), a power- and performance-efficient implementation of new algorithms on the platform requires a DSE process. The designer has to select, without support from existing software development utilities, the right fine-tuning parameters of the mapping configurations, including partitioning of layers in sub-layers, assignment of operations to processors, precise scheduling of operations and data transfer.

AutoML[1] provides tools for assessing with a design space exploration the impact of changing hyperparameter values on the precision and on the performance of a target algorithm. Automated algorithm design approaches mostly optimize for classification precision and only as a secondary objective try to reduce the computing workload. ALOHA aims at advancing state of the art in this field by implementing an application-level DSE environment that will tune all the mentioned porting-related parameters. The architectural features, described by means of an adequate architecture model, will be considered to define an optimal porting configuration. Implementation of such porting will be later automated using platform-specific support. The DSE will be made scenario aware to capture the different operation modes of the DL algorithms, thereby allowing to make runtime trade-off choices regarding various extra-functional properties such as system performance, energy consumption, precision.

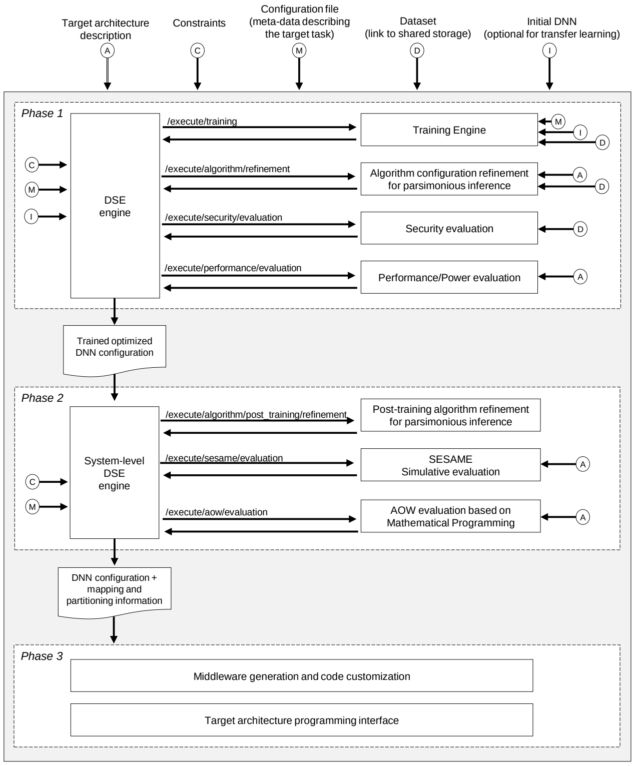
Target architecture description
A
Constraints
C
Configuration file (meta-data describing the target task)
M
Dataset (link to shared storage)
D
Initial DNN (optional for transfer learning)
I
Phase 1
C
M
I
DSE engine
/execute/training
Training Engine
M
I
D
/execute/algorithm/refinement
Algorithm configuration refinement for parsimonious inference
A
D
/execute/security/evaluation
Security evaluation
D
/execute/performance/evaluation
Performance/Power evaluation
A
Trained optimized DNN configuration
Phase 2
C
M
System-level DSE engine
/execute/algorithm/post_training/refinement
Post-training algorithm refinement for parsimonious inference
/execute/sesame/evaluation
SESAME Simulative evaluation
A
/execute/aow/evaluation
AOW evaluation based on Mathematical Programming
A
DNN configuration + mapping and partitioning information
Phase 3
Middleware generation and code customization
Target architecture programming interface

Figure 1: General toolflow overview

3 DESIGN FLOW OVERVIEW

An overview of the proposed toolflow is shown in Figure 1. The toolflow receives as inputs, a dataset, a set of use-case related constraints (security, performance and power), a configuration file, initial DNN(s) and hardware architecture/specification files. The main output of the flow is an *architecture aware* partitioned and mapped DNN configuration ready to be ported on the target computing platform. The overall tool flow can be divided into three different phases.

The first phase aims at automating the algorithm design process. It generates the optimal algorithm configuration taking into account the target task, the set of constraints and the target architecture that will execute the inference task. This is possible thanks to a decision-taking tool, called *Design Space Exploration (DSE) engine*, that creates a Pareto graph populated with design points corresponding to candidate algorithm configurations. To populate the Pareto graph, the DSE engine requests evaluation of the design points to a set of satellite tools, shown on the right-hand side of Figure 1, that assess different design points with respect to different metrics (accuracy, security, power, performance, etc..). The DSE uses design-space pruning techniques to reduce the number

[1] http://automl.org/

of evaluations to be performed, however exploration can require several iterations. At the end the DSE selects the optimal algorithm configuration, that is propagated to the next stage of the flow.

The second phase of the toolflow aims at automating a system-level design process, optimizing the partitioning and the mapping of the algorithm configuration selected by the DSE in the previous phase on the target processing platform. This phase generates an *architecture aware* partitioned and mapped application configuration ready to be transferred to the last stage of the toolflow. Similarly to phase 1, the design process is driven by a DSE engine, indicated as *System-level DSE engine* in Figure 1.

Finally, the third phase automates the porting of the target inference application on the target architecture, translating mapping information in adequate calls to computing and communication primitives exposed by the architecture. This phase exploits also the power- and performance-related knobs exposed by the platform (VFS, power and clock gating etc.).

In the following sections, a description of the components in each phase, and an overview of the integration methodology that allow them to work towards a common unified toolflow are presented.

3.1 Toolflow integration methodology

The ALOHA Toolflow integration is based on RESTful Microservice architecture, which is a software architectural style that designs an application as a collection of a loosely coupled, collaborating services. Microservice architecture has been used by a widely set of software leader companies such as Amazon, Ebay, Netflix or Uber between others[2]. To decompose the application into services, decomposition based on business capabilities[3] has been used.

Each of the Toolflow components implements a HTTP/REST API that can be accessed from other parts of the application. Containers are utilized to achieve the required level of isolation between modules. Given that components are exposing a stateless interface, an orchestrator module was implemented as controller of the application and data flows.

3.2 Toolflow components

This section outlines the role and functionality of the main toolflow components.

3.2.1 DSE Engine. The DSE engine drives the search for the optimal algorithm configuration through the vast design space, using iterative evaluation of candidate design points. It reads all the input files from a shared storage, which includes constraints, configuration, initial DNN(s) and hardware architecture/specification files, and converts representational formats, if needed, to communicate effectively with the different satellite tools. Subsequently, it initiates the exploration process. When the exploration is finished, the DSE engine triggers the next phase of the toolflow for system-level DSE. If no initial DNN is provided, by default, the DSE engine will generate a population of design points using random or minimum topologies.

Since performing a full exploration would be unfeasible due to excessive runtimes, the DSE engine uses a Genetic Algorithm (GA) to explore and prune the design space. Figure 2 shows the workflow of the proposed GA methodology. The process is initiated with a set of DNNs which then creates a genotype for use in the GA. This set of DNNs together make up the initial 'Population'. Each genotype in the population is evaluated and given a fitness score. Evaluation of parameters is provided by satellite tools (described below in more detail) and performed iteratively. With each iteration, we get solutions that are better than the ones in the previous generation and these have a higher probability of creating offspring for the next generation. These new off-springs are created through gene altering operations, like crossover and mutations, and these replace the low scoring, poorly performing solutions.

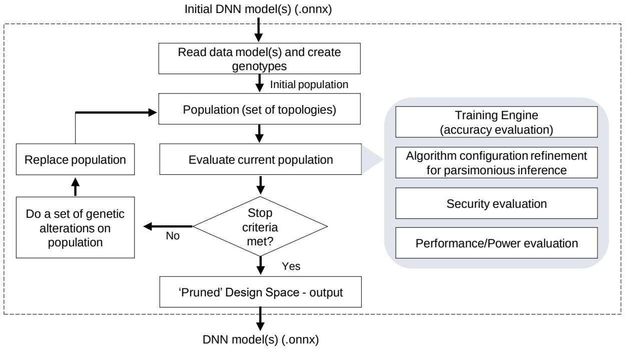
Initial DNN model(s) (.onnx)
Read data model(s) and create genotypes
Initial population
Population (set of topologies)
Evaluate current population
Replace population
Do a set of genetic alterations on population
No
Stop criteria met?
Yes
'Pruned' Design Space - output
DNN model(s) (.onnx)
Training Engine (accuracy evaluation)
Algorithm configuration refinement for parsimonious inference
Security evaluation
Performance/Power evaluation

Figure 2: Overview of the exploration process workflow

The iterations continue until the stopping criteria are met, which can be pre-defined satisfactory performance scores or a specified maximum number of iterations. The last generation achieved in this way gives us a set of topologies that satisfy the evaluation constraints and are best among the explored topologies. A Pareto graph is built using these resulting topologies along with their fitness scores in the respective evaluation modules.

3.2.2 Training Engine. The Training engine is the principal utility specifically dedicated for training in the proposed toolflow. It is accessed by the DSE engine to request the accuracy evaluation of a candidate algorithm configuration. It supports different training methods, and may start training from scratch or applying transfer learning to reuse pre-trained networks in a different use-case. Conceptionally, the training engine relies on the following specifications:

1. information baseline (domain description; training data, pre-trained models);
2. task (classification, regression, semantic segmentation etc.);
3. training scenario (training from scratch, fine-tuning, transfer learning);
4. model architecture (type of model; specification which parameters are free and which are fixed);
5. learning scheme (dataset augmentation, loss function, regularization, dropout, adversarial training; probabilistic metric for domain adaptation etc.);
6. optimization technique for hyper parameter tuning.

The output of the training engine comprises numerical values for the free parameters (weights and bias; hyper parameters) and meta information from analysis of the training and final model.

Figure 3 shows an overview of the Training engine workflow in conjunction with the toolflow components. Core part of the

[2] https://microservices.io/articles/whoisusingmicroservices.html
[3] https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

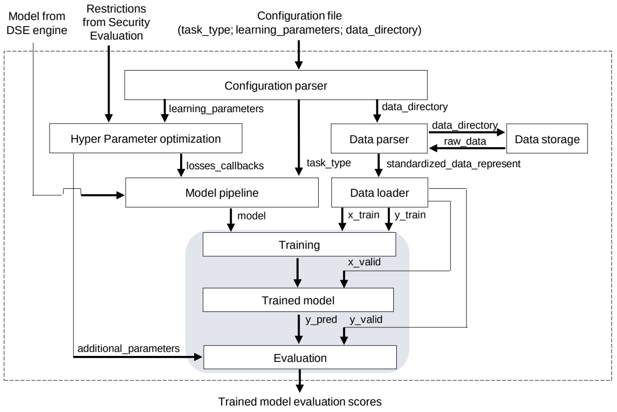

Figure 3: Overview of the Training engine workflow

Training engine serves for model training and evaluation. Moreover, it performs a local Hyper Parameter optimization to identify some training related configuration parameters that are not included in the model. Such optimization can be restricted (with directives propagated from the Security Evaluation component) avoiding exploration of some hyperparameters due to the security reasons (eg. optimizer function). Optimization of the hyper parameters and training are performed simultaneously on the model provided by the DSE Engine. At the end of this procedure, evaluation scores and trained model are send back to the DSE Engine.

3.2.3 Algorithm configuration refinement for parsimonious inference. This component of the toolflow, when requested by the DSE engine, tries to reduce the computing effort and the energetic cost of the execution of inference of a candidate design point. To this aim it can apply transformations from two classes to the DNN under refinement:

1. QUANTIZE: reduction of data precision (using different numerical representation formats in activations and weights).
2. PRUNE: remove low-impact connections between network layers.

The QUANTIZE class of transformations is meant to lower the data representation from the one used for the original floating-point training to one which allows for parsimonious inference on the target embedded device. Transformations of this class include: low-precision calibration, low-precision calibration + fine-tuning, Q-bit integer quantization [12], Q-bit INQ quantization [21], binarization [2], ABC-Net binarization [13].

The PRUNE transformations include both iterative pruning [11] and INQ pruning [21] to prune less relevant network weights. The INQ pruning uses fine-tuning of a pre-trained network. It can be used only together with INQ quantization, as an additional option. This component provides in output a modified algorithm description, after performing an optimization process that can be seen as "local search" within the overall exploration process.

3.2.4 Security evaluation. The security evaluation component evaluates design points proposed by the DSE engine in terms of security under adversarial input perturbations.

As shown in Figure 1, the security evaluation module receives a trained network model (corresponding to the current design point) and the dataset as inputs. For each data point, it then generates an adversarial perturbation that, when applied to the data point, maximizes its probability of being misclassified by the targeted trained network model. The adversarial perturbation is quantified by a distance measure computed between the source data point and its perturbed version, bounded by a maximum distance value *epsilon*. These attacks are known as evasion attacks or adversarial examples, and can be generated with state-of-the-art gradient-based algorithms. The security evaluation procedure amounts to measuring the attack success rate as a function of the maximum admissible input perturbation epsilon, i.e., how the classification accuracy drops as the maximum admissible input perturbation epsilon increases. Three levels corresponding to low, medium and high security will be defined to measure how quickly the accuracy drops as epsilon increases. According to the evaluation results, the security evaluation tool may also associate to a candidate algorithm configuration a prospective action (among a set of pre-defined measures) that may be used to improve its security level.

3.2.5 Performance/Power evaluation. This satellite tool evaluates the performance and the power consumption associated with the execution of the inference of a candidate design point on the target architecture. It receives as inputs one or several DNN models coming from the DSE engine, and the target architecture description (see Figure 1). The tool generates as output, for every DNN model, the following set of evaluated parameters: the DNN inference execution time in seconds (Performance), the DNN inference energy consumption in joules (Energy), the number of processors prospectively required for DNN inference (Processors), and the memory required for DNN inference in bytes (Memory). The evaluation of a DNN model is performed in three main steps. First, an internal DNN model representation is extracted from a specification of the input DNN. Second, a Cyclo-Static Dataflow (CSDF) model is generated from the DNN model, as a graph of concurrent tasks communicating data via FIFOs. Third, the CSDF model is evaluated in terms of performance, power/energy consumption, and resource usage. During the evaluation, the target architecture is taken into account. This step will be implemented by extending the open-source tool DARTS [4] with techniques for estimating the power/energy consumption of the SDF graph when executed on the target architecture.

3.2.6 System-level DSE engine. The system-level DSE engine controls the exploration of the design space exposed by different partitioning and mappings of the different inference software tasks and creates a Pareto graph populated with design points corresponding to candidate system-level configurations, featuring:

1. a partitioning of DNN actors in sub-actors;
2. a mapping of the DNN actors (or partitioned sub-actors) on the processing elements available in the target architecture;
3. a mapping of task-to-task communication items and intermediate variables on communication and storage structures available in the target architectures.

Granularity and nature of the mappable actors is extracted from the architecture description format. This ensures, on one hand, architectural awareness for the optimization-related decisions taken in this phase. On the other hand, this permits mapping decisions

[4] http://daedalus.liacs.nl/darts/

to be translated to actual code in the porting phase. To populate the mentioned Pareto graph, the system-level DSE engine requests evaluation of the design points to two satellite tools: Sesame framework [16] and Architecture Optimization Workbench (AOW) [14] (see Figure 1). The main difference between Sesame and AOW lays in the level of details. AOW explores the whole design space subject to system requirements and resource constraints (e.g., serializing processing cores and communication buses) using coarse-grain models for computation and communication, while Sesame can perform more precise simulation of both computation and communication over a more limited search space for better mapping. In the proposed toolflow, the synergy between AOW and Sesame is explored, where AOW finds "sweet" design spots and Sesame fine-tunes and verifies them. The system-level DSE engine uses design-space pruning techniques to reduce the number of evaluations to be performed. Exploration can require several iterations. At the end, the tool selects the optimal design point to be propagated to lower level of the design flow.

To find more efficient mappings of DNN actors to the underlying platform architecture and to optimize the usage of the available resources in the target architecture, the system-level DSE engine may also deploy transformations on the DNN algorithm graph by, for example, merging or splitting actors (i.e., increasing or decreasing the concurrency in the DNN algorithm). Alternatively, the DSE engine may also invoke the post-training algorithm refinement for parsimonious inference to achieve a workload reduction by considering specific features of the target architecture.

3.2.7 Post-training algorithm refinement for parsimonious inference. This component is responsible for performing, when invoked by the system-level DSE engine on a candidate DNN, a post-selection refinement of the DNN inference algorithm, to reduce the computation burden during inference. This satellite tool is based on Irida Labs‘ PI Technology [20]. At one usage mode, this technique can serve as a self-pruning mechanism for the underlying DNN model, thus enabling the elimination of unnecessary components and redundancies in the DNN structure, through an additional specialized post-training process. If the properties of the target hardware architecture are favorable, the same technique can act as a method for converting a static computing graph, to a dynamic graph which can exhibit significant reduction in the average computational load during inference. In such a graph, several components (i.e. convolutional kernels, groups of kernels, layers etc.) are conditionally executed according to learned rules, and based on the respective data being processed, by the means of some special, trainable processing modules called LKAMs (Learning Kernel Activation Modules).

To perform either of these operations, an internal model analysis process is initially used to identify the demanding nodes and algorithmic components that can deliver the largest computational gains. Upon identifying the components of interest, a set of specialized hyper-parameters are being defined based on the architecture of the processed DNN model, and some utility modifications (i.e. insertion of LKAMs) are applied without user interaction. After the completion of the initial analysis, a specialized post-training refinement process is undertaken in order to appropriately refine the model.

If the process can converge to a solution that delivers a more parsimonious inference, retaining at the same time the accuracy of the initial model within specified margins, the tool returns the modified model, otherwise notifies the DSE engine to proceed with the initial trained model.

3.2.8 Middleware generation and code customization. The Middleware generation and code customization support component includes a set of utilities and guidelines that:

(1) abstract the characteristics of the target platform;
(2) automate the translation of the partitioning and mapping description into a platform-specific code that exploits the programming primitives exposed by the target processing platform, that must be used to execute a processing or communication task on the hardware architecture;
(3) customize and instrument the code to reduce as much as possible the power consumption of the target hardware, using, when available, power reduction techniques such as power gating, clock gating, frequency scaling and others.

To interface these utilities with the rest of the toolflow, we have outlined a first version of the architecture description format. Such piece of information is intended to be produced by a prospective user, when targeting a specific architecture for the first time, or by the platform producer, to foster the adoption of its platform by the community. The format aims to provide a general description of the hardware platforms in terms of population of computing elements, connectivity, and available operating modes (data types, working frequency and gating conditions). Moreover, the architecture description format describes, for the target reference platform, a set of operators/actors representing the elementary computation and communication tasks that can be triggered on the computing resources exposed by the processing platform. These actors will correspond to those managed by the utilities in the previous phase, ensuring that the mapping information received as input by the middleware component are prone to be implemented on the target platform.

4 EXPERIMENTAL RESULTS

In this section we present a set of experiments that present the potential of the optimization techniques that are used in the ALOHA components.

4.1 Architecture-aware algorithm selection

A first experiment shows the potential of architecture-aware selection of algorithm parameters. We have performed a short exploration considering VGG-16 [17] as initial algorithm. We consider NEURAghe as target platform, whose capabilities are presented in Table 1[5]. NEURAghe is highly parameterizable, thus it can be configured to fit in different devices with different costs and power figures. In this experiment we have considered the LOSA configuration. Two custom DNNs, derived from VGG-16 have been identified, trained and tested using the architectural model and the training engine. DNNs have slightly different layer configurations than the original, respectively with a number of neurons increased (over-) or descreased (under-) to better match the size of the matrix of multiply-and-accumulate modules instantiated inside LOSA, to increase their utilization over time and computing efficiency. Results

[5] The name of the accelerator template derives from the ancient megalithic edifices named *nuraghes*, typical of the prehistoric culture in Sardinia. The different configurations are named after most important *nuraghes*. https://en.wikipedia.org/wiki/Nuraghe

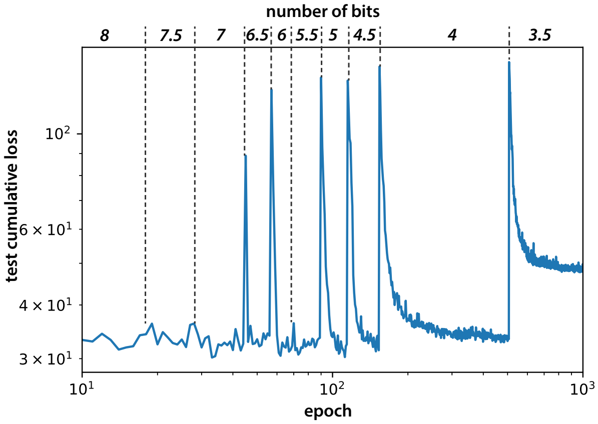

Figure 4: Cumulative test loss on CIFAR-10 for VGG-16 fine-tuned to different precisions from 8 to 3.5 bits, with weights and activations scaling simultaneously.

are reported in Table 2. Both custom DNNs execute more efficiently than the original. The over-dimensioned configuration undergoes an increased workload in the same execution time as the the original, allowing for a costless increase in accuracy. The downsized algorithm also increases efficiency, reduces execution time, at the expenses of a reduced accuracy.

4.1.1 *Performance* vs *accuracy trade-off.*

As detailed in Section 3.2.3, the ALOHA toolflow includes functionality to trade off an algorithm's accuracy with its energy cost. To test this functionality, we initially focused on the relatively simple scenario of applying optimization for parsimonious inference, to the VGG-16 topology trained on CIFAR-10. To this aim we moved to stricter constraints in terms of data representation, using a methodology derived from Hubara et al.'s [12] quantized neural networks (QNN). In our case, we apply the QNN method to a pre-trained CNN, after applying a set of standard transformations so that each CONV layer is always followed by a batch normalization layer, on turn followed by an activation quantization. We applied this methodology using a pretrained version of this algorithm as input and applying the following heuristic *relaxation strategy*. First, we keep track of $\Delta_{\mathcal{L}}$, the variation of the overall total training loss between different epochs, and we calculate it's running average and standard deviation ($\mu_{\Delta_{\mathcal{L}}}$ and $\sigma_{\Delta_{\mathcal{L}}}$) over E epochs (typically 10-20).

(1) if both $\mu_{\Delta_{\mathcal{L}}}$ and $\sigma_{\Delta_{\mathcal{L}}}$ are below hyperparametric thresholds μ_τ and σ_τ, we consider the training *stale*.
(2) if the training is *stale* for a N consecutive iterations (e.g. 2), and the absolute loss is below a given threshold L_τ, we drop the arithmetic precision by a given factor F.
(3) at each drop of precision, we reset the state of $\Delta_{\mathcal{L}}$.

We used $F = \sqrt{2}$ (corresponding to dropping precision of the equivalent of 0.5 bits at a time), $L = 100$, $E = 20$, $\mu_\tau = \sigma_\tau = 5$. At the start of the procedure we set the representation to Q1.15, i.e. a minimum representable value of $\varepsilon = 2^{-15} = 3.052 \times 10^{-5}$. By tweaking these parameters, the overall ALOHA flow can choose how much effort to dedicate to the quantization procedure. We applied the precision drop simultaneously to weights and activations; for inputs, we stopped dropping precision after reaching 8 bits (the native precision of input data).

Accuracy losses where negligible from 16 to 8 bits; Figure 4 shows the latter part of the relaxation procedure where quality decreases a bit more with each drop in precision, but is typically recoverable up to a certain degree. As shown in Figure 4, the relaxation procedure becomes slower with each drop, due to the increased difficulty in finding good solutions. We stopped this preliminary version of the procedure at 3.5 bits, where the impact on accuracy is more substantial and the network does not converge satisfactorily with respect to the hyperparameters we chose. Overall, we were able to keep the quality drop in terms of accuracy below 5% when switching from a 16-bit fixed point to a 4-bit network, capable to represent only 16 discrete values with $\varepsilon = 2^{-3} = 0.125$, with a 4× compression (only from quantization not accounting e.g. for pruning and/or additional compression opportunities granted by the correlation between kernel values). Table 1 provides an hint of the performance increase achievable on NEURAghe, when moving from 16 to 8 bit data precision.

4.2 Security evaluation

In this section we discuss the capability of the framework to: (*i*) assess the security of deep networks to adversarial examples, through the notion of security evaluation curves [1], i.e., showing how the performance of a model decreases under attacks crafted with an increasing level of perturbation; and (*ii*) improve the security of deep networks with *adversarial training*, i.e., by re-training the neural network including such attacks as part of the training data [10, 18]. It is worth remarking that the framework will include also other state-of-the-art defenses against adversarial examples, including explicit detection or rejection of such samples, and the use of specific hyperparameter configurations to mitigate this threat (e.g., varying the regularization term in the loss function optimized during training). We refer the reader to [1] for a more comprehensive discussion of such defenses, as well as of the algorithms used to craft the attacks. We discuss now an example of application of our framework to improve the security of a deep network on a task involving the recognition of MNIST handwritten digits. In particular, we consider a well-known convolutional neural network used for this task,[6] consisting of different convolutional layers with pooling and ReLU activations and a fully-connected output layer. We trained it on the MNIST training set (consisting of 60, 000 images), after normalizing all images in [0, 1] by dividing the pixel values by 255, and manipulated 10, 000 test samples with the Fast Gradient Sign Method (FGSM) attack algorithm [10]. This attack bounds the max-norm distance between the source image $\mathbf{x}$ and its adversarial counterpart $\mathbf{x}'$ as $\|\mathbf{x} - \mathbf{x}'\|_\infty \leq \varepsilon$. This basically means that every pixel p in the image $\mathbf{x}'$ is manipulated independently, in the interval $[p - \varepsilon, p + \varepsilon]$. We run this attack for $\varepsilon \in \{0, 0.01, 0.02, \ldots, 1\}$, and report the corresponding security evaluation curve in Fig. 5, showing how classification accuracy degrades under attacks characterized by an increasing perturbation ε. We finally applied a basic defense known as *adversarial training* [10], which suggests retraining the convolutional network by incorporating the attack samples in the training set. To this end, we run the attack on the training data and retrained the network on such samples along with the initial training samples. We then generated the attack samples with

[6] https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py

Table 1: Main features of different NEURAghe configurations.

	LOSA single 4x4	ARRUBIU dual 2x4	SABINA single 2x2	LOSA single 4x4	ARRUBIU dual 2x4	SABINA single 2x2	BANZOS single 1x1
Device	Z-7045	Z-7045	Z-7020	Z-7045	Z-7045	Z-7020	Z-7007s
DSP [#]; Freq [MHz]	864; 140	864; 140	216; 80	864; 140	864; 140	216; 80	54; 80
Benchmark net	ResNet 18	ResNet 18	ResNet 18	VGG 16	VGG 16	VGG 16	SqueezeNet
Actual (16 bit)	61.91	103	18.34	172.67	184	29.32	3.84
GOps/s per Watt (16 bit)	6.19	10.3	5.24	17.26	18.4	8.37	1.53
GOps/s per k$ (16 bit)	25	41	41	69	74	65	43
Actual (8 bit)	111.12	-	33.07	335.09	-	56.51	-

Table 2: Comparison betweeen original VGG-16 and custom NEURAghe-aware configurations

Benchmark	Performance (GOps/s)	Accuracy (Top-1)
VGG16	172.67	88.4%
NEURAghe-aware VGG16 (over)	182.43	89.6%
NEURAghe-aware VGG16 (under)	183.75	79.7%

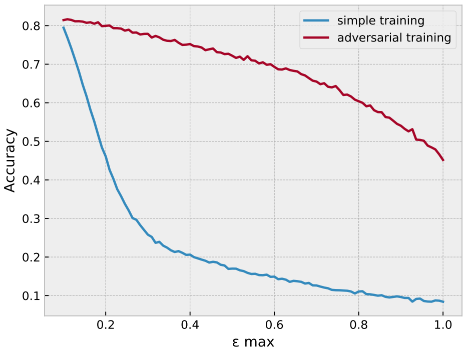

Figure 5: Security evaluation of the considered convolutional neural network under the FGSM attack with increasing perturbation ε.

the FGSM attack against the robust network. As shown in Fig. 5, the security of this network has been substantially increased by this simple countermeasure. However, it is worth remarking that stronger attack algorithms may be much more effective in this case. We refer the reader to [1] for further details. In Fig. 6, we show some examples of manipulated MNIST handwritten digits, able to mislead classification, in along with their corresponding adversarial perturbations (magnified to improve visibility).

4.3 Post-training parsimonious inference experiment

In this section we explain what kind of additional savings can be obtained applying post-training PI, and we present the results of the evaluation of PI technique on a VGG-16 model. We performed experiments using the VGG16 model on ILSVRC. The PI process involves addition of LKAMs into all convolutional layers but the first. The model was trained using the publicly provided pre-trained model for initialization, and with a rather aggressive pruning hyperparameter setting, aiming to a more lossy but economical inference. The resulting activity profiles of the trained parsimonious model are illustrated in Figure 7. The resulting model achieves a respectable of 70.4% accuracy, presenting a 2% deficit to the reference model, but with an impressive 48.31% reduction in the required FLOPs. The average kernel utilization is at the 66.14%, but more importantly, as can be seen in figure 10, seven of the convolutional layers are operating in a static or close-to-static mode, enabling the permanent pruning of the redundant kernels from the model.

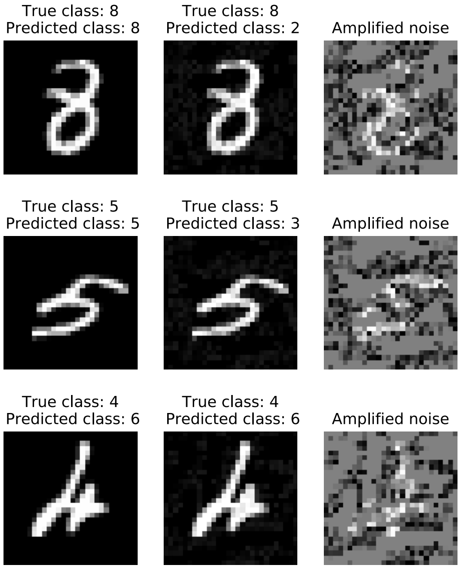

Figure 6: Manipulated MNIST handwritten digits that mislead classification by a convolutional neural network, crafted with the FGSM attack algorithm [10] with $\varepsilon = 0.05$. Note that, within this setting, the adversarial perturbations are almost imperceptible to the human eye, though still effective to mislead recognition.

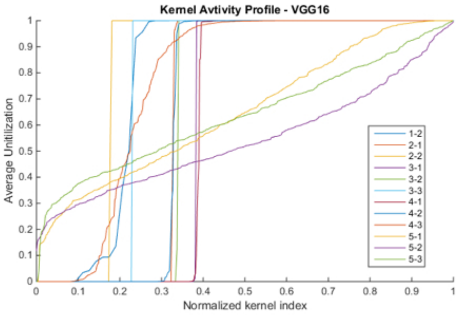

Figure 7: Kernel Activity Profile for every layer of VGG16 model on ILSVRC2012 classification challenge.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have highlighted the main features of the ALOHA framework, explaining the main components automating the development process of deep learning inference tasks on low-energy resource-constrained computing nodes. We have presented a first set of experiments, as a proof-of-concept demonstrating the potential of the proposed development techniques.

ACKNOWLEDGMENTS

This work was supported by EU Commission for funding ALOHA Project (H2020) under Grant Agreement n. 780788. The authors would like to thank all the participants taking part in the project for their support, including Giuseppe Desoli and Giulio Urlini from STMicroelectronics srl, Adriano Souza Ribeiro and Werner Kloihofer from PKE Electronics AG, Cristina Chesta from Santer Reply SpA, and Yaniv Ben Zriham from Max-Q Artificial Intelligence LTD.

REFERENCES

[1] B. Biggio and F. Roli. 2018. Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning. *Pattern Recognition* 84 (2018), 317–331.

[2] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]* (Feb. 2016). arXiv:cs/1602.02830

[3] Dong Yu et al. 2014. *An introduction to computational networks and the computational network toolkit.* Technical Report.

[4] Giuseppe Desoli et al. 2017. 14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC '17)*. IEEE, 238 – 239. https://doi.org/10.1109/ISSCC.2017.7870349

[5] MartÃŋn Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association Berkeley, 265 – 283.

[6] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, 1 – 12. https://doi.org/10.1145/3079856.3080246

[7] Sharan Chetlur et al. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:arXiv:1410.0759

[8] Yangqing Jia et al. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia (MM '14)*. ACM, 675 âĂŞ– 678. https://doi.org/10.1145/2647868.2654889

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press, Cambridge, MA.

[10] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations.*

[11] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]* (Oct. 2015). arXiv:cs/1510.00149

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *arXiv:1609.07061 [cs]* (Sept. 2016). arXiv:cs/1609.07061

[13] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards Accurate Binary Convolutional Neural Network. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 345–353.

[14] Michael Masin, Lio Limonad, Aviad Sela, David Boaz, Lev Greenberg, Nir Mashkif, and Ran Rinat. 2013. Pluggable Analysis Viewpoints for Design Space Exploration. *Procedia Computer Science* 16 (2013), 226–235. https://doi.org/10.1016/j.procs.2013.01.024

[15] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. 2017. NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs. *CoRR* abs/1712.00994 (2017). arXiv:1712.00994 http://arxiv.org/abs/1712.00994

[16] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.* 55, 2 (Feb 2006), 99–112. https://doi.org/10.1109/TC.2006.16

[17] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large Scale Image Recognition. *CoRR* abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556

[18] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *International Conference on Learning Representations.*

[19] The Theano Development Team and Rami et al. Al-Rfou. 2016. Theano: A Python framework for fast computation of mathematical expressions. (05 2016). arXiv:arXiv:1605.02688

[20] Ilias Theodorakopoulos, V. Pothos, Dimitris Kastaniotis, and Nikos Fragoulis. 2017. Parsimonious Inference on Convolutional Neural Networks: Learning and applying on-line kernel activation rules. *CoRR* abs/1701.05221 (2017). http://arxiv.org/abs/1701.05221

[21] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2016. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. (Nov. 2016).