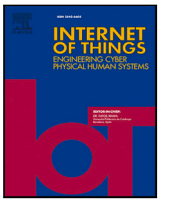
Internet of Things 21 (2023) 100665

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot



Research article

DeepEdgeSoC: End-to-end deep learning framework for edge IoT devices

Mhd Rashed Al Koutayni *, Gerd Reis, Didier Stricker

German Research Center for Artificial Intelligence, DFKI, 67663 Kaiserslautern, Germany

ARTICLE INFO

Keywords:
IoT
Edge computing
Artificial intelligence
Deep learning
Hardware acceleration
Quantization
High-level synthesis
FPGA

ABSTRACT

The acceleration of deep neural networks (DNNs) on edge devices is gaining significant importance in various application domains. General purpose graphics processing units (GPGPUs) are typically used to explore, train and evaluate DNNs because they offer higher processing and computational capability compared to CPUs. However, this comes at the cost of increased power consumption required by these devices for operation, which prevents efficient deployment of networks on edge devices. In the Internet of Things (IoT) domain, Field programmable gate arrays (FPGAs) are considered a powerful alternative since their flexible architecture can run the DNNs with much less energy. The enormous amount of effort and time required for the entire end-to-end edge-aware deployment urged us to develop DeepEdgeSoc, an integrated framework for deep learning (DL) design and acceleration. DeepEdgeSoc is an overarching framework under which DNNs can be built. DeepGUI, a visual drag-and-drop DNN design environment, plays an important role in accelerating the network design phase. In DeepEdgeSoc, the networks can be quantized and compressed to suite the underlying edge devices in terms of size and energy. DeepEdgeSoc goes beyond the software level by converting the networks to appropriate FPGA implementations that can be directly synthesized and integrated within a System-on-Chip (SoC).

1. Introduction

The availability of large amounts of data and the advancement in the processor industry have pushed deep learning (DL) to become the backbone for numerous application areas. Major sectors such as healthcare, the autonomous driving industry and security surveillance are increasingly relying on DL algorithms to solve their day-to-day challenges. Although DL is replacing the traditional hand-crafted methods, it comes at the cost of higher computational complexity. Therefore, data-centers are considered a suitable environment for the operation of DL networks. Nevertheless, there is a constant need for edge computing, i.e., to move DL networks from data-centers to user locations due to many reasons. For example, communication latency between data-centers and users limits application speed. Privacy requirements may also pose additional legal issues in data transfer to and from data-centers. Finally, moving processing to the edge provides additional protection against security threats such as sniffing. Trying to achieve plausible performance at the edge has always been a challenge. This is because it is not possible to leverage high-speed general-purpose graphics processing units (GPGPUs), with their enormous power requirements, at the edge. Furthermore, in contrast to the bulky hardware in data-centers, edge devices, such as car sensors and security cameras, tend to have small sizes. Moreover, environmental and operational constraints of edge devices require special care when moving from the center to the edge.

To overcome these limitations and perform computations closer to the user, appropriate embedded hardware platforms should be used. Advances in Internet of Things (IoT) technologies have led to the use of smart sensors that take over some of the processing

* Corresponding author.
E-mail addresses: rashed.al_koutayni@dfki.de (M.R. Al Koutayni), gerd.reis@dfki.de (G. Reis), didier.stricker@dfki.de (D. Stricker).

https://doi.org/10.1016/j.iot.2022.100665
Received 12 August 2022; Received in revised form 6 December 2022; Accepted 7 December 2022
Available online 22 December 2022

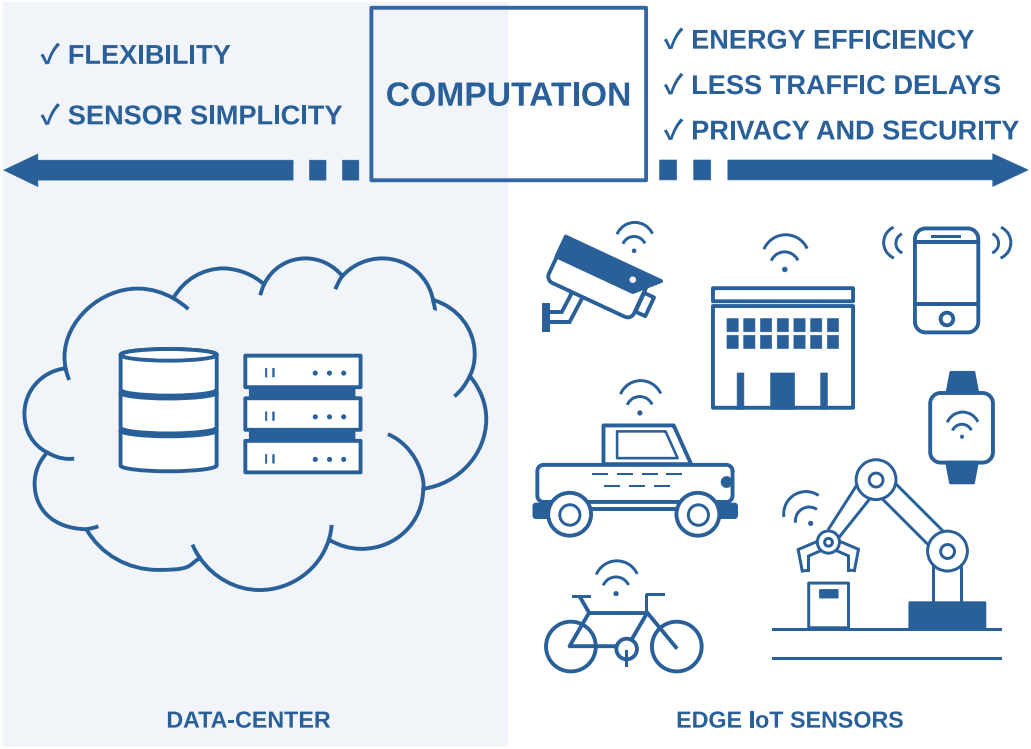


Fig. 1. Trade-offs between data-center and edge computation.

burden from data-centers. The responsibilities of such sensors range from simple data pre-processing to full execution of algorithms, where data-centers become superfluous. Undoubtedly, the higher the performance expectations of the sensor, the more complicated the hardware required and the more energy consumed. For example, in computer vision and image processing applications, the intelligent camera sensor must capture and process sequential images within a designated energy budget while maintaining run-time speed and result quality. Fig. 1 illustrates the trade-off that should be made based on how much computation is expected from the sensors. Shifting computation to the edge increases energy efficiency at the expense of more complex sensors and less development flexibility. On the other hand, sensors can be simple signal transducers when the power-hungry data-center handles the computational tasks. Field Programmable Gate Arrays (FPGAs) have emerged as a credible solution that can meet latency and energy requirements and close the performance-energy gap. In a previous work, the authors illustrated the efficiency of running a convolutional neural network (CNN) on an FPGA as compared to its GPGPU counterpart [1] and the literature is full of similar work. However, this does not mean that accelerating DL on edge FPGAs is a straightforward procedure. This is due to the large development gap that software developers encounter when trying to target the FPGAs to run their networks.

Originally, when it comes to DL research, a significant amount of experimentation time is spent on implementing and debugging novel DNN models from scratch. Furthermore, modifying and upgrading existing network architectures require noteworthy efforts. Since the resulting network uses the floating-point data type, it is less suitable to be directly deployed on an edge embedded device such as the FPGA due to the large memory footprint and increased power consumption required for operation. Therefore, additional optimizations are required in the quantization and compression of the network. Once the software version of the compressed network has been created, an efficient mapping into the hardware design space must be performed, which requires additional time and effort.

Taking all these challenges into account, we bridge the DL software–hardware development gap by introducing DeepEdgeSoC, an overarching framework that accelerates the design phase of DL research. DeepEdgeSoC provides a graphical interface that allows researchers to design, visualize and export complicated DL architectures. The DeepEdgeSoC framework accelerates the end-to-end DL design process and facilitates efficient edge-aware deployment. Specifically, we provide a powerful graphical interface to design DNNs from scratch, as well as parse and import existing Open Neural Network Exchange [2] (ONNX) models. Once the network is designed or imported, the computational flow is represented visually as a directed acyclic graph (DAG). This means that DL layers can be added, modified or deleted with just a few clicks. Furthermore, the hyper-parameters of each layer can be easily tweaked via the graphical interface. Network arithmetic (e.g., the height and width of a given layer's output feature maps) can be performed automatically during and after building the model. In addition to that, the DeepEdgeSoC framework allows quantization of network parameters and activation functions to arbitrary bitwidths. Finally, the hardware version of the network can be obtained thanks to our DNN2HLS sub-framework, which includes efficient hardware implementations for numerous DL layers and operations. Although DeepEdgeSoC primarily targets Xilinx FPGAs, it can be extended to support FPGA chips from other vendors (such as Intel/Altera)

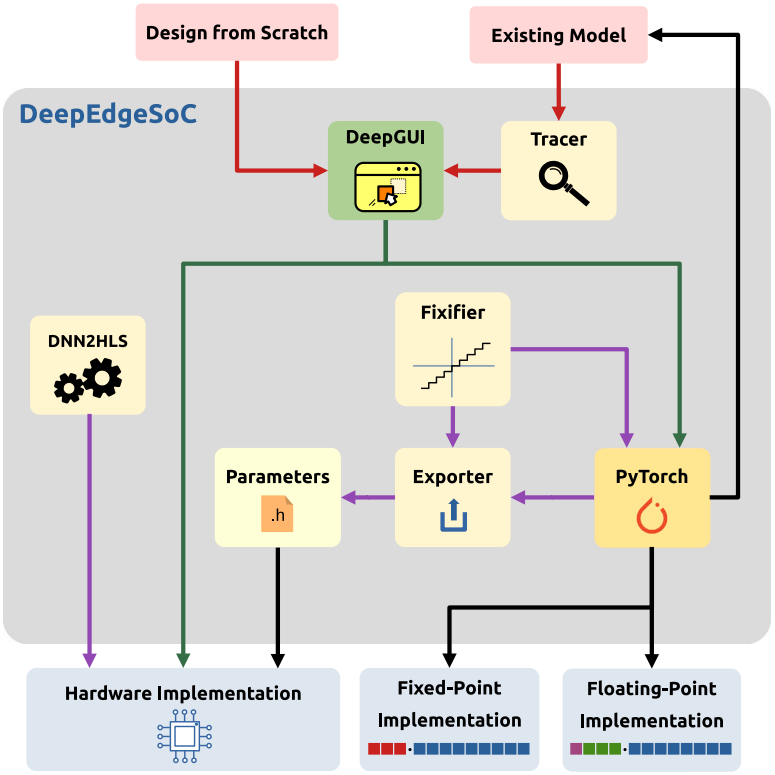


Fig. 2. The DeepEdgeSoC framework. DeepEdgeSoC is capable of converting existing network models into visually editable graphs thanks to the Tracer module. Also, novel network architectures can be graphically designed and modified using DeepEdgeSoC. The floating-point version of the network be exported directly along with training and evaluation scripts. Furthermore, a fixed-point version of the network can be created for post-training quantization (PTQ) and/or quantization-aware training (QAT) using the Fixifier module. Network parameters can be arranged in an appropriate format to be used in the hardware implementation, which can be readily obtained through DNN2HLS.

as well as other embedded platforms. To better explain the DeepEdgeSoC workflow, we provide two illustrative case studies in which we implement a CNN to classify a subset of 43 different German traffic signs and an LSTM-based bubble detector. We start the process by designing the network from scratch, as in the first example, or importing it using the DeepGUI, as in the second example. Then we train the networks to get a valid floating-point version. Afterwards, we compress the networks to get an acceptable fixed-point representation. Lastly, we obtain the hardware implementations of the networks and we port them to the PYNQ-Z1 FPGA development board using DNN2HLS. To assess the quality of the resulting system, we provide a comparison between the performance of the FPGA-based implementations and the implementations on a GPGPU as well as other embedded platforms, namely NVIDIA Jetson Xavier and RaspberryPi 3B+. Currently, DeepEdgeSoC mainly supports parsing PyTorch-based ONNX models [3] as input networks. Also, it is only possible to export the designed networks as PyTorch models. However, this work can be easily extended to support other frameworks such as Tensorflow and Keras.

The main contributions of this work can be summarized in the following points:

- An integrated DL framework that covers the development process of edge intelligence, starting with the design and training of neural networks and ending with their hardware implementation.
- PyTorch quantization layers that use straight-through estimators (STE) for arbitrary precision.
- A parameterized hardware framework that comprises various DL layers, including linear, convolutional, recurrent and activation layers in addition to skip connections.

The paper is structured as follows: in Section 2, we compare our work to the related work in the literature. In Section 3, we provide technical details about each component of the framework. Section 4 provides two test cases that were implemented using DeepEdgeSoC, namely German traffic signs classifier and bubble detector. Lastly, we discuss the paper in Section 5 and we analyze the strengths and limitations of the study.

2. Related works

The vital role that FPGAs play in accelerating DNNs in IoT systems has attracted attention to the development of numerous frameworks that perform the conversion of DNN models into equivalent hardware implementations.

2.1. Frameworks for IoT sensors

Recently, researchers have shown increased interest in taking the challenge of pushing algorithms towards IoT devices. Smart Ambient Behavior Observation System (SABOS), proposed by Irfan et al. [4], is an IoT solution for monitoring elderly care homes using non-visual-based sensors while maintaining privacy. Interestingly, they introduced a data reduction algorithm to minimizes the amount of traffic between SABOS and the cloud server. In their comprehensive survey, Adeel et al. [5] highlight the important role of Wireless Sensor Networks (WSNs) in disaster monitoring and management systems where battery lifetime of mobile nodes is critical. Furthermore, they compared various communication network technologies on which WSNs rely. Lin et al. introduced MCUNet [6], a framework that jointly explores neural architecture and schedules inference while targeting small-memory microcontrollers. Similarly, a bi-directional co-design approach [7] is proposed to jointly optimizes DNN models and their deployment on FPGAs.

2.2. Frameworks for deploying DNNs on FPGAs

The closest framework available in the literature is CNN2Gate [8] as it shares most of the features with our DeepEdgeSoC. CNN2Gate, built upon PipeCNN [9], is able to parse ONNX models, quantize them, export the parameters and generate the hardware implementation. fpgaConvNet [10] can also optimize the mapping Caffe or Torch CNNs onto FPGAs. hls4ml [11] is a framework mainly designed for the development of FPGA-based trigger and data acquisition systems used in the field of particle physics. This framework is capable of converting CNNs, including fully-connected layers, into corresponding High-Level Synthesis (HLS) implementations. Based on a template-driven workflow, CNN-Grinder [12] generates accelerated CNNs for programming low-end-low-cost FPGA System-on-Chip (SoC). FP-DNN [13] supports mapping Tensorflow CNNs, RNNs and Residual Nets onto low-end FPGA boards. DeepHLS [14] converts Keras DNN descriptions to synthesizable C codes. It also supports floating-point and fixed-point data types. Nevertheless, the previously mentioned frameworks, except PF-DNN, do not provide a mechanism to map Recurrent Neural Networks (RNNs) to the FPGA and are mainly concerned with CNNs. Furthermore, the network description has to be available beforehand and cannot be designed or extracted through a graphical interface.

2.3. Frameworks for DNN visual design

Although Tensorboard [15] provides the ability to visualize DNNs, it is not possible to modify the computation graphs because they are read-only. There exist a number of DNN design frameworks that offer powerful drag and drop GUIs for DL design. PrototypeML [16], for example, facilitates modular DNN design with multiple inputs and outputs. It also supports skip connections, which are fundamental to ResNet-like architectures. DeepCognition DL Studio [17], Sony's Neural Network Console [18] and IBM Watson [19] are integrated frameworks that enable developers to design, train and validate different DNN models. However, there is still a gap between the powerful frameworks mentioned above and the deployment of DL on edge devices. This is due to the lack of quantization support, as well as the ability to produce FPGA-friendly implementations.

Appendix contains detailed tables that show similarities and differences between our presented work and previous frameworks.

3. Framework anatomy

Fig. 2 shows the overall architecture of the DeepEdgeSoC framework. It consists of 5 main components: *Tracer*, *DeepGUI*, *Fixifier*, *Exporter* and *DNN2HLS*. Tracer is an interface module that parses ONNX representations of deep neural networks and converts them into graphically presentable graphs. DeepGUI is a graphical interface for visualizing architectures, allowing researchers to add, modify or delete layers without writing a single line of code. Fixifier is a modified version of PyTorch's neural network module that allows parameters and activation functions to be quantized on the fly to arbitrary bitwidths . The Exporter module automatically extracts the parameters of each layer and prepares them in a suitable format for hardware use. DNN2HLS is an AXI4 stream-based library that provides efficient hardware implementations for various DL layers and operations. The following subsections provide informative explanations for each of the mentioned components.

3.1. The tracer module

The Tracer module allows researchers and developers to import existing models instead of having to redesign them from scratch. Under the hood, this module takes a pre-processed ONNX network graph and converts it to a scope-aware graph. The resulting hierarchical graph contains the computational nodes as well as their parameters and hyper-parameters. During this process, it is possible for the user to exclude undesired nodes from the graph, i.e., shunt them. The FPGA hardware streaming architecture requires each computational node to have a single input and a single output. Therefore, each node with multiple outputs is replaced by a single output node of the same type, followed by a special node called Split. Similarly, each multiple input node is replaced by a single input node of the same type, followed by another special node called Merge. To visualize the network, the graph visualization algorithm by Gansner et al. [20] is used to assign the appropriate coordinates (x, y) to each node while trying to avoid overlapping with other nodes and edges as much as possible. Finally, the Tracer module exports the processed network graph to the DeepGUI format. The exported graph is a union of node and edge sets. Each element in the node set is a network layer coupled with the corresponding metadata such as the hyper-parameters, (x, y) coordinates and the hierarchy level to which the node belongs. Elements of the edge set are represented by an ordered pair of nodes, where each pair (n_i, n_j) indicates a directed edge from n_i to n_j.

3.2. DeepGUI

DeepGUI is a browser-based, portable framework that replaces programming effort with an easy-to-use drag-and-drop interface that accelerates the process of designing new networks from scratch and improving existing network models. In addition, DeepGUI supports the design of hierarchical neural networks by enabling the grouping of associated network layers as a reusable module. In DeepGUI, it is easy to visually draw the network graph by adding or removing layers. Furthermore, a layer's hyper-parameters can be modified using a side palette. In contrast to the few platforms available for DL visualization, our presented framework goes beyond the software-level design. Specifically, DeepGUI can automatically generate an edge computing-aware version of the network that can be ported to the FPGA with negligible effort.

Behind the scenes, DeepGUI traces the deep neural network graph and optimally maps each node to a corresponding hardware-level processing element (PE), while mapping each edge in the graph to a corresponding hardware-level stream. Fig. 3 illustrates a design example using DeepGUI. In this screenshot, a modular CNN with a skip connection was built. The palette on the left provides different DL layer types categorized by their functionalities. Moreover, the configuration palette on the lower left allows the user to change the hyper-parameters of a particular layer. As can be seen in the figure, DeepGUI offers two export formats: Python for software and C++ for hardware.

3.3. Fixifier

Deep neural network compression plays an important role in the process of deploying DL algorithms on an edge-computing platform. This is because portable edge devices are constrained by their battery capacity, on-chip resources and limited memory. Thus, the network architecture should be as lightweight as possible, consuming as little power as possible while maintaining acceptable accuracy and speed. There are two main approaches for quantizing neural networks: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) [21].

In PTQ, the model's pre-trained parameters, as well as the activations are quantized to a fixed-point representation with specific bitwidths. While in QAT, the network is retrained under quantization constraints for parameters and activations. For both approaches, there should be a mechanism to quantize parameters and activations adopted by the DL layers. Existing DL frameworks provide fake quantization layers capable of converting network arithmetic from floating-point to 8-bit integer operations. Although this may be suitable for processor architectures such as GPUs and CPUs, we are targeting FPGA platforms that allow the efficient use of arbitrary bitwidths for arithmetic. Thus, 8-bit integer arithmetic is not always the optimal quantization scheme for a network to be deployed on an FPGA.

We introduce DeepEdgeSoC's Fixifier, a customized PyTorch neural network module that supports quantization using arbitrary bitwidths for parameters and activation functions. This module is mainly responsible for converting a floating-point-based network into a corresponding lightweight fixed-point network, hence the name "Fixifier". In Fixifier, the quantization operation for weights, bias and activation of a particular layer can be performed or bypassed using a set of control flags. Moreover, the bitwidths for weights and activation functions can be configured on the fly during training. What makes Fixifier so desirable is its backward compatibility with PyTorch's original neural network module. In other words, if all quantization control flags are set to false, the behavior of Fixifier is identical to that of the original PyTorch module, whereas the quantization mode is activated as soon as the control flags are set correctly. In this case, PTQ can be applied simply by specifying the desired bitwidths and evaluating the resulting quantized network directly. Fixifier supports the QAT quantization scheme by allowing parameter and activation quantization only in forward propagation, while the quantization functions will be bypassed during back propagation. This trick, called a "straight-through estimator" [22], is necessary because the quantization function is a piece-wise constant function that breaks the back propagation chain by its vanishing gradients.

3.4. Exporter

The functionality of this module is to export each computational node's parameters (i.e., weights and biases) to a hardware-friendly format. The Exporter module allows parameters to be exported in two formats, namely floating-point and fixed-point. The arbitrary fixed-point precision is suitable for deploying a quantized version of a network onto hardware. The Exporter module automatically detects the dimensions of each layer's parameters and exports the corresponding array definition.

3.5. DNN2HLS

After training and quantizing a deep neural network, substantial time and effort is needed to convert the software network model into a corresponding optimized hardware architecture. To accelerate this process, a mechanism is required to perform the mapping into hardware space. Consequently, a generic and flexible hardware library for various DL layers is required. Therefore, we introduce DNN2HLS, an optimized DL hardware framework that can efficiently transform DNNs into HLS hardware implementations for Xilinx FPGAs. This framework provides the basic DL layers such as convolution, pooling, non-linearity, linear and recurrent layers. Table 1 lists the supported DL layers and hyper-parameters that can be tuned via DeepEdgeSoC. The parallelization (unroll) factor in a layer describes the degree of parallelization in that particular layer. At the hardware level, DNN2HLS converts the DNN into a hardware streaming architecture by mapping each layer to a separate hardware block. These blocks are then interconnected via AXI4 stream channels to form a pipeline. In contrast to the traditional computation flow, where each layer has to wait until all input is available, a

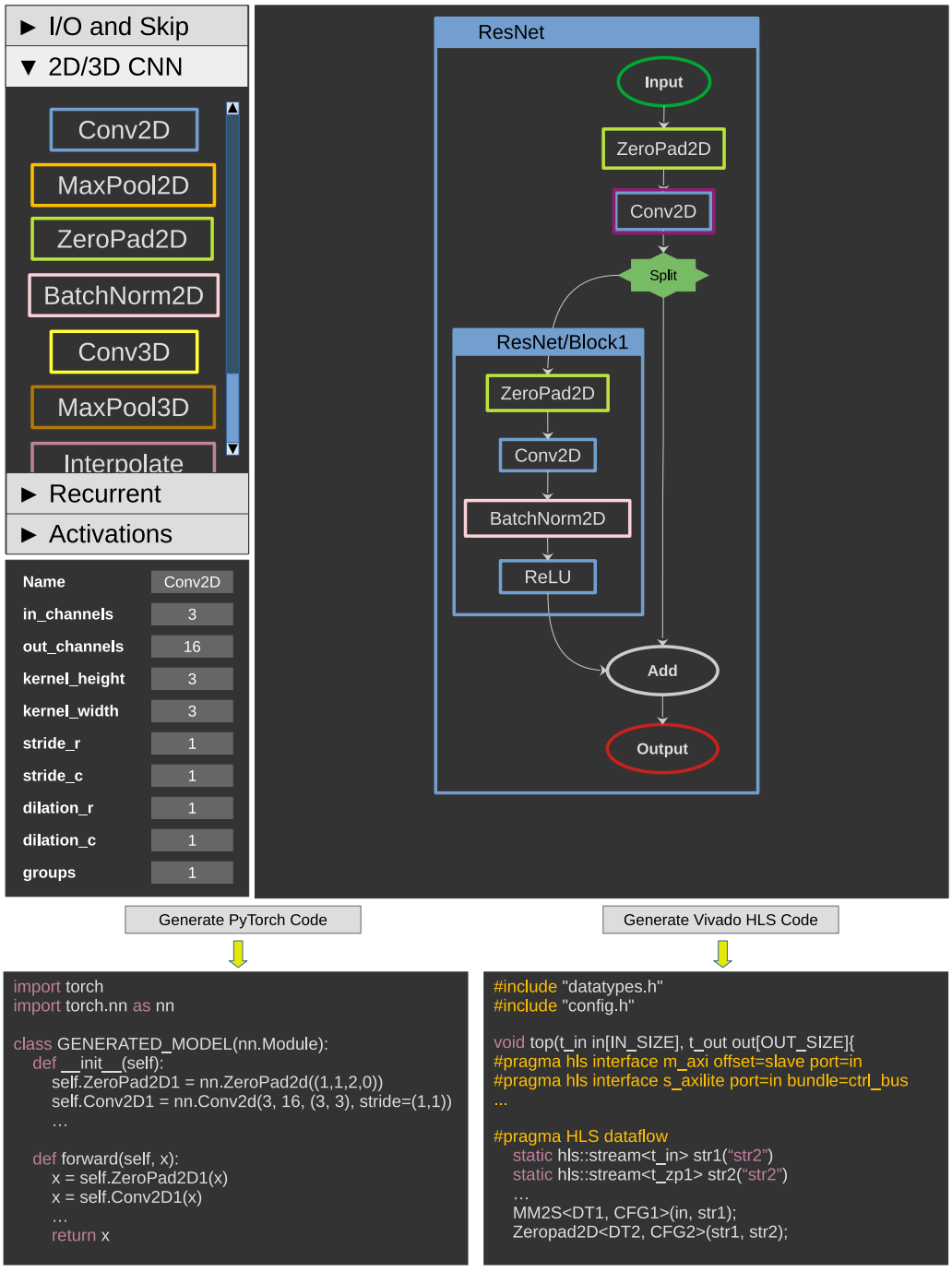


Fig. 3. DeepGUI. This screenshot shows an example of how a simple modular ResNet model can be designed using the drag and drop interface. Also, it illustrates the ability to tune the hyper-parameters and export the network in various formats, namely Python for software and C++ for hardware.

streaming layer can begin computation as soon as the previous layer provides the minimum required input. Thanks to C++ template generalization, hardware implementation of DNN layers is highly customizable. Specifically, each layer can be easily customized by tuning hyper-parameters, data types and bitwidths used for parameters and activation functions within that particular layer. This greatly minimizes the time spent on research and development, allowing for more prototyping. Currently, reusability of the same hardware module for multiple layers in a time division multiplexing manner is not supported. If the network, together with its parameters, does not fit on the selected FPGA, a larger FPGA with more available resources should be selected. Moreover, any change in the network layers or parameters requires the whole design to be re-synthesized.

Table 1
Supported layers and their tunable hyper-parameters.

Layer	Hyper-Parameters
Linear	Input size, output size, parallelization factor
2D Convolution	Input channels, output channels, input dims, kernel dims, stride, dilation, input groups, unroll factor
2D Max/Avg Pooling	In/out channels, input dims, kernel dims, stride
2D Zero Padding	In/out channels, input dims, padding rows (top+bottom), padding columns (left+right)
2D Batch Norm	In/out channels, input dims
3D Convolution	Input channels, output channels, input dims, kernel dims
3D Max Pooling	In/out channels, input dims, kernel dims
RNN	Input size, output size, parallelization factor
LSTM	Input size, output size, parallelization factor
GRU	Input size, output size, parallelization factor
ReLU	–
LeakyReLU	Slope
Sigmoid	–
TanH	–
Softmax	–
Split	In/out channels, input dims
Concat	In/out channels, input dims

If the targeted platform is a pure FPGA, the architecture produced by DNN2HLS is sufficient to configure it and make the network run. However, in the case of SoCs, the chip comprises two parts: the Programmable Logic (PL) and the Processing System (PS). This means that in addition to the architecture generated by DNN2HLS for the PL, the PS must be programmed to be able to communicate with it. There is no unique way to program the PS, as the code can change according to the intended application, the format of available data, input and output devices, and the memory addresses. For this case, a PS code template in Python for PYNQ boards and in C++ for general use are provided.

In the following, we provide an overview of our custom hardware architectures for common DL layers.

3.5.1. Convolutional and pooling layer architecture

The convolutional layer is considered the basic feature extraction block, where one or more learned filters (kernels) are applied to the input tensor by cumulatively multiplying each weight by the corresponding input value. For an input tensor of dimension ($C_{in}\times H_{in}\times W_{in}$), a kernel tensor of dimension ($C_{out}\times C_{in}\times K_H\times K_W$) is needed to obtain an output tensor of dimension ($C_{out}\times H_{out}\times W_{out}$). C_{in}, C_{out}, H_{in}, W_{in}, H_{out}, W_{out}, K_H and K_W denote the number of input channels, the number of output channels, the input height, the input width, the output height, the output width, the kernel height and the kernel width respectively. Normally, the kernel height and width are smaller than those of the input tensor. Since the kernels are shifted line by line across the input, an output cannot be computed until the appropriate number of lines are buffered in a special internal memory called the Line Buffer. The dimensions of the line buffer are given as ($C_{in}\times K_H\times W_{in}$). Synchronously with the line buffer, the input patches are copied to a Window Buffer of size ($C_{in}\times K_H\times K_W$) where they are made temporarily available for the convolution operation. The data inside the window buffer is then passed to a multiplier array so that the convolution multiplications can be performed in parallel. The resulting partial products are then passed to an adder tree to perform the accumulation part of the convolution operation and add the appropriate learned bias. Finally, the resulting value is streamed out via the output buffer. Fig. 4 illustrates the hardware architecture of the 2D convolutional layer. An important hyper-parameter, called the unroll factor, is tuned at design time to determine the degree of parallelization in the convolutional layer. In case of no parallelization, the convolutional multiplications are performed sequentially using a single multiplier. The next level of parallelization corresponds to performing (K_W) multiplications in parallel, while processing the rows in sequence. If the parallelization degree is further increased, the hardware architecture depicted in Fig. 4 emerges, where the ($K_H\times K_W$) multiplications are performed simultaneously. This means that the input channels are processed in sequence, unless the parallelization is increased to perform the whole ($Cin\times K_H\times K_W$) multiplications in a single clock cycle. Obviously, the unroll factor is mainly responsible for the space–time-trade-off, since increasing this factor speeds up the convolution at the expense of more hardware resources.

The pooling layer is implemented in a very similar manner using a set of line and window buffers. However, instead of the multiplier array, a set of comparators is used to find the maximum (minimum) value of a given input patch in case of max (min) pooling. For average pooling, an adder tree and a divisor are used to calculate the average input value. We refer the reader to our past paper [1] for more analytical and technical details about convolutional and pooling layers. In that work, the reader can also find further information about stride, dilation and group hyper-parameters.

3.5.2. Linear layer architecture

Linear layers (also known as "fully connected layers") play an important role in DL tasks such as feature extraction, classification and regression. The name of this layer comes from the fact that every input contributes to every output by a certain factor, called weight. Therefore, the total amount of weights in this layer is given as ($N_{in}\times N_{out}$) where N_{in} and N_{out} are the input and output sizes, respectively. The output is calculated as the vector–matrix multiplication between the input vector and the weight array. Additionally, a bias vector of size (N_{out}) is added to the output.

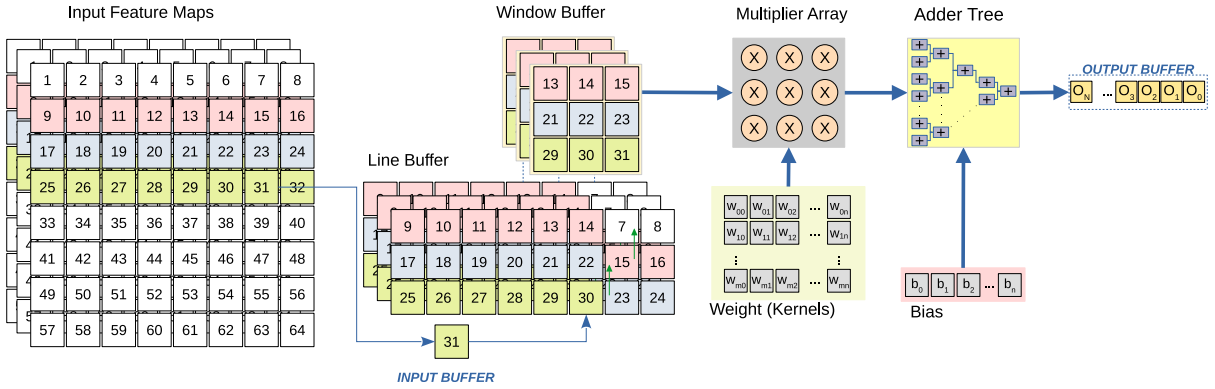


Fig. 4. Convolutional layer architecture.

The hardware realization of the linear layer consists of a set of multipliers, adders, a router, output accumulators, an output multiplexer and control logic as shown in Fig. 5. Each input value is read from the input stream and multiplied with the corresponding weights. These partial multiplications are then used to update the output accumulators. An output multiplexer is needed to decide whether the values in the accumulators are ready to be streamed out (to the output stream) or still have to be sent back to the adders for accumulation. The accumulators are initialized by the biases instead of zeros to save (N_{out}) additions.

Using ($N_{in} \times N_{out}$) multipliers would perform the task of the linear layer in a single clock cycle. However, this is usually not feasible due to limited hardware resources. Another limitation is that only one value can be read from the input stream at a time. Otherwise, the input values must be additionally buffered. Conversely, a single multiplier can be used to perform the total amount of multiplications in ($N_{in} \times N_{out}$) clock cycles, which is basically undesirable due to the very long latency for practical values of N_{in} and N_{out}. Therefore, we offer the flexibility to choose the degree of output parallelization by introducing the parallelization factor (p) of this layer. In this case, a router is needed to conduct the accumulated values for updating the corresponding subset of accumulators. The control logic is then required to orchestrate the work of the different hardware blocks. Particularly, the control logic determines when to pop a value from the input stream, which subset of weights to be used by the multipliers, when to initialize the accumulators, where the partial accumulations should be routed, and whether the calculation is finished.

3.5.3. Batch normalization layer architecture

The use of batch normalization layers is considered advantageous in DL because they speed up the training and handle the internal covariate shifts. This type of layer processes every input channel separately and therefore maintains the same number of channels at its output. After training and during inference, the running mean μ and variance σ^2 are used to normalize the input x, as in Eq. (1).

$$\hat{x} = \frac{x - \mu}{\sigma^2} \tag{1}$$

Afterwards, each output feature map z is calculated by multiplying the corresponding normalized input feature map $\hat{x}$ by a learned channel weight γ and adding a learned channel bias β, as shown in Eq. (2).

$$z = \gamma \cdot \hat{x} + \beta \tag{2}$$

The aforementioned operations (especially division) are expected to require a huge amount of on-chip resources. Thus, these two steps are folded into a single step by finding the relationship between the final output z and the original input x by substituting Eq. (1) into Eq. (2):

$$z = \gamma \cdot \hat{x} + \beta = \gamma \cdot \frac{x - \mu}{\sigma^2} + \beta \tag{3}$$

which leads to:

$$z = W \cdot x + B \tag{4}$$

where:

$$W = \frac{\gamma}{\sigma^2} \qquad B = \frac{-\gamma \cdot \mu}{\sigma^2} + \beta \tag{5}$$

In this case, we only need to calculate the constant parameter vectors W and B offline and store them on-chip instead of storing μ, σ^2, γ and β. Furthermore, the batch normalization can be performed using a single multiplication by W and a single addition to B instead of the division mentioned before. It is noticeable that the size of W and B is equal to the number of input channels. In other words, all values in a particular input feature map C_i are multiplied by the same $W[i]$ and added to the same $B[i]$. The input tensor is streamed in channel-last manner, i.e., consecutive stream values are taken from consecutive channels. Therefore, the weights and biases must be queried in a round-robin fashion. The W and B vectors are illustrated as ring buffers in Fig. 6.

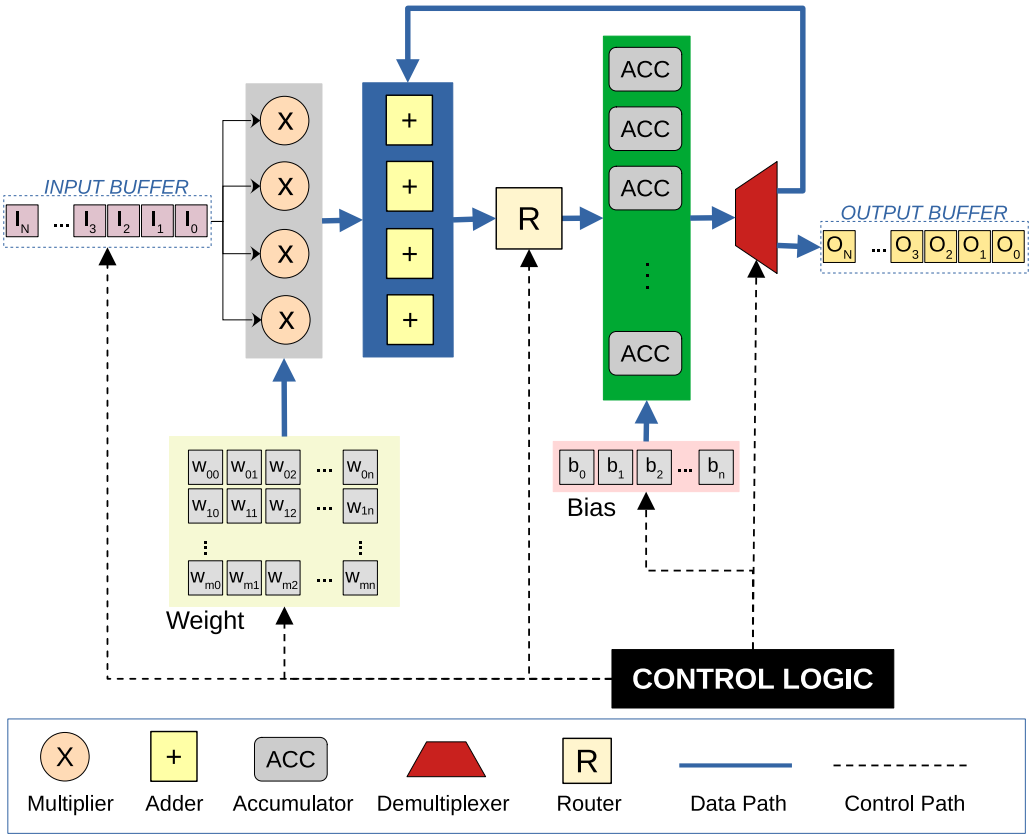


Fig. 5. Linear layer architecture.

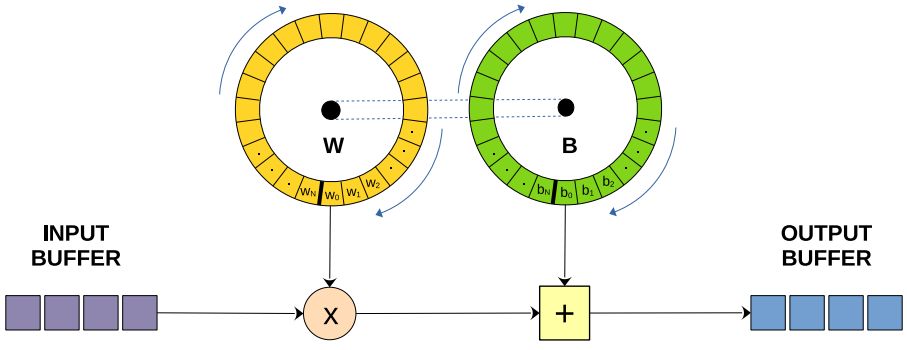


Fig. 6. Batch normalization architecture.

3.5.4. Recurrent layer architecture

Unlike feed forward networks, in which the current output is derived from the current input independently of the past, Recurrent Neural Networks (RNNs) are a family of neural networks that produce their output or sequence of outputs based on a temporal input sequence. At the core of recurrent networks, a memory cell is required to maintain information from the past. Intuitively, one can feed the output back as an auxiliary input to preserve the memory effect. This implementation, known as Classic or Vanilla RNN, suffers from vanishing or exploding gradients. That is, the gradients might quickly decay to zero or approach infinity during training, which prevents the convergence of the trained network and consequently limits its memory range to short input sequences. To overcome this issue, Hochreiter and Schmidhuber [23] introduced the Long Short-Term Memory networks (LSTMs) where the gradients are allowed to flow unchanged without vanishing. Besides, the problem of exploding gradients can be easily solved by squashing or clipping the gradients. As a result, LSTMs are able to overcome the gradient issues and therefore process longer input sequences. Traditional problems such as speech recognition, video analysis and object tracking can be efficiently addressed using DL thanks to LSTMs.

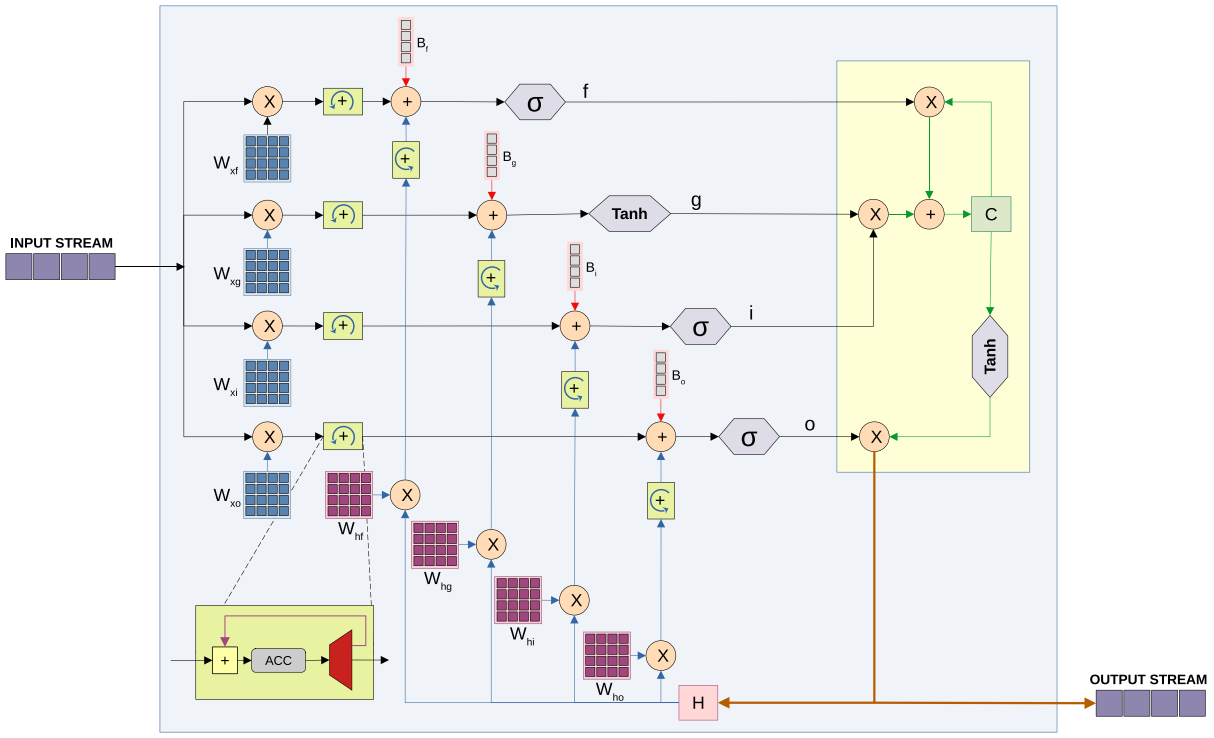


Fig. 7. LSTM architecture.

An LSTM layer can be characterized mainly by its input size (S_{in}) and its hidden or output size (S_{out}). Internally, LSTMs employ 2 different memories, the hidden state (h_t) and the cell state (c_t), each of size S_{out}. Furthermore, 4 different gates work together to control the contribution of the input and the memories to the output. The following formulae describe the mechanism behind LSTMs:

$$i_t = \sigma(x_t \cdot W_{xi} + h_{t-1} \cdot W_{hi} + b_i) \tag{6}$$

$$o_t = \sigma(x_t \cdot W_{xo} + h_{t-1} \cdot W_{ho} + b_o) \tag{7}$$

$$f_t = \sigma(x_t \cdot W_{xf} + h_{t-1} \cdot W_{hf} + b_f) \tag{8}$$

$$g_t = tanh(x_t \cdot W_{xg} + h_{t-1} \cdot W_{hg} + b_g) \tag{9}$$

While the internal state update formulae are given as below:

$$c_t = \sigma(g_t \cdot i_t + c_{t-1} \cdot f_t) \tag{10}$$

$$h_t = o_t \cdot tanh(c_t) \tag{11}$$

where i_t, o_t, f_t and g_t are the input, output, forget and cell gate tensors, respectively, each having the size (S_{out}). W_{x*}, W_{h*} and b_* are the input weight, hidden state weight and bias tensors which have the size ($S_{out} \times S_{in}$), ($S_{out} \times S_{out}$) and (S_{out}), respectively. Functions $\sigma(.)$ and $tanh(.)$ are the Sigmoid and TanH activations, respectively. The subscripts t and $t-1$ denote the current and the previous temporal step, respectively.

Fig. 7 shows the hardware architecture for LSTM. It is clear from Formulae (6)–(9) that multiply–accumulate operations (MACs) must be performed between the weights (W_{x*}, W_{h*}) and the corresponding tensors (x_t, h_{t-1}). Therefore, an accumulator block is used, consisting of an adder, a result accumulator cell (ACC), and a multiplexer to route the accumulated value. The workflow within an LSTM layer begins by reading an input value x_t from the input stream and consequently replicating it four times to be multiplied by the corresponding weights W_{x*}. Meanwhile, the internal state h_{t-1} is also multiplied by its corresponding weights W_{h*}. Once the multiplications are done, the results are forwarded to the activation functions ($Sigmoid, TanH$) as mentioned in the gates' formulae. Finally, the internal states (h_t, c_t) are updated according to the update Formulae (10) and (11), as mentioned earlier. An LSTM layer can be dynamically accelerated by specifying the parallelization factor during the design phase (default value is 1) so that more accumulations can be performed in parallel, reducing the total number of clock cycles required for inference.

3.5.5. Zero padding layer architecture

As we go deeper into a DNN, the feature maps almost always shrink in height and width while the number of channels increases. This causes the input feature maps of a particular layer to become slightly smaller than what that layer expects. In order to fix this

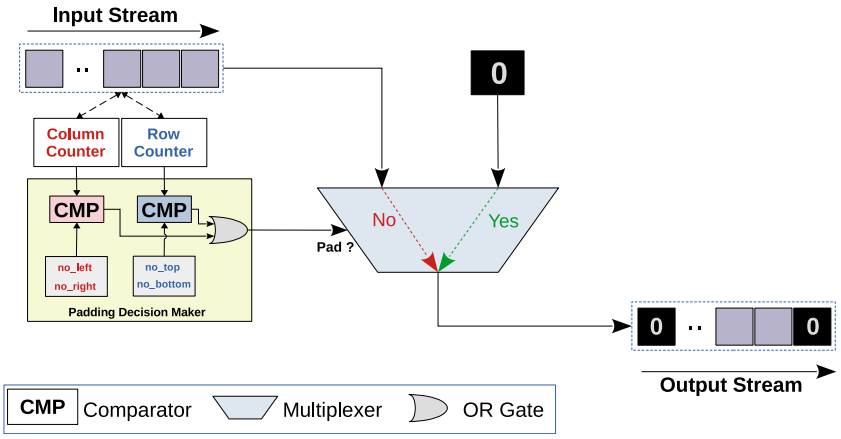


Fig. 8. Zero padding architecture.

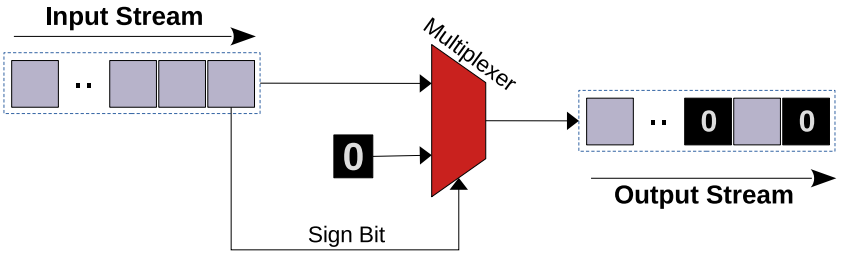


Fig. 9. ReLU architecture.

issue, a zero padding layer is required in between. In particular, adding a set of zero-valued rows and/or columns at the borders solves this problem and adapts the input to the next layer.

Fig. 8 shows the hardware architecture of the zero padding layer. A padding decision maker keeps track of input feature maps horizontally and vertically by reading the values of two counters (namely, row and column counters). Based on that, it decides whether to shunt the input to the output or push out a zero instead. The decision signal controls the multiplexer, which routes the correct value to the output.

3.5.6. Activation layer architectures

ReLU. The rectified linear unit (ReLU) is a very famous activation function that, despite being very simple to implement, plays an important role in adding non-linearity to a DNN and allowing it to work as a general function approximator. Unlike other activation functions (e.g., Sigmoid), the ReLU function is efficient, fast to compute, and does not suffer from vanishing gradients during training. At its core, it clones every positive value from the input to the output while setting negative values to zero. Fig. 9 shows our implementation of the ReLU function in hardware. Interestingly, the sign bit of the input value can be used to drive the multiplexer. In other words, if the input is positive, the sign bit is zero and the multiplexer passes the input to the output. Otherwise, a zero is pushed into the output stream.

Sigmoid and TanH. The Sigmoid function $\sigma(t)$ is given as:

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{12}$$

while the TanH function $tanh(t)$ is given as:

$$tanh(t) = \frac{e^{t} - e^{-t}}{e^{t} + e^{-t}} \tag{13}$$

The implementation of such computationally-expensive mathematical functions (since they consist of exponentiation and division) on FPGA is a challenging process. FPGAs provide powerful acceleration solutions for various algorithms, but typically have limited on-chip computation and storage resources. A naive, straightforward implementation of a computationally-expensive function might quickly exceed the available resources. Furthermore, executing such an implementation might incur major delays that affect the overall run-time of a particular algorithm. An alternative way to implement these functions is to convert them into look-up tables by storing sampled function values in a vector. In such manner, a sought function value can be looked-up in the table instead of being calculated. For this purpose, an input t should be converted to an index corresponding to the nearest function value.

Softmax. We consider the implementation of the Softmax function:

$$Softmax(x_1|(x_1, x_2, \dots, x_n)) = \frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}} \tag{14}$$

Implementing a multi-variable function as a look-up table can quickly become impractical. This is because in the general case the values of the function $y = f(x_1, x_2, \dots, x_n)$ must be stored in an n-dimensional look-up table, whereas only a 1-dimensional look-up table is needed for single-variable functions. The size of the multi-variable look-up table grows exponentially with the number of variables. For the n-dimensional Softmax function, we need to generate an n-dimensional look-up table. If we consider an example of $n = 4$ where each variable is in the range $[-5, +5]$ and we sample this range to 128 values, we get a look-up table of size:

$$S = (128)^4 = 268,435,456 \tag{15}$$

Therefore, we follow the same methodology proposed by [24] where the Softmax function is decomposed into two functions, namely $f_1(x) = e^x$ and $f_2(x) = \frac{1}{x}$. Since these two functions are unbounded, we need to choose a suitable input range for our application. Next, we create two look-up tables, *exp* for the function $f_1(x)$ and *inv* for the function $f_2(x)$. Now for each set of inputs: $x_1, x_2, \dots, x_n$:

- every e^{x_i} is looked-up in the *exp* table.
- the sum of e^{x_i} is calculated to get the Softmax's denominator.
- the inversion of the sum *inv_sum* is looked-up in the *inv* table.
- every Softmax value is calculated as: $e^{x_i} * inv\_sum$.

As can be seen, we need 3 loops to cover all inputs, and we need to perform $n + 1$ look-ups.

In the special case of the 2-variable Softmax function, it is possible to obtain a single-variable Softmax function that matches the Sigmoid function:

$$Softmax(x_1|(x_1, x_2)) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \longrightarrow \sigma(t) = \frac{1}{1 + e^{-t}} \tag{16}$$

This can be done by dividing the numerator and the denominator by the non-zero value e^{x_1}:

$$Softmax(x_1|(x_1, x_2)) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{1}{1 + e^{x_2 - x_1}} = \frac{1}{1 + e^{-(x_1 - x_2)}} = \sigma(x_1 - x_2) \tag{17}$$

That is, by setting $t = x_1 - x_2$, one can easily converts the two-variable Softmax function into a single-variable Sigmoid function. Now we can simply calculate t and use the exact same Sigmoid table generated previously. This is very efficient for binary classification tasks such as face detection, where each output is classified as either *face* or *background*.[1]

4. Case studies

This section presents two examples of using DeepEdgeSoC to design and implement DNNs with different layer types. In the first example, a network consisting of convolutional, pooling layers and skip connections is designed from scratch. The capabilities of DeepEdgeSoC are further illustrated by the second example, in which a convolutional recurrent network is implemented.

4.1. German traffic sign classification

We demonstrate how the end-to-end design process is accelerated using our proposed framework by experimenting with an example network model for traffic sign classification. For this purpose, the German Traffic Sign Recognition Benchmark [25] (GSTRB) dataset, which consists of 43 different traffic sign classes is used. GSTRB is distributed on 40,482 RGB images, each of size 48 × 48. The dataset is split into a training set (35,600 images), a validation set (3,609 images) and a test set (1,273 images).

We conduct this experiment in three phases. First of all, a network is designed from scratch using the DeepGUI. Afterwards, the network is exported and trained in full-precision mode. Then, we move to the quantization mode.

4.1.1. Network design phase

In this phase, the DeepGUI web client is used to design a simple CNN network that performs the task of traffic sign classification. The input image is first passed to a 7 × 7 convolutional layer, which produces 32 channels for the output feature maps. A non-linear activation function, namely ReLU, is then applied to these feature maps. Subsequently, the dimensions are reduced using a max-pooling layer (kernel size = 2 × 2). Afterwards, the resulting feature maps are fed to 2 branches. The first branch consists of two convolutional layers (kernel size = 4 × 4), each with a ReLU activation and a max-pooling layer (kernel size = 2 × 2). The second branch contains a convolutional layer (kernel size = 7 × 7) along with a ReLU activation and a max-pooling layer (kernel size = 5 × 5). Throughout both branches, the number of channels is kept at 32. The output of both branches is then concatenated along the channel dimension and passed to the fully connected layer with output size 43 followed by a log softmax layer. Since the second branch has less propagation delay, a FIFO buffer must be inserted in this particular branch after the max-pooling layer to synchronize the two branches. Fig. 10 illustrates the example network used in this work.

[1] The Python3 source code for converting the Sigmoid and softmax functions to look-up tables is available under: https://github.com/dfki-av/FunctionLUT/.

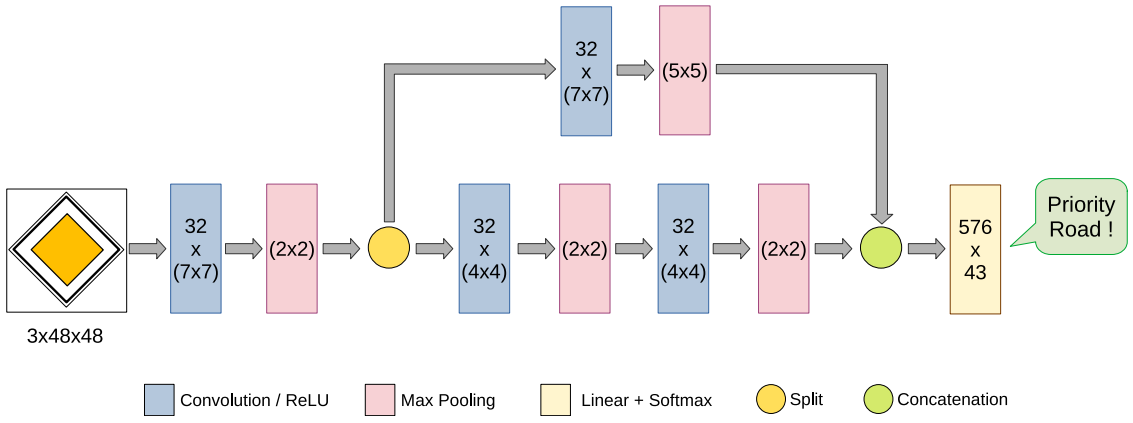


Fig. 10. The architecture of the CNN-based traffic sign classification algorithm. For the convolutional/pooling layers, the numbers between brackets indicate the kernel size, while the other number indicates the number of output channels. The number for the fully connected layer indicates the input/output sizes of that particular layer.

Table 2
PYNQ-Z1 Resource utilization for the floating-point implementation of the traffic sign classifier.

Resources	Used	Available	Util
Block RAM Tile	545	280	195%
DSPs	69	220	31%
CLB Flip Flop	47,788	106,400	45%
CLB LUTs	40,556	53,200	76%

4.1.2. Full-precision phase

The full-precision version of the network is exported and trained for 100 epochs using the SGD optimizer with an initial learning rate of 0.05. Validation is performed at the end of each epoch, and the learning rate is further reduced by a factor of 0.1 every time the validation loss does not improve for 3 consecutive epochs. The batch size used for training is 64. The negative log-likelihood loss is used as it is suitable for multi class training tasks.

The performance of the network is evaluated using the test set, and the achieved test accuracy is 91.52%. After the network training is completed, the weights are exported to the appropriate C++ header file format. The VHLS implementation is also obtained from DeepEdgeSoC. In Vitis HLS, the hardware network is first cross-checked by simulation. Afterwards, the algorithmic high-level description of the network is converted into the corresponding RTL representation using high-level synthesis. The PYNQ-Z1 evaluation board, which contains an xc7z020-clg400-1 SoC chip, is used for prototyping. The PL clock frequency is set to 250 MHz. The utilization and the on-chip resources can be seen in Table 2.

4.1.3. Fixed-precision phase

As can be seen in the first phase, the chip is over-utilized by the resulting full-precision hardware implementation. This is because floating-point arithmetic consumes plenty of on-chip resources. Therefore, a trimmed-down version of the network is obtained by exploiting quantization. First of all, we export the pre-trained network and attempt to quantize it without performing any training steps (i.e., PTQ). The resulting network is evaluated using the test dataset. We choose 4 quantization bits for the weights and biases of the convolutional layers and the fully connected layer. The mechanism for selecting the optimal combination of quantization bits for the parameters and the activations is beyond the scoop of this work and therefore not discussed here. Since the accuracy has dropped notably to 71.17%, QAT must be performed. After training the network for 100 epochs using the same hyper-parameters mentioned in the previous design phase, the achieved test accuracy is 90.89%. Since the accuracy drop in this case is acceptable (0.63% abs.; 0.688% rel.), we proceed to the hardware implementation. At this stage, the quantized parameters are exported and the same hardware implementation as in the previous section is used. However, the data type is changed from floating-point to fixed-point before the high-level synthesis process is performed again. The FPGA chip utilization is drastically reduced, as shown in the 3 table. The exported RTL modules can then be imported into Vivado IP Integrator (IPI), where the interface between the PL and the PS is configured. Subsequently, the integrated system can be synthesized into a bitstream that can be used directly to configure the FPGA. The same PYNQ-Z1 board is used to deploy the obtained bitstream of the traffic sign classification network.

4.1.4. Comparison and discussion

The results of both phases reveal that, thanks to the quantization, the network could be compressed to fit the chip. To illustrate the efficiency of the accelerated FPGA implementation, a comparison is made with the original implementation on the GPGPU used for training, the GeForce GTX 1070. Network inference is performed on both platforms, and the run-time and energy consumption

Table 3
PYNQ-Z1 Resource utilization for the fixed-point implementation of the traffic sign classifier.

Resources	Used	Available	Util
Block RAM Tile	145	280	52%
DSPs	42	220	19%
CLB Flip Flop	21,201	106,400	20%
CLB LUTs	22,653	53,200	43%

Table 4
FPGA vs. GPU performance analysis for the traffic sign classifier.

Platform	Run-Time [ms]	Energy [mJ]
FPGA (ours)	4.80	5.28
GeForce GTX 1070	1.33	35.91

Table 5
FPGA vs. different embedded platforms performance analysis for the traffic sign classifier.

Platform	Run-Time [ms]	Energy [mJ]
FPGA (ours)	4.80	5.28
NVIDIA Jetson Xavier (GPU)	5.05	30.30
NVIDIA Jetson Xavier (CPU)	8.49	119.71
RaspberryPi 3B+	97.31	506.01

per frame are measured for each platform. In order to measure the image run-time, the average run-time per image is calculated after running the inference on the entire test set (1,273 images). For the power measurement, only the dynamic energy consumed for image inference is considered: $E_{image} = (P_{total} - P_{idle}) * \Delta T_{image}$, where P_{total} is the total power consumed during computation, P_{idle} is the power consumption in idle mode, i.e., without inference and ΔT_{image} is the image run-time.

Table 4 shows that the run-time delay for one image on the GPU is 1.33 ms while it is 4.80 ms on the FPGA. Furthermore, the energy consumption per image on the GPU reached 35.91 mJ compared to 5.28 mJ on the FPGA. In summary, although the FPGA implementation is 3.6× slower, it is 6.8× more energy efficient compared to its GPU counterpart.

To provide a fair comparison, the FPGA implementation is compared to two existing embedded platform, namely NVIDIA Jetson Xavier and RaspberryPi 3B+. NVIDIA Jetson Xavier consists of a 512-Core Volta embedded GPU with Tensor Cores, 8-Core ARM v8.2 64-Bit CPU and 16 GB 256-Bit RAM. Moreover, RaspberryPi 3B+ employs a Broadcom BCM2837B0, Quad core Cortex-A53 (ARMv8) 64-bit SoC CPU running at a maximum frequency of 1.4 GHz with 1 GB SDRAM. The original PyTorch model obtained by the GPU is deployed on these devices. No quantization is performed on these embedded platforms because they use floating-point as the native data type for computations. In a live demo scenario, an image should be processed immediately after capture. Thus, image batches are not used for inference.

Table 5 illustrates the run-time delay and energy consumption per image for the mentioned platforms.

To visualize the numerical results, Fig. 11 shows the performance analysis for the different embedded platforms mentioned above. The numbers on the arrows illustrate the factors by which the performance differs for different implementations.

4.2. Bubble detection

In this section, we give another example of deep neural network design using DeepEdgeSoC by implementing the convolutional recurrent neural network (CRNN) presented by Tan et al. in [26]. The network plays an important role in monitoring continuous bladder irrigation (CBI) by making the correct decision to flush sterile fluid through the catheter into the bladder. This is essential to prevent blood clot formation and retention after hemorrhagic surgeries on prostate and bladder, allowing the free flow of urine. The authors have used a CRNN to detect the presence of an air bubble in the measured fluid and therefore improve the reliability of blood measurement. In this experiment, the pre-trained network is imported instead of designing it from scratch. Afterwards, the quantization step is performed by first trying the PTQ and then moving to QAT. Finally, the hardware implementation is exported and the deployment is performed on the FPGA.

4.2.1. Network setup phase

In this phase, we examine the aforementioned CRNN network and import it to DeepEdgeSoC. The network expects a 80×70 grayscale image. In the beginning, the first convolutional layer generates 8 feature maps using its 5×5 kernels. Afterwards, a max-pooling layer reduces the dimensions using a 2×2 pooling kernel. The second convolutional layer unifies the 8 feature maps to a single feature map by applying 5×5 convolutional kernels. Subsequently, the output feature map is scaled down using a 2×2 pooling layer. A ReLU activation function is appended to each convolutional layer. The resulting feature map is then flattened before being sent to the first LSTM layer (360×32) whose output is forwarded to the second LSTM layer (32×32). Lastly, a linear layer is used to classify the input image as bubble or no bubble. Fig. 12 illustrates the CRNN for bubble detection.

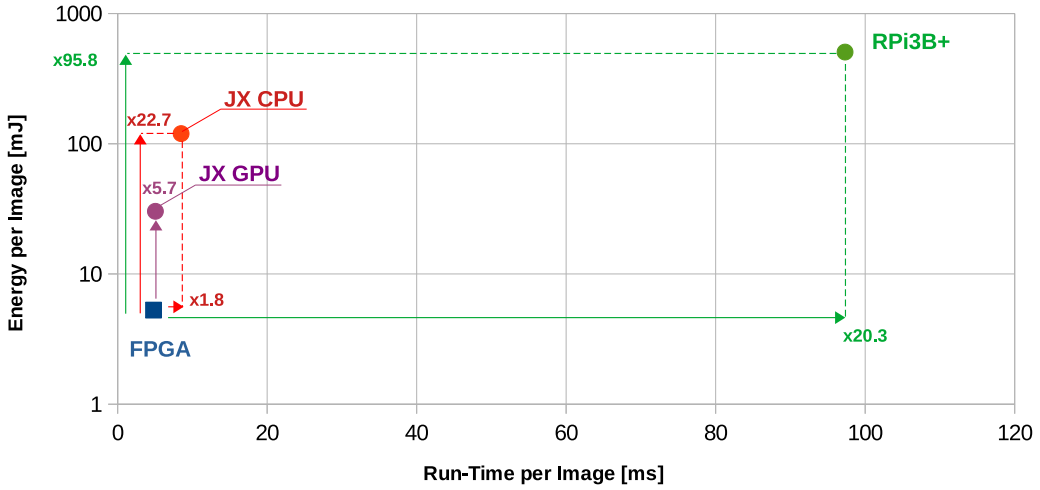


Fig. 11. Performance analysis for different embedded platforms; FPGA, JX GPU, JX CPU and RPi3B+ denote the performance of our PYNQ-Z1, Jetson Xavier embedded GPU, Jetson Xavier embedded CPU and Raspberry Pi 3B+ CPU respectively.

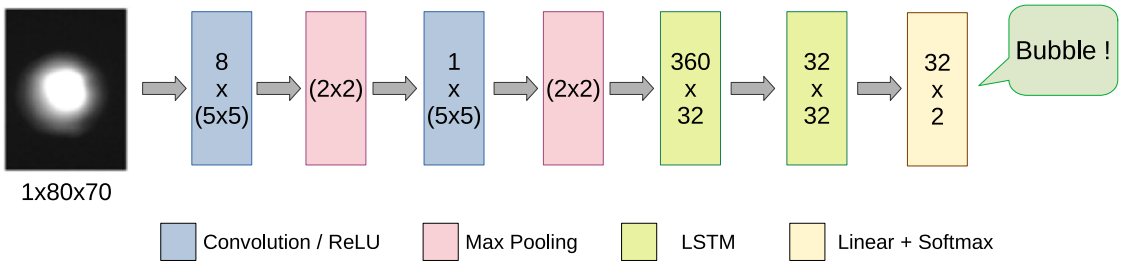


Fig. 12. The architecture of the Convolutional Recurrent Neural Network (CRNN)-based bubble detection network. The numbering annotation is used in a similar manner as in Fig. 10. However, the number for the LSTM layer indicates the input/output sizes of that particular layer.

Table 6
PYNQ-Z1 Resource utilization for the floating-point implementation of the bubble detector.

Resources	Used	Available	Util
Block RAM Tile	233	280	83%
DSPs	429	220	195%
CLB Flip Flop	73,320	106,400	67%
CLB LUTs	79,928	53,200	150%

4.2.2. Full-precision phase

DeepEdgeSoC is used here to generate the hardware implementation of the given pre-trained network. As stated in the original paper, the accuracy of this network is 97.4%. Since the network is relatively small (59099 parameters), the PYNQ-Z1 development board is again chosen for this experiment. By following the same development process as before, i.e., simulation first then high-level synthesis, we obtain an implementation that is too large to fit on the chip (Table 6).

4.2.3. Fixed-precision phase

This time, the network parameters are quantized to 5 bits. A quantized version of the network is obtained by first applying PTQ. In this case, the accuracy drops notably to 82.41%. Therefore, the network must be re-trained using the same hyper-parameters mentioned in the original work. As a result, the accuracy reaches 96.61% which has a negligible difference (0.76% abs.; 0.81% rel.) with the full-precision accuracy.

Table 7 illustrates the resource utilization of the FPGA implementation using the previously mentioned implementation procedure. The operating frequency is 166.67 MHz and the input is processed frame-by-frame, i.e., one frame is fed to the network at a time.

4.2.4. Comparison and discussion

The average values of the inference run-time and energy consumption per frame are measured on GeForce GTX 1070 and PYNQ-Z1 using the whole testset (15454 images). The average results mentioned in Table 8 are obtained by taking the dynamic energy

Table 7
PYNQ-Z1 Resource utilization for the fixed-point implementation of the bubble detector.

Resources	Used	Available	Util
Block RAM Tile	49	280	18%
DSPs	30	220	14%
CLB Flip Flop	9,056	106,400	9%
CLB LUTs	8,439	53,200	16%

Table 8
FPGA vs. GPU performance analysis for the bubble detector.

Platform	Run-Time [ms]	Energy [mJ]
FPGA (ours)	1.38	0.83
GeForce GTX 1070	1.03	13.66

Table 9
FPGA vs. different embedded platforms performance analysis for the bubble detector.

Platform	Run-Time [ms]	Energy [mJ]
FPGA (ours)	1.38	0.83
NVIDIA Jetson Xavier (GPU)	1.83	21.46
NVIDIA Jetson Xavier (CPU)	2.69	39.24
RaspberryPi 3B+	32.83	154.32

Table 10
Resource utilization summary on xa7a25t-cpg238-1I for the fixed-point based implementation.

Resources	Used	Available	Util
Block RAM Tile	44	90	48%
DSPs	20	80	25%
CLB Flip Flop	9,183	29,200	31%
CLB LUTs	14,000	14,600	95%

into consideration. The inference takes 1.38 ms on the PYNQ-Z1 board, while it needs 1.03 ms on the GPU. However, a single frame processing consumes only 0.83 mJ on the FPGA compared to 25.87 mJ on the GPU. This makes the PYNQ board 1.3× slower but 31.2× more energy efficient than the GPU.

Similarly to the previous case study, we show a comparison with other embedded platforms in Table 9.

Since the FPGA chip can be selected in the HLS step, the bubble detection network was synthesized several times for different FPGAs. It was found that the chip xa7a25t-cpg238-1I from the Artix7 family is the smallest chip on which the network fits.

Table 10 shows the on-chip resource utilization for the given network and highlights how small this chip is compared to the previously used PYNQ-Z1. Since the chip is not physically present in our lab, we could not perform timing or power measurements.

5. Discussion and future works

In this work, we have presented DeepEdgeSoC, an integrated framework that optimizes the DL workflow based on a graphical interface that facilitates the design, visualization and export of various DL architectures. A powerful feature of DeepEdgeSoC is the ability to parse and import existing network models using the Tracer module. Furthermore, DeepEdgeSoC provides both floating-point and fixed-point models of the network. Using the DNN2HLS framework, the network can be automatically mapped to the hardware design space. The network parameters required to run the network can be obtained using the Exporter module. To better explain the workflow with DeepEdgeSoC, we presented an illustrative case study in which we used a CNN to identify a subset of 43 different German traffic signs. To evaluate the consistency of the resulting implementation, we compare the performance of the FPGA-based implementation to the performance of the GPGPU-based implementation, as well as other embedded platforms, such as NVIDIA Jetson Xavier and RaspberryPi 3B+. Finally, we enrich the results of this work by providing another implementation example of a CRNN network for bubble detection.

While this work contributes to the acceleration of DL on the edge, it has some limitations. First of all, DeepEdgeSoC does not provide a mechanism to automatically find the optimal combination of quantization bits for a given network. A brute-force search could take decades even for a small network that consists of only a few layers, as the total number of possible combinations grows exponentially with the size of the DNN. Thus, our framework should be able to determine the appropriate quantization bit combination(s) by employing a smarter search algorithm. Secondly, changing the parallelization factors in each hardware layer leads to a new hardware implementation of the same software network. Each hardware implementation is characterized by its own total latency, resource utilization, and ultimately energy consumption. Automatic exploration of the design space to find the optimal implementation would further improve the results. Lastly, we do not yet provide a mechanism to partition large DNNs in case they do not fit on the FPGA.

Table A.11
Comparison with related works.

	Visualization				Layers			Features					Frameworks					
	Visualization	Drag and Drop GUI	Multiple I/O	Modular Design	CNN	RNN/LSTM	Skip Connections	Quantization	HLS	OpenCL	Interface Synthesis	ASIC	ONNX	PyTorch	TF/Keras	Theano	Caffe(2)	Torch Lua
HLS4ML [24]	×	×	×	×	✓	×	✓	✓	✓	×	×	✓	✓	✓	✓	×	×	×
fpgaConvNet [10]	×	×	×	×	✓	×	✓	✓	✓	×	×	×	×	×	×	×	✓	✓
CNN2Gate [8]	×	×	×	×	✓	×	×	✓	×	✓	✓	×	✓	✓	✓	×	✓	✓
Caffeine [27]	×	×	×	×	✓	×	*	**	✓	×	×	×	×	×	×	×	✓	×
CNN-Grinder [12]	×	×	×	×	✓	×	✓	✓	✓	×	✓	×	×	×	×	×	✓	×
CascadeCNN [28]	×	×	×	×	✓	×	*	✓	✓	×	✓	×	×	×	×	×	✓	×
FP-DNN [13]	×	×	×	×	✓	✓	✓	**	×	✓	✓	×	×	×	✓	×	×	×
deepHLS [14]	×	×	×	×	✓	×	*	✓	✓	×	✓	×	×	×	✓	×	×	×
LeFlow [29]	×	×	×	×	✓	×	×	×	✓	×	×	×	×	×	✓	×	×	×
PipeCNN [9]	×	×	×	×	✓	×	✓	×	×	✓	×	×	×	×	×	×	×	×
FINN [30]	×	×	×	×	✓	×	✓	✓	✓	×	✓	×	×	✓	×	✓	×	×
PrototypeML [16]	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	✓	✓	×	×	×	×
DL Studio [17]	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	✓	✓	×	✓	×
Sony's NNC [18]	✓	✓	×	×	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
IBM Watson [19]	✓	✓	×	×	✓	✓	×	×	×	×	×	×	✓	✓	✓	✓	✓	×
Tensorboard [15]	✓	×	✓	✓	✓	✓	✓	×	×	×	×	×	✓	✓	✓	✓	✓	✓
Ours	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	×	✓	✓	***	×	×	×

* We could not get precise information about these features.
** Caffeine and FP-DNN support only 16-bit quantization.
*** Through ONNX.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The datasets used in this work are taken from other papers or benchmarks available online.

Acknowledgment

This work has been partially funded by the Federal Ministry of Education and Research of the Federal Republic of Germany as part of the research project VIDETE (Grant number 01IW18002) and the research project DECODE (Grant number 01IW21001).

Appendix. Deeper comparison with related works

Table A.11 show the similarities and differences between our proposed work and the most related previous frameworks.

References

[1] M.R. Al Koutayni, V. Rybalkin, J. Malik, A. Elhayek, C. Weis, G. Reis, N. Wehn, D. Stricker, Real-time energy efficient hand pose estimation: A Case Study, Sensors 20 (10) (2020) 2828.
[2] J. Bai, F. Lu, K. Zhang, et al., ONNX: Open neural network exchange, 2019, https://github.com/onnx/onnx.
[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.
[4] M. Irfan, H. Jawad, B.B. Felix, S.F. Abbasi, A. Nawaz, S. Akbarzadeh, M. Awais, L. Chen, T. Westerlund, W. Chen, Non-wearable iot-based smart ambient behavior observation system, IEEE Sens. J. 21 (18) (2021) 20857–20869.
[5] A. Adeel, M. Gogate, S. Farooq, C. Ieracitano, K. Dashtipour, H. Larijani, A. Hussain, A survey on the role of wireless sensor networks and IoT in disaster management, in: Geological Disaster Monitoring Based on Sensor Networks, Springer, 2019, pp. 57–66.
[6] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al., Mcunet: Tiny deep learning on iot devices, Adv. Neural Inf. Process. Syst. 33 (2020) 11711–11722.
[7] X. Zhang, C. Hao, Y. Li, Y. Chen, J. Xiong, W.-m. Hwu, D. Chen, A bi-directional co-design approach to enable deep learning on IoT devices, 2019, arXiv preprint arXiv:1905.08369.

[8] A. Ghaffari, Y. Savaria, CNN2Gate: An implementation of convolutional neural networks inference on FPGAs with automated design space exploration, Electronics 9 (12) (2020) 2200.
[9] D. Wang, K. Xu, D. Jiang, Pipecnn: An opencl-based open-source FPGA accelerator for convolution neural networks, in: 2017 International Conference on Field Programmable Technology, ICFPT, IEEE, 2017, pp. 279–282.
[10] S.I. Venieris, C.-S. Bouganis, FpgaConvNet: A framework for mapping convolutional neural networks on FPGAs, in: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2016, pp. 40–47.
[11] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, et al., Fast inference of deep neural networks in FPGAs for particle physics, J. Instrum. 13 (07) (2018) P07027.
[12] P.G. Mousouliotis, L.P. Petrou, CNN-Grinder: From algorithmic to high-level synthesis descriptions of CNNs for low-end-low-cost FPGA SoCs, Microprocess. Microsyst. 73 (102990) (2020) http://dx.doi.org/10.1016/j.micpro.2020.102990.
[13] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, J. Cong, FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2017, pp. 152–159.
[14] M. Riazati, M. Daneshtalab, M. Sjödin, B. Lisper, Deephls: A complete toolchain for automatic synthesis of deep neural networks to FPGA, in: 2020 27th IEEE International Conference on Electronics, Circuits and Systems, ICECS, IEEE, 2020, pp. 1–4.
[15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, software available from tensorflow.org. URL https://www.tensorflow.org/.
[16] D.R. Harris, Prototypeml: A neural network integrated design and development environment, 2020, arXiv preprint arXiv:2007.01097.
[17] Deep learning studio, https://deepcognition.ai/. (Accessed: 2021-05-19).
[18] Sony Neural Network Console. https://dl.sony.com/. (Accessed: 2021-05-19).
[19] S.G. Tamilselvam, N. Panwar, S. Khare, R. Aralikatte, A. Sankaran, S. Mani, A visual programming paradigm for abstract deep learning model development, in: Proceedings of the 10th Indian Conference on Human-Computer Interaction, 2019, pp. 1–11.
[20] E.R. Gansner, E. Koutsofios, S.C. North, K.-P. Vo, A technique for drawing directed graphs, IEEE Trans. Softw. Eng. 19 (3) (1993) 214–230.
[21] R. Krishnamoorthi, Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018, arXiv preprint arXiv:1806.08342.
[22] Y. Bengio, N. Léonard, A. Courville, Estimating or propagating gradients through stochastic neurons for conditional computation, 2013, arXiv preprint arXiv:1308.3432.
[23] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[24] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L.P. Carloni, G. Di Guglielmo, P. Harris, J. Krupa, et al., Hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices, 2021, arXiv preprint arXiv:2103.05579.
[25] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, C. Igel, Detection of traffic signs in real-world images: The German traffic sign detection benchmark, in: International Joint Conference on Neural Networks, (1288) 2013.
[26] X. Tan, G. Reis, D. Stricker, Convolutional recurrent neural network for bubble detection in a portable continuous bladder irrigation monitor, in: Conference on Artificial Intelligence in Medicine in Europe, Springer, 2019, pp. 57–66.
[27] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, J. Cong, Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 38 (11) (2018) 2072–2085.
[28] A. Kouris, S.I. Venieris, C.-S. Bouganis, Cascadeˆ CNN: Pushing the performance limits of quantisation in convolutional neural networks, in: 2018 28th International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2018, pp. 155–1557.
[29] D.H. Noronha, B. Salehpour, S.J. Wilton, LeFlow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks, in: FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers, VDE, 2018, pp. 1–8.
[30] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, Finn: A framework for fast, scalable binarized neural network inference, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017, pp. 65–74.