Journal of Real-Time Image Processing (2021) 18:2583–2599
https://doi.org/10.1007/s11554-021-01147-2

ORIGINAL RESEARCH PAPER

Automated CNN back-propagation pipeline generation for FPGA online training

A. Mazouz[1] · **C. P. Bridges**[1]

Received: 13 January 2021 / Accepted: 25 June 2021 / Published online: 23 July 2021

Abstract
Training of convolutional neural networks (CNNs) on embedded platforms to support on-device learning has become essential for the future deployment of CNNs on autonomous systems. In this work, we present an automated CNN training pipeline compilation tool for Xilinx FPGAs. We automatically generate multiple hardware designs from high-level CNN descriptions using a multi-objective optimization algorithm that explores the design space by exploiting CNN parallelism. These designs that trade-off resources for throughput allow users to tailor implementations to their hardware and applications. The training pipeline is generated based on the backpropagation (BP) equations of convolution which highlight an overlap in computation. We translate the overlap into hardware by reusing most of the forward pass (FP) pipeline reducing the resources overhead. The implementation uses a streaming interface that lends itself well to data streams and live feeds instead of static data reads from memory. Meaning, we do not use the standard array of processing elements (PEs) approach, which is efficient for offline inference, instead we translate the architecture into a pipeline where data is streamed through allowing for new samples to be read as they become available. We validate the results using the Zynq-7100 on three datasets and varying size architectures against CPU and GPU implementations. GPUs consistently outperform FPGAs in training times in batch processing scenarios, but in data stream scenarios, FPGA designs achieve a significant speedup compared to GPU and CPU when enough resources are dedicated to the learning task. A 2.8×, 5.8×, and 3× speed up over GPU was achieved on three architectures trained on MNIST, SVHN, and CIFAR-10 respectively.

Keywords Convolutional neural networks · Automated hardware design · Online training · Continuous learning · FPGA design

1 Introduction

Recent literature shows a clear demand for embedded deep learning solutions for hardware-constrained designs and novel compression techniques. CPUs and GPUs have been prominent for executing CNNs on offline training settings however, their energy efficiency and low throughput have made them less attractive for embedded use. For their power-efficient performance and highly parallelised flexible architecture, FPGAs presented themselves as a viable option for hard real-time computation of heavy, deep learning applications [1]. Also, FPGAs allow designers to develop modular IP cores allowing for easier prototyping with the option to selectively deploy design areas at runtime without risk to the overall application. This is crucial given the rapid changes in CNN architectures. Until recently, most works only investigated the hardware implementation of forward pass CNNs as inference engines and accelerators, there is plenty of research done to map the CNN forward pass unto FPGAs for embedded inference [2–18], in contrast, there is a clear lack of work in the areas of online deployment and training on FPGAs. But with the recent breakthroughs in the new field of Continuous Learning [19–23], online training on embedded platforms has attracted more research. In [20] the authors test continuous learning scenarios on benchmark datasets, they investigate the required parameters and conditions for continuous learning to be effective, their main findings show that it is possible and effective if the models account for the catastrophic forgetting problem by

✉ A. Mazouz
a.mazouz@surrey.ac.uk

C. P. Bridges
c.p.bridges@surrey.ac.uk

[1] Surrey Space Centre, University of Surrey, Guildford, Surrey, UK

retaining previously learned information. In [23], the authors proposed an adaptive hierarchical network structure composed of CNNs that can grow to accommodate new classes and tasks when new data becomes available. Similarly, in [21], Yoon et al. propose an expandable architecture capable of selective retraining when presented with new tasks and data. The continuous learning research community is mainly interested in scenarios [24] where datasets are unavailable in the form of neatly organized fully tagged datasets, but in an evolving application where new samples are made available at different times after the models are trained on the main dataset. An example of this would be a self-driving car trained on an object detection dataset receiving new data samples during deployment. Space agents such as Satellites and Rovers [25, 26] can benefit from such capabilities as well, changes to the mission parameters or visual environment can necessitate online learning to remotely account for these unexpected changes. Such new online data could be vital for the system's real-world performance as opposed to the simulated offline performance benchmarked on an organized dataset. For these reasons, training on FPGA platforms becomes even more appealing.

In this work, we identify the significant computational overlap between the inference task and training task in CNNs and translate it into a new FPGA pipeline. This allows for the design of CNN hardware models capable of both inference and online training whilst minimizing the resource overhead. Despite the overlap, the training process of CNNs is still computationally complicated and requires significant changes to the data path of the hardware design, managing the data from the forward pass, updating parameters, and calculating gradients. Compared to inference, the training phase involves a much higher number of operations ($>3\times$) with increased mathematical complexity [27], and a backward pass can take twice the time of a forward pass [28]. The training phase also involves high intermediate datasets, necessitating high memory bandwidth, and large storage. We acknowledge that GPUs have been the de-facto for training tasks to meet immense computation requirements. However, compared to FPGAs, typical GPUs' energy efficiency is poor [1]. Nvidia's Jetson family provides an alternative low power approach with a focus on embedded AI operating at 5–30 W. However, the Jetson GPUs were designed as low-power inference engines and not optimized for training purposes like our proposed combined real-time training and inference FPGA solution.

2 Related work

In both academic and professional fields, GPUs are still the platform of choice for CNN training, and companies such as Facebook and Google maintain datacentres with GPU clusters. Such clusters are employed for training on large amounts of data and incur high costs due to the GPUs' high-power consumption rates and the energy overhead needed for cooling, making them expensive to maintain operationally. Google sought a solution in researching the possibility of using ASIC and FPGAs within their servers so the workload of training and running deep learning models is offloaded. This resulted in the successful design and wide-scale deployment of the Tensor Processing Unit (TPU) [29] for training and inference. Other companies such as Microsoft [30, 31] and Amazon's "AWS EC2 F1" instance followed suit in using FPGA clusters within their data centres and servers for back-end training and inference at a lower power cost—highlighting the trend for low-power solutions utilising FPGAs. CNN training on FPGA platforms has not been investigated thoroughly with only two exceptions that focus on batch training which uses FPGA platforms as replacements for GPUs clusters in offline training [28, 32]. In [27] Wenlai et al. presented F-CNN, the first CPU/FPGA hybrid design for deploying and training CNN networks. The CPU is used as a controller and the FPGA as an accelerator. They employ an analytical model to generate a backpropagation model. Their implementation of CNN training involves a direct translation of backpropagation equations for error calculation and parameter updates. This requires the introduction of significant resource overheads since it does not fully consider the overlap in calculations within the forward pass. In [32] Venkataramanaiah et al. extends work from [28] and introduces a hardware CNN training RTL compiler. Their work is purely FPGA and relies on static processing element arrays for convolutional calculations. It uses pre-optimized and precompiled Verilog CNN hardware modules, but unlike [27] has no analytical model to inform the compilation and provide design space exploration. Both works do not present extensive experimental results and only cover Ler-Net 5 [33] and custom architectures rather than modern CNNs as benchmarks. In addition, they deploy different data representation schemes, 32-bit and 16-bit, respectively, and use different development boards operating at different frequencies which makes it challenging to compare the implementations in terms of resource and latency overheads.

One area of research that remains unexplored is continuous learning on embedded platforms, mainly FPGAs. This requires on-board implementations of training algorithms for CNN models which are flexible and allow users to easily explore the design space for suitable designs based on the hardware and its applications. In our previous work [18] we proposed an FPGA compiler for CNN inference, the compiler generates FPGA pipelines from high-level CNN input architectures, it generates a Simulink hardware model which is validated then translated into RTL/VHDL. For this research, we expand on the same compiler to include an improved design space exploration step and a training

pipeline with the aim of reusing most of the original forward pass pipeline resources.

The main contributions of this work are,

- Providing deep learning engineers with an automated CNN compiler tool for FPGAs which supports inference and online training.
- Using analytical equations to model the complied design's expected performance trade-space using a multi-objective optimization genetic algorithm, allowing the user to automate the design space exploration for trade-offs in accuracy, latency, and resources.

This paper will cover the fundamentals of CNN back-propagation and highlight the areas where they overlap with the forward pass can be exploited in Sect. 2. Then in Sect. 3, the proposed research will be presented, this includes the hardware models of these functionalities with the analytical equations for latency and resources. The section also covers design space exploration using the multi-objective optimization algorithm. Section 4 will be dedicated to the experiments and results, first, MOGA results will be presented on three different datasets and four architectures. Finally, the section will cover the implementation of the models obtained from the previous sections on a Xilinx FPGA, a breakdown of latency and resources results will be given followed by discussion and comparison with related works.

3 Mathematical basis for convolutional backpropagation

3.1 Convolution

A convolutional layer and a convolutional network, as a whole, can be treated as a computational graph. Let us say we have a gate f in a computational graph with inputs X and Y which outputs Z. For a simple function F which takes X and Y as inputs and outputs Z, local gradients can be computed by differentiating Z with respect to X and Y as $\partial z/\partial x$ and $\partial z/\partial y$. For a convolutional forward pass, the inputs process through the CNN layer, and at the output, the loss is obtained using a loss function which quantifies the error between the ground truth and the real output. When we process the loss backward, layer across layer, we obtain the gradient of the loss from the previous layer as $\partial L/\partial z$. For the loss to be propagated to the other gates, we need to find $\partial L/\partial x$ and $\partial L/\partial y$. Using the chain rule, we can calculate $\partial L/\partial x$ and $\partial L/\partial y$, which would propagate to the other layers. Let us assume the function F is a convolutional operation between input matrix X and a filter matrix F. convolution between Input X and Filter F, gives us an output matrix O. This process describe the convolutional Forward Pass (FP) and the output matrix O will be forwarded to the next layer as an input matrix X. For the Backward Pass (BP) we find the loss gradient with respect to the Output O from the next layer as $\partial L/\partial O$.

As seen in Fig. 1, we can find the local gradients $\partial O/\partial X$ and $\partial O/\partial F$ with respect to Output O. Using chain rule and the loss gradient from the previous layer $\partial L/\partial O$, we can calculate $\partial L/\partial X$ and $\partial L/\partial F$. In the following subsections, we briefly show the process of calculating both gradients to help determine the critical overlapping elements.

3.1.1 Finding filter local gradient $\partial L/\partial F$

The first step is to calculate the local gradient $\partial L/\partial F$ which will be used to update the new weight matrix $F_{\text{new}} = F + \alpha\left(\frac{\partial L}{\partial F}\right)$ with α being the learning rate, it is a configurable hyperparameter used during training to affect the amount the weights are updated, it has a small positive value, often in the range between 0.0 and 1.0.

Thus, we calculate $\frac{\partial L}{\partial F} = \frac{\partial L}{\partial O} \times \frac{\partial O}{\partial F}$ where $\frac{\partial L}{\partial F}$ is the gradient to update Filter F, $\frac{\partial L}{\partial O}$ is the loss gradient from the previous layer, $\frac{\partial O}{\partial F}$ is the local gradient.

To find the local gradient—$\partial O/\partial F$ we must differentiate Output Matrix O with respect to Filter matrix F. Matrix O values are known from the FP:

Therefore, finding derivatives with respect to F gives Eq. (1), for every element of F where M and N are the dimensions of matrix O:

$$\frac{\partial L}{\partial F_{i,j}} = \sum_{K=1}^{M}\sum_{L=1}^{N} \frac{\partial L}{\partial O_{K,L}} \times \frac{\partial O_{K,L}}{\partial F_{i,j}}. \tag{1}$$

Expanding and substituting the values of $\partial O/\partial F$ with X gives a set of equations which represent a convolutional operation between input X and loss gradient $\partial L/\partial O$ as shown in Fig. 2.

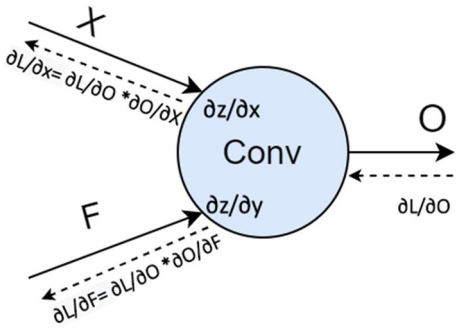

Fig. 1 Function F during a backward pass to calculate input local gradients

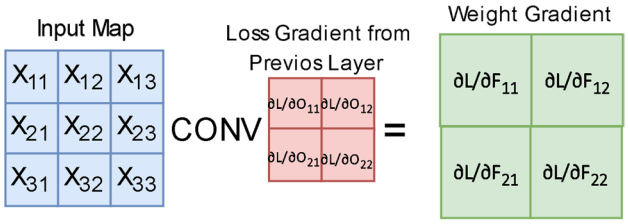

Fig. 2 Demonstration of how $\partial L/\partial F =$ Convolution of input matrix X and loss gradient $\partial L/\partial O$

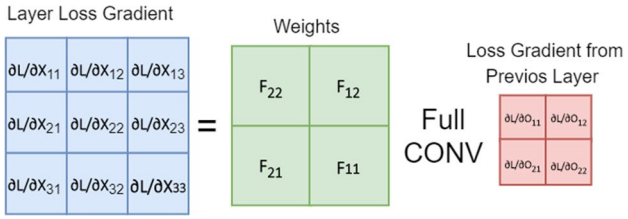

Fig. 3 Demonstration of how full convolution generates the values of $\partial L/\partial X$ and can be used to represent $\partial L/\partial X$

3.1.2 Finding previous layer local gradient $\partial L/\partial X$

Finding the local gradient $\partial O/\partial X$ which will be backpropagated to the previous layer is achieved using the Chain rule as in Eq. (2): For every element of X where M and N are the dimensions of matrix O:

$$\frac{\partial L}{\partial X_{i,j}} = \sum_{K=1}^{M} \sum_{L=1}^{N} \frac{\partial L}{\partial O_{K,L}} \times \frac{\partial O_{K,L}}{\partial X_{i,j}}. \tag{2}$$

Expanding Eq. (2) and substituting $\partial O/\partial X$ with F which is equal to the derivative of O with respect to X gives a full set of equations which show that $\partial L/\partial X$ can be represented as a full convolution between a 180-degree rotated Filter F and loss gradient $\partial L/\partial O$ as seen in Fig. 3. Full convolution is different from regular or valid convolution where a filter only scans elements within the feature map without going outside (adding padding), full convolution applies filters starting from the top-left element being centered, and padding is needed for the elements outside the feature map.

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input, Loss Gradient } \frac{\partial L}{\partial O}\right), \tag{3}$$

$$\frac{\partial L}{\partial X} = \underset{\text{Conv}}{\text{Full}} \left(180^\circ \text{ weights, Loss Gradient } \frac{\partial L}{\partial O}\right). \tag{4}$$

Equations (3) and (4) highlight an overlap in computation between the forward and backward pass, convolutional computations are required for feature extraction and gradient calculations. An efficient design should take advantage of this by reusing computational resources dedicated to the forward pass during the backward pass, we describe this in more detail in the following sections, we will also use dF and dX to refer to $\frac{\partial L}{\partial F}$ and $\frac{\partial L}{\partial X}$ in the following sections.

3.2 Pooling

No learning takes place on the pooling layers. The function of the pooling layer is to progressively reduce the convolution spatial size and reduce the number of parameters and computation in the network. This also controls overfitting by reducing the feature space available during training. There are two common types of pooling: average and maximum.

3.2.1 Average pooling

During the FP, average pooling outputs the average of the input elements using a scanning window where a K by K window scans through the input matrix from the top left to the bottom right. During the BP, the error $\partial L/\partial O$ is multiplied by $\frac{1}{K^2}$ where K is the dimension of the scanning window, the result is assigned to the whole pooling block (all units get the same value).

3.2.2 Max pooling

During the FP, max-pooling outputs the maximum of the input elements (winning element) using a scanning window. The BP error is simply assigned to the "winning element" because other units in the previous layer's pooling blocks did not contribute to the output, hence all the others are assigned values of zero.

3.3 Rectified linear unit activation (ReLU)

During the FP, the ReLU layer changes all negative elements to zero while retaining the value of the positive elements. No learning takes place and no spatial/depth information is changed. During the BP, gradients of the positive elements retain their value while the rest become zero.

3.4 Fully connected

During the FP, the fully connected layer calculates the dot product of the vectorized input $X = [X_1\ X_2\ X_n]$, the weight matrix W, and bias vector B, as described in Eq. (5).

$$W = \begin{pmatrix} W_{11} & W_{12} & W_{1n} \\ W_{21} & W_{22} & W_{2n} \\ W_{k1} & W_{k2} & W_{kn} \end{pmatrix}$$
$$B = [b_1\ b_2\ \ldots\ b_k] \tag{5}$$
$$Y = (W \cdot X) + B.$$

Equation (6) and (7) describe the backward pass to calculate local gradients dW and dX using the loss gradient from the previous layer (softmax) and input and weight vectors X and W:

$$\partial W = [\mathrm{d}L_1, \mathrm{d}L_2, \mathrm{d}L_k]\, X, \tag{6}$$

$$\partial X = [\mathrm{d}L_1, \mathrm{d}L_2, \mathrm{d}L_k]\, W. \tag{7}$$

3.5 Softmax loss function

Softmax function takes an N-dimensional vector of real numbers x from the fully connected layer and transforms it into a vector of real number probabilities p, size k, in range (0, 1) which add up to 1, as described in Eq. (8).

$$p_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}. \tag{8}$$

Cross entropy indicates the distance between what the model believes the output distribution should be, and what the distribution is. It is defined as Eq. (9)

$$F(y, p) = -\sum_i y_i \log\left(p_i\right). \tag{9}$$

Equation (10) describes the final derivative of the Cross-entropy loss function with a softmax, with p being the output vector and y being the ground truth target, this is the global gradient for the Softmax layer which will be backpropagated to the previous layers.

$$\frac{\partial L}{\partial z_i} = (p_i - y_i). \tag{10}$$

4 Research proposal

4.1 Automated hardware design generation

The model-based hardware design workflow shown in Fig. 4, automatically generates target agnostic RTL for CNN architectures for series-networks and directed acyclic graphs. Our tool first parses a pre-trained input graph provided by the user for network information and parameters. This information is used to initialize and generate PEs for the different functionalities, these PEs are then fetched to populate the design space creating a hardware model representation of the input graph. This is achieved in Simulink, this hardware model is explored using a Multi-Objective Genetic Algorithm (MOGA) by taking advantage of the inter-layer parallelism in Conv layers and dedicating more or fewer PEs for convolutional operations. Once the hardware model is

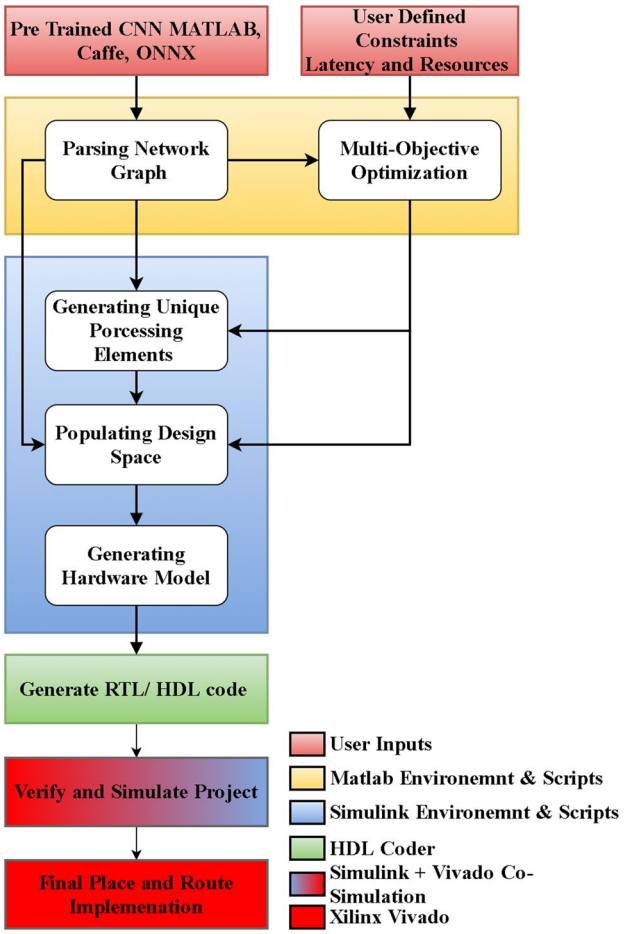

Fig. 4 Overall system diagram from CNN input to generated and verified HDL design

verified, RTL code is generated for the whole design. The tool uses MATLAB scripts, Simulink environment, and Vivado for a complete compilation of a pre-trained CNN. The layers presented in Sect. 2 are supported by the compiler to be translated into hardware for FPGA use within a streaming interface where data inputs are streamed through the design pipeline with control signals. The control signal is generated alongside the pixel stream to schedule and control the process, this signal specifies the beginning and end of rows and columns plus the validity of each pixel using a 5-bit binary vector as seen in Fig. 5. The different layers are converted into separate processing units, each unit contains a number of Processing Elements (PEs) such as convolution, pooling, non-linearity, and fully connected operations. The PEs for the FP are described first, then changes to the design will be introduced to accommodate backpropagation and learning.

The verification environment, seen in Fig. 6, is a co-simulation environment between the hardware Simulink model and an HDL simulation tool, Vivado in our case. The tool

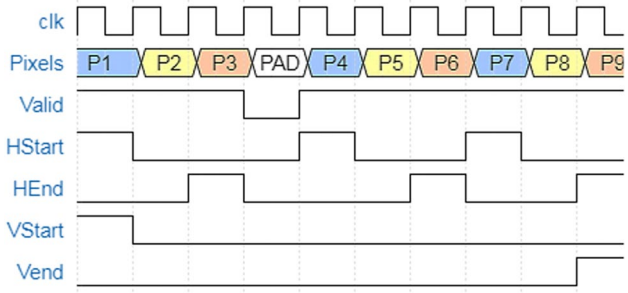

Fig. 5 Control signal for a 3×3 window with 1-pixel padding interval [17]

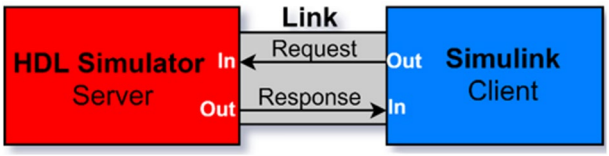

Fig. 6 Co-simulation and verification environment using Simulink as a client and Vivado as server allowing for sending stimuli and visualizing responses

compiles an HDL project and generates a Simulink co-simulation block. The block is used to communicate between the Simulink environment and the HDL code. The client-side Simulink environment allows the user to provide input stimulus in the form of a MATLAB classification dataset or test images. The server reads the dataset samples and corresponding ground truth information, it generates a data stream and a control signal from the images and feeds it into the HDL code block. The HDL code block communicates with the HDL simulation tool by sending this data and reading its outputs. This is achieved using Tool Command Language (TCL) scripts. The output data is sent back to the Simulink environment where it can be displayed, saved, visualized, and analysed.

Our implementation uses a streaming interface that lends itself well to real-time data streams instead of static data reads from memory. This means we do not use the standard array of PEs approach, which is typically efficient for offline inference setups. Instead, we translate the architecture into a pipeline where data is streamed through allowing for new samples to be read as they become available. The blocks using this interface do not need a configuration option for the exact image size or the size of the inactive regions. In addition, if image parameters are changed, there will be no need to update each block. Instead, an update to the image parameters once at the serialization step is sufficient and subsequent tuneable parameters for the other blocks such as the size of the line buffers will be changed as a function of the new input data size by the compiler. Once these parameters are fixed and a hardware model is generated, it is not possible to modify them at runtime. By using a streaming pixel interface with control signals, each block or object starts computation on a new segment of pixels at the start-of-line or start-of-frame signal. The computation occurs whether the block or object receives the end signal for the previous segment or not. The work on the automated generation of the FP pipeline is described at length in our previous publication [18]. To develop our new online system, we revisit and update critical information for the scientific community.

4.1.1 Proposed convolutional layer design

The convolutional PE (C_{PE}) requires two main processing blocks, a line buffer, and a Multiply and Accumulate (MAC) block. The line buffer uses FIFOs to buffer the input feature maps as they are streamed into the PE, then tap out the elements required for the currently active window, the active window is always of the size $K \times K$ which is the size of the applied filter. For instance, a 3×3 filter requires 9 elements from different rows and is synchronized using a control signal. The MAC core simply receives the tapped-out elements, multiplies them by their corresponding filter elements, and accumulates them into an element of the output feature map using an adder tree. The output feature map is streamed further into the pipeline for subsequent processing as new elements become available each cycle [18]. Equations (11), (12), and (13) show the resource modelling equations Where K is the filter dimension, N_{mult} is the number of multipliers,N_{add} is the number of adders and $N_{\mathrm{add_{stages}}}$ is the number of stages needed in the adder tree. Equation (14) shows the main latency modelling equation for the convolutional pipeline.

$$N_{\mathrm{mult}} = K^2, \tag{11}$$

$$\mathrm{Max\_}N_{\mathrm{add\_stages}} = \log_2(K^2) + 1, \tag{12}$$

$$N_{\mathrm{add}} = \frac{[k^2 - 1]}{\left(\frac{\mathrm{Max}_{N_{\mathrm{add_{stages}}}}}{N_{\mathrm{add_{stages}}}}\right)}, \tag{13}$$

$$C_{\mathrm{PE}}(T) = \mathrm{Clk} \times \left\lceil \left\lceil \begin{array}{c} \mathrm{In}_{\mathrm{Delay}} + \left(\mathrm{BP} + \frac{1}{2}\right) + \left\lceil \begin{array}{c} (\mathrm{FMi}_W + \mathrm{BP} + \mathrm{FP}) \\ \times (\mathrm{FMi}_H) \\ +\mathrm{FMi}_H \end{array} \right\rceil \\ +\mathrm{Padding}_t + \mathrm{Tap}_{\mathrm{Out}} \\ +\mathrm{Multiply}_{\mathrm{Out}} + \mathrm{AddTree}_{\mathrm{Out}} \\ +\mathrm{Out}_{\mathrm{Delay}} \\ +\mathrm{ReLU}_T \end{array} \right\rceil \right\rceil . \tag{14}$$

The variables for the latency modelling equation are:

- BP, FP: Back Porch and Front Porch values for vertical blanking. The total of Back porch + Front porch must be at least 2 times the largest filter size.
- FMi_W FMi_H: Feature Map width and height in pixels.
- Padding: Time needed to implement vertical and horizontal padding on input data and move it to the output.
- $\mathrm{Tap}_{\mathrm{Out}}$: Time needed to move data from the line buffer into registers which taps out relevant elements needed for the multiplier, K Clk Cycles.
- $\mathrm{Multiply}_{\mathrm{Out}}$: Time needed to move data from Tap_out registers to Multipliers, K Clk Cycles.
- Add - $\mathrm{Tree}_{\mathrm{Out}}$: Time needed to move Multiply_out results from registers to adder tree. Takes $(N_{\mathrm{add}_{\mathrm{stages}}} \times \mathrm{Clk}) + 2$ delay Cycles to put intermediate data in registers.
- $\mathrm{In}_{\mathrm{Delay}}$, $\mathrm{Out}_{\mathrm{Delay}}$: Time needed to move data into registers and give the system enough time for processing. Set to 4 Clk each, $\mathrm{In}_{\mathrm{Delay}}$ is only needed for the first layer, $\mathrm{Out}_{\mathrm{Delay}}$ is included for every convolutional PE.
- ReLU: Time needed to apply rectified linear unit on elements, a conditional switch is used, it takes 1 clock cycle per element.

For the backward pass, two more C_{PEs} are required to calculate dF and dX as described in Sect. 3.1 and Eq. (3) and (4). These PEs require weights and input data from the last forward pass, thus every training cycle requires saving the input feature maps and last updated weights for the FP C_{PE} into an internal memory block. Weights for the C_{PE} are K^2 elements saved as FixP = 16-bit fixed point and thus require $\lceil (K^2 \times \mathrm{FixP})/8 \rceil$ Bytes of memory. A Conv Layer with N Conv PEs, therefore, requires $N \times \lceil (K^2 \times \mathrm{FixP})/8 \rceil$ Bytes of internal memory to buffer in the weights for a fully parallelised convolutional layer. The feature maps require buffering of $N \lceil (\mathrm{FMw} \times \mathrm{FMh} \times \mathrm{FixP})/8 \rceil$ Bytes while considering that feature map dimensions are spatially reduced deeper in the network from pooling and convolution, dimensionality is described by the following equation $\mathrm{FMw} = ((\mathrm{FMw} - K - P/S) + 1$ where P is padding and S is stride. During backpropagation, weights and input feature maps are read by the dF and dX units requiring additional registers to save the intermediate results and two more C_{PEs} to calculate new local gradients and update the weights. The results are saved into BRAM. dF is used to update the weights and the new weights are read again next FP cycle, dX is read by the next layer. This process needs to be synchronized so needed inputs are available to be read and written from/to memory when needed. This is achieved using a new scheduler and a memory-control block added to each convolutional unit as needed. The C_{PE} used for the FP convolution can be reused during the BP to perform the 2D convolution required to calculate dF and update the weights. The same C_{PE} can be adjusted to calculate dX as well, if latency is not a priority. Otherwise, parallelising dF and dX calculations would require an additional C_{PE}. Figure 7 shows a backpropagation unit, input X from the previous FP is saved and read from the memory block to be used in calculating the gradients, two C_{PE} are dedicated to calculating dF and dX. dX is calculated using the rotated F matrix and dY of the next layer, dX is passed to the next layer as dY. dF from the current cycle is used to update the weights F for the next FP using learning rate α. The new weights F are used for the next FP and the outputs are again saved in memory for the next BP cycle. The generated Simulink hardware model with the same process can be seen in Fig. 8 the inputs are a BP enable signal, previous layer dL stream, learning factor,

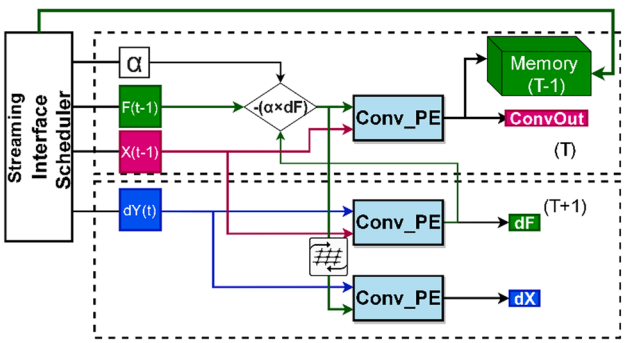

Fig. 7 Backward pass diagram for the Convolutional Unit, T for the FP cycle, $T+1$ for backpropagation, $T-1$ for the input data saved from the previous cycle

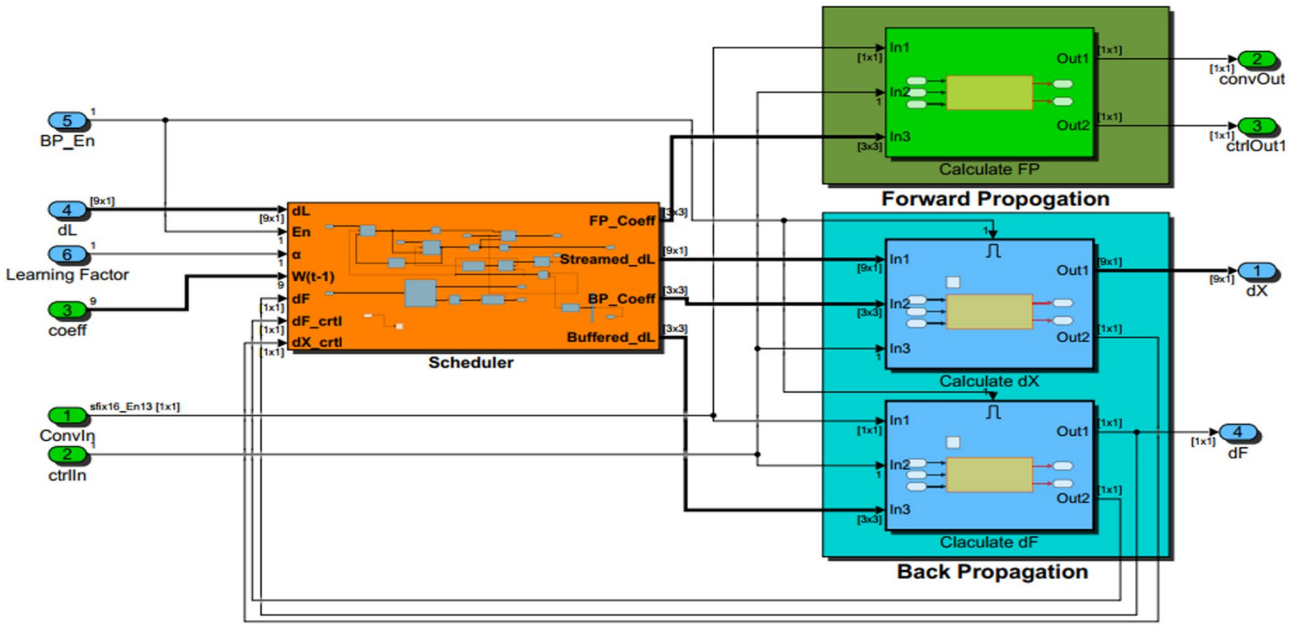

Fig. 8 Generated hardware model for a convolutional unit in simulink

input frame and a control signal, outputting an output frame, dX, and dF plus a control signal.

Our new workflow gives the option of generating designs that implement a Forward Pass CNN pipeline alone or adding the CNN training pipeline. Hardware resources required for each can be estimated from the generated generic PEs and layer parameters (Eqs. 11–14), the estimations allow the tool to generate multiple designs based on resource limitations and user input. Block Ram (BRAM) tiles are currently assumed to be RAMB36E1 which can be used as RAMB18E1 and FIFO18E1 [34] when needed. DSP is assumed to be (18 × 25 MACCs) Zynq-7000 generation architecture [34].

4.1.1.1 Resource estimation for convolutional units and C_{PEs} for FP:

The FPGA hardware representation of the FP pipeline of the convolutional PEs uses these estimations in addition to the equations described earlier to create an analytical model:

- DSP Slices: K^2 DSP slices are mapped to each unique Conv PE, the mapping maximises performance by dedicating a multiplier for each coefficient of the Conv unit filter.
- LUT slices: an upward of ~ 800 LUT slices are dedicated to each generic Conv unit, these are used as shift registers for the Line Buffer, and as logic and Muxes. ~ 300 of which are used for the Line Buffer.
- Block RAM: at FixP = 16-bit fixed-point representation, 2 18 k BRAMs are dedicated to each Conv unit for the line Buffer FIFO.
- $\text{BRAM}_{\text{linebuffer}} = \text{Ceiling}\left(\text{FM}_{\text{Size}} \times K \times \frac{\text{FixP}}{18\,\text{Kb}}\right)$.

4.1.1.2 Resource estimation for convolutional units and C_{PEs} for BP

The FPGA hardware representation of the BP pipeline of the convolutional PEs uses these estimations:

- DSP Slices: $\left(K^2 \times 2\right) + 1$ DSP slices are mapped to each unique Conv unit, K^2 to dX C_{PE}, K^2 to dF C_{PE}, and one for the scheduler to update the weights by multiplying the gradient by the learning rate.
- LUT slices: An upward of ~ 2200 LUT slices are dedicated to each generic Conv unit, these are used as shift registers for the Line Buffer, logic, and Muxes. ~ 800 for the two dX and dF PEs, ~ 800 for the scheduler and memory control.
- Block RAM: At FixP = 16-bit fixed-point representation, 5 18 k BRAMs are dedicated to each Conv unit, 2 for the dX line-buffer, 2 for the dF line-buffer, and 1 for the memory controller using a FIFO to save intermediate feature maps of size 32 × 32 or less.

$$\text{BRAM}_{\text{Memory}} = \left(\text{FM}_{\text{Height}} \times \text{FM}_{\text{width}} \times \frac{\text{FixP}}{18\,\text{Kb}}\right),$$

$$\mathrm{BRAM}_{\mathrm{linebuffer}} = \left(\mathrm{FM}_{\mathrm{Size}} \times K \times \frac{\mathrm{FixP}}{18\,\mathrm{Kb}}\right).$$

4.1.2 Proposed pooling layer design

For our work, we consider both Maximum and average pooling implementations:

4.1.2.1 Average pooling A stripped-down version of the $\mathrm{CL}_{\mathrm{PE}}$ is used for average pooling, the same memory control, and kernel core structure is used to apply weights that result in averaged values of the scanning windows, no registers are needed to save the kernel coefficients and no weights and biases are read from memory. For the BP, the same line buffer FIFOs are used to tap a zero mask for every element from dX and add the value of that element to the zero mask.

4.1.2.2 Maximum pooling Maximum pooling uses the same Memory control structure but replaces the MAC core with a comparator tree. For the Backward pass, there is no gradient with respect to non-maximum values. Thus, the gradient dX from the next layer is passed back to only the winning element (maximum element). All other neurons get zero gradients. To implement, this the same line buffers are used to store elements from dX, the input X feature map elements from the FP are saved into on-chip memory FIFO18E1 or FIFO36E1 alongside their indices. These elements are used to upsample the input feature map using demultiplexers.

4.1.3 Proposed fully connected layer design

The fully connected PE ($\mathrm{FC}_{\mathrm{PE}}$) is implemented as parallelized MAC units. These units multiply the feature-map elements streamed in from the last layer and multiply them by their corresponding weights and accumulates them into an output register. This is done for each FC output head. The process can be further parallelized to alleviate the processing bottleneck of streamed data by using multiple $\mathrm{FC}_{\mathrm{PEs}}$ in parallel to process the data channels from the last layer concurrently. Equation (15–17) show the resource modelling equations while Eq. (18) shows the latency modelling equation.

$$N_{\mathrm{mult}} = \mathrm{FC}_{\mathrm{out}} \times N, \tag{15}$$

$$N_{\mathrm{add}} = \left[\left[\mathrm{FC}_{\mathrm{out}} \times N\right] + \left(\mathrm{FC}_{\mathrm{out}} \times N_{\mathrm{add}}\right)\right], \tag{16}$$

$$N_{\mathrm{reg}} = \mathrm{FC}_{\mathrm{out}} \times P, \tag{17}$$

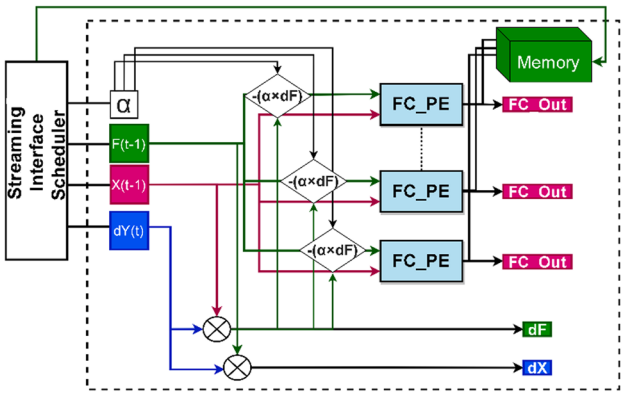

Fig. 9 Backward pass diagram for the Fully Connected Unit

$$\mathrm{FC}_{\mathrm{PE}}(T) = \mathrm{Clk} \times \left[\left(\mathrm{FMi}_W + \mathrm{BP} + \mathrm{FP}\right) \times \left(\mathrm{FMi}_H - 1\right) + \mathrm{FMi}_H\right] \times P, \tag{18}$$

where $\mathrm{FC}_{\mathrm{out}}$ is the number of output heads in the FC layer. N is the number of $\mathrm{FC}_{\mathrm{PE}}$ dedicated for every output head. N_{add} is the number of adders needed for the adder tree. N_{reg} is the number of registers needed to accumulate the outputs of the different PEs.

P is the Parallelism-coefficient, a new term equal to the ratio between the number of input data channels over the number of $\mathrm{FC}_{\mathrm{PEs}}$. So $\mathrm{Ch}_{\mathrm{D}}/\mathrm{FC}_{\mathrm{PE}}$ equals to 1 when the number of $\mathrm{FC}_{\mathrm{PE}}$ is equal to the number of input data channels, fully paralyzing the computation at the expense of chip resources.

For the BP, the FP PEs can also be reused for the MAC calculations to calculate dF and dX which require matrix multiplications, dX calculations are parallelized to improve throughput and thus require additional resources. Also, a scheduler and a memory control block are required to both synchronize the process and save intermediate results from the forward pass. Figure 9 shows how the FC PEs are used to both calculate the FP output FC_Out which are saved in memory for the BP, the gradients dF used to update the weights, and dX which is backpropagated to the next layer.

4.1.3.1 Resource estimation for fully connected units and $\mathrm{FC}_{\mathrm{PEs}}$ for FP The generated hardware model uses generic PEs for the FP pipelines, this allows us to estimate the resources needed for the generated pipeline.

- DSP Slices: 1 DSP slice is mapped to every FC unit dedicated to multiplying and accumulating weights assigned to the unit with incoming feature map elements.
- LUT slices: An upward of ~ 20 slices are dedicated to every FC PE as logic.

- Block RAM: No BRAM is required for the FC units and PEs.

4.1.3.2 Resource estimation for fully connected units and FC_{PEs} for BP The FPGA hardware representation of the BP pipeline of the fully connected PEs follows the following estimations.

- DSP Slices: 1 DSP slices is mapped to every FC unit dedicated to multiplying and accumulating weights assigned to the unit with incoming feature map elements. For an FC layer with FC_{Out} output heads, FC_{Out} DSP slices are required, an additional FC_{Out} is needed to calculate dX assuming the FP slices are reused to calculate dF.
- LUT slices: An upward of ~ 400 is dedicated to every FC head as logic. 10 LUT Slices for dX, ~ 200 for dF, and ~ 100 for the memory control and scheduler. An upward of ~ 20 is dedicated to every FC PE as logic.
- Block RAM: One 18 K BRAM tile is required to save one feature map vector of 32×32 (one data channel) at 16 bits.

$$\mathrm{BRAM}_{\mathrm{Memory}} = \left(\mathrm{FM}_{\mathrm{Height}} \times \mathrm{FM}_{\mathrm{width}} \times \frac{\mathrm{FP}_{\mathrm{rep}}}{18\,\mathrm{Kb}} \right).$$

4.1.4 Proposed softmax and loss function design

Softmax is optional to include, it is implemented using MathWorks's Native Floating Point library [35] and requires 80 DSP slices, 20 K LUT slices, and no BRAM.

4.1.5 Scheduler

The new scheduler is implemented as an automated block using Simulink. It uses conditional switches to route input data streams between the FP and BP pipelines. The streaming architecture allows for variable stream sizes as the data is always accompanied by a control signal. The MAC Core size however is fixed and presents a challenge since the BP path convolutions would have different size Conv operation "filters". This is circumvented by buffering the results and reusing the existing MAC core as needed. It is possible to compile the MAC core to support BP convolutions and reduce the latency of that stage. For our experiments, we use MAC cores that only support the original FP size. A BP_En signal acts as the main synchronizing signal for switching between the two modes. This signal enables the data reshaping, rotating the $F(t-1)$ weight matrix, buffering the dL gradients for the MAC operations, and updating the weight and bias matrices with the new gradients. The scheduler

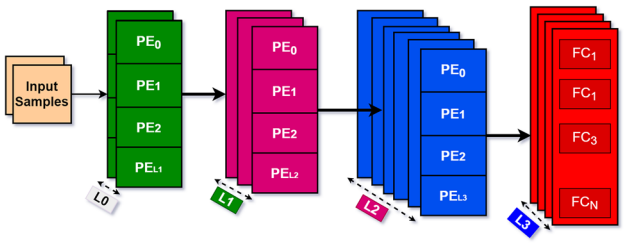

Fig. 10 Design-Space generations using $(L0+L1+L2)\times 4$ CPEs

block receives a BP_En signal, learning rate variable, dL, dF, and dX plus their control signals when needed.

4.2 Automated design generation using MOGA

When all processing elements are generated from the high-level architecture data (number of layers, network graph, layer-specific parameters), they are used to populate a design space. The process to generate a design space for an architecture requires three steps:

Parse an input architecture for parameters and connections in MATLAB.
Programmatically generate generic PEs based on the data in MATLAB/Simulink.
Use the PEs, the latency, resources estimation equations, and the architecture data to model, explore and generate a design space.

Each architecture has a high-level description which will include the number of layers, the connections between them, and PEs initialized using filter size, stride, and padding parameters. The high-level description also allows for optionally including the SoftMax layer and training pipeline or just implementing an inference pipeline. Lastly, the FPGA board's working frequency and resources in terms of available DSP slices, BRAM, and logic are considered. The design space exploration process is treated as a multi-objective optimization problem. Our goal is to find different configurations of the high-level architecture using intra-layer parallelism. Multi-objective optimization involves minimizing multiple objective functions subject to a set of constraints, for our case; the latency and FPGA resources. The constraints will depend on the architecture and user input. Figure 10 shows how the convolutional PEs are mapped to the pipeline, pooling and non-linearity follow each layer but are not included in the figure and the number of FC PEs can be adjusted to parallelize the FC operations. PEs are dedicated to each active data channel streaming a feature map from the previous layer. The number of active PEs tasked with applying the layer's filter is adjusted to reduce the resource overhead. For instance, a direct mapping means every filter will have

a dedicated PE, to trade-off latency and resources the number of dedicated PEs is changed. This number of dedicated PEs is what we aim to optimize, this is done by exploring the design space. Our optimization algorithm of choice is a genetic multi-objective optimization algorithm [36]. We provide the algorithm with the latency estimation equations per layer and PE, the resource estimation equations per PE, the number of convolutional layers, the number of filters per layer, the size of the filters, and the number of DSP slices available on the chip. The optimization goal is to minimize the number of PEs needed to implement an architecture while minimizing latency.

We provide the algorithm with three input arguments, a fitness function, an input vector whose size is equal to the number of variables to be optimized in the problem, and bounding constraints. The output returned by the MOGA is the points on the Pareto front, and the objective function values at the found Pareto front. For each iteration, the population in the current generation is the number of samples the MOGA picks for optimization returning a fitness score for each variable, these scores inform the sampling of the next generation's population. With a large population size, the genetic algorithm searches the solution space more thoroughly, thereby reducing the chance that the algorithm does not find a global minimum. However, a large population size also causes the algorithm to run more slowly, for our tests, we use higher population sizes for the deeper networks. The number of variables in the optimization is set to the number of convolutional layers, with an input vector P of size n elements constrained by a lower bound of 1 and an upper bound of *ub*. where *ub* is the number of convolutional filters in that layer. The optimization objectives are described by *y*, a vector of size four, where latency, DSP, LUT slices, and BRAM are optimized. DSP objective optimization is described by Eqs. (19) and (20). *k* is a vector of size n where each element is the dimension of the filters per layer (3×3 for example). *L* is a vector of size *n*, it holds the values of the maximum PEs needed to fully parallelize computations as calculated in Eq. (19). $L(i)$ therefore holds the number of PEs required for layer *i* in the current configuration. For instance, if the number of input channels to $L(2)$ is $P(1)=3$, and $P(2)$ is randomly set by the optimizer to 3 then $L(2)$ is 9, meaning 9 PEs are required for the fully parallelized computations in that layer.

Table 1 Architectures used for validation

Dataset	Architecture	Parameters	Operations
MNIST [33]	8-8	155.16 K	2.86 M
MNIST	8-16-32	333.72 K	6.79 M
SVHN [37]	8-16-32-64	639.58 K	32.2 M
CIFAR-10 [38]	8-16-32-64-64	676 K	83 M

Table 2 Datasets used for training and validation

Dataset	Type	Size	Classes	Train/Test
MNIST	Grayscale	28×28	10	50 K/10 K
SVHN	RGB	32×32	10	50 K/10 K
CIFAR-10	RGB	32×32	10	75 K/25 K

$$L(i) = P(i) \times P(i-1) \quad \text{With } (1 \leq P(i) \leq ub(i)), \tag{19}$$

$$Y(\text{DSP}) = [L(1) \times k(1)^2 + L(2) \times k(2)^2 \ldots + L(n)) \times k(n)^2] \leq \left(\text{DSPmax}/k^2\right). \tag{20}$$

5 Experiments and results

5.1 MOGA results

The architectures used for all experiments in this section and Sects. 5.2 and 5.3 are listed in Tables 1 and 2. Four custom architectures with differing depths and number of operations are used to simulate possible implementations of different complexity and computational load. For instance, a custom network of the shape $a1-a2-a3-a4$ represents a network with four convolutional layers with *a* filter each layer, these layers are each followed by non-linearity ReLU, a pooling layer (average pooling was used for all the architectures in

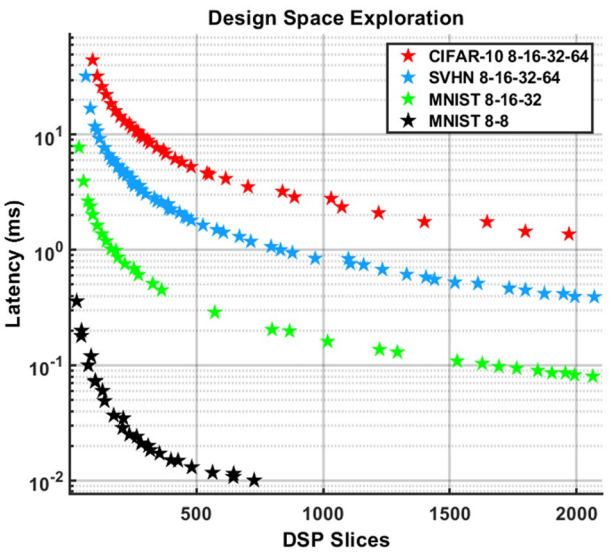

Fig. 11 Multi-Objective Optimization of Resources against Latency on MNIST 8-8, MNIST 8-16-32, SVHN 8-16-32-64 and on CIFAR-10 8-16-32-64-64, with Population Sizes 100, 250,600 and 800 respectively

this work), and finally a fully connected softmax output layer.

Y(DSP), *Y*(LUT), and *Y*(BRAM) use the resource estimation data from the previous section and the latency equations Eqs. (14) and (18) to model the active configuration's latency estimation. This is based on the number of C_{PEs} used per convolution layer and the number of FC_{PEs} mapped to parallelize the FC layer. We use DSP slices as an optimizable objective alone with latency in the examples below because we have noticed that DSP slices estimation is the most accurate, unlike LUT slices and BRAM. This is because DSP slices are easier to predict as they are directly mapped for multiplications which can be estimated with high accuracy. When DSP slices are used for optimizing resources against latency, the other metrics consistently satisfy the constraints and conditions set by the user as well. Figure 11 shows the MOGA results, a clear trade-off between latency and DSP slices can be seen in all architectures. For MNIST 8-8, the optimizer found fewer configurations bounded between a minimal mapping and a direct mapping which is possible for such a small architecture, unlike the other three which were all constrained by the maximum number of DSP slices available on the Zynq-7100 which is 2020 slices.

5.2 FPGA experimental setup

In Sects. 5.2 and 5.3, we aim to perform place and route implementations to assess performance metrics of the proposed pipelines, latency and resources will be used to explore the performance trade-offs between the different designs, also to estimate the resources overhead of including the learning pipeline. Multiple designs are generated for each architecture using the MOGA, these designs will provide a trade-off between throughput and resources by reducing the number of active data channels per layer. Four designs for each architecture will be used for the experimental setup. Training is achieved using stochastic gradient descent. We specify a mini-batch size (B_s) which sets the size of the subset batch we use for training from all available data for an iteration. We update weights after passing the data samples in each mini-batch, meaning the gradients are calculated once for each mini-batch. We stream the mini-batch through the network one sample at a time, this does not affect the final training results. We choose to do this for two reasons. First, streaming mini-batch samples lends its self well to our main design objective which is generating CNN inference and online training designs, continues learning inherently necessitates small sample size processing for training due to the limited availability of new data, this stands in contrast to having access to a full dataset off-chip and reading a mini-batch to train offline. Second, smaller mini-batch sample sizes, starting from one, allow

Table 3 Results on Architecture-8-8, MNIST, $F = 200$ MHz, training times for FPGA, GPU and CPU, Epochs = 10, Bs = 32 and Bs = 1

Design space PEs	Inference latency (ms)	Reported resources-inference						Reported resources-learning				Training time			
		DSP slices		LUT slices		BRAM		DSP slices		LUT slices		BRAM		FPGA		GPU	CPU
														BS32 (s)	BS1 (ms)		
72	0.005	728	36%	88.6 K	31%	80	10%	1528	75%	146.2 K	52%	368	47%	3.25	0.04	(BS32) 30 s	(BS32) 48 s
20	0.031	220	11%	32.6 K	12%	28	4%	460	23%	48.6 K	17%	110	14%	11.6	0.15		
6	0.112	74	3.6%	13.2 K	5%	14	2%	154	8%	18 K	7%	40	5%	33.6	0.45	(BS1) 6.7 ms	(BS1) 9 ms
2	0.368	28	1.3%	6.2 K	3%	10	1%	58	3%	7.8 K	3%	24	3%	82.8	1.10		

Table 4 Results on Architecture-8-16-32, MNIST, $F = 200$ MHz, training times for FPGA, GPU and CPU, Epochs = 30, Bs = 32 and Bs = 1

Design space PEs	Inference latency (ms)	Reported resources-inference						Reported resources-learning						Training time			
		DSP slices		LUT slices		BRAM		DSP slices		LUT slices		BRAM		FPGA		GPU	CPU
														BS32 (s)	BS1 (ms)		
164	0.05	1556	77%	194 k	72%	186	24%	3166	156%	376 k	136%	836	55%	33	1.46	(BS32) 96 s	(BS32) 121 s
104	0.112	1096	54%	132 k	47%	156	20%	2020	100%	263 k	95%	536	35%	65	2.88		
50	0.425	485	24%	65 k	24%	50	7%	958	48%	96 k	35%	218	15%	296	13.2	(BS1) 8.3 ms	(BS1) 12.6 ms
11	1.06	178	8.8%	27 k	10%	16	2%	289	14%	26 k	10%	59	4%	572	25.4		

Table 5 Results on Architecture-8-16-32-64, SVHN, $F = 200$ MHz, training times for FPGA, GPU and CPU, Epochs = 40, Bs = 32 and Bs = 1

Design space PEs	Inference latency (ms)	Reported resources-inference						Reported resources-learning						Training time			
		DSP slices		LUT slices		BRAM		DSP slices		LUT slices		BRAM		FPGA		GPU	CPU
														BS32 (m)	BS1 (ms)		
170	0.912	1920	95%	242 k	81%	213	27%	3550	175%	380 k	138%	866	57%	32.44	1.63	(BS32) 15.m	(BS32) 75 m
88	1.23	832	41%	114 k	41%	132	17%	1840	91%	235 k	85%	544	36%	43.56	2.56		
43	2.56	485	24%	72 k	26%	55	7%	977	48%	98 k	36%	213	14%	90.6	4.61	(BS1) 15 ms	(BS1) 55 ms
4	36	46	2.8%	9 k	3%	8	1%	96	5%	9 k	3%	21	2%	1275	82.1		

Table 6 Results on Architecture-8-16-32-64-64, CIFAR-10, $F = 200$ MHz, training times for FPGA, GPU and CPU, Epochs = 60, Bs = 32 and Bs = 1

Design space PEs	Inference latency (ms)	Reported resources-inference						Reported resources-learning						Training time			
		DSP slices		LUT slices		BRAM		DSP slices		LUT slices		BRAM		FPGA		GPU	CPU
														BS32 (m)	BS1 (ms)		
107	3.36	1062	53%	121 k	45%	124	16%	2000	98%	238 k	86%	543	36%	168	4.61	(BS32) 56.1 m	(BS32) 171 m
68	5.16	710	35%	83 k	30%	85	11%	1372	68%	152 k	55%	344	23%	256	10.28		
39	10.1	606	30%	67 k	24%	74	9%	801	40%	87 k	31%	198	13%	518	20.46	(BS1) 34 ms	(BS1) 109 ms
22	12	224	11%	31 k	11%	28	3%	382	19%	41 k	14%	92	6%	617	24.44		

for successfully training on smaller designs which only use on-chip memory and can require less than 2% of the chip's DSP slices as our results will show in Sect. 5.3, smaller input sample sizes allow for on-board training with limited resources. Larger sample sizes can be used depending on a design's requirements but for our experiments, all sample sizes were set to one to simulate a continuous learning scenario where the design is deployed on an embedded setting and new data is available in smaller samples for fine-tuning an already-trained network. Training a model from scratch using an offline dataset is not the objective of our work, for such scenarios discrete GPU setups remain to be more suitable. It is worth noting that the new data made available is of known classes, continuous learning of new classes is not possible using our setup since it would require an online change to the architecture's output layers, this is possible using the FPGA's online reconfiguration capabilities but not supported in this work. Tables 3, 4, 5, 6 show different implementations of the designs generated from the architectures listed in Table 1 trained on the datasets in Table 2. The generated designs can support FP inference alone or include backpropagation. Multiple designs with differing resource requirements and throughput performance were used for each architecture to highlight the different design options a user can pick from based on hardware limitations and task-specific performance and objectives. All simulations and implementations were carried out on the Xilinx Zynq-7100 board. The post place and route implementations of the designs highlight the overhead of including backpropagation support in the designs. The overhead is consistent with the resource estimation details given in the previous section. Additional DSP slices are mapped to account for the additional parallelized processing needed to calculate local gradients and update weights. Additional BRAM blocks are dedicated to buffer intermediate data from the FP required for BP calculations and additional LUT slices are needed for the local schedulers and the overall new BP glue logic. Input data is sent to the testbench using the Vivado/Simulink co-simulation environment described in Sect. 4.1. The architectures were trained on a single GeForce GTX 1050 Ti GPU and an i7-6700 3.40 GHz × 8 CPU, the hyperparameters used for each training session were replicated for the FPGA training, the results are reported in Tables 3, 4, 5, 6. GPUs performance is optimal on batches because of the GPU's built-in parallelizing capabilities. A different approach to comparing the performance based on our practical objectives is one-sample latency simulating data streams. Accuracies achieved using these hyperparameters specified in Tables 3, 4, 5, 6 were 96.2, 99.6, 88.93, and 76.37% for MNIST-8-8, MNIST 8-16-32, SVHN 8-16-32-64, and CIFAR 8-16-32-64-64 respectively.

Table 7 Online training comparisons with related works

FPGA				GPU (1050 Ti)		CPU (i7)	
	This Work (ms)	[28] (ms)	[32] (ms)	BS1 (ms)	BS32 (ms)	BS1 (ms)	BS32 (ms)
CIFAR-10	4.61	NA	1.96	15	1.11	55	3.42
MNIST	2.88	53	NA	8.3	0.4	12.6	0.64

5.3 FPGA experimental results and analysis

All tables show the resources of the different designs for FP and BP highlighting the overhead of including a training pipeline into the design which pushes some implementations beyond the available on-chip resources (seen highlighted in red) in Tables 4 and 5. These designs were included to highlight the overhead and the possibility of implementation when a larger board is available, the latency results included however are estimations from pre place-and-route simulations. The different designs generated based on the number of dedicated PEs also highlight a clear trade-off between latency and resources, for instance, in Table 3 designs-1 and 4 show an 200× speed up for 9× more DSP slices allowing the user to tailor the design process to their specific application. In Training times, the GPU consistently performs better than the FPGA except for design-1 and 2 in Tables 3 and 4, this is because the FPGA is processing data as a stream one sample at a time while the GPU is built to process batches in parallel. A more accurate comparison can be seen in the stream-mode simulation on the GPU, it shows that the FPGA consistently outperforms the GPU on single-sample scenarios like online continuous learning. Latency results highlighted in green indicate that the FPGA outperformed the GPU for that specific design configuration and dataset, orange indicates the opposite. The results highlight FPGA platforms as strong candidates for future online learning applications, the per-sample latency, and energy overhead consistently outperform GPUs in online training scenarios with limited input data rates. Table 7 shows comparisons with related works, there is a clear lack of research in the area of online training of CNNs on FPGA, to the best of our knowledge [27] and [30] are the only works that have implemented training architectures on FPGAs. In [27] and [30] Stratix-V and Startix-10 boards were used with 150 MHz and 240 MHz frequencies, 32-bit and 16-bit data representations with 1963 and 5760 DSP usage to implement LeNet-5 MNIST and CIFAR-10 custom architecture respectively. Our training pipeline allows for new data to be processed as early as it becomes available while previous samples are still being processed, however the results we reported were for single samples, meaning the resources for the already processed areas of the pipeline become idle. When data is provided as early as an area of the pipeline becomes idle, we achieved an effective improvement in latencies upward of 4× times for deeper architectures. It is also worth mentioning that [30] used 5760 DSP slices at 240 MHz against our 2020 DSP slices at 200 MHz.

6 Conclusion

In this work, we have presented a fully automated design backpropagation pipeline for CNNs with a focus on online training. The pipeline uses a streaming interface and a modular design approach, generic PEs are generated and used to populate a design space based on the user's specific requirements including the option to compile a training pipeline. We explore the design space using a multi-objective genetic algorithm and an analytical model of the network's latency and estimated resources, these allow for optimizing the hardware model by exploiting CNN's intralayer parallelism. Latency trade-offs of 95× for MNIST, 71× for CIFAR-10, and 18× for SVHN were achieved. Trade-offs in resource utilization in terms of DSP Slices were 44× for MNIST, 52× for SVHN, and 24× for CIFAR-10. The training pipeline is generated based on an overlap in computation between the forward and backward passes. We translate the overlap into hardware by reusing most of the forward pass pipeline reducing the resources overhead. The design also minimizes the need for off-chip memory by utilizing BRAM to buffer in parameters and intermediate feature maps when needed, feature maps are streamed during the FP. We minimize data movement by placing BRAM memory near PEs, this can be used to buffer in data when external memory is required for larger designs which was not covered in this work as we only used small to medium scale architectures. The results show that the FPGA implementations of these architectures outperform their GPU and CPU counterparts in most online learning scenarios where samples are streamed at a limited rate, A 2.8×, 5.8×, and 3× speed up over GPU was achieved on three deeper architectures trained on MNIST, SVHN, and CIFAR-10 respectively. Using batch processing, GPUs consistently outperform FPGA for larger designs. Our main objective in developing these tools is to support the onboard deployment of deep learning models for both inference and online learning. This tool can provide deep learning engineers with an easy and accessible design cycle. There is a difficult challenge in scaling these tools efficiently to account for the rapid progress in modern CNNs, both in terms of the shape and size of new architectures. Future works in the field of on-board learning should tackle the limitations

of this work and the ones that preceded it, this includes the support for deeper architectures, and the different layers that are present in modern CNNs, automating the design and implementation of external memory protocols is, therefore, necessary as well.

References

1. Fowers, J., Brown, G., Cooke, P., Stitt, G.: A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Paper presented at the Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, Monterey, California, USA
2. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA, 22 Feb 2015, pp. 161–170. ACM, 2689060
3. Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., Yang, H.: Going deeper with embedded FPGA platform for convolutional neural network. In: Paper presented at the Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA
4. Gan, F., Zuyi, H., Song, C., Feng, W.: Energy-efficient and high-throughput FPGA-based accelerator for Convolutional Neural Networks. In: 2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), 25–28 Oct. 2016, pp. 624–626 (2016)
5. Venieris, S.I., Bouganis, C.: fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs. In: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 1–3 May 2016, pp. 40–47 (2016)
6. Wang, Y., Xu, J., Han, Y., Li, H., Li, X.: DeepBurning: automatic generation of FPGAbased learning accelerators for the neural network family, in: Design Automation Conference (DAC) (2016)
7. Zhiqiang, L., Yong, D., Jingfei, J., Jinwei, X.: Automatic code generation of convolutional neural networks in FPGA implementation. In: 2016 International Conference on Field-Programmable Technology (FPT), 7–9 Dec. 2016, pp. 61–68 (2016)
8. Hwang, W.J., Jhang, Y.J., Tai, T.M.: An efficient FPGA-based architecture for convolutional neural networks. In: 2017 40th International Conference on Telecommunications and Signal Processing (TSP), Barcelona, Spain, 5 July 2017, pp. 582–588. IEEE (2017)
9. Hao, Y., Quigley, S.: The implementation of a deep recurrent neural network language model on a xilinx fpga. arXiv preprint arXiv:1710.10296 (2017)
10. Kaiyuan Guo, S.Z., Jincheng, Y., Yu, W., Huazhong, Y.: [DL] A survey of FPGA-based neural network inference accelerators. ACM Trans Reconfig TechnolSyst **12**, 1 (2019)
11. Wu, R., Guo, X., Du, J., Li, J.: Accelerating neural network inference on FPGA-based platforms—A survey. Electron **10**,1025 (2021). https://doi.org/10.3390/electronics10091025
12. Guo, K., Sui, L., Qiu, J., Yu, J., Wang, J., Yao, S., Han, S., Wang, Y., Yang, H.: Angel-eye: a complete design flow for mapping CNN onto embedded FPGA. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(1), 35–47 (2018). https://doi.org/10.1109/TCAD.2017.2705069
13. Abdelouahab, K., Pelcat, M., Serot, J., Berry, F.: Accelerating CNN inference on FPGAs: a survey. arXiv preprint arXiv:1806.01683 (2018).
14. Solovyev, R.A., Kalinin, A.A., Kustov, A.G., Telpukhov, D.V., Ruhlov, V.S.: FPGA implementation of convolutional neural networks with fixed-point calculations. 2018. [Online]. Available: arXiv preprint arXiv:1808.09945
15. Venieris, S.I., Kouris, A., Bouganis, C.-S.: Toolflows for mapping convolutional neural networks on FPGAs: a survey and future directions. ACM Comput. Surv. **51**(3), Article 56 (2018). https://doi.org/10.1145/3186332
16. Rivera-Acosta, M., Ortega-Cisneros, S., Rivera, J.: Automatic tool for fast generation of custom convolutional neural networks accelerators for FPGA. Electronics **8**(6), 641 (2019)
17. Mazouz, A., Bridges, C.P.: Adaptive hardware reconfiguration for performance tradeoffs in CNNs. In: 2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 22–24 July 2019, pp. 33–40 (2019)
18. Mazouz, A., Bridges, C.P.: Automated offline design-space exploration and online design reconfiguration for CNNs. In: 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS), 27–29 May 2020, pp. 1–9 (2020)
19. Hayes, T.L., Kanan, C.: Lifelong machine learning with deep streaming linear discriminant analysis. In CVPR-W (2020)
20. Käding, C., Rodner, E., Freytag, A., Denzler, J.: Fine-tuning deep neural networks in continuous learning scenarios. In: Chen C.-S., Lu J., Ma K.-K. (eds.) Computer Vision—ACCV 2016 Workshops, Cham, 2017, pp. 588–605. Springer International Publishing (2017)
21. Yoon, J., Yang, E., Lee, J., Hwang, S.J.: Lifelong learning with dynamically expandable networks, in ICLR (2018)
22. Parisi, G.I., Kemker, R., Part, J.L., Kanan, C., Wermter, S.: Continual lifelong learning with neural networks: a review. Neural Netw. **113**, 54–71 (2019). https://doi.org/10.1016/j.neunet.2019.01.012
23. Roy, D., Panda, P., Roy, K.: Tree-CNN: a deep convolutional neural network for lifelong learning. ArXiv abs/1802.05800 (2018)
24. Posewsky, T., Ziener, D.: Throughput optimizations for fpga-based deep neural network inference. Microprocess Microsyst. 60:151–161 (2018)
25. Stimpson, A.J., Tucker, M.B., Ono, M., Steffy, A., Cummings, M.L.: Modeling risk perception for mars rover supervisory control: before and after wheel damage. In: Aerospace Conference, 2017 IEEE, Montana. USA, Mar 4 2017, pp. 1–8. IEEE (2017)
26. Mazouz, A., Bridges, C.P.: Multi-sensory CNN models for close proximity satellite operations. In: 2019 IEEE Aerospace Conference, 2–9 March 2019, pp. 1–7 (2019)
27. Choi, S., Sim, J., Kang, M., Kim, L.-S.: TrainWare: a memory optimized weight update architecture for on-device convolutional neural network training. In: Paper presented at the Proceedings

of the International Symposium on Low Power Electronics and Design, Seattle, WA, USA
28. Wenlai, Z., Haohuan, F., Luk, W., Teng, Y., Shaojun, W., Bo, F., Yuchun, M., Guangwen, Y.: F-CNN: an FPGA-based framework for training convolutional neural networks. In: 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), London, England, 6 July 2016, pp. 107–114. IEEE (2016)
29. Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T.V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C.R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., Yoon, D.H.: In-datacenter performance analysis of a tensor processing unit. In: Paper presented at the Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, ON, Canada
30. Hao, Y., Quigley, S.: The implementation of a deep recurrent neural network language model on a xilinx fpga. arXiv preprint arXiv:1710.10296 (2017).
31. Caulfield, A.M., Chung, E.S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., Burger, D.: A cloud-scale acceleration architecture. In: Paper presented at the The 49th Annual IEEE/ACM International Symposium on Microarchitecture, Taipei, Taiwan
32. Venkataramanaiah, S.K., Ma, Y., Yin, S., Nurvithadhi, E., Dasu, A., Cao, Y., Seo, J.: Automatic compiler based FPGA accelerator for CNN training. In: 2019 29th International Conference on Field Programmable Logic and Applications (FPL), 8–12 Sept. 2019, pp. 166–172 (2019)
33. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
34. Xilinx: Zynq-7000 Soc Data Sheet: Overview. In. Xilinx (2018). https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
35. Mathworks: Generate Target-Independent HDL Code with Native Floating-Point. https://uk.mathworks.com/help/hdlcoder/ug/generate-target-independent-hdl-code-with-native-floating-point-libraries.html (2015). Accessed 04 Aug 2020
36. Konak, A., Coit, D.W., Smith, A.E.: Multi-objective optimization using genetic algorithms: a tutorial. Reliab. Eng. Syst. Saf. **91**(9), 992–1007 (2006). https://doi.org/10.1016/j.ress.2005.11.018
37. Krizhevsky, A.: Learning Multiple Layers of Features from Tiny Images. University of Toronto, Toronto (2012)
38. Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., Ng, A.: Reading Digits in Natural Images with Unsupervised Feature Learning. NIPS (2011).

A. Mazouz received a B.S. from the Institute of Electrical and Electronic Engineering at the University of Boumerdes, Algeria in 2014. An MSc in Power Engineering from the same institute in 2016. 2nd year Ph.D. student at the Surrey Space Centre (SSC), member of the On-Board Data Handling group researching runtime adaptive deep learning for embedded systems.

C. P. Bridges received the B.Eng. degree from the University of Greenwich, London, U.K., in 2005, and the Ph.D. degree from the University of Surrey, Guildford, U.K., in 2009, both in electronic engineering. In 2013, he designed, built, and still operates the U.K.'s first CubeSat (STRaND-1) with Surrey Satellite Technology (SSTL) and now contributes toward computing hardware and software with SSTL, on ESA's ESEO mission and also the NASA-JPL/CalTech AAReST mission. He currently leads the On-Board Data Handling (OBDH) research group within Surrey Space Centre, University Of Surrey. His research interests include software-defined radios, real-time embedded systems, agent computing, Java processing, multicore processing in FPGAs, and astrodynamic computing methods in many spaceflight payloads.