Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA

Roman Solovyev, Alexander Kustov, Dmitry
Telpukhov, Vladimir Rukhlov
Institute for Design Problems in Microelectronics of
Russian Academy of Sciences (IPPM RAS)
Moscow, Russia
zf-turbo@yandex.ru

Alexandr Kalinin,
Department of Computational Medicine and Bioinformatics,
University of Michigan,
Ann Arbor, MI, USA

Abstract — **Modern mobile neural networks with a reduced number of weights and parameters do a good job with image classification tasks, but even they may be too complex to be implemented in an FPGA for video processing tasks. The article proposes neural network architecture for the practical task of recognizing images from a camera, which has several advantages in terms of speed. This is achieved by reducing the number of weights, moving from a floating-point to a fixed-point arithmetic, and due to a number of hardware-level optimizations associated with storing weights in blocks, a shift register, and an adjustable number of convolutional blocks that work in parallel. The article also proposed methods for adapting the existing data set for solving a different task. As the experiments showed, the proposed neural network copes well with real-time video processing even on the cheap FPGAs.**

Keywords—Neural network hardware; Field programmable gate arrays; Fixed-point arithmetic; 2D convolution

I. INTRODUCTION

Recent research in artificial neural networks has demonstrated their ability to perform well on a wide range of tasks [1], [2]. Most of the modern neural network architectures for computer vision include convolutional layers and thus are called convolutional neural networks (CNNs). They have high computational requirements. However, there is a compelling need for the use of deep convolutional neural networks on mobile devices and in embedded systems. This is particularly important for video processing in, for example, autonomous cars and medical devices [3].

Following properties of many modern high-performing CNN architectures make their hardware implementation feasible:

- high regularity: all commonly used layers have similar structure (Conv3x3, Conv1x1, MaxPooling, FullyConnected, GlobalAvgPooling);

- typically small size of convolutional filters: 3×3;

- ReLU activation function (comparison of the value with zero): easier to compute compared to previously used Sigmoid and Tanh functions.

Due to high regularity, size of the network can be easily varied, for example, by changing the number of convolutional blocks. In the case of field programmable gate arrays (FPGAs), this allows to program the network on different types of FPGAs, providing different processing speed. For example, implementation of higher number of convolutional blocks on an FPGA can directly lead to a speed-up in processing.

Related direction in neural network research considers adapting NNs for the use on mobile devices [4]. Mobile networks typically have reduced number of weights and require relatively small number of arithmetic operations. However, they are still executed at the software level and use floating-point calculations. For some tasks such as real-time video analysis that requires processing of 30 frames per second mobile networks still can be not fast enough without further optimization. In order to use an already trained neural network in a mobile device, a set of optimizations can be used to speed up computation. There exist a number of approaches to do so, including weight compression or computation using low-bit data representations. Since hardware requirements for neural networks keep increasing, there is a need for design and development of specialized hardware block for the use in ASIC and FPGA. The speed up can be achieved by following:

- hardware implementation of the convolution operation, which is faster than software convolution;

- using fixed-point arithmetic instead of floating-point calculations;

- reducing the network size while preserving the performance;

- modifying structure of a network architecture while preserving the same level of performance and decreasing footprint of the hardware implementation and saved weights.

For example, Zhang C. et al. [6] quantitatively analyzed computing throughput and required memory bandwidth for various CNNs using optimization techniques, such as loop tiling and transformation. This allowed their implementation to achieve peak performance of 61.62 GFLOPS. Qiu J. et al. [5] proposed an FPGA implementation of pre-trained deep neural networks from VGG family. They used dynamic-precision quantization with 48-bit data representation and singular vector decomposition to reduce the size of fully-connected layers,

which led to smaller number of weights that had to be passed from the device to the external memory. Higher level solution is proposed in [7], which considers the use of the OpenGL compiler for deep networks, such as AlexNet and VGG. Duarte et al. [8] have recently suggested the protocol for automatic conversion of neural network implementations in high-level programming language to intermediate format (HLS) and then into FPGA implementation. However, their work is mostly focused on the implementation of fully-connected layers. In this work we propose a design and implementation of FPGA-based CNN with fixed-point calculations that allows achieving the exact performance of the corresponding software implementation on the live hand written digit recognition problem. Due to the reduced number of parameters we avoid common issues with memory bandwidth. Suggested method can be implemented on very basic FPGAs, but also is scalable for the use on FPGAs with large number of logical cells. Additionally, we demonstrate how existing open datasets can be modified in order to better adapt them for real-life applications. Finally, in order to promote the reproducibility of results, facilitate open-scientific development, and enable collaborative validation we make our source code, documentation, and all results from this study available online.
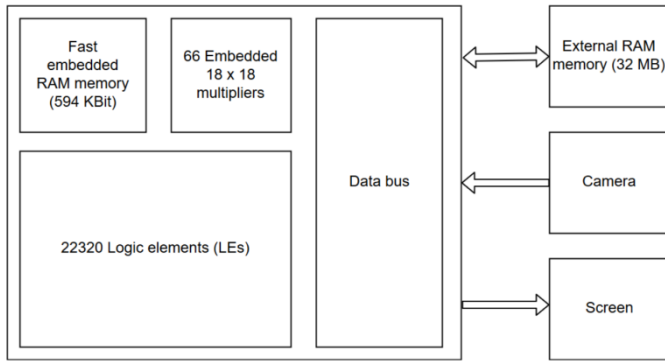

Fig. 1.   DE0-Nano development board and external devices

II. METHODS

A. Implementation requirements

To demonstrate our approach, we implement a solution for the problem of recognizing handwritten digits received from a camera in real time. The results are displayed on an electronic LED screen. The minimal speed of digit recognition should exceed 30 FPS, that is, neural network should be able to process a single image in 33ms. The resulting hardware implementation should be ready for transfer to separate custom VLSI device for mass production.

B. Hardware specifications

We use the compact development board DE0-Nano due to the following reasons:

- *Intel (Altera) FPGA is installed on this board, which is mass-produced and cheap;*

- *Cyclone IV FPGA has rather low performance and small number of logic cells, assuming increased performance if re-implemented with most of other modern FPGAs;*

- *it makes connecting peripherals, such as camera and touchscreen, easier;*

- *the board itself has 32 MB of RAM, which can be used to store weights of a neural network. The general scheme of the board and external devices is shown in Fig. 1.*

OV7670 camera module is chosen for image acquisition due to the high quality/price ratio. In this application, high resolution video is not required, since every image is reduced to the size of 28 × 28 pixels and converted to grayscale. The camera module also has a simple connection mechanism.

Display module with 320 × 240 resolution TFT screen is chosen as the output device. Display module driven by microcontroller is equipped by 2.4 inches color touchscreen (18-bitcolor, 262,144 color variations). It also has backlight and is convenient to use due to the large viewing angle. Contrast and dynamic properties of H24TM84A LCD indicator allow displaying video. LCD controller contains RAM buffer that lowers the requirements for the device microcontroller.

Final scheme for connecting camera and screen modules to De0-Nano board is shown in Fig. 2.
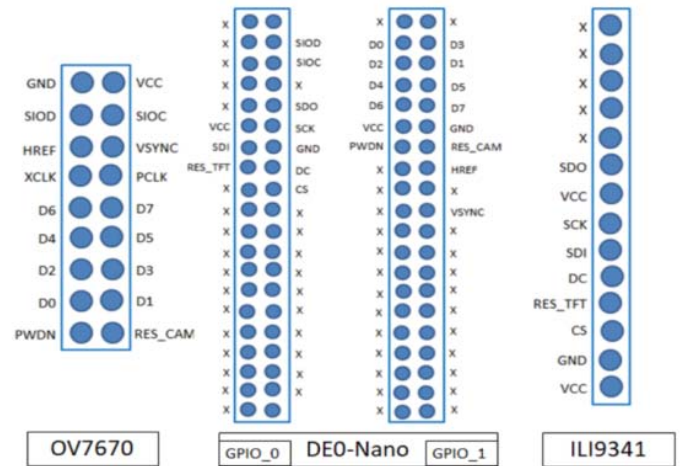

Fig. 2.   Scheme for connecting camera and screen modules to De0-Nano board. Pins with the same name are connected. Pins marked as 'x' remain unconnected

C. Dataset preparation

The MNIST dataset for handwritten digit recognition [9] is widely used in the computer vision community. However, it is not well suited for training a neural network in our application, since it differs greatly from the camera images (Fig. 3).

Major differences include:

- *MNIST images are light digits over dark background, opposite to those from the camera feed;*

- *camera produces color images, while MNIST is grayscale;*

- *size of MNIST image is 28 × 28 pixels, while camera image size is 320 × 240 pixels;*

- *unlike centrally placed digits and homogenious background in MNIST images, digits can be shifted and slightly rotated in camera images, sometimes with noise in the background;*

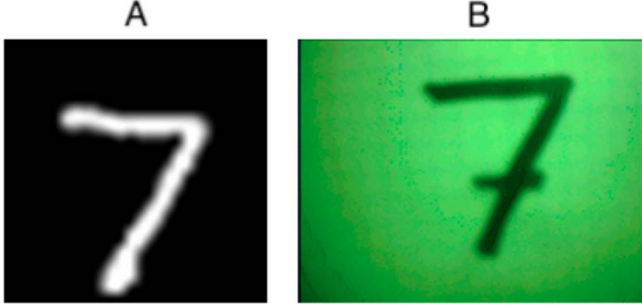- *MNIST does not have a separate class of images without digits.*

Fig. 3. Different appearance of (A) an image from the MNIST dataset; and (B) an image from the camera feed

Given that the recognition performance on the MNIST dataset is very high, we reduce the size of image from camera to 28 × 28 pixels and convert them into grayscale. This helps us to address following problems:

- there is no significant loss in accuracy, as even in small images digits are still easily recognized by humans;

- color information is excessive for digit recognition;

- noisy images from camera can be cleaned by reducing and averaging neighboring pixels.

Since image transformation is also performed at hardware level, it is necessary to consider in advance a minimum set of arithmetic functions that can effectively bring image to the desired form. The suggested algorithm for modifying camera images goes as following:

1) We crop a central part measuring 224 × 224 pixels from a 320 × 240 image, which subsequently allows easy transition to the desired image size, since 224 = 28 × 8.

2) Then, a cropped image part is converted to a grayscale image. Because of the peculiarities of human visual perception, we take weighted, rather than simple, average. To facilitate the conversion at the hardware level, the following formula is used:

$$B W = (8 \times G + 5 \times R + 3 \times B)/16 \qquad (1)$$

Multiplication by 8 and division by 16 are implemented using shifts.

3) Finally, 224 × 224 image is split into 8 × 8 blocks. We calculate average value for each of these blocks, forming a corresponding pixel in 28 × 28 image.

Resulting algorithm is simple and works very fast at the hardware level.

In order to use MNIST images for training a neural network, on-the-fly data augmentation is used. This method implies that during the creation of the next mini-batch for training, a set of different filters is arbitrarily applied to each image. This technique is used to easily increase the dataset size, as well as to bring images to the required form, as in our case.

The following filter set was used for augmenting MNIST images:

- color inversion;

- random 10 degrees rotation in both directions;

- random expansion or reduction of an image by 4-pixels;

- random variation of image intensity (from 0 to 80);

- adding random noise from 0% to 10%.

Optionally, images from camera can be mixed into mini-batches.

D. CNN architecture design

Despite recent developments in CNN architectures, the essence remains the same: the input size decreases from layer to layer and the number of filters increases. At the end of a network, a set of characteristics is formed that are fed to the classification layer (or layers), and the output neurons indicate the likelihood that the image belongs to a particular class.

The following set of rules for constructing a neural network architecture is proposed to minimize the total number of stored weights (which is critical for mobile systems) and facilitate the transfer to fixed-point calculations:

- minimize number of fully connected layers, which consume major part of memory for storing weights;

- reduce number of filters of each convolution layer as much as possible without degrading the classification performance;

- stop using bias, which is important when shifting from floating-point to fixed-point, because adding a constant hinders monitoring range of values, and rounding bias error over each layer tends to accumulate;

- use simple type activation, such as RELU, since other activations, such as Sigmoid and Tahn, contain division, exponentiation, and other functions that are harder to implement in hardware;

- minimize number of heterogeneous layers, so that one hardware unit can perform calculations at a large number of flow stages.

Before translating the neural network onto hardware, we train it on a prepared dataset and save the software implementation for testing. We create software implementation using Keras with Tensorflow backend.

In our previous work, we have proposed a VGG Simple neural network [10], which is a lightweight modification of the popular VGG architecture [9]. Despite the high performance,

the major disadvantage of this model is the number of weights, size of which exceeds FPGA capacity. Besides, the exchange with external memory imposes additional time costs. Moreover, this model involves a "bias" term, which also has to be stored, requires additional processing blocks, and tends to accumulate error if implemented in fixed-point representation. Therefore, we propose a further modification of this architecture that we call Low Weights Digit Detector (LWDD).
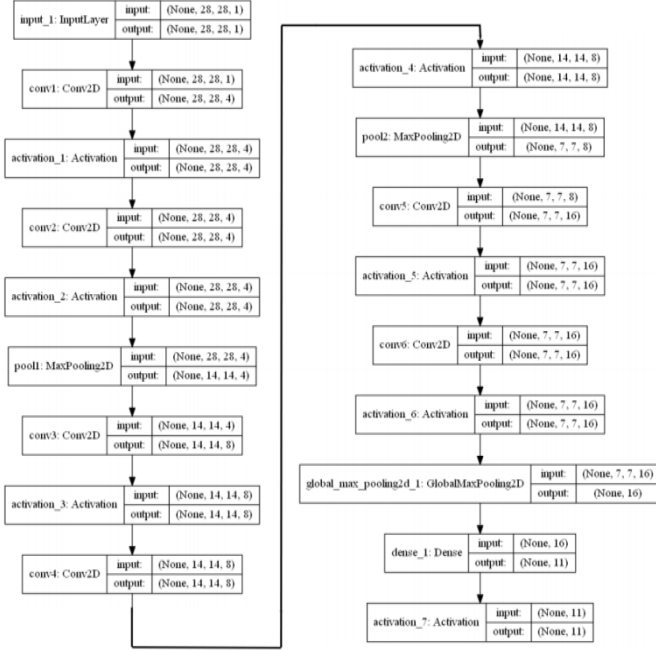

Fig. 4. Low Weight Digit Detector (LWDD) neural network architecture

First, we remove large fully connected layers and bias terms. Then, GlobalMaxPooling layer is added to neural network, instead of GlobalAvgPooling, which is traditionally used, for example, in ResNet50. The efficiency of these layers is approximately the same, while the hardware complexity of finding a maximum is much simpler than mean value calculation from the computational point of view. These introduced changes do not lead to the decrease in network performance. New architecture is shown in Fig. 4. Changes in neural network structure allow to reduce number of weights from 25,000 to approximately 4,500, and to store all weights in the internal memory of the FPGA. On the modified MNIST dataset with image augmentations, LWDD neural network achieves 96% accuracy.

E. Fixed-point calculation implementation

In neural networks, calculations are traditionally performed with floating point either on GPU (fast) or CPU (slow), for example, using float32 type. When implemented at the hardware level, floating-point calculations are slower than fixed-point due to the difficulty of controlling the mantissa and the exponent for various operations.

Let's consider the first convolutional layer of a neural network, which is the main building block of convolutional architectures.

At the layer input is a two-dimensional matrix (original picture) 28×28 with values from $[0; 1)$. It is also known that if $a \in [-1, 1]$ and $b \in [-1, 1]$, then $a \cdot b \in [-1, 1]$.

For 3×3 convolution, the value of the certain pixel (i, j) in the second layer can be calculated as follows:

$$n_{i,j} = b + w_{00}p_{i-1,j-1} + w_{01}p_{i-1,j} \\ + w_{02}p_{i-1,j+1} + w_{10}p_{i,j-1} + w_{11}p_{i,j} \quad (2) \\ + w_{12}p_{i,j+1} + w_{20}p_{i+1,j-1} \\ + w_{21}p_{i+1,j} + w_{22}p_{i+1,j+1}$$

Since weights $w_{i,j}$ and bias b are known, it is possible to calculate potential minimum mn and maximum mx of the second layer. Let $M = \max(|mn|, |mx|)$. If we divide $w_{i,j}$ and b by the value of M, we can guarantee that for any configuration of input data, the value on the second layer does not exceed 1. We call M a reduction coefficient of the layer. For the second layer, we use the same principle, namely, the value at layer input belongs to interval $[-1;1]$, so we can repeat our reasoning. For the proposed neural network after all weight reductions to the last layer, the position of the maximum of the last neuron is not changed, that is, the network works equivalently to the neural network without reductions from the point of view of floating-point calculations.

After performing this reduction on each layer, we can move from floating-point calculations to fixed-point calculations, since we know exactly the range of values at each stage of computation. We use the following notation to represent the numbers of bits: $x_b = [x \cdot 2^N]$.

If $z = x + y$, then addition can be expressed as: $z' = x_b + y_b = [x \cdot 2^N] + [y \cdot 2^N] = [(x + y) \cdot 2^N] = [z \cdot 2^N] = z_b$. Multiplication can be expressed as: $z' = x_b + y_b = [x \cdot 2^N] \cdot [y \cdot 2^N] = [(x \cdot y) \cdot 2^N \cdot 2^N] = [z \cdot 2^N \cdot 2^N] = [z_b \cdot 2^N]$, that is, we have to divide multiplication result by 2^N to get the real value, or just shift it by N positions.

If we sort through all possible input images and focus on the potential minimum and maximum values, we can get very large reduction coefficients, such that the accuracy will rapidly decrease from layer to layer. This can require a large width of fixed-point representation of weights and intermediate computational results. To avoid this, we can use all (or a part) of the training set to find most likely maximum and minimum values in each layer. As our experiments are show, usage of the training set makes it possible to decrease reduction coefficients. At that, we should scale up coefficients by a small margin, either focusing on the value of 3σ or increasing the maximum by several per cent.

However, under certain conditions, overflow and violation of the calculated range are possible. To address this issue, a hardware implementation requires a detector of such cases and the mechanism for replacement of overflowed values with the maximum for given layer. This can be achieved by minor modifications of a convolutional unit.

For fixed-point calculations with the limited width of weights and intermediate results, rounding errors inevitably arise, accumulate from layer to layer, and can lead to "inaccurate" predictions. We consider "inaccurate" predictions

to happen when the predicted value is compared with the prediction by the software implementation, rather then with the true image label. To validate the "accuracy" of predictions, we run all test images through both the floating-point software implementation and fixed-point software implementation (or Verilog benchmark) and then compare predictions. Ratio of mismatches to the total number of tests is a measure of is "inaccuracy" measure for the given width of weights and intermediate results. We choose the bit width at which the number of errors is 0.

When using fixed-point calculations with convolution blocks, two different strategies are possible:

• rounding after each elementary operation of addition and multiplication;
• calculation with full accuracy and rounding at the very end of convolution operation.

Two experiments are carried out to determine the most effective approach. To achieve zero difference from the floating-point model, the number storage requires 17 bits in case of rounding at the beginning, and only 12 bits in case of rounding at the end.

Rounding after each operation slightly increases performance, and significantly increases memory overhead. Therefore, it is advantageous to perform rounding after convolution block.

F. FPGA-based hardware implementation

In FPGA-based realization, SDRAM is used to store a video frame from camera. In SDRAM memory on De0-Nanocard used in this study, two equal areas for two frames are allocated — current frame is recorded in the first area, and previous frame is read from the other memory area. And after the output is finished, these areas change their roles. When using SDRAM memory in this study, we consider two important issues. First, memory operates at high frequency of 143MHz, thus, we face one more problem of transferring of data from the clock domain of camera to the clock domain of SDRAM. Second, in order to achieve maximum speed, writing to SDRAM should be performed by whole transactions, or in "burst". FIFO directly built in FPGA memory is the best way to solve both of these problems. Basic idea is that camera fills FIFO at low frequency, then SDRAM controller reads data at high frequency, and immediately writes them to memory in one transaction. Data output to TFT screen is organized in the same way. Data from SDRAM are written to screen FIFO, and then are read at the frequency of 10MHz. After FIFO has been cleared, the operation is repeated.

A picture from the camera, after passing through SDRAM, is displayed on the screen as is, and also is fed to neural network for its recognition through block that converts image to grayscale and decreases resolution. When neural network operation is finished, the result is also output directly to the screen.

After conversion, input image is stored in the database, which also stores weight coefficients for each layer that were calculated and wired-in beforehand. As necessary, data from

there is downloaded through the controller to the small memory unit for the further use. In the hardware realization, not all layers of neural network under test are used; some of them are replaced by other functions. For example, there is no ZeroPadding layer, instead of it module of intermediate image edge detection is applied, which allows to reduce chip memory usage. GlobalMaxPooling layer is replaced by the function from the Convolution layer that immediately gets GlobalMaxPooling layer result by finding the largest value in the intermediate image. The rest of the layers are implemented as separate modules. Since Convolution and Dense layers can use convolutional blocks for calculations, both of them have access to these blocks. Modules contain ReLU activation function, which is used as needed. In the last layer, Softmax activation function is applied. It is implemented as traditional Maximum, because position of a neuron with the maximum value is always the same for these functions. To implement the neural network, the specialized Convolution block is used, which performs convolution of 3×3 in one clock cycle. This block is a scalar product of vectors and contains 9 multiplications and 8 additions. The same block is used for calculations in the fully connected Dense layer due to splitting the entire set of additions and multiplications into blocks of 9 neurons.

Fig. 5.   Shift register operation: blue indicates data for previous convolution operation obtained at the previous step
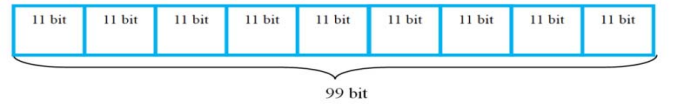

Fig. 6.   Storage of all layer weights as single block

TABLE I.   INFORMATION ON RESOURCES USED ON FPGA AFTER PLACE & ROUTE STAGE

	Logiccells (available: 22320)	9-bit elements (132)	Internal memory (available: 608256 bit)	PLLs (4)
Input image converter	964 (4%)	0 (0%)	0 (0%)	0 (0%)
Neural network	4014 (18%)	23 (17%)	285428 (47%)	0 (0%)
Weights database	0 (0%)	0 (0%)	70980 (12%)	0 (0%)
Storage of intermediate calculation results	1 (< 1%)	0 (0%)	214448 (35%)	0 (0%)
Total usage	5947 (27%)	23 (17%)	371444 (61%)	2(50%)

G. Additional optimization of calculations

To increase the performance, a number of techniques are applied that made it possible to reduce the number of cycles required for one image classification.

1) Increasing of convolution blocks number: If there is enough free space in FPGA, we can improve the performance by increasing the number of convolution blocks, thereby multiplying productivity. Consider the second convolutional block in the proposed neural network LWDD. There are 4 of 28×28 images at the layer input, and 16 blocks of weights are given. To calculate the set of outputs for this layer, also consisting of 4 images, we have to perform four multiplications of the same set of pixels by different sets of weights. If there is only one convolutional block, this takes at least 4 cycles, but if there are 4 such blocks, then only one clock cycle is needed, thus Convolution layer calculation speeds up 4 times.

2) Shift register: To perform an elementary convolution operation, we have to get values of 9 neighboring pixels from an input image, then next 9 pixels, 6 of which have already been received in the previous step (see Fig. 5). To shorten the time for the necessary data call up, shift register is developed to keep new data at their input and at the same time to "push out" old data. Thus, each step requires only 3 new values instead of 9.

TABLE II.   CLOCK CYCLES PER CONVOLUTION BLOCK

	Clock cycles per 1 frame	Processing speed-up
1st convolution block	236746	-
2nd convolution block	125320	1,89
4th convolution block	67861	3,49

3) Storing of all data for one Convolution operation at the same address: When we call up data that are necessary for calculations, one clock cycle is used for each value. Therefore, in order to reduce time spent on downloading required data, as well as convenience of access, prior to putting into internal memory of FPGA, data are stacked in blocks of 9 pieces, after which they are accessible at one address. With such memory arrangement, we can perform the extraction of weights in one clock cycle and, thus, speed up calculations for convolutional and fully connected layers. Example is shown in Fig. 6.

TABLE III.   TOTAL RESOURCE USAGE FOR THE PROJECT

Weight dimensions	Convolutional blocks	Logical cells	Memory	Embedded M9 elements	Critical path delay	Max. FPS
11 bit	1	3750	232111	25	21,84	193
	2	4710	309727	41	22,628	352
	4	6711	464959	77	23,548	625
12 bit	1	3876	253212	25	24,181	174
	2	4905	337884	41	24,348	327
	4	10064	589148	77	-	-
13 bit	1	3994	274313	25	22,999	183
	2	5124	366041	41	25,044	318
	4	8437	54949	77	-	-

		7		

III. PERFORMANCE RESULTS

The proposed design is successfully implemented in FPGA. Details on logic cells number and memory usage are given in Table I. These numerical results demonstrate low hardware requirements of the proposed model architecture. Moreover, for this implementation, the depth of the neural network can be further increased without exhausting the resources of this specific hardware.

In this implementation, input images are processed in real time, and the original image is displayed along with the result. Classification of one image requires about 230 thousand clock cycles and we achieve overall processing speed with the large margin over 150 frames/sec.

If performance is insufficient and spare logic cells are available, we can speed up calculations by adding convolutional blocks that perform computations in parallel. Table II shows number of clock cycles required to process one frame using different number of convolutional blocks. Table III shows total resources required for entire FPGA based project implementation for different weight dimensions and different number of convolution blocks. Missing values denotes the cases that Quartus could not synthesize due to the lack of FPGA resources. Source code for both software and hardware implementations as well as video demonstrating the real-time digit classification from the mobile camera video feed is available on GitHub [11].

IV. CONCLUSIONS

In this work we propose a design and an implementation of FPGA-based CNN with fixed-point calculations that allows to achieve the exact performance of the corresponding software implementation on the live handwritten digit recognition problem. Due to the reduced number of parameters we avoid common issues with memory bandwidth. Suggested method can be implemented on a very basic set of FPGAs, but also is scalable for the use on FPGAs with large number of logical cells. Additionally, we demonstrate how existing open datasets can be modified in order to better adapt them for real-life applications. Finally, in order to promote the reproducibility of results, facilitate open-scientific development, and enable collaborative validation, source code, documentation, and all results from this study are made available online. There are many possible ways to improve performance of hardware implementations of neural networks. While we explored and implemented some of them in this work, only relatively shallow neural networks were considered, without additional architectural features, such as skip connections. Implementing even deeper networks with multiple dozens of layers is problematic, since all layer weights would not fit into the FPGA memory and will require the use of the external RAM, which can lead to the decrease in performance. Moreover, due to the large number of layers, error accumulation will increase and will require wider bit range to store fixed-point weight values. In the future, we plan FPGA-based implementation of specialized lightweight neural network architectures that are currently successfully used on mobile devices. This will allow

to use the same hardware implementation for different tasks by fine-tuning the architecture using pre-trained weights.

REFERENCES

[1] Huang, Gao, et al. "Densely connected convolutional networks." CVPR. Vol. 1. No. 2. 2017.

[2] Chen, Liang-Chieh, et al. "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs." IEEE transactions on pattern analysis and machine intelligence 40.4 (2018): 834-848.

[3] A. Shvets, A. Rakhlin, A. A. Kalinin, and V. Iglovikov, "Automatic instrument segmentation in robot-assisted surgery using deep learning," arXiv preprint arXiv:1803.01207, 2018.

[4] Sandler M. et al. "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation" arXiv preprint arXiv:1801.04381, 2018.

[5] J. Qiu et al., "Going deeper with embedded FPGA platform for convolutional neural network," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser.FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[6] C. Zhang et al., "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[7] N. Suda et al., "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in Proceedings of the 2016ACM/SIGDA International Symposium on Field-Programmable GateArrays, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847276

[8] J. Duarte et al., "Fast inference of deep neural networks in fpgas for particle physics," arXiv preprint arXiv:1804.06913, 2018.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86,no. 11, pp. 2278–2324, Nov 1998.

[10] R. A. Solovyev, A. G. Kustov, V. S. Ruhlov, A. N. Schelokov, and D. V. Puzyrkov, "Hardware implementation of a CNN in FPGA based on fixed point calculations," Izvestiya SFedU. Engineering Sciences, July 2017, in Russian.

[11] "Verilog generator of neural net digit detector for FPGA," GitHub. [Online]. Available: https://github.com/ZFTurbo/Verilog-Generator-of-Neural-Net-Digit-Detector-for-FPGA