Towards a Multi-array Architecture for Accelerating Large-scale Matrix Multiplication on FPGAs

Junzhong Shen[1,2], Yuran Qiao[1,2], You Huang[1,2], Mei Wen[1,2] and Chunyuan Zhang[1,2]
[1]College of Computer, [2]National Key Laboratory for Parallel and Distributed Processing
National University of Defense Technology, Changsha, China 410073
Email: shenjunzhong@nudt.edu.cn

***Abstract*—Large-scale floating-point matrix multiplication is a fundamental kernel in many scientific and engineering applications. Most existing work only focus on accelerating matrix multiplication on FPGA by adopting a linear systolic array. This paper towards the extension of this architecture by proposing a scalable and highly configurable multi-array architecture. In addition, we propose a work-stealing scheme to ensure the equality in the workload partition among multiple linear arrays. Furthermore, an analytical model is developed to determine the optimal design parameters. Experiments on a real-life convolutional neural network (CNN) show that we can obtain the optimal extension of the linear array architecture.**

I. Introduction

Large-scale floating-point matrix multiplication is widely used in many complicated computation tasks such as scientific computing [1] and deep learning [2]. Recently, field-programmable gate arrays (FPGAs) have become a particularly attractive option for accelerating large-scale matrix multiplication due to their reconfigurability and abundant logic resources. Previous studies [3–8] have primarily focused on accelerating matrix multiplication on FPGA by using an efficient architecture, i.e. the one-dimensional systolic array. This architecture was demonstrated successfully in matrix multiplication acceleration, which contributes to low bandwidth requirement and good scalability of the accelerator designs.

In this paper, we focus on the extension of the linear array architecture. According to our studies, there exist three approaches to extend the linear array architecture: 1. increasing the length of the linear array; 2. adopting multiple parallel linear arrays; 3. combining approaches 1 and 2. As a result, the design space is significantly expanded, which make it more difficult to reach the optimal solution than previous work. In addition, the computation efficiency of the accelerator is hard to be ensured if we adopt a fixed architecture for various problem sizes. This paper addresses these challenges by proposing a highly configurable and scalable multi-array architecture, which allows users to dynamically change the number of PEs in used as well as the number of parallel PE arrays. In addition, a work-stealing scheme is adopted to guarantee the equality of the workloads partition among the PE arrays. To determine the optimal design option for the proposed architecture, we also propose an analytical model to evaluate the realistic memory bandwidth as well as quantifying data traffic volumes.

The rest of this paper is organized as follows. We review the background in Section II. The proposed multi-array architecture is illustrated in Section III. Section IV presents the performance model. The experimental results are presented in Section V. Section VI concludes this paper.

II. Background

In this paper, we focus on the dense matrix multiplication (MM) algorithm: $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where matrix $\mathbf{A} \in \mathbb{R}^{M \times K}$, $\mathbf{B} \in \mathbb{R}^{K \times N}$ and $\mathbf{C} \in \mathbb{R}^{M \times N}$, respectively. Equation 1 describes the computation pattern of this algorithm:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}. \tag{1}$$

Since it is infeasible to store the entire input and output matrices on the FPGA, abundant researches [4–10] adopt a similar block matrix multiplication algorithm in their designs. Here, we introduce the block matrix multiplication algorithm in Dou [6], which has been proved to be successful in matrix multiplication acceleration. Initially, matrix $\mathbf{A}$ is split into $\lceil M/S_i \rceil$ sub-blocks (namely SA) of size $S_i \times K$ and matrix $\mathbf{B}$ is partitioned into $\lceil N/S_j \rceil$ sub-blocks (namely SB) of size $K \times S_j$. In this way, the result matrix $\mathbf{C}$ can be calculated by performing sub-block matrix multiplications on the SA_i and SB_j, where $i \in [1, \lceil M/S_i \rceil]$ and $j \in [1, \lceil N/S_j \rceil]$. The basic idea of this algorithm is to split the multiplication of SA_i and SB_j into multiple inner-product operations of two vectors, i.e U_k and V_k, where U_k is the k^{th} column of SA_i and V_k is the k^{th} row of SB_j ($k \in [1, K]$). Here we define $C_{i,j}$ as the product of SA_i and SB_j, then $C_{i,j}$ can be calculated by accumulating C_1, C_2,.., and C_K iteratively, shown as

$$C_{i,j} = SA_i \times SB_j = \sum_{k=1}^{K} V_k \times U_k \tag{2}$$

III. Proposed Architecture

Fig. 1 presents an overview of our proposed architecture for acceleration of matrix multiplication. Due to limited amount of on-chip memory on FPGAs, the source data and final results are stored in the off-chip memory (i.e. the DDR). It can be seen that the accelerator is composed of several modules, namely the Memory Access Controller *(MAC)*, the Workload Queue Management (*WQM*), and the Matrices Processing Engine (*MPE*).

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

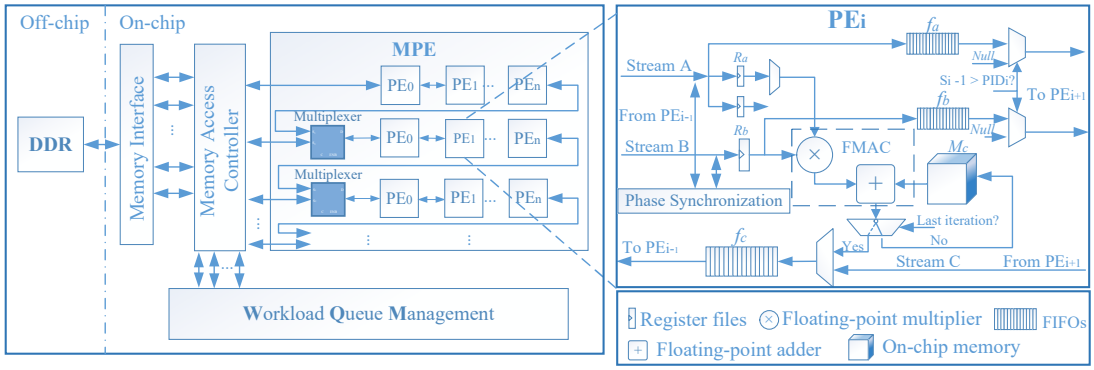


Fig. 1. An overview of our proposed architecture.

A. MPE Design

The MPE is the kernel computation module of our architecture. As shown in Fig. 1, the *MPE* module consists of several linear arrays of PEs, in addition with some multiplexers placed between adjacent PE arrays.

We apply two operation modes in the two adjacent PE arrays, namely the *Independent* mode and the *Cooperation* mode. In the *Independent* mode, the multiplexer between the PE arrays is disabled, meaning that the PE arrays can execute computation tasks independently without any data communication. While in the *Cooperation* mode, the multiplexer between the PE arrays is enabled. As a result, the data paths of the PE arrays are connected by the multiplexer. As shown in Fig. 1, the PE array that placed behind a multiplier can fetch data from the proceeding PE array in this mode. In *Cooperation* mode, the required memory bandwidth of the PE arrays is lower since the PE arrays share the same memory interface when they are connected. In addition, larger block sizes can be supported in the *Cooperation* mode since the number of PEs in the connected array has increased. Note that the multipliers are initialized by the host CPU, meaning that the multi-array architecture is highly configurable. More importantly, our architecture preserves the scalability of the linear array architecture.

The fully pipelined structure of PE is presented in the right part of Fig. 1. It can be seen that the PE consists of two sets of data registers for input data buffering, three First-In-First-Outs (FIFOs) for delivering data between PEs, local memory for temperate data storing, and floating-point multiply-and-accumulate unit ($FMAC$). Different from previous studies, we implement additional control units to support arbitrary block size. In addition, we implement a phase synchronization unit (PSU) to guarantee the correctness of the final results when the block sizes for matrices **A** and **B** are different. By conditionally inserting stalls into the computation pipeline of the PE, the PSU ensures that the k^{th} column of SA and k^{th} row of SB are fetched into each PE simultaneously. The dataflow in each PE consists of three stages:

Prefetch. In this stage, the PE picks up the corresponding element in V_1 (i.e. the first column of SA) based on the PE identifier (PID), then stores the data into the local register R_a. For instance, PE_1 with $PID = 1$ will picks up the second element in V_1.

Compute. In this stage, the k^{th} row of SB (i.e. U_k) and the $(k+1)^{th}$ column of SA (i.e. V_{k+1}) are fetched into the PE simultaneously, where $1 < k \leq K$. The buffered element in R_a is multiplied with all the elements of SB_k in order. Therefore, the data buffered in R_a is reused S_j times. In the meantime, the PE also buffers the corresponding data in V_{k+1} into R_a. Note that we apply double buffering in R_a to overlap buffering data of the next iteration and computation of the current iteration. The products of the multipliers in *FMAC* are then added with the intermediate results generated in the previous iteration, which are stored in the local memory M_c. Finally, the newly sums are written back into the memory M_c. Note that in the last iteration (i.e. $k = K$), the final results are written into FIFO f_c instead of memory M_c.

Write back. In this stage, the PE sends its local results to the proceeding PE or the *MAC* module (only PE_0) from the f_c. As a results, the result data are delivered from the end of each independent PE arrays to the *MAC* module.

B. WQM Design

The *WQM* module is responsible for workloads assignment for the PE arrays. It manages multiple workload queues to buffer the computation tasks for the PE arrays. Note that one workload queue corresponds to one PE array. For the proposed multi-array architecture, the steadiness of an even partition of workloads among PE arrays is the key to achieve better performance. However, it is difficult to ensure that the workloads are always equally partitioned during the whole computation procedures of PE arrays. For example, sometime the PE array which is assigned with less workloads could obtain higher memory bandwidth, which may worsen the inequality of workloads partition. As a result, the system performance would bottlenecked by the PE array with the most workloads. To address this issue, we adopt the work-stealing scheme [11] in the design of the *WQM* module.

The basic idea of the work-stealing scheme is to enable an idle PE array to acquire computation tasks from the overloaded PE array(s). Fig. 2 depicts the working procedure of the work-stealing scheme. Note that a controller (omitted in Fig. 2) is designed to manage the workload delivery among the

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

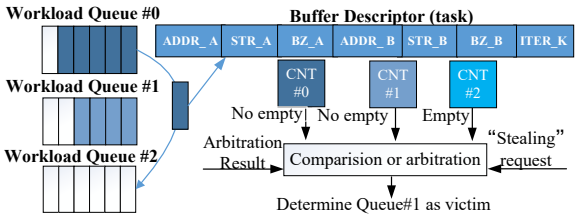


Fig. 2. Illustration of our proposed work-stealing scheme.

workload queues. It can be seen that a counter is implemented to record the number of tasks for each workload queue. Once the controller detects that a workload queue becomes empty, it will immediately steal a task from a non-empty queue, then load it into the empty workload queue. If there exist more than one non-empty queues, the controller will select the workload queue that with the most workloads as target, by comparing the corresponding counters of the workload queues. It is important to note that we implement a round-robin arbiter in the controller to arbitrate multiple concurrent work-stealing requests. The controller repeats the detection and arbitration during the entire computation procedure of the PE arrays.

C. MAC Design

The *MAC* module is responsible for managing data transfer between the external memory and the accelerator. As shown in Fig. 2, the workloads executed by the *MAC* module are organized by a self-defined data structure named *buffer descriptor*. A *buffer descriptor* contains the following parameters: $ADDR$ specifies the memory locations that store the sub-matrices; STR specifies the stride of each memory transfer; BZ specifies the block sizes and $ITER\_K$ specifies the iteration (K).

As mentioned in the above context, elements of matrix **A** are fetched into the PE arrays in column-major order. However, the matrix **A** is stored in row-major order. Therefore, the access of matrix **A** may cause inefficient memory bandwidth utilization. To improve the effective memory bandwidth, we transpose matrix **A** to allow its data to be fetched in row-major order. In this way, the burst transfer mode that favored by the external memory can be used to access both matrices **A** and **B**. As a result, the memory bandwidth for the accelerator is significantly improved, which contributes to performance improvement of the overall system.

IV. Performance Modeling

In this section, we will illustrate how to determine the optimal solution of mapping the block matrix multiplication algorithm onto the multi-array architecture.

Let the bandwidth of the off-chip memory be BW (MB/s), the number of PE in a single PE array be P (when all the multiplexers are disabled), the number of PE arrays work in parallel be N_p, block size of the matrix **A** (on rows) be S_i and block size of the matrix **B** (on columns) be S_j. For **A** of size $M \times K$ and **B** of size $K \times N$, the average number of sub-block matrix multiplications performed by each PE array can be expressed as

$$N_{work} = \lceil \frac{1}{N_p} \times \lceil \frac{M}{S_i} \rceil \times \lceil \frac{N}{S_j} \rceil \rceil. \tag{3}$$

Note that we pad matrices **A** and **B** with zeros if M and N are not integer multiples of S_i and S_j. In addition, the time (in seconds) taken to load a workload (i.e. SA_i and SB_j) and write back the corresponding $C_{i,j}$ can be calculated by

$$T_{work} = \frac{4 \times (S_i \times K + S_j \times K + S_i \times S_j)}{BW}. \tag{4}$$

To simplify the model, we assume that all the workloads are equally partitioned. Therefore, the time taken to transfer data between the external memory and the PE arrays can be expressed as

$$T_{trans} = N_{work} \times T_{work}. \tag{5}$$

According to the data path described in section III, the computation time $T_{compute}$ (in seconds) of a single PE array can be determined as follows:

$$T_{compute} = \frac{N_{work} \times (S_i + max\{S_i, S_j\} \times K + Stage_{fmac})}{F_{acc}}, \tag{6}$$

where $Stage_{fmac}$ denotes the stages of the computation pipeline in each PE, and F_{acc} is the working frequency of the accelerator. Since the memory access and computation process are overlapped in our architecture, it is difficult to directly estimate the execution time of the accelerator. However, the lower bound and upper bound of the execution time T_{total} can be determined by

$$T_{compute} < T_{total} < T_{trans} + T_{compute}. \tag{7}$$

To simplify the discussion on the parameters that affect T_{total}, we assume $S_i = S_j$ for the rest of this paper. It can be inferred that the attainable memory bandwidth BW for each PE array is mainly affected by N_p and S_i, which can be expressed by

$$BW = f(N_p, S_i). \tag{8}$$

This is because S_i determines te burst length of memory access, and N_p affects the conflicts of memory accesses of the PE arrays. From equations 3, 4, 5, 6 and 8, it can be seen that N_p and S_i are the key factors that affect the performance of our accelerator. To reduce the size of design space, we consider the constraints on N_p and S_i. We observe that there exists a relationship between S_i and N_p. For better understanding, we denote $P_m = 4$ as the maximum number of the independent PE arrays (when all the multiplexers are disabled), then the relationship between N_p and S_i can be determined as

$$\begin{cases} N_p \in \{1,2,3,4\}, & if \quad 1 \le S_i \le P \\ N_p \in \{1,2\}, & if \quad P < S_i \le 2P \\ N_p = 1, & if \quad 2P < S_i \le 4P \end{cases}. \tag{9}$$

To this end, the size of design space can be reduced with the assistance of equation 9. Given the fixed problem size and $P_m * P$ (i.e. the total number of PEs), the proposed analytical model can be used to determine the optimal $< S_i, N_p >$ that minimizes the range of T_{total}.

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

TABLE I
POST-SYNTHESIS RESOURCE UTILIZATION.

Resource	DSP48Es	BRAMs	Flip-Flops	LUTs
Utilization	1032	560.50	292016	192493
percentage(%)	28.67	38.13	33.70	44.44

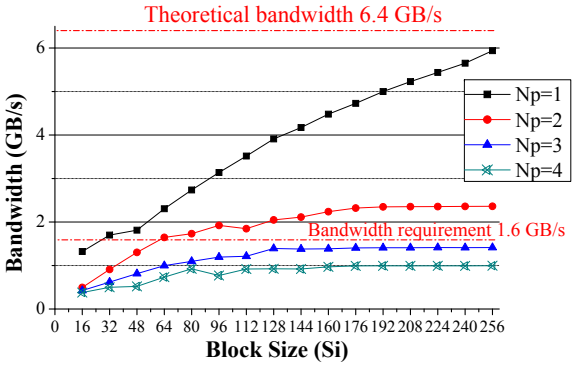
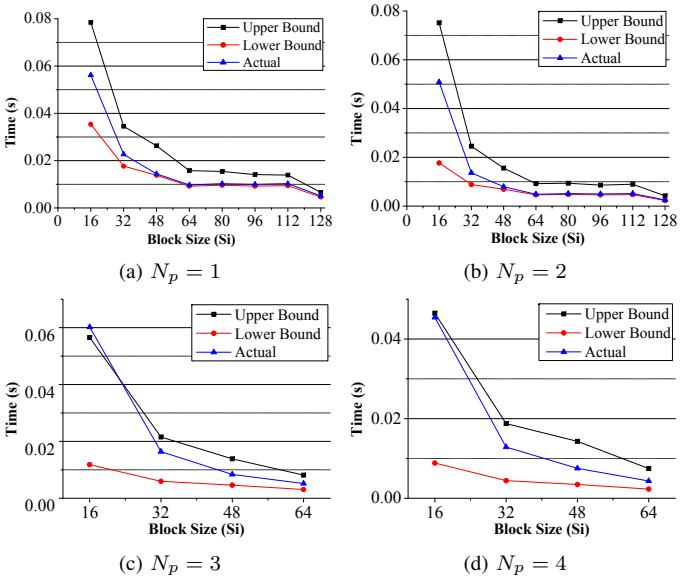


Fig. 3. Effective memory bandwidth.



(a) $N_p = 1$ (b) $N_p = 2$

(c) $N_p = 3$ (d) $N_p = 4$

Fig. 4. Comparison of the estimated execution time and actual execution time for conv-2.

V. EXPERIMENTAL RESULTS

The FPGA platform used in our experiment is the Xilinx VC709 board, which equips with a XC7VX690T FPGA and two DDR3 DRAMs. In addition, all synthesized results are obtained from Xilinx Vivado 2016.4.

In order to quantify function f, we evaluate the average effective memory bandwidth of a PE array in terms of block sizes and number of PE arrays. As shown in Fig. 3, two observations can be found. First, the effective memory bandwidth goes up with the increase of block size. Second, the effective bandwidth declines when we increase the number of PE arrays.

As a case study, we use a real-life CNN model, AlexNet [12], to validate our analytical model. Note that AlexNet comprises of five convolutional layers and three fully connected layers, and the computation patterns of these layers can be converted to matrix multiplication [13]. Note that we are

TABLE II
OPTIMAL (N_p, S_i) OF ALL LAYERS IN ALEXNET.

Layers	$M * K * N$	Optimal (N_p, S_i)	Performance (GFLOPS)		
			Optimal	$N_p = 4$	$N_p = 1$
Conv-1	96*363*3025	(2,128)	59.7	57.1	49.2
Conv-2	128*1200*729	(2,128)	87.8	70.3	61.4
Conv-3	384*2304*169	(2, 96)	64.9	62.9	57.4
Conv-4	192*1728*169	(2, 96)	64.1	54.8	51.2
Conv-5	128*1728*169	(2,128)	62.9	44.9	43.9
fc-6	128*9216*4096	(2,128)	100.9	79.3	70.7
fc-7	128*4096*4096	(2,128)	99.3	78.1	69.5
fc-8	128*4096*1000	(2,128)	96.9	83.6	67.8

mainly focus on determining the optimal design parameters under fixed P_m and P, therefore we do not make full use of the resource of the FPGA to pursuit maximum performance. In our experiment, we set $P_m = 4$ and $P = 64$. After post-synthesis, a maximum frequency of 200 MHz (F_{acc}) is achieved. Table I summarized the resource utilization of the overall system. It can be seen that the overall resource utilization is below 50%, which contributes to the high frequency of our accelerator.

We give a detailed comparison between the predicted and actual execution time for conv-2 in Fig. 4. It can be seen that the predicted lower bound of execution time closely follows the actual measurement when the memory requirement of each PE array is satisfied. However, when the memory bandwidth requirement is unsatisfied, the actual time of each PE array becomes more close to the upper bound of predicted execution time. In addition, it can also be found that using multiple PE arrays does not ensure the optimal performance. For example, the case of $(N_p, S_i) = (1, 32)$ achieves lower execution time than the case of $(N_p, S_i) = (2, 16)$. The main reason is that both of the cases are memory-bound (< 1.6 GB/s), and the case of $(N_p, S_i) = (1, 32)$ can reach higher memory bandwidth (it can be confirmed by Fig. 3), which contributes to its higher performance.

The optimal $< N_p, S_i >$ of all the layers in AlexNet is given in Table II. It can be seen that when compared to other extension approaches, i.e extending the number of PEs only ($N_p = 1, P = 256$) and extending the number of PE arrays only ($N_p = 4, P = 64$), our accelerator that implemented with the optimal $< N_p, S_i >$ achieves the highest performance for all layers. In addition, our accelerator achieves 100.9 GFLOPS for fc-6, which demonstrates that our multi-array architecture can achieve high ratio (i.e. up to 98.6%) of sustained performance to theoretical peak performance (which is denoted by $2 \times F_{acc} \times P_m \times P$ [6]).

VI. CONCLUSION

In this paper, we focus on the architecture extension of the linear array architecture for matrix multiplication on FPGA, by proposing a highly configurable and scalable multi-array architecture. We employ a work-stealing scheme to realize workload balancing among PE arrays. An efficient analytical model is developed to determine the optimal design options for the architecture extension. Experimental results show that our optimal extension of the linear array architecture can reach the highest performance and computation efficiency.

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

REFERENCES

[1] G. Govindu, R. Scrofano, and V. K. Prasanna, "A library of parameterizable floating-point cores for fpgas and their application to scientific computing," in *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms*, 2005, pp. 137–148.

[2] G. Lei, Y. Dou, R. Li, and F. Xia, "An fpga implementation for solving the large single-source-shortest-path problem," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, no. 5, pp. 473–477, 2016.

[3] J.-w. Jang, S. Choi, and V. Prasanna, "Area and time efficient implementations of matrix multiplication on FPGAs," in *Proc. IEEE International Conference on Field-Programmable Technology, 2002. (FPT'02)*, Dec. 2002, pp. 93–100.

[4] L. Zhuo and V. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs," in *Proc. 18th International Parallel and Distributed Processing Symposium*, Apr. 2004.

[5] J.-w. Jang, S. Choi, and V. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Trans. VLSI Syst.*, vol. 13, no. 11, pp. 1305–1319, 2005.

[6] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proc. ACM International Symposium on Field-programmable Gate Arrays(FPGA'05)*, New York, NY, USA, 2005, pp. 86–95.

[7] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, "FPGA based high performance double-precision matrix multiplication," in *Proc. International Conference on VLSI Design (VLSI'09)*, 2009, pp. 341–346.

[8] Z. Jovanovic and V. Milutinovic, "FPGA accelerator for floating-point matrix multiplication," *IET Computers & Digital Techniques*, vol. 6, no. 4, pp. 249–256, 2012.

[9] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 4, pp. 433–448, 2007.

[10] G. Wu, Y. Dou, and M. Wang, "High performance and memory efficient implementation of matrix multiplication on FPGAs," in *IEEE International Conference on Field-Programmable Technology (FPT'2010)*, 2010, pp. 134–137.

[11] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[13] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. International Conference on Artifical Neural Networks*, 2014, pp. 281–290.

978-1-5386-4881-0/18/$31.00 ©2018 IEEE