Hardware/software co-design for Neural Network trained by improved Particle Swarm Optimization

by

DANG TUAN LINH

Student ID Number: 1188001

A dissertation submitted to the
Engineering Course, Department of Engineering,
Graduate School of Engineering,
Kochi University of Technology,
Kochi, Japan

in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Assessment Committee:	
Supervisor:	Yukinobu Hoshino
Co-Supervisor:	Masayoshi Tachibana
Co-Supervisor:	Yukio Mitsuyama
	Shinichi Yoshida
	Kiminori Matsuzaki

September 2017

Abstract

Nowadays, using the particle swarm optimization (PSO) algorithms to train neural networks (NN) has become an attractive research. A famous method used to train the NNs is the back-propagation (BP) algorithm. Recently, previous studies have shown that the NN trained by PSO algorithms (NN-PSO) obtained higher recognition rates and lower learning errors when compared to the NN trained by the conventional BP algorithm. However, the standard PSO (SPSO) algorithm may stick to a local minimum in the training process. In this situation, the SPSO algorithm is stopped, leading to a high learning error rate or a low recognition rate of the trained NN. So far few studies have tried to overcome this problem by adding improving the standard PSO algorithms. Normally in previous studies, the improved versions of the PSO required to add more compute-intensive tasks that often increase computational burden the SPSO algorithm. Typically, previous researchers have implemented the NN trained by PSO algorithms only in hardware or only in software. Normally, the hardware implementation of the NN-PSO in these studies has been tested in a simulation using ModelSim program with the SPSO algorithm but it has not been tested in the real classification tasks. The FPGA-based NN was commonly investigated with a function-approximation, and the researchers just focused on reducing learning errors in the training phase.

To deal with the issues described above, this research has three objectives. The first objective is to introduce the improved particle swarm optimization (PSO) algorithms which overcome the premature convergence of the standard PSO (SPSO) algorithm without adding many computational tasks or compute-intensive functions to the SPSO algorithm.

The second objective is to propose co-design architectures between hardware and software for the NN-PSO. Compared to the hardware-only approach, the proposed co-design approach not only maintains the testing speed but also reduces the required FPGA resources concerning the logic elements and the memory bits previously reserved for the hardware implementation of the PSO algorithms. Compared to the software-only approach, the proposed co-design approach preserves the flexibility during in the training phase while obtaining the higher operating speed in the testing phase. The flexibility of the co-design is an easiness to modify the PSO parameters or change the PSO algorithms without redesigning or rebuilding the FPGA part.

The third objective is to investigate the performances concerning the learning errors and recognition rates in classification tasks of the NN trained by PSO algorithms implemented in a real FPGA device with the proposed co-design architectures comparing to the NN trained by the standard and dissipative PSO (SPSO, DPSO) presented in

previous studies. The DPSO algorithm is chosen because it keeps the particles out of the local minimum without adding many tasks to the PSO algorithm.

Concerning the first objective, three different PSO algorithms which only change the velocity update function of the SPSO algorithm were proposed called the wPSOd_CV, the PSOseed, and the PSOseed2 algorithm.The first PSO algorithm introduced in this chapter is the wPSOd_CV algorithm that has two main components called the velocity control and the weight control. The velocity control has a jump phase to increase the velocity of the particle so that particle can move to another searching area to avoid the premature convergence. The weight control is used to balance between the exploitation task and the exploration task of the wPSOd_CV algorithm. The wPSOd_CV algorithm has some drawbacks. Even modifying only the velocity update function, this algorithm adds two operations, increasing the compute-intensive tasks of the SPSO algorithm. In addition, the jumping phase in the velocity control mechanism always has very big jumps. If the searching area has many solutions, the wPSOd_CV has the possibility to meet other solutions and jump to other searching space before the before meeting the best solution. This issue leads to a possibility to ignore the best solution or require more iterations to reach the best solution. To overcome these drawbacks, the PSOseed algorithm is proposed. Avoiding jumping phases, for each particle the PSOseed algorithm uses a new variable called the seed position that is randomly generated in the initial phase of the algorithm. In each iteration, each particle is attracted and pulled to the position of its seed. The seed mechanism could reduce the possibility that the particle falls in a local minimum. Compared to the wPSOd_CV algorithm, the PSOseed algorithm does not use any division operator and use fewer multiplication operators. A limitation is that the PSOseed algorithm highly depends on the generated seeds. The performance of the PSOseed significantly is reduced if the seeds are poorly generated. This thesis also introduces the PSOseed2 algorithm to solve the issue concerning the seed positions of the PSOseed algorithm by proposing a seed control mechanism. The operation of the PSOseed2 algorithm is similar to the PSOseed algorithm. However, in each iteration, all seed positions will be reseeded if the particles in the PSOseed2 algorithm cannot find a better position when the fitness value of the algorithm does not change.

Regarding the second objective, the NN is implemented in hardware to maintain the testing speed of the hardware-based program while the PSO is implemented in software to keep the flexibility of the software-based program. Furthermore, the software implementation of the PSO algorithms can also reduce the FPGA resources previously reserved for the PSO algorithms. Three different co-design architectures were presented based on the proposed partitioning methodology. The first architecture uses the NIOS II processor. The second architecture replace the NIOS II by the ARM processor. This architecture has also other improvements such as the using of direct-memory access or

random-access memory. The third architecture combines the ARM approach in the second architecture with the FPGA-based PCA to reduce the required resources but still maintain the accuracy of the NN trained by PSO algorithms concerning the recognition rates and the learning errors. Experimental results confirmed both the speed advantage of the proposed co-design approach when compared with the software-only approach and the resource advantage of the proposed co-design approach when compared with the hardware-only approach. Results also showed that the hardware-based PCA can reduce the required resources of the program while keeping high recognition rates.

In terms of the third objective, the NN was trained by three proposed PSO algorithms (wPSOd_CV, PSOseed, PSOseed2) and two PSO algorithm presented in previous studies (SPSO, DPSO) in all three proposed co-design architectures. Experimental results demonstrated that the NN trained by the proposed algorithms, especially the PSOseed2 algorithm, had the highest performances concerning the recognition rates and the learning errors in all three architectures.

Acknowledgements

First of all, I would like to thank Professor Yukinobu Hoshino for his willingness to coach me, his promptness, the eye-opening discussions we had and his valuable comments on the progress of my research. In the same light, I thank my co-Supervisors, Professor Masayoshi Tachibana and Professor Yukio Mitsuyama for their excellent observations, ideas, and creativity that inspired my dissertation. I would like to thank all the professors at the Kochi University of Technology who were responsible for the Ph.D. courses that I have attended during these years. I have acquired considerable knowledge through their presentations and interactive scientific discussions. I would like to present all my gratitude to Dr. Cao Thang at the University of Tokyo for his constructive advice and interesting ideas. Thank you also to Mr. Keita Mitani, Mr. Yuki Shinomya and other friends in Hoshino Laboratory at Kochi University of Technology for creating the best working conditions. I would also like to thank all Vietnamese friends of Kochi University of Technology for their friendship and support over the past three years. To all of them, I would like to express my sincere greetings. Additionally, I would like to thank all office staffs of International Relations Division, Kochi University of Technology. Finally, countless thanks and acknowledgements go to my family, especially, my wife and my daughter, for giving me their caring support and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

A neural network (NN) has emerged as an attractive target for research. Several companies have also used the NN in the practical applications [1, 2]. The NN is invented for representing a human brain concerning two aspects. The first aspect relates to a knowledge acquisition. Both the NN and the human brain have a learning phase to acquire the knowledge from an environment. The second aspect is the knowledge presentation. The acquired knowledge after the training phase is stored in the form of the weights and biases of each node of both the NN and the human brain [3, 4].

Before the testing phase, the NN needs to be trained. During the training phase, the NN tried to find a set of parameters concerning the weights and biases which minimize the learning error. A famous method used in the training of the NN is the back-propagation (BP) algorithm. However, in previous studies have mentioned that the NN trained by the particle swarm optimization (PSO) algorithm had obtained higher recognition rates and lower learning errors when compared with the NN trained by the BP algorithm [5–7]. The PSO algorithm was introduced to simulate the social behavior such as a bird flocking or a fish schooling. At any given time, the movement of each particle in the swarm, for example, a bird or a fish, is always based on the knowledge of the best personal position of this particle and the best global position of the swarm. The best global position is evaluated from the best personal position of all particles [8–10].

In the training phase of the NN using PSO algorithm, the position of each particle is an encoding vector that consists of all weights and biases of the NN. Each vector is a potential solution for the weights and biases of the NN. In this situation, if the NN has N_D weights and biases, the PSO particle is the N_D-dimensional vector. If the PSO training uses P particles, P encoding vectors will be employed. During the training phase, the movements of these P vectors are based on the PSO algorithm. In each position of the encoding vector, the learning error of the NN will be calculated using a fitness function. The goal of the PSO training is to find a position of the encoding vector which has the minimization of the learning error [5, 11].

In previous studies, the software implementation of the NN trained by PSO algorithm (NN-PSO) is already investigated [5, 11]. Several practical problems have been solved using the software-based NN-PSO [7, 12–15].

However, the standard PSO (SPSO) algorithm may stick to a local minimum during the training of the NN, and the training phase will be stopped. In this case, the NN cannot be trained. Thus, its learning error will be high, and its recognition rate will be low. To overcome this premature convergence of the SPSO algorithm, several different approaches have been introduced. The attractive and repulsive PSO (ARPSO) adds the repulsive phase. The diversity of the swarm calculated in each iteration of the swarm to determine whether the ARPSO operates in the attractive phase (the SPSO algorithm) or the repulsive phase [16]. The PSO with spatial particle extension (SEPSO) that adds the radius r to detect the collisions of all particles in the swarm. If the collisions occur, the particles will be bounced backward to prevent the situation when all particles stick to the local minimum [17]. The opposition-based PSO calculates the opposite particle of each particle in each iteration. If the opposite particle has a better fitness value than the original particle, this opposite particle will replace the original particle [18]. Other researchers use the multi-swarm strategy [19] or use not only the information of a particle but also the information of its neighborhood to calculate the new position of this particle [20]. Other solutions are to use the mutation operations such as the Cauchy mutation [21, 22], the Gaussian mutation [23, 24], or to combine the SPSO algorithm with the genetic algorithm [25]. The improved versions of the PSO in previous studies normally add more functions, more compute-intensive tasks to the SPSO algorithm. It is beneficial to investigate an improved version of the PSO algorithm to keep the particle out of the local minimum which does not add many functions or tasks to the SPSO algorithm.

Currently, a field-programmable gate array (FPGA) has also attracted many researchers. Compared with the software-based program, the FPGA-based program could obtain a higher operating speed because of the parallelism [26–28]. Compared with the graphics processing unit (GPU)-based program, the FPGA-based program has a lower power consumption [28–30]. In addition, the FPGA-based program is also portable because it can be implemented in a single FPGA chip and can operate without the need of having a processor. Several studies related hardware implementation of the NN [31], the SPSO [27], or even the NN-PSO have been published [32–34].

However, the hardware implementation of the NN-PSO was only tested in a simulation using ModelSim program, with the SPSO algorithm, or was not tested in the classification tasks. In these studies, the FPGA-based NN normally was used to investigate the function approximation, or the researchers only focus on the training phase of the NN-PSO by showing only the reduction of the learning errors [32–34]. It needs to investigate the hardware implementation of the NN-PSO in classification jobs using a real FPGA device.

In addition, the previous papers normally focus only on the hardware-only architecture [32–34]. In this situation, all components of the program were implemented in hardware. This approach required many FPGA resources regarding the logic elements and the memory bits. On the other hand, the FPGA device has the resource constraints because the FPGA resources are very expensive [35]. In addition, the hardware-only program is not flexible as the software-based program. It needs to rebuild and recompile the program each time we change the parameters [36]. Therefore, it is necessary to have a co-design architecture which can keep both the speed advantage of the FPGA-based program and the flexibility of the software-based program. This approach can also reduce the required FPGA resources previously reserved for the components which are already moved to the software-side.

1.2 Research objectives

To deal with the issues described above, this research has three main objectives. The first objective is to introduce the improved PSO algorithms which overcome the premature convergence of the SPSO algorithm without adding many computational tasks or compute-intensive functions to the SPSO algorithm.

The second objective is to propose co-design architectures between hardware and software for the NN-PSO. Compared to the hardware-only approach, the proposed co-design approach not only maintains the testing speed but also reduces the required FPGA resources concerning the logic elements and the memory bits previously reserved for the

hardware implementation of the PSO algorithms. Compared to the software-only approach, the proposed co-design approach preserves the flexibility during the training phase while obtaining the higher operating speed in the testing phase. The flexibility of the co-design is an easiness to modify the PSO parameters or change the PSO algorithms without redesigning or rebuilding the FPGA part.
The third objective is to investigate the performances concerning the learning errors and recognition rates in classification tasks of the NN trained by PSO algorithms implemented in a real FPGA device with the proposed co-design architectures comparing to the NN trained by the standard and the dissipative PSO [37] (SPSO, DPSO) presented in previous studies. The DPSO algorithm is chosen because it keeps the particles out of the local minimum without adding many tasks to the PSO algorithm.

1.3 Structure of the dissertation

This dissertation is divided into eight chapters, each of which deals with different accounts. The content of the dissertation is as follows.

- Chapter 1 briefly presents the overview and objectives of this research.
- Chapter 2 presents related work concerning the NN, the SPSO algorithm, and the DPSO algorithm. It also details how the NN can be trained using the PSO algorithms.
- Chapter 3 focuses on the premature convergence issue of the SPSO algorithm by proposing three improved versions of the SPSO algorithm called wPSOd_CV, PSOseed, and PSOseed2.
- Chapter 4 describes the proposed partitioning methodology between hardware and software for the NN-PSO. The operations on the software side of the proposed partitioning methodology are also presented in this chapter.
- Chapter 5 focuses on the first co-design architecture based on the partitioning methodology presented in chapter 4 which uses the NIOS II processor. This chapter also discusses the performances of the NN trained by proposed PSO algorithms in this architecture.
- Chapter 6 deals with the performances of the NN trained by proposed PSO algorithms in the second co-design architecture that uses the ARM processor. The

second co-design architecture is introduced to overcome the drawbacks of the first architecture using the NIOS II processor presented in chapter 5.

- Chapter 7 presents a method to improve performances of the NN trained by proposed PSO algorithms in the proposed co-design architectures. To reduce the required resources while maintaining the high recognition rate, this chapter proposes the third co-design architecture which is a combination between the co-design architecture using ARM processor introduced in chapter 6 and the FPGA-based principal component analysis. The performances of the NN trained by the proposed PSO algorithm in the third co-design architecture are also discussed in this chapter.
- Chapter 8 concludes all chapters and discusses the future directions for continuing the research.

References

[1] Google uses neural networks to translate without transcribing - new scientist. `https://www.newscientist.com`. (Accessed on 06/05/2017).

[2] Swiftkey debuts world's first smartphone keyboard powered by neural networks - swiftkey blog. `https://blog.swiftkey.com`. (Accessed on 06/05/2017).

[3] S. Haykin. *Neural networks and learning machines.* Pearson Education, Upper Saddle River, NJ, third edition, 2009.

[4] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[5] V. G. Gudise and G. K. Venayagamoorthy. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium*, pages 110–117, Apr 2003.

[6] I. Vilovic, N. Burum, and D. Milic. Using particle swarm optimization in training neural network for indoor field strength prediction. In *2009 International Symposium ELMAR*, pages 275–278, Sep 2009.

[7] Z. A. Bashir and M. E. El-Hawary. Applying wavelets to short-term load forecasting using pso-based neural networks. *IEEE Transactions on Power Systems*, 24(1):20–27, Feb 2009.

[8] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, Nov 1995.

[9] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, 2007.

[10] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Oct 1995.

[11] R. Mendes, P. Cortez, M. Rocha, and J. Neves. Particle swarms for feedforward neural network training. In *Proceedings of the 2002 International Joint Conference on Neural Networks*, volume 2, pages 1895–1899, 2002.

[12] N. Yançin, G. Tezel, and C. Karakuzu. Epilepsy diagnosis using artificial neural network learned by pso. *Turkish Journal of Electrical Engineering & Computer Sciences*, 23:421–432, 2015.

[13] P. Khoury and D. Gorse. Investing in emerging markets using neural networks and particle swarm optimisation. In *Proceedings of the 2015 International Joint Conference on Neural Networks*, pages 1–7, July 2015.

[14] K.W. Chau. Application of a pso-based neural network in analysis of outcomes of construction claims. *Automation in Construction*, 16(5):642–646, 2007. ISSN 0926-5805.

[15] W. Z. Lu, H. Y. Fan, A. Y. T. Leung, and J. C. K. Wong. Analysis of pollutant levels in central hong kong applying neural network method with particle swarm optimization. *Environmental Monitoring and Assessment*, 79(3):217–230, 2002.

[16] J. Riget and J.S. Vesterstrm. A diversity-guided particle swarm optimizer - the arpso. Technical report, 2002.

[17] T. Krink, J. S. Vesterstrom, and J. Riget. Particle swarm optimisation with spatial particle extension. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1474–1479, 2002.

[18] H. Wang, H. Li, Y. Liu, C. Li, and S. Zeng. Opposition-based particle swarm algorithm with cauchy mutation. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation*, pages 4750–4756, Sep 2007.

[19] J. Jie, J. Zeng, C. Han, and Q. Wang. Knowledge-based cooperative particle swarm optimization. *Applied Mathematics and Computation*, 205(2):861 – 873, 2008.

[20] R. Mendes, J. Kennedy, and J. Neves. The fully informed particle swarm: simpler, maybe better. *IEEE Transactions on Evolutionary Computation*, 8(3):204–210, Jun 2004.

[21] H. Wang, C. Li, Y. Liu, and S. Zeng. A hybrid particle swarm algorithm with cauchy mutation. In *Proceedings of the 2007 IEEE Swarm Intelligence Symposium*, pages 356–360, Apr 2007.

[22] A. Stacey, M. Jancic, and I. Grundy. Particle swarm optimization with mutation. In *Proceedings of the 2003 Congress on Evolutionary Computation*, volume 2, pages 1425–1430, Dec 2003.

[23] N. Higashi and H. Iba. Particle swarm optimization with gaussian mutation. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium*, pages 72–79, Apr 2003.

[24] Q. Wu. Power load forecasts based on hybrid pso with gaussian and adaptive mutation and wv-svm. *Expert Systems with Applications*, 37(1):194 – 201, 2010. ISSN 0957-4174.

[25] B. Yang, Y. Chen, and Z. Zhao. A hybrid evolutionary algorithm by combination of pso and ga for unconstrained and constrained optimization problems. In *Proceedings of the 2007 IEEE International Conference on Control and Automation*, pages 166–170, May 2007.

[26] E. Monmasson and M. N. Cirstea. Fpga design methodology for industrial control systems - a review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, Aug 2007.

[27] Y. Maeda and N. Matsushita. Simultaneous perturbation particle swarm optimization using fpga. In *Proceedings of the 2007 International Joint Conference on Neural Networks*, pages 2695–2700, Aug 2007.

[28] J. A. G. Pulido, M. A. V. Rodriguez, J. M. S. Perez, S. P. Mendes, and V. Carreira. Accelerating floating-point fitness functions in evolutionary algorithms: a fpga-cpu-gpu performance comparison. *Genetic Programming and Evolvable Machines*, 12 (4):403–427, 2011.

[29] D. Zou, Y. Dou, and F. Xia. Optimization schemes and performance evaluation of smithwaterman algorithm on cpu, gpu and fpga. *Concurrency and Computation: Practice and Experience*, 24(14):1625–1644, 2012.

[30] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian. High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP. *International Journal of Reconfigurable Computing*, 2012, 2012.

[31] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, May 2007.

[32] Mehmet Ali Cavuslu, Cihan Karakuzu, and Fuat Karakaya. Neural identification of dynamic systems on fpga with improved pso learning. *Applied Soft Computing*, 12(9):2707 – 2718, 2012.

[33] A. Bezborah. A hardware architecture for training of artificial neural networks using particle swarm optimization. In *Proceedings of the 2012 Third International Conference on Intelligent Systems Modelling and Simulation*, pages 67–70, Feb 2012.

[34] C. J. Lin and H. M. Tsai. Fpga implementation of a wavelet neural network with particle swarm optimization learning. *Mathematical and Computer Modelling*, 47 (910):982 – 996, 2008.

[35] D. G. Bailey. *Design for Embedded Image Processing on FPGAs*. Wiley Publishing, 1st edition, 2011.

[36] S. A. Li, C. C. Hsu, C. C. Wong, and C. J. Yu. Hardware/software co-design for particle swarm optimization algorithm. *Information Sciences*, 181(20):4582–4596, 2011.

[37] X. F. Xie, W. J. Zhang, and Z. L. Yang. Dissipative particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1456–1461, 2002.

Chapter 2

Background

Chapter 2 presents related work concerning the operation of the NN, the SPSO, and the DPSO algorithms. It also details how the NN can be trained using the PSO algorithms.

2.1 Neural network

In previous studies, an artificial NN was proposed to represent a human brain. Similar to the operation of the human brain, the NN acquires the knowledge from the environment during the training phase. This collected knowledge is stored in the form of the weights and biases of each node in the NN. In each iteration of the training phase, each node in the NN tries to obtain a set of weights and biases which minimizes the learning error of the NN.

The resources of the hardware-based program concerning the logic elements and the memory bits are very expensive. On the other hand, the loops and the cycles in the hardware implementation require many resources of the FPGA device. To reduce the required resources of the FPGA-based program, this research focuses only on the feedforward NN because the data in the feedforward NN only move in one direction without loops or cycles [1, 2].

Typically, the feedforward NN has three different types of layers called the input layer, the hidden layer, and the output layer, respectively. The NN has only one input layer and one output layer, but it may have more than one hidden layers. Each layer consists of several nodes that connect to other nodes in the next and previous layers. The inputs of a node in one layer are multiplied by the weights in this layer. Then, this result becomes the input of nodes in the next layer through an activation function [1, 2].

The operation of one node in the NN is illustrated in Fig. 2.1. The calculated results x before the activation function is expressed in Eq. (2.1). This study uses the Sigmoid function as the activation function. Therefore, the output y is calculated according to Eq. (2.2). This output y is used as the input of the nodes in the next layer of the NN [1, 2].

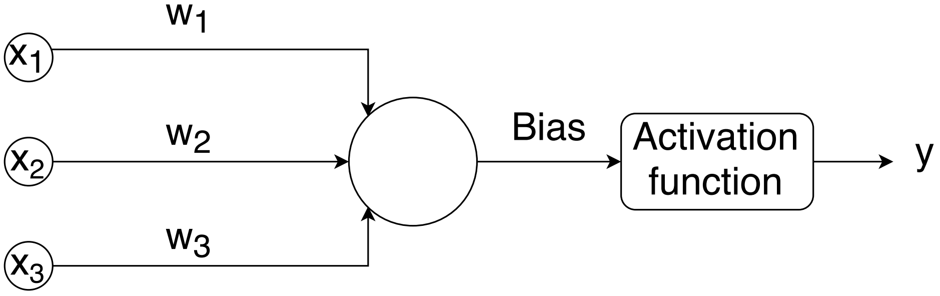


Figure 2.1: Operation of one node in the NN

$$x = Bias + x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3 \tag{2.1}$$

where $Bias$ is the bias value, x_1, x_2, and x_3 are the inputs, $Weight1$ and $Weight2$ are the weights, x is the input for the activation function.

$$y = \frac{1}{1 + e^{-x}} \tag{2.2}$$

where y is the output data after the activation function.

The NN is shown in Fig. 2.2. This NN has N_I nodes in the input layer, N_O nodes in the output layer, N_H nodes in one hidden layer, and N_L hidden layers.

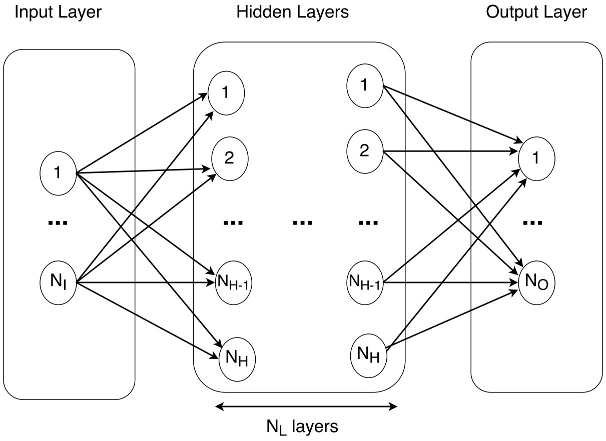


Figure 2.2: Architecture of the NN

Fig. 2.3 shows the weights and biases in each layer of the NN. This NN has $N_I \times N_H$ weights and N_H biases in the connections between the input layer and the first layer of the hidden layers. The connections between the last layer of the hidden layers and the output layer have $N_H \times N_O$ weights and N_O biases. The connections inside the hidden layers have $(N_H + 1) \times N_H \times (N_L - 1)$ weights and $N_H \times (N_L - 1)$ biases.

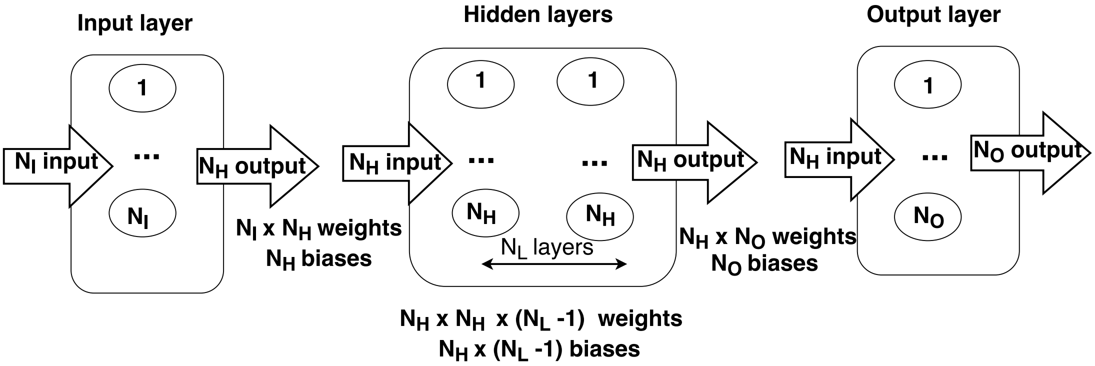


Figure 2.3: Weights and biases of the NN

In this study, two types of the NN are used. The first NN type has one hidden layer ($N_L = 1$) and the second type of the NN has two hidden layers ($N_L = 2$). The N_{D1} number of weights and biases for one hidden layer NN is calculated in Eq. (2.3).

$$N_{D1} = (N_I + 1) \times N_H + (N_H + 1) \times N_O \tag{2.3}$$

The calculation of N_{D2} weights and biases for the two hidden layers NN is illustrated in Eq. (2.4).

$$N_{D2} = (N_I + 1) \times N_H + (N_H + 1) \times N_H + (N_H + 1) \times N_O \tag{2.4}$$

2.2 Particle swarm optimization algorithms in previous studies

2.2.1 Standard particle swarm optimization algorithm

The SPSO algorithm is a well-known method for training of the NN. The idea behind the SPSO algorithm comes from the social behaviors such as a bird flocking or a fish schooling. At any given time t, each particle p in the swarm tends to move to the best personal position found by this particle and the best global position of the swarm. The best global position is evaluated from the best personal position of all particles in the swarm [3, 4]. The operation of the SPSO algorithm can be described as follows.

1. Initialize the random value for all particles in the swarm at time t . For example, particle p has position $\overrightarrow{x_p}(t)$, velocity $\overrightarrow{v_p}(t)$, best personal position $\overrightarrow{xPbest_p}(t)$, best global position $\overrightarrow{xGbest}(t)$. The fitness value for the best personal position $\overrightarrow{xPbest_p}(t)$ is $Pbest_p(t)$. In a similar way, the fitness value for the best global position $\overrightarrow{xGbest}(t)$ is $Gbest(t)$.

2. Calculate the new velocity at time $(t+1)$ according to Eq. (2.5).

$$\begin{aligned}\overrightarrow{v_p}(t+1) = w \times \overrightarrow{v_p}(t) + c_1 \times r_1 \times (\overrightarrow{xPbest_p}(t) - \overrightarrow{x_p}(t)) \\ +c_2 \times r_2 \times (\overrightarrow{xGbest}(t) - \overrightarrow{x_p}(t))\end{aligned} \tag{2.5}$$

   where w is the inertia weight, r_1 and r_2 are the random numbers, c_1 and c_2 are the coefficients, $\overrightarrow{xPbest_p}$ is the best personal position found by particle p, $\overrightarrow{xGbest}$ is the best position found by any particle in the swarm

3. Evaluate the new position at time $(t+1)$ based on Eq. (2.6).

$$\overrightarrow{x_p}(t+1) = \overrightarrow{x_p}(t) + \overrightarrow{v_p}(t+1) \tag{2.6}$$

4. Estimate the new fitness values at time $(t+1)$ based on Eqs. (2.7) and (2.8). The positions corresponding to these new fitness values are also discovered. As presented, the best personal position $\overrightarrow{xPbest_p}(t+1)$ is the position that obtains the fitness value $Pbest_p(t+1)$, and the best global position $\overrightarrow{xGbest}(t+1)$ is the position that achieves the fitness value $Gbest(t+1)$.

$$Pbest_p(t+1) = \begin{cases} f(\overrightarrow{x_p}(t+1)) \;\; if \;\; f(\overrightarrow{x_p}(t+1)) < Pbest_p(t) \\ Pbest_p(t) \;\; if \;\; f(\overrightarrow{x_p}(t+1)) \geqslant Pbest_p(t) \end{cases} \tag{2.7}$$

   where $f(.)$ is a fitness function.

$$Gbest(t+1) = \operatorname*{argmin}_p Pbest_p(t+1) \tag{2.8}$$

5. The stopping criterion is checked. In the PSO algorithm, the stopping criterion could be a number of iterations or a threshold of the global fitness value *Gbest*. If the stopping condition is satisfied, the PSO algorithm is stopped. Otherwise, the operation of the SPSO algorithm returns to step 2 to continue a new iteration at time $(t + 2)$.

2.2.2 Dissipative PSO algorithm

The SPSO could be trapped by a local minimum. In this situation, the PSO algorithm will be stopped. If the SPSO sticks to the local minimum during the training of the NN. The NN cannot be trained. The recognition rate of the testing phase will be low, and the learning error of the training phase will be high. A well-known algorithm was created to overcome the premature convergence called the dissipative PSO algorithm [5]. This is the simple variation of the SPSO algorithm which adds Eqs. (2.9) and (2.10) after calculating the new velocity $\overrightarrow{v_p}(t+1)$ and the new position $\overrightarrow{x_p}(t+1)$.

$$If(rand() < c_v)\ \ Then\ \ \overrightarrow{v_p}(t+1) = rand() \times \overrightarrow{v_{max}} \tag{2.9}$$

$$If(rand() < c_l)\ \ Then\ \ \overrightarrow{x_p}(t+1) = Rand(lo, up) \tag{2.10}$$

c_v and c_l are numbers in the range [0, 1], $Rand(lo, up)$ is the random number between [lo, up], $rand()$ is the random number in the range [0, 1], and $\overrightarrow{v_{max}}$ is the maximum velocity.

2.3 NN trained by the PSO algorithms

For the training of the NN by PSO algorithms, one important aspect is the encoding strategy. In this approach, the weights and the biases of the NN are encoded in the particle of the PSO algorithm.

For encoding, the position $\overrightarrow{x_p}$ of particle p in the swarm is considered as a vector which represents all weights and biases of the NN. If the PSO training has P particles, P vectors will be used in the training phase. In this situation, the number of weights and biases in the NN equals to the number of dimensions of one particle vector. Each element of the vector corresponds to one parameter of the NN. When the elements of PSO vector are changed during the PSO algorithm, the corresponding parameters of the NN are also updated [6, 7].

As presented in the previous section, two different type NNs are investigated in this research. The first NN has one hidden layer and N_{D1} parameters. The calculation of N_{D1} parameters is shown in Eq. (2.3). The second NN is two-hidden-layer NN which has N_{D2} weights and biases which are described in Eq. (2.4). Therefore, there are two different encoding vectors. The first vector is N_{D1}-dimensional vector and the second vector has N_{D2} dimensions. These encoding vectors are illustrated in Fig. 2.4.

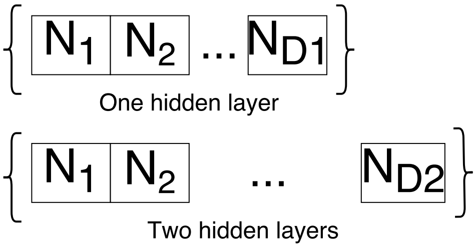


Figure 2.4: Encoding vectors for one PSO particle in two different types of NN

For demonstration, the details of the encoding vector for two-hidden-layer NN can be observed in Fig. 2.5. The N_{D2} parameters contain $N_I \times N_H$ weights and N_H biases in the connections between the input layer and the first hidden layers, $N_H \times N_H$ weights and N_H biases in the connections between the first hidden layer and the second hidden layer, $N_H \times N_O$ weights and N_O biases in the connections between the last layer of the hidden layers and the output layer.

Finishing the encoding procedure, the training phase is executed. The training data are sent to the NN whose weights and biases are already encoded in the form of the PSO vectors. Each PSO vector represents a position of a particle. This position is a set of potential solutions for the parameters (the weights and the biases) of the NN. The goal of the training phase is to find the position of the particles, in other words, the weights and biases of the NN which could minimize the error between the actual output of the NN and the desired output of the NN.

In each iteration, the actual output data of the NN are compared with the labeled data (the desired output data) using an objective function, for all data. In this research, the mean square error function is used as the objective function as shown in Eq. (2.11). This is the single objective function [8].

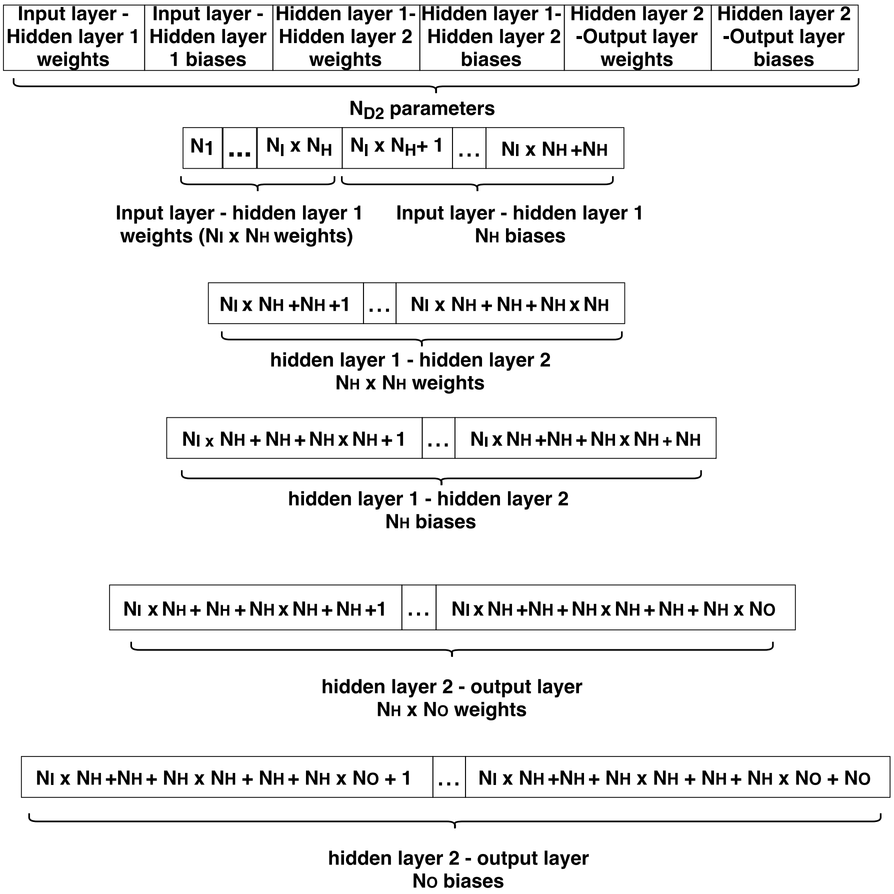

Input layer - Hidden layer 1 weights
Input layer - Hidden layer 1 biases
Hidden layer 1- Hidden layer 2 weights
Hidden layer 1- Hidden layer 2 biases
Hidden layer 2 -Output layer weights
Hidden layer 2 -Output layer biases
ND2 parameters
N1 ... NI x NH | NI x NH+ 1 ... NI x NH +NH
Input layer - hidden layer 1 weights (NI x NH weights)
Input layer - hidden layer 1 NH biases
NI x NH +NH +1 ... NI x NH + NH + NH x NH
hidden layer 1 - hidden layer 2 NH x NH weights
NI x NH + NH + NH x NH + 1 ... NI x NH +NH + NH x NH + NH
hidden layer 1 - hidden layer 2 NH biases
NI x NH + NH + NH x NH + NH +1 ... NI x NH +NH + NH x NH + NH + NH x NO
hidden layer 2 - output layer NH x No weights
NI x NH +NH + NH x NH + NH + NH x NO + 1 ... NI x NH +NH + NH x NH + NH + NH x NO + NO
hidden layer 2 - output layer No biases

Figure 2.5: Details of the encoding vectors for two-hidden-layer NN

$$fitness_p = \frac{1}{T}\sum_{i=1}^{T}(labeled\_data_i(j) - output\_data_{ip}(j))^2 \tag{2.11}$$

where $fitness_p$ is the fitness value of particle p, T is the number of training samples, $labeled\_data_i(j)$ is the j^{th} component of the labeled data i $(1 \le i \le T)$, $output\_data_{ip}(j)$ is the j^{th} output data i of particle p.

At any given iteration, for each particle p, the position ever visited by this particle which has the lowest learning error (lowest fitness value $fitness_p$) calculated by Eq. (2.11) becomes the personal best position $\overrightarrow{xPbest_p}$. The fitness value corresponds to the personal best position becomes the personal best fitness value $Pbest_p$. In the similar way, the position visited by all particles in the swarm that obtains the lowest learning error calculated by Eq. (2.11) is the global best position $\overrightarrow{xGbest}$ of the swarm. The corresponding fitness value of the global best position is the global best fitness value $Gbest$.

The particles move in each iteration to minimize the objective function (Eq.2.11). The changing of the position of each particle means the weights updating and the biases updating of the NN.

The training continues until the achievement of the stopping conditions. Normally, the stopping conditions are the number of iterations or the final global fitness value *Gbest*. The weights and biases of the NN which correspond to the global best position of the PSO algorithm after the training phase is used in the testing phase of the NN.

References

[1] S. Haykin. *Neural networks and learning machines.* Pearson Education, Upper Saddle River, NJ, third edition, 2009.

[2] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[3] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, Nov 1995.

[4] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, 2007.

[5] X. F. Xie, W. J. Zang, and Yang Z. L. Dissipative particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1456–1461, 2002.

[6] J. R. Zhang, J. Zhang, T. M. Lok, and M. R. Lyu. A hybrid particle swarm optimizationback-propagation algorithm for feedforward neural network training. *Applied Mathematics and Computation*, 185(2):1026 – 1037, 2007.

[7] M. T. Das and L. C. Dulger. Signature verification (sv) toolbox: Application of pso-nn. *Engineering Applications of Artificial Intelligence*, 22(45):688 – 694, 2009.

[8] E. L. Lehmann and G. Casella. *Theory of Point Estimation , Second Edition Springer Texts in Statistics*, volume 41. 1998.

Chapter 3

Proposed PSO algorithms

Chapter 3 focuses on the premature convergence issue of the SPSO algorithm by proposing three improved versions of the SPSO algorithm called wPSOd_CV, PSOseed, and PSOseed2. Several parts of this chapter have been published in our research articles [1, 2].

3.1 Introduction

In the training of the NN, the SPSO algorithm may be stuck in a local minimum. In this situation, the SPSO will be stopped, the NN cannot be trained, and the recognition rate will be low. Therefore, it is necessary to have an improved version of the SPSO algorithm to solve the premature convergence of this algorithm.

Several different approaches have been introduced to overcome the premature convergence of the PSO algorithm. The attractive and repulsive PSO (ARPSO) adds the repulsive phase. The diversity of the swarm calculated in each iteration of the swarm to determine whether the ARPSO operates in the attractive phase (the SPSO algorithm) or the repulsive phase [3]. The PSO with spatial particle extension (SEPSO) that adds the radius r to detect the collisions of all particles in the swarm. If the collisions occur, the particles will be bounced backward to prevent the situation when all particles stick to the local minimum [4]. The opposition-based PSO calculates the opposite particle of each particle in each iteration. If the opposite particle has a better fitness function than the original particle, this opposite particle will replace the original particle [5]. Other researchers use the multi-swarm strategy [6] or use not only the information of a particle but also the information of its neighborhood to calculate the new position of this particle [7]. Other solutions are to use the mutation operations such as the Cauchy mutation [8, 9], the Gaussian mutation [10, 11], or to combine the SPSO algorithm with the genetic algorithm [12]. The improved versions of the PSO in previous studies normally add more functions, more compute-intensive tasks to the SPSO algorithm. It is beneficial to investigate an improved version of the PSO algorithm to keep the particles out of the local minimum which does not add many functions or tasks to the SPSO algorithm.

The main contribution of this chapter is to propose three new PSO algorithms for the training of the NN called the wPSOd_CV algorithm, the PSOseed algorithm, and the PSOseed2 algorithm, respectively. These PSO algorithms do not add many compute-intensive tasks to the SPSO algorithm because these algorithms only modify the velocity update function of the SPSO algorithm. All these three PSO algorithms were used to investigate the operation of the proposed co-design architectures. Experimental results will be shown in chapter 5, 6, and 7 of this thesis.

This chapter is presented as follows. Section 3.2 presents the wPSOd_CV algorithm. The drawback of the wPSOd_CV algorithm is also described in this section. Section 3.3 introduces the PSOseed algorithm to overcome the disadvantage of the wPSOd_CV algorithm. The limitation of the PSOseed also discussed in this section. Finally, the improved version of the PSOseed algorithm called the PSOseed2 algorithm is introduced in section 3.4. Section 3.5 concludes this chapter.

3.2 Proposed wPSOd_CV algorithm

3.2.1 Introduction

The first PSO algorithm introduced in this chapter is the wPSOd_CV algorithm. This algorithm contains two different strategies. The first strategy is the velocity control, and the second strategy is the weight control. The wPSOd_CV algorithm only changes the velocity update function. Thus, the position update function, the calculation of fitness values $Pbest_p$ and $Gbest$ are similar to the SPSO algorithm. These calculations can be seen in Eqs. (2.6), (2.7), and (2.8) presented in chapter 2 of this thesis.

3.2.1.1 Velocity control

The velocity control of the PSO algorithm is a crucial aspect. If the velocities of the particles are too slow, the training time becomes very long. On the other hand, the training phase becomes unstable if the particles have high velocities.

The velocity control mechanism is added to the velocity update function. This mechanism adds the c_3 part to the SPSO algorithm as can be observed in Eq. (3.1) [13].

$$\begin{aligned}\vec{v}_p(t+1) &= w \times \vec{v}_p(t) + c_1 \times (\overrightarrow{xPbest}_p(t) - \vec{x}_p(t)) \\ &+c_2 \times (\overrightarrow{xGbest}(t) - \vec{x}_p(t)) + \frac{c_3 \times r}{(\vec{v}_p(t))^2}\end{aligned} \tag{3.1}$$

where w is the inertia weight, r is the random numbers, c_1, c_2, and c_3 are the coefficients, $\vec{x_p}(t)$ is the position of particle p, $\overrightarrow{xPbest_p}(t)$ is the best position found by particle p, $\overrightarrow{xGbest(t)}$ is the best position found by any particle in the swarm, $\vec{v_p}(t+1)$ is the velocity of particle p at time $(t+1)$.

The velocity control is shown in Fig. 3.1 and is explained as follows.

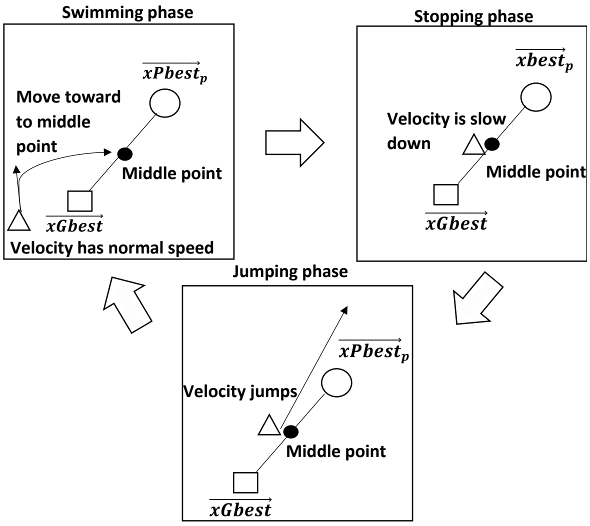

Swimming phase
$\overrightarrow{xPbest_p}$
Move toward to middle point
Middle point
$\overrightarrow{xGbest}$
Velocity has normal speed
Stopping phase
$\overrightarrow{xbest_p}$
Velocity is slow down
Middle point
$\overrightarrow{xGbest}$
Jumping phase
$\overrightarrow{xPbest_p}$
Velocity jumps
Middle point
$\overrightarrow{xGbest}$

Figure 3.1: Mechanism of the velocity control

1. Swimming phase: particle p moves with a high velocity to the middle position between $\overrightarrow{x\_Gbest}$ and $\overrightarrow{x\_Pbest}_p$. In this phase, the c_3 part has value zero, and the newly calculated velocity does not depend on this part.

2. Stopping phase: particle p is near the middle position between $\overrightarrow{x\_Gbest}$ and $\overrightarrow{x\_Pbest_p}$. In this phase, particle p reduces the speed, and the c_3 part starts to affect the newly calculated velocity.

3. Jumping phase: when particle p has a very small speed, the c_3 part will have a very high value. The newly calculated velocity will increase drastically, and particle p will jump to another searching area to start a new swimming phase.

3.2.1.2 Weight control

The inertia weight w also makes a significant impact to the PSO algorithm. When w has a high value, the focus of the PSO algorithm is the global area. On the other hand, when w has a small value, the PSO algorithm tends to search the local area. With the using of the linear decreasing strategy, the PSO algorithm addresses the exploitation task to search in the global area at the beginning of the PSO algorithm. Then, the exploration task will be investigated to focus on the local area. The linear decreasing strategy is illustrated in Eq. (3.2) [14].

By combination between the velocity control and weight control (the linear decreasing strategy of the inertia weight), the PSO algorithm may have the possibility to search the solutions which are skipped in the velocity-only approach.

$$w = w_{max} - \frac{w_{max} - w_{min}}{N_{ite}} \times ite \tag{3.2}$$

The proposed wPSOd_CV algorithm consists of both the velocity control and the inertia weight control. The velocity update function of the wPSOd_CV algorithm is given in Eq. (3.3).

$$\begin{aligned}\vec{v}_p(t+1) = w_{max} - \frac{w_{max} - w_{min}}{N_{ite}} \times ite \times \vec{v}_p(t) + c_1 \times (\overrightarrow{xPbest_p}(t) - \vec{x}_p(t)) \\ +c_2 \times (\overrightarrow{xGbest}(t) - \vec{x}_p(t)) + \frac{c_3 \times r}{(\vec{v}_p(t))^2}\end{aligned} \tag{3.3}$$

where N_{ite} is the number of iterations, ite is the current iteration, c_1, c_2, and c_3 are the coefficients, r is the random number.

3.2.2 Discussion

This section proposed a novel PSO algorithm for the training of the NN called the wPSOd_CV algorithm. This algorithm has two main components which are velocity control and weight control. The wPSOd_CV algorithm only modifies the velocity update function to alleviate the premature convergence. However, even modifying only the velocity update function, this algorithm already adds several arithmetic operators that are shown in the velocity control c_3 part shown in Eq. (3.1), and in the weight control strategy presented in Eq. (3.2). It is necessary to propose another PSO algorithm which adds fewer arithmetic operators to the velocity update function.

In addition, the velocity control mechanism also has a drawback. As described in section 3.2.1.1, the c_3 part always creates a big jump to another searching area in the jumping phase. If the searching ara has many solutions, the wPSOd_CV has the possibility to meet other solutions and jump to other searching space before the before meeting the best solution. This issue leads to a possibility to ignore the best solution or require more iterations to reach the best solution.

This chapter also introduces the PSOseed algorithm to overcome the disadvantages of the PSOd_CV algorithm. This PSOseed algorithm which does not add many computational tasks as the wPSOd_CV algorithm will be presented in the next section.

3.3 Proposed PSOseed algorithm

3.3.1 Introduction

To overcome the drawbacks of the wPSOd_CV algorithm, the PSOseed algorithm is proposed. Avoiding jumping phases, for each particle the PSOseed algorithm uses a new variable called the seed position that is randomly generated in the initial phase of the algorithm. In each iteration, each particle is attracted and pulled to the position of its seed. The seed mechanism could reduce the possibility that the particle falls in a local minimum. Compared to the wPSOd_CV algorithm, the PSOseed algorithm does not use any division operator and use fewer multiplication operators.

In the PSOseed algorithm, the velocity update function was modified as can be seen in Eq. (3.4). This PSOseed algorithm introduces a new c_3 part when compared with the SPSO algorithm. This c_3 part has a new variable called the seed positions of the particles. In the PSOseed algorithm, each particle has its seed position. At the beginning of the PSO training, the seed positions of all particles in the swarm are assigned the random values.

$$\begin{aligned}\overrightarrow{v_p}(t+1) = w \times \overrightarrow{v_p}(t) + c_1 \times (\overrightarrow{xPbest_p}(t) - \overrightarrow{x_p}(t)) + c_2 \times (\overrightarrow{xGbest}(t) - \overrightarrow{x_p}(t)) \\ + c_3 \times r \times (\overrightarrow{xSeed_p} - \overrightarrow{x_p}(t))\end{aligned} \tag{3.4}$$

where $\overrightarrow{xSeed_p}$ is the seed position of particle p, c_1, c_2, and c_3 are the coefficient, r is the random number.

With the appearance of the c_3 part, the calculated velocity $\overrightarrow{v_p}(t+1)$ at time $(t+1)$ of particle p depends not only on the current current velocity $\overrightarrow{v_p}(t)$, the best position $\overrightarrow{xPbest_p}(t)$ found by particle p, the best position found by all particle $\overrightarrow{xGbest(t)}$ but also on the seed position $\overrightarrow{xSeed_p}$ of this particle.

The c_3 part or the seed factor of particle p usually attracts particle p by pulling this particle to the seed position $\overrightarrow{xSeed_p}$ even when the particle p is stuck in a local minimum. Using this concept, particle p may be kept out of the local minimum and continue the operation in another searching area.

3.3.2 Discussion

This section proposes the PSOseed algorithm which slightly modifies the velocity update function of the SPSO algorithm.

The proposed PSOseed algorithm uses the seed factors to keep the particles out of the local minimum. However, the seed positions of all particles are randomly generated in the initial phase of the PSOSeed algorithm without any control. The performance of the PSOseed algorithm highly affected by these seed positions. If the seeds are generated in the proper positions, the NN trained the by PSOseed algorithm could have a high recognition rate and low learning error. On the other hand, the accuracy of the NN trained by the PSOseed algorithm will reduce significantly if the positions of the seeds are poorly generated.

Therefore, an improved version of the PSOseed algorithm called the PSOseed2 algorithm is also introduced in the next section. This PSOseed2 algorithm has a mechanism to control the seed positions of all particles.

3.4 Proposed PSOseed2 algorithms

3.4.1 Introduction

This section presents the PSOseed2 algorithm to overcome the drawback of the PSOseed concerning the seed positions. This PSOseed2 algorithm which combines the PSOseed algorithm and a reseed mechanism also does not add many compute-intensive tasks to the SPSO algorithm.

Similar to the operation of the PSOSseed algorithm, the seed positions of all particles are also initialized at the beginning of the PSOseed2 algorithm. The PSOseed2 algorithm also uses the velocity update function of the PSOseed algorithm as expressed in Eq.(3.4). However, the seed positions of all particles are fixed in the PSOseed algorithm while the reseed control is conducted in each iteration of the PSOseed2 algorithm. For each particle, the newly calculated fitness value $Gbest(t+1)$ at time $(t+1)$ are compared with the fitness value $Gbest(t)$ at time t. If this newly calculated $Gbest(t+1)$ does not have a lower fitness value than the fitness value $Gbest(t)$, all seed factors are reseeded. The operation of the PSOseed2 algorithm is described as follows.

1. At the beginning of the algorithm at time t, assign the initial values for the seed positions of all particles.

2. At time $(t+1)$, conduct the PSOseed algorithm for all particles using:

   - Eq. (3.4) presented in section 3.4 of this chapter to update the new velocities.
   - Eq. (2.6) presented in section 2.2 of chapter 2 to calculate the new positions.
   - Eq. (2.7) presented in section 2.2 of chapter 2 to evaluate the new fitness values $Pbest(t+1)$ and the corresponding positions $\overrightarrow{xPbest(t+1)}$, for all particles.
   - Eq. (2.8) presented in section 2.2 of chapter 2 to estimate the new fitness values $Gbest(t+1)$ and the corresponding positions $\overrightarrow{xGbest(t+1)}$.

3. Compare the newly calculated $Gbest(t+1)$ at time $(t+1)$ with the $Gbest(t)$ at time t:

   - If the newly calculated $Gbest(t+1) \geq Gbest(t)$ then renew all seed factors.
   - If the newly calculated $Gbest(t+1) < Gbest(t)$ then update the global fitness value $Gbest(t) = Gbest(t+1)$.

4. Return to step 2 to start a new iteration at time $(t+2)$ until the stopping criteria are satisfied.

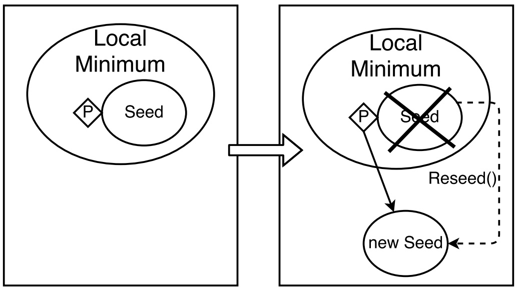


Figure 3.2: Operation of the PSOseed2 algorithm

The operation of the PSOseed2 algorithm could be illustrated in Fig. 3.2. In this figure, the reseed mechanism is used to overcome the problem of the seed positions. The PSOseed algorithm will finish when the positions of both particle p and its initial seed are in the local minimum. In this situation, the NN cannot be trained by the PSOseed algorithm. On the contrary, the PSOseed2 may activate the reseed mechanism to generate the new seed position of particle p which pulls the particle p out of the local minimum. The NN can continue to be trained by the PSOseed2 algorithm.

3.4.2 Discussion

This section already presents the PSOseed2 algorithm which is based on the concept of the PSOseed algorithm presented in section 3.3. The PSOseed2 algorithm is the combination of the PSOseed algorithm and the reseed control which could solve the seeding problem of the PSOseed algorithm.

The PSOseed2 also solves the premature convergence of the SPSO without adding many computational tasks or functions to the SPSO algorithm.

3.5 Discussion

Three different PSO algorithms are proposed in this chapter. All three PSO algorithms only change the velocity update function of the SPSO algorithm. The first PSO algorithm called the wPSOd_CV that has two different strategies called the velocity control and the inertia weight control. However, the wPSOd_CV still adds several tasks to the SPSO algorithm. In addition, the jumping phase of the velocity control may skip the solutions. Therefore, another PSO algorithm is introduced called the PSOseed algorithm. This algorithm employs the seed factors to pull the particle out of the local minimum. The new part in the velocity update function of this algorithm does not use any multiplication or division operations. However, the PSOseed algorithm highly depends on the seeds. If the seeds are poorly generated, the performance of the PSOseed will be decreased significantly. The PSOseed2 is also proposed in this chapter to improve the PSOseed algorithm. The PSOseed2 is the PSOseed algorithm with the reseed mechanism.

All these three PSO algorithms were used to train the NN in the proposed co-design architectures. Experimental results of these PSO algorithms will be detailed in next chapters.

From next chapter, the co-design for the NN-PSO will be presented. The first issue described in chapter 4 is the proposed partitioning methodology of the NN-PSO.

References

[1] T. L. Dang, T. Cao, and Y. Hoshino. A proposed psoseed2 algorithm for training hardware-based and software-based neural networks. *International Journal of Innovative Computing, Information and Control*, 13(4):1205–1219, Aug 2017.

[2] T. L. Dang and Y. Hoshino. An-fpga based classification system by using a neural network and an improved particle swarm optimization algorithm. In *2016 Joint 8th International Conference on Soft Computing and Intelligent Systems (SCIS) and 17th International Symposium on Advanced Intelligent Systems (ISIS)*, pages 97–102, Aug 2016.

[3] J. Riget and J.S. Vesterstrm. A diversity-guided particle swarm optimizer - the arpso. Technical report, 2002.

[4] T. Krink, J. S. Vesterstrom, and J. Riget. Particle swarm optimisation with spatial particle extension. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1474–1479, 2002.

[5] H. Wang, H. Li, Y. Liu, C. Li, and S. Zeng. Opposition-based particle swarm algorithm with cauchy mutation. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation*, pages 4750–4756, Sep 2007.

[6] J. Jie, J. Zeng, C. Han, and Q. Wang. Knowledge-based cooperative particle swarm optimization. *Applied Mathematics and Computation*, 205(2):861 – 873, 2008.

[7] R. Mendes, J. Kennedy, and J. Neves. The fully informed particle swarm: simpler, maybe better. *IEEE Transactions on Evolutionary Computation*, 8(3):204–210, Jun 2004.

[8] H. Wang, C. Li, Y. Liu, and S. Zeng. A hybrid particle swarm algorithm with cauchy mutation. In *Proceedings of the 2007 IEEE Swarm Intelligence Symposium*, pages 356–360, Apr 2007.

[9] A. Stacey, M. Jancic, and I. Grundy. Particle swarm optimization with mutation. In *Proceedings of the 2003 Congress on Evolutionary Computation*, volume 2, pages 1425–1430, Dec 2003.

[10] N. Higashi and H. Iba. Particle swarm optimization with gaussian mutation. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium*, pages 72–79, Apr 2003.

[11] Q. Wu. Power load forecasts based on hybrid pso with gaussian and adaptive mutation and wv-svm. *Expert Systems with Applications*, 37(1):194 – 201, 2010. ISSN 0957-4174.

[12] B. Yang, Y. Chen, and Z. Zhao. A hybrid evolutionary algorithm by combination of pso and ga for unconstrained and constrained optimization problems. In *Proceedings of the 2007 IEEE International Conference on Control and Automation*, pages 166–170, May 2007.

[13] Y. Hoshino and H. Takimoto. Pso training of the neural network application for a controller of the line tracing car. In *Proceedings of the 2012 IEEE International Conference on Fuzzy Systems*, pages 1–8, Jun 2012.

[14] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence*, pages 69–73, May 1998.

Chapter 4

Proposed partitioning methodology for the NN-PSO

Chapter 4 describes the proposed partitioning methodology between hardware and software for the NN-PSO. The operations on the software side of the proposed partitioning methodology are also presented in this chapter. Several parts of this chapter have been published in our research articles [1–3].

4.1 Introduction

The FPGA-based program may obtain a higher operating speed than the conventional software-based program because of the parallelism of the FPGA-based program [4, 5]. Compared to the GPU-based program, the FPGA-based program has a lower power consumption [5–7]. In addition, all components of the FPGA-based program such as the FPGA chip, the random access memory (RAM), the buttons... could be implemented in a single FPGA board, and the FPGA-based can operate without the need of having a processor that is very portable. Nowadays, the using of a system on chip (SoC) FPGA emerges. The single SoC FPGA device consists of both a processor such as an ARM processor and FPGA components. This approach could increase the operating speed because of higher bandwidth, and increase the portability because of lower board size [8].

However, the hardware-based approach also has the disadvantage in terms of the resource constraints. Each FPGA design project needs to take care of the FPGA resources because the resources of the FPGA device concerning the logic elements and the memory bits are very expensive which could exceed the financial condition of the project [9]. In addition, the FPGA-based program also has the drawback regarding the flexibility. Compared with the software-based approach, the FPGA-based approach requires more time and efforts to modify the program. For example, each time the parameters of the PSO algorithm such as the number of iterations or the number of particles are changed, the waiting time for rebuilding the program is unavoidable [10].

The operating speed of the software-based program could be slower than the hardware-based program. However, the software implementation approach has the flexibility advantage. It is easy to modify the parameters without rebuilding and recompiling the program. Furthermore, the software-based program could be implemented by a processor such as the ARM processor or the Intel processor which is not expensive as compared to the FPGA resources.

Therefore, this research investigates the co-design architectures between hardware and software for the training of the NN by PSO algorithms. These architectures have the speed advantage of the hardware-only approach, the flexibility and the resources benefits of the software-only approach.

In the co-design, one of the most important issues is the partitioning methodology. It is necessary to determine what component of the system is executed by a processor on the software side, and what components are implemented on the hardware side.

The main contribution in this chapter is to propose the partitioning methodology for the NN trained by PSO algorithms. The operations on the software side are also presented in this chapter.

This chapter is presented as follows. Section 4.2 presents the proposed partitioning methodology for the NN-PSO. The operations on the software side are detailed in section 4.3.

4.2 Proposed partitioning methodology

This research proposes a co-design architecture which could take advantage of both software approach and hardware approach.

In the NN-PSO system, the PSO algorithms are used only in the training phase of the NN while the NN is used in both the training phase and the testing phase. Hence, the PSO algorithms are moved to the software side and implemented by a processor by the C programming language. On the other hand, the NN is still implemented on the hardware side by the SystemVerilog programming language. The advantages of the proposed architecture are as follows.

- The necessary FPGA resources for the proposed co-design architecture is reduced when compared with the hardware-only architecture because the PSO algorithms are already moved to the software side. The FPGA resources reserved for the implementation of the PSO algorithms are not used in the co-design approach.
- During the training of the NN, the parameters of the PSO algorithms could be modified without rebuilding or redesigning the FPGA-based NN. It is easy to change the parameters of the PSO algorithms such as the number of iterations, the number of particles, or even the new PSO algorithm to investigate the operation of the proposed architecture in different situations. Having this flexibility, the software-based PSO may have a higher ratio of performance to cost than the FPGA-based PSO.
- The testing phase still has the FPGA-based NN which maintains the operating speed. Compared with the software-only approach, the co-design approach may obtain a higher operating speed in the testing phase.
- The processors used for the implementation of the PSO algorithms in this research are the NIOS II processor and the ARM processor which are already embedded in the FPGA board. Therefore, the co-design still has the portability because all components are implemented in a single FPGA board.

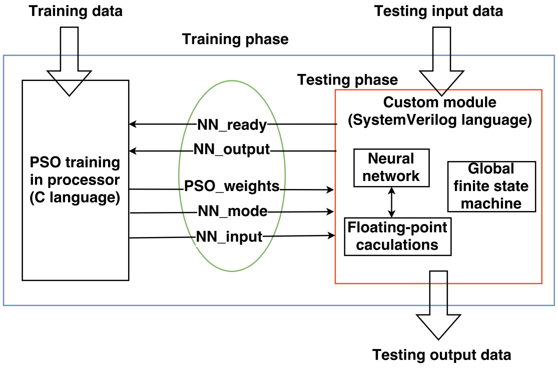


Figure 4.1: Proposed partitioning methodology

The proposed co-design architecture can be observed in Fig. 4.1. The components of the training phase are shown inside the blue rectangle. These components are the

PSO training module implemented by a processor, the NN implemented in an FPGA-based custom module, and the connections between the software-based module and the hardware-based module. The NN is wrapped inside the custom module along with the floating-point calculation module for the floating-point calculations and the finite state machine for control the operations of the custom module.

There are three different co-design architectures presented in this thesis. The first architecture uses the NIOS II as the processor. The second architecture replaces the NIOS II processor by the ARM processor. The third architecture combines the ARM approach of the second architecture with the hardware-based PCA. The details of these architectures along with the connections between the hardware side and the software side will be discussed in chapters 5, 6, and 7.

Finishing the training phase, the FPGA-based components inside the red rectangle are used in the testing phase.

4.3 Operations on the software side

This section presents the operations during the training phase on the software side. These operations are executed by a processor. The operations on the hardware side and the connections between hardware and software are presented in chapters 5, 6, and 7.

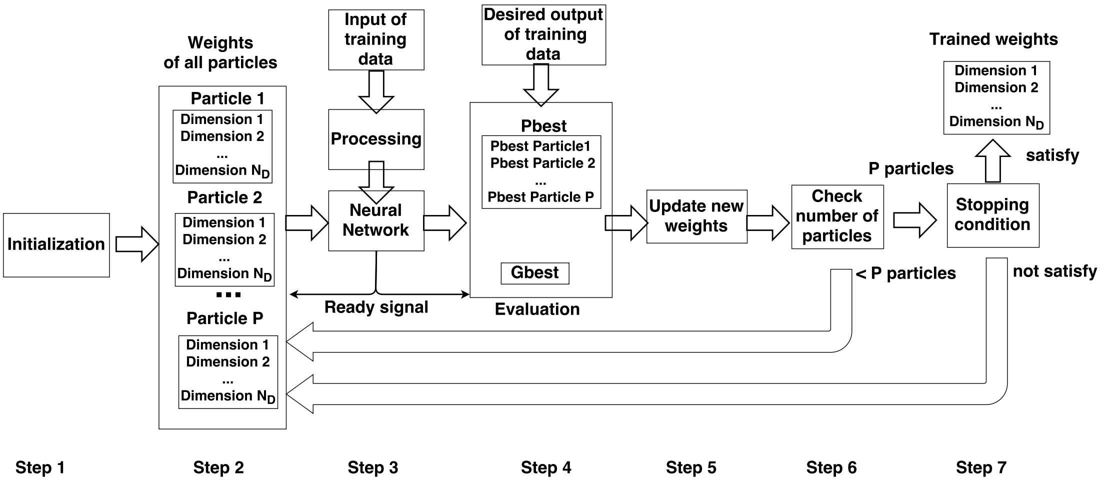


Figure 4.2: Operations on the software side

Fig. 4.2 illustrates the training phase of our co-design. All components except the NN are executed by a processor. The operation of the training is as follows.

Step 1 The initial values of all velocities and positions are randomly generated. As discussed in chapter 2, the position of each particle is a set of potential solution for the weights and biases of the NN.

Step 2 At any given iteration, the position vector ($\overrightarrow{PSO\_weights}$) of particle p is sent to the NN where $0 \leqslant p \leqslant P$, and P is the number of particles. The $\overrightarrow{PSO\_weights}$ has N_D dimensions that represent N_D parameters of the NN. In our research, $N_D = N_{D1}$ if one-hidden-layers NN is used, $N_D = N_{D2}$ if two-hidden-layer NN is employed. The calculations of D_1 and D_2 are presented in Eqs. (2.3) and (2.4) presented in chapter 2.

Step 3 The software side waits for the results from the hardware-based NN by checking a ready signal. Based on our experiments, the training of the NN obtained better accuracy when the training data are normalized and examined in the random order. Thus, the processing module is added to shuffle the data. At any given iteration of the training phase, this module creates a new hash table for all training data. In this processing module, the data are also processed using z-score normalization as can be seen in Eq. (4.1) [11]. The attributes are processed one-by-one. The first attribute of all data samples is normalized in the first iteration. This normalization continues to process the second attribute in the second iteration, the third attribute in the third iteration.... This process repeats until all attributed are already normalized.

$$attribute_{ij} = \frac{attribute_{ij} - mean\_attribute_i}{sd\_attribute_i} \tag{4.1}$$

where $attribute_{ij}$ is the j^{th} data of the $attribute_i$, $mean\_attribute_i$ and $sd\_attribute_i$ denote the mean and the standard deviation of the $attribute_i$.

Step 4 The evaluation module checks the ready signal from the NN before receiving the processed data from the NN. The actual output from NN is compared with the desired output of the NN by using the objective function as shown in Eq. (2.11). The fitness values calculated from this fitness function are used to identify the new $Pbest_p$ of particle p and the new $Gbest$ of all particles. The particle positions $\overrightarrow{x\_Pbest_p}$ corresponding to the fitness value $Pbest_p$ and $\overrightarrow{x\_Gbest}$ corresponding to the fitness value $Gbest$ are saved. As presented in section 2.3, $\overrightarrow{x\_Pbest_p}$ is the position of the particle p that obtains the $Pbest_p$ value, and $\overrightarrow{x\_Gbest}$ is the position of any particle in the swarm that gets the $Gbest$ value.

Step 5 The new velocity and new position of particle p is updated based on Eq. (2.5) and (2.6) presented in chapter 2.

Step 6 This step is reserved for the checking of the particle numbers. If all particles are already processed, the training phase moves to step 7. Otherwise, the training returns to step 2, and next particle will be addressed.

Step 7 This step is used for the checking of the stopping condition which is the number of iterations in this study. If the stopping criterion is met, the training phase is terminated, and the trained weights will be stored.

4.4 Discussion

This chapter presents the proposed co-design between hardware and software for the training of the NN by PSO algorithms. As discussed, the NN is implemented in hardware to maintain the testing speed of the hardware-based program while the PSO is implemented in software to keep the flexibility of the software-based program. Furthermore, the software implementation of the PSO algorithms can also reduce the FPGA resources previously reserved for the PSO algorithms.
The details of the co-design architectures will be discussed in the next chapters of this thesis. Chapter 5 focuses on the first architecture using the NIOS II processor.

References

[1] T. L. Dang, T. Cao, and Y. Hoshino. A proposed psoseed2 algorithm for training hardware-based and software-based neural networks. *International Journal of Innovative Computing, Information and Control*, 13(4):1205–1219, Aug 2017.

[2] T. L. Dang, T. Cao, and Y. Hoshino. Hybrid hardware-software architecture for neural networks trained by improved pso algorithm. *ICIC Experss Letters*, 11(3): 565–574, Mar 2017.

[3] T. L. Dang and Y. Hoshino. An-fpga based classification system by using a neural network and an improved particle swarm optimization algorithm. In *2016 Joint 8th International Conference on Soft Computing and Intelligent Systems (SCIS)*

and 17th International Symposium on Advanced Intelligent Systems (ISIS), pages 97–102, Aug 2016.

[4] E. Monmasson and M. N. Cirstea. Fpga design methodology for industrial control systems - a review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, Aug 2007.

[5] J. A. G. Pulido, M. A. V. Rodriguez, J. M. S. Perez, S. P. Mendes, and V. Carreira. Accelerating floating-point fitness functions in evolutionary algorithms: a fpga-cpu-gpu performance comparison. *Genetic Programming and Evolvable Machines*, 12 (4):403–427, 2011.

[6] D. Zou, Y. Dou, and F. Xia. Optimization schemes and performance evaluation of smithwaterman algorithm on cpu, gpu and fpga. *Concurrency and Computation: Practice and Experience*, 24(14):1625–1644, 2012.

[7] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian. High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP. *International Journal of Reconfigurable Computing*, 2012, 2012.

[8] Altera. What is an SoC FPGA? page 4, 2014.

[9] D. G. Bailey. *Design for Embedded Image Processing on FPGAs*. Wiley Publishing, 1st edition, 2011.

[10] S. A. Li, C. C. Hsu, C. C. Wong, and C. J. Yu. Hardware/software co-design for particle swarm optimization algorithm. *Information Sciences*, 181(20):4582–4596, 2011.

[11] J. Han, M. Kamber, and J. Pei. Data preprocessing. In *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 83 – 124. Morgan Kaufmann, Boston, third edition edition, 2012.

Chapter 5

Proposed co-design architecture using the NIOS II processor

Chapter 5 focuses on the first co-design architecture, based on the partitioning methodology presented in chapter 4 which uses the NIOS II processor. This chapter also discusses the performances of the NN trained by proposed PSO algorithms in this architecture. Several parts of this chapter have been published in our research articles [1, 2].

5.1 Introduction

In the co-design architecture, choosing a processor is an important issue. The first proposed architecture uses the NISO II processor developed by Altera [3, 4]. This processor is selected because of several advantages concerning the flexibility, the high performance, the low cost, and the long life cycle as claimed by Altera [3, 4].

In addition, many intellectual property (IP) cores are provided along with the NIOS II processor to reduce the design time, increase the performance of the system. These IP cores can be freely downloaded from the website of Altera [5–7].

The main contribution of this chapter is to introduce a hybrid framework between software and hardware for the training of the FPGA-based NN by software-based PSO algorithms. In this framework, the NN is hardware implemented by SystemVerilog programming language while the PSO algorithms are executed in the NIOS II processor. In our experiments, three proposed PSO algorithms (wPSOd_CV, PSOseed, PSOseed2) introduced in chapter 3 and two other PSO algorithms (SPSO, DPSO) presented chapter 2 were used to test with two publicly recognized databases (heart disease dataset and iris dataset). The operating speed and the required resources were also investigated.

This chapter is presented as follows. Section 5.2 presents the proposed architecture. The experimental results are shown in section 5.3. Finally, section 5.4 concludes this chapter.

5.2 Proposed architecture

5.2.1 Overview

As discussed in chapter 4 of this thesis, the NN is implemented by SystemVerilog in hardware to accelerate the testing speed of the co-design architecture. The PSO algorithms are moved to the software side to maintain the flexibility of the software-based program and to reduce the required FPGA logic elements and memory bits. In this approach, the NIOS II processor developed by Altera is used as the processor. This processor is one of the most famous embedded softcore processors that is optimized for the FPGA programs. Using the NIOS II softcore processor, all components of the NN trained by PSO algorithms (NN-PSO) system are implemented in an FGPA device which could be portable and low power consumption. In addition, it is not difficult to customize the NIOS II processor to satisfy the new requirements or the new features of the design [3, 4].

Fig. 5.1 shows the components of the proposed co-design approach. The FPGA-based module contains five components which are an NN, a floating-point (FP) submodule, a buffer_in, a buffer_out, and a ready submodule. In this design, the floating-point submodule is reserved for the floating-point calculations. This submodule implements the floating-point IP cores provided by Altera to reduce the design time [6].

In this proposed co-design architecture, the synchronization between the FPGA side and the NIOS II processor is crucial. The program will not work if the design could not overcome this issue. Two buffers and the *ready* signal are employed to control the connections between the hardware side and the software side. These two buffers are implemented using the on-chip memory of the FPGA device. The ready submodule will send the *ready* signal to the NIOS II processor when buffer_in is empty or the buffer_out is full. The buffer_in whose size is $N_D + N_I$ is used to store the parameters (weights and biases), and the input data of the NN. As described in chapter 2, N_I is the number of nodes in the input layer, N_D is the number of weights and biases. In this situation, $N_D = N_{D1}$ if one-hidden-layer NN is used, and $N_D = N_{D2}$ if two-hidden-layers NN is used. The calculations of the N_{D1} and N_{D2} are presented in Eqs. (2.3) and (2.4) of chapter 2.

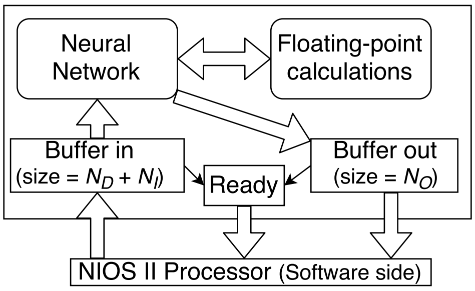

Neural Network
Floating-point calculations
Buffer in (size = $N_D + N_I$)
Ready
Buffer out (size = N_O)
NIOS II Processor (Software side)

Figure 5.1: Proposed co-design with NIOS II processor

The operations of the PSO training executed by the NIOS II processor on the software side are already discussed in section 4.3 of chapter 4. The next sections in this chapter are dedicated to the operations on the hardware side and the connections between software and hardware.

5.2.2 The FPGA-based custom module

The operation of the FPGA-based custom module is based on a global finite-state machine (FSM) as shown in Fig. 5.2. Normally, the FSM is kept in the *idle* state. When the buffer_in is full, the FSM changes to the *NN_running* state. In this state, the NN starts the operation which is controlled by another FSM called FSM-NN. The activation function of the NN is the hardware implementation of the Sigmoid function which is shown in Eq. (2.2) of chapter 2. When the NN needs to use the floating-point calculations, the NN will send a *request* signal to the FP submodule. After sending *request* signal, the FSM of the hardware-based module will move to the *FP_running* state. Inside the floating-point submodule, there is a *calculation_counter*. When the *calculation_counter* reduces to zero, the FSM returns to the *NN_running* state. Upon finishing the operation, NN will send the output data to the buffer_out. When the buffer_out is full, the FSM comes back to the *idle* state.

The FP submodule which handles the floating-point calculation of the FPGA-based module implements the floating-point IP cores provided by Altera [6]. The *calculation_counter* is used in the FP submodule to synchronize the operation of the NN with the operation of the FP submodule. The connections between the FP submodule and the NN are illustrated in Fig. 5.3. The details of the operation are as follows.

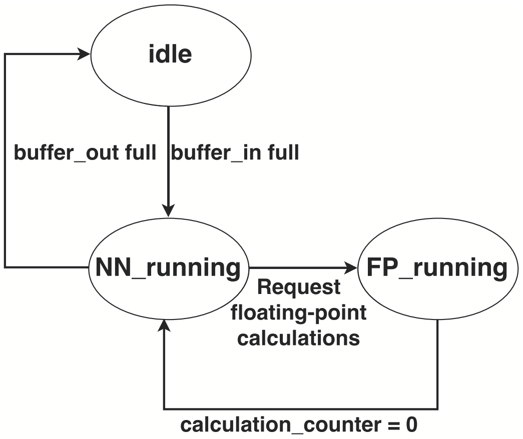


Figure 5.2: FSM of the FPGA-based component

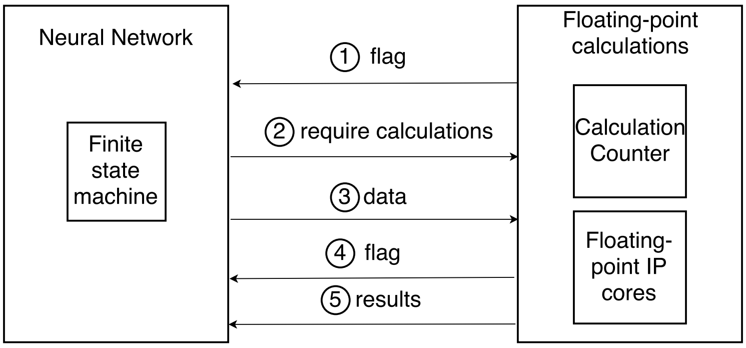


Figure 5.3: Synchronization between NN and floating-point submodule

1. The FSM-NN controls the operation of the hardware-based NN. This FSM has two main states called *waiting_phase* and *running_phase*. In the *running_phase*, the FSM has several smaller states to calculate the output from each node of the NN using Eqs. (2.1) and (2.2). During the calculation, if the NN needs to execute a floating-point operation, the NN will check the current status of the FP submodule by investigating the $flag$ signal come from the FP submodule. If the signal is not ready ($flag = 0$), the FSM of the NN moves to *waiting_phase*. Otherwise, the algorithm changes to step 2.

2. The *require_calculations* signal is sent from NN to the FP submodule. This signal depends on the type of the operation. For example, if the subtraction is required, the NN will send $float\_alu\_mode\_sub$ signal to the FP submodule.

3. The input data are also forwarded to the FP submodule using *data*.

4. The NN determines whether the operation of the FP module is finished or not by rechecking the $flag$ signal. Inside the FP module, the $flag$ signal is controlled by the *calculation_counter*. As discussed, the FP module implements the floating-point IP cores provided by Altera. The output latency for each arithmetic operation is specified in its user guide [6]. The values of these latencies are employed in the *calculation_counter*. Upon receiving the *require_calculations* signal, the $flag$ will be set to value zero ($flag = 0$), and *calculation_counter* will be assigned to a corresponding value to the latency of the required operation. For example, the latency of the subtraction is mentioned in the user guide as seven clock cycles. Hence, when the *require_calculations* is the *float_alu_mode_sub* signal, the *calculation_counter* has value seven which is reduced one unit on each clock cycle. When the counter value equals zero, the subtraction is finished. The $flag$ signal will be set to value one ($flag = 1$). Otherwise, the value of the $flag$ signal is still zero.

5. In the NN side:

    - If $flag$ signal has value one ($flag = 1$), the NN will collect the outputs of the floating-point calculations from the FP submodule. After that, the FSM of the NN continues to operate in the *running_phase*. When the NN needs the floating-point calculations again, the algorithm returns to step 2.
    - If $flag$ signal still has value zero ($flag = 0$), the FSM of the NN changes to the *waiting_phase* to wait for the results of the FP submodule.

5.2.3 Connections between hardware and software

5.2.3.1 Operation

The connections between the hardware side and the software side become important issues in the proposed architecture. If the operation on the hardware side cannot synchronize with the operation on the software side, the co-design architecture may not produce the correct results.

Fig. 5.4 shows the synchronization. The details are described as follows.

1. The NIOS II processor checks whether the FPGA-based module is in the ready state or not by checking the *ready* signal sent from the FPGA-based module. In the FPGA-based module, the value of the *ready* signal depends on the state of

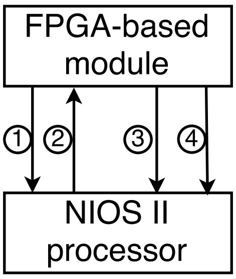


Figure 5.4: The FPGA-based component

the buffer_in. If this buffer is empty, the *ready* signal has value one ($ready = 1$). Otherwise, the *ready* are set to zero ($ready = 0$).

2. In the NIOS II side:

   - if $ready = 1$, the NIOS II processor will send $N_D + N_I$ data to the buffer_in of the FPGA side (as presented, N_D is the number of weights and biases of the NN, N_I is the number of input nodes of the NN). After that, the *ready* is set to value zero ($ready = 0$). The data from buffer_in are sent to the NN to process. Finishing the processing, the output data from NN module will be sent to the buffer_out whose size is N_O (N_O is the number of output nodes). If the buffer_out is full, the *ready* will be set again to value one ($ready = 1$).
   - if $ready = 0$, the NIOS II is put in the waiting state.

3. The NIOS II uses the *ready* signal again to determine whether the operation of the FPGA-based module is finished or not.

4. If $ready = 1$, the NIOS II processor will collect the output data from the buffer_out. Otherwise, the NIOS II processor will wait.

5.2.3.2 Implementation

The proposed architecture uses the IP cores from Altera to increase the performance, and decrease the design time. Besides the floating-point IP cores presented in the previous section, the Avalon memory-mapped interface (Avalon-MM) is also used. This interface is used to create the connections between hardware side and software side. The communication and the data transmission between the NIOS II processor and the FPGA-based module are conducted using this interface.

The operation of the Avalon-MM is based on several signals. The full list of the signal could be observed in its user guide [5]. Five signals are used in this implementation. The *write* signal is used for the write request. The *read* signal is reserved for the read request. The address request employs the *address* signal. To send the data from the NIOS II to the FPGA modules, the *writedata* signal is needed. In the opposite direction, the *readdata* signal is dedicated to the data sent from FPGA modules to the NIOS II processor. Fig. 5.5 illustrates the Avalon-MM connection between the NIOS II processor and the custom module. The NIOS II is the master, and the custom module is considered as the slave in this connection. The FPGA-based NN is wrapped inside this custom module.

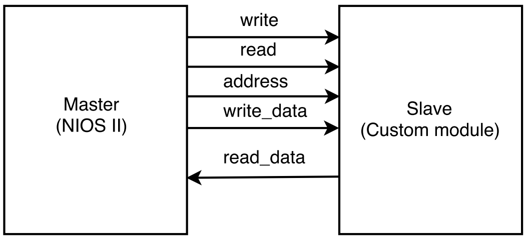


Figure 5.5: Avalon MM interface

The connection ports of the FPGA-based custom module is illustrated in Fig. 5.6. This module has two Avalon-MM connections called *avalon_slave_*0 and *avalon_slave_*1, respectively. The former is used for the data signal, and the latter is reserved for the control signal.

The connections of the whole system are presented in Fig. 5.7. In this figure, the FPGA-based module is called *new_component_*0. Two connections between the hardware-based module and the NIOS II processor have address 0000_0000 and address 0000_0400, respectively. These addresses could be accessed using the pointers in the C programming language executed by the NIOS II processor.

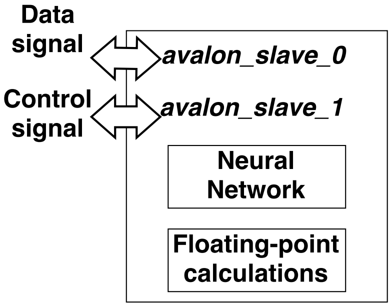
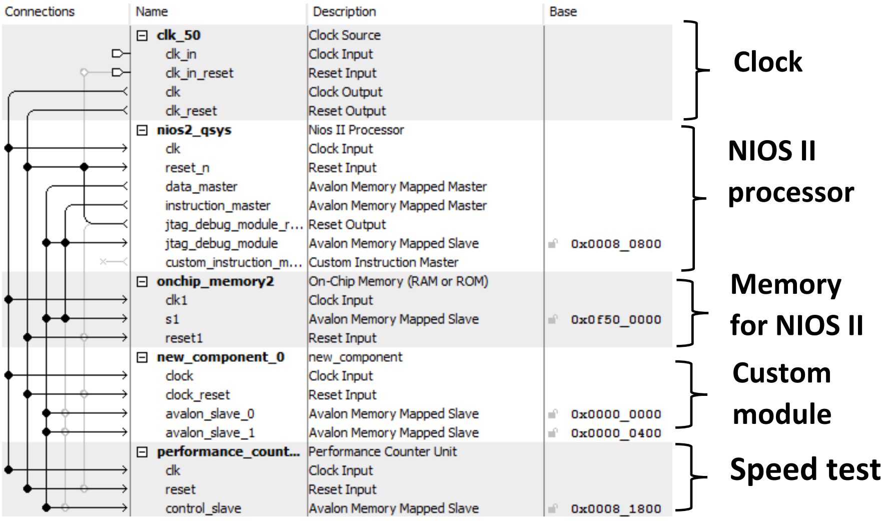


Figure 5.6: Custom module



Connections	Name	Description	Base
	⊟ **clk_50**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **nios2_qsys**	Nios II Processor	
	clk	Clock Input	
	reset_n	Reset Input	
	data_master	Avalon Memory Mapped Master	
	instruction_master	Avalon Memory Mapped Master	
	jtag_debug_module_r...	Reset Output	
	jtag_debug_module	Avalon Memory Mapped Slave	0x0008_0800
	custom_instruction_m...	Custom Instruction Master	
	⊟ **onchip_memory2**	On-Chip Memory (RAM or ROM)	
	clk1	Clock Input	
	s1	Avalon Memory Mapped Slave	0x0f50_0000
	reset1	Reset Input	
	⊟ **new_component_0**	new_component	
	clock	Clock Input	
	clock_reset	Reset Input	
	avalon_slave_0	Avalon Memory Mapped Slave	0x0000_0000
	avalon_slave_1	Avalon Memory Mapped Slave	0x0000_0400
	⊟ **performance_count...**	Performance Counter Unit	
	clk	Clock Input	
	reset	Reset Input	
	control_slave	Avalon Memory Mapped Slave	0x0008_1800

Figure 5.7: Connections in NIOS II approach

5.3 Experiments

All experiments are conducted using the DE1-SoC board which has the Cyclone V chip [8]. Fig. 5.8 shows the DE1-SoC board. Five different PSO algorithms were investigated in the experiments included three proposed PSO algorithms (wPSOd_CV, PSOseed, PSOseed2), the SPSO algorithm, and the DPSO algorithm (presented in chapter 2). The DPSO is chosen in the experiments because this is also an improved version of the SPSO that keeps the particle avoid the premature convergence without adding many compute-intensive tasks to the SPSO.

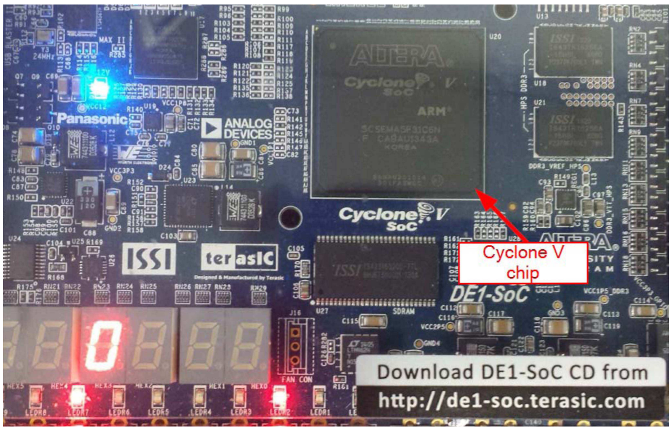


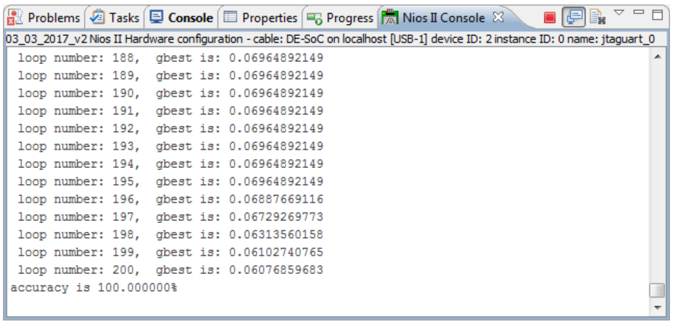
Figure 5.8: DE1-SoC board

Based on the experimental results, a set of parameters which obtained high recognition rates of the NN and low learning errors of the PSO algorithms was as follows.

- $w = 0.9$, $c_1 = c_2 = 0.5$ in the SPSO algorithm.

- w reduced from 0.9 to 0.00001, $c_1 = c_2 = 0.5$, $c_3 = 0.00001$ in the wPSOd_CV algorithm.
- $w = 0.9$, $c_1 = c_2 = 0.5$, $c_3 = 0.3$ in the PSOseed and the PSOseed2 algorithms.
- $c_v = 0.0$, $c_l = 0.001$ in the DPSO algorithm (similar to previous research [9]).

The parameters for the PSO training such as the number of particles or the number of iterations were changed in each experiment to investigate the operations of five PSO algorithms in different situations. The experimental results can be observed using the NIOS II console as can be seen in Fig. 5.9.



03_03_2017_v2 Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtaguart_0

```
 loop number: 188,  gbest is: 0.06964892149
 loop number: 189,  gbest is: 0.06964892149
 loop number: 190,  gbest is: 0.06964892149
 loop number: 191,  gbest is: 0.06964892149
 loop number: 192,  gbest is: 0.06964892149
 loop number: 193,  gbest is: 0.06964892149
 loop number: 194,  gbest is: 0.06964892149
 loop number: 195,  gbest is: 0.06964892149
 loop number: 196,  gbest is: 0.06887669116
 loop number: 197,  gbest is: 0.06729269773
 loop number: 198,  gbest is: 0.06313560158
 loop number: 199,  gbest is: 0.06102740765
 loop number: 200,  gbest is: 0.06076859683
accuracy is 100.000000%
```

Figure 5.9: NIOS II console

In our experiments, two publicly recognized databases were employed called the heart disease dataset and the iris dataset.

5.3.1 Performances of he NN trained by PSO algorithms

To observe the operation of the NN trained by PSO algorithms in the proposed co-design architecture using the NIOS II processor, the data in each database were randomly divided into two different sets to conduct the cross-validation. When the first set was used as the training data, the second set was considered as the testing data, and vice versa. The full list of the experiments is shown in Table 5.1.

Table 5.1: List of experiments conducted with NIOS II approach

Dataset	Features
Heart disease dataset	13 attributes, 2 classes Cross validation data: 100 samples in set 1, 80 samples in set 2 (presented in section 5.3.1.1)
Iris dataset	4 attributes. 3 classses Cross validation data: 105 samples in set 1, 45 samples in set 2 (presented in section 5.3.1.2)

5.3.1.1 Heart disease database

The heart disease database contains the information to predict whether a patient has the heart disease or not (two classes). Each sample in this data set consists of thirteen attributes of four different types called real, ordered, binary, and nominal [10]. With this database, 100 data samples were chosen randomly as set 1, and other 80 samples were selected randomly as set 2. The conducted experiments used one-hidden-layer NN which had 13 input nodes corresponded to 13 attributes, 2 output nodes corresponded to 2 classes, and 24 hidden nodes. The cross-validation was carried out. When set 1 was used as the training set, set 2 became the testing set and vice versa.

Scenario 1: in this scenario, the training data were set 1, and 80 samples of set 2 were chosen as the testing data.

The first measurement of the NN trained by three proposed PSO algorithms (wPSOd_CV, PSOseed, and PSOseed2) and two PSO algorithms presented in previous studies (SPSO, DPSO) in the co-design architecture was the *Gbest*, the minimum value of the learning error. Fig. 5.10 illustrates the reduction curves of the *Gbests* of all five PSO algorithms when the parameters of the PSO training were 10 particles, 230 iterations. After the training phase, the *Gbest* of the NN trained by PSOseed2 algorithm decreased to the lowest value. The full list of the experimental results is shown in Table 5.2. At iteration 230, the final *Gbest* of the NN trained by the PSOseed2 algorithm was 0.1060 while the learning errors of the NN trained by PSOseed, wPSOd_CV, DPSO, and SPSO were 0.1111, 0.1243, 0.1271, and 0.1497, respectively.

The second measurement for the efficiency of the NN trained by different PSO algorithms was the recognition rate of the NN. The full list of the recognition rates is also shown in Table 5.2. Similar to the learning errors, the NN trained by PSOseed2 algorithm also obtained the highest recognition rate at 96.25% when compared to the NN trained by other four PSO algorithms (SPSO, DPSO, wPSOd_CV, and PSOseed).

Table 5.2: Results with heart disease dataset, 100 training samples, 80 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1497	0.1271	0.1243	0.1111	0.1060
Recognition rate	78.75%	85.00%	86.25%	92.50%	96.25%

Experimental results of this scenario confirmed the efficiency of the NN trained by the PSOseed2 algorithm concerning the recognition rate and the learning error.

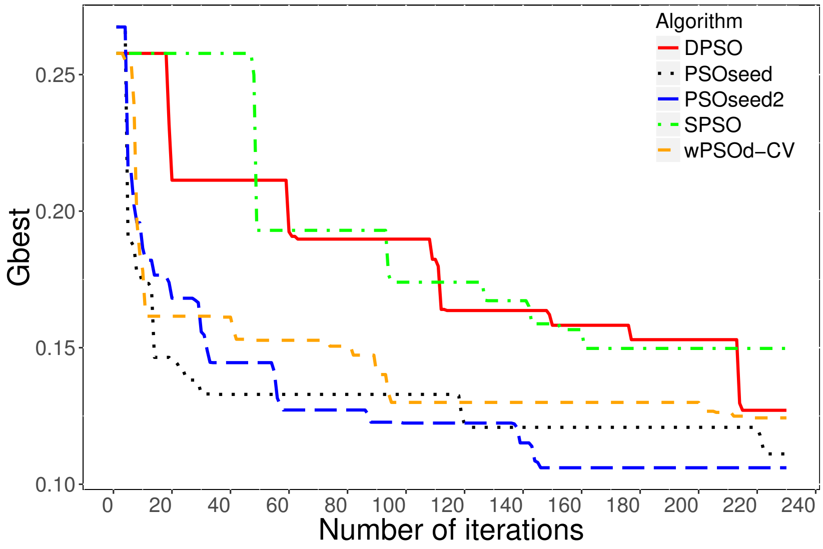


Figure 5.10: Reduction of *Gbest* with heart disease dataset, 100 training samples, 80 testing samples

Scenario 2: to conduct the cross-validation, set 2 was used as the training data in the second scenario, and set 1 became the testing set. In this situation, the number of particles was 25 particles, and the number of iterations was 135 iterations. This scenario observed the operation of the NN trained by PSO algorithms when the number of training data was lower than the number of testing data.

The reduction of the fitness values *Gbest* of the NN trained by five different PSO algorithms is presented in Fig. 5.11. The learning error of the FPGA-based NN trained by the software-based PSOseed2 algorithm once again had the lowest value.

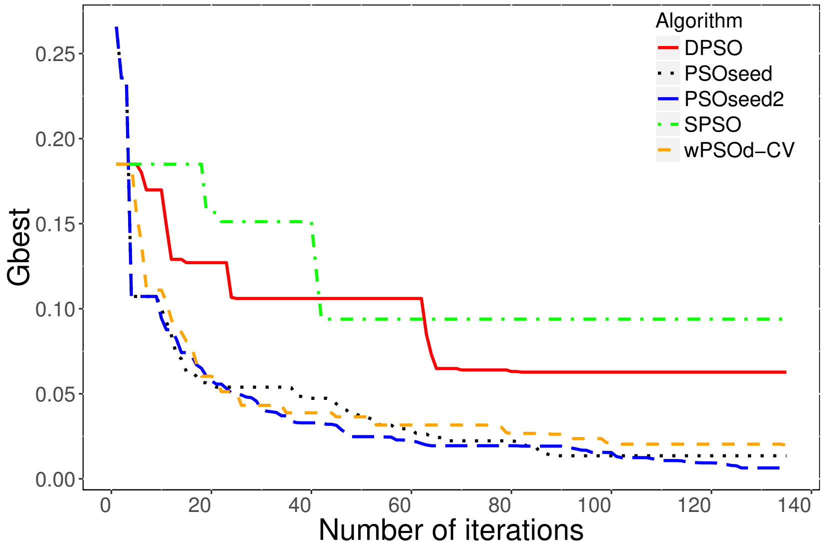


Figure 5.11: Results with heart disease dataset, 80 training samples, 100 testing samples

As illustrated in Table 5.3, the NN trained by the PSOseed2 algorithm also achieved the higher recognition rate (78.00%) than the NN trained by other PSO algorithms (76% with the PSOseed algorithm, 73.00% with the wPSOd_CV and the DPSO algorithm, and 72% with the SPSO algorithm).

Table 5.3: Results with heart disease dataset, 80 training samples, 100 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.0939	0.0627	0.0198	0.0136	0.0064
Recognition rate	72.00%	73.00%	73.00%	76.00%	78.00%

Experimental results with the cross-validation suggested that the PSOseed2 algorithm could be a potential solution for the improvement of the SPSO algorithm for the training of the NN.

5.3.1.2 Iris database

The iris database has 150 samples of three different flowers that are iris Setosa, iris Versicolour, and iris Virginica. Each sample in this dataset contains four attributes called the sepal width, the sepal length, the petal width, and the petal length, respectively [10]. To conduct the experiment with this dataset, the NN has four input nodes, three output nodes, ten nodes in each hidden layer, and two hidden layers. Four input nodes of this NN correspond to four attributes of the iris database, and three output nodes were chosen because the iris database has three classes. Using this database, 105 samples were selected randomly as set 1, and 45 remaining samples were considered as set 2.

Scenario 1: In the first test, the smaller dataset (set 2 with 45 samples) was the training data, 105 samples of set 1 were selected as the testing data.
Fig. 5.12 and Table 5.4 demonstrate the experimental results of this scenario. The settings for all PSO algorithms were 40 particles, and 270 iterations. As can be observed, the NN trained by the PSOseed2 algorithm obtained the lowest learning error at 0.0105 among the NNs trained by five PSO algorithms. Concerning the recognition rate, the NN trained by the proposed PSOseed2 algorithm and the proposed PSOseed algorithm got the highest percentage of the correct recognition at 96.19% after 270 iterations.

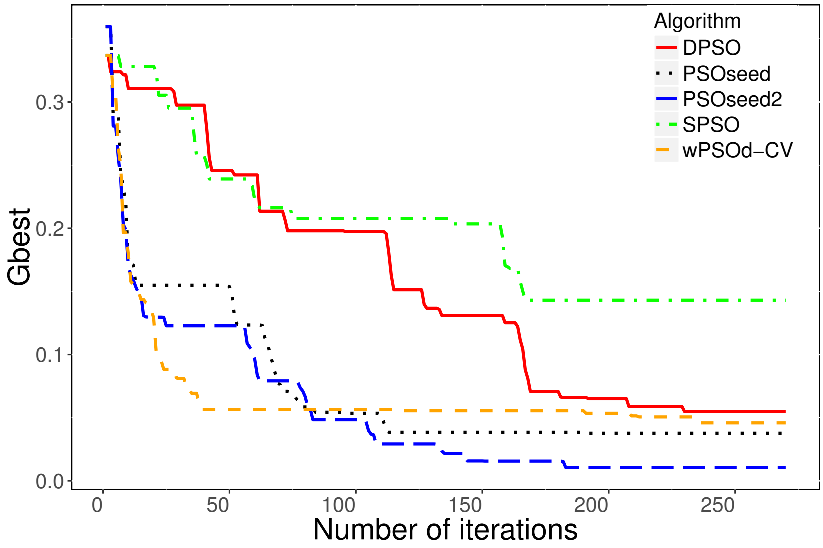


Figure 5.12: Reduction of *Gbest* with iris dataset, 45 training samples, 105 testing samples

Table 5.4: Results of FPGA-based NN with iris dataset, 45 training samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1430	0.0548	0.0459	0.0379	0.0105
Recognition rate	87.62%	94.29%	95.24%	96.19%	96.19%

Scenario 2: In the first test, 45 samples of set 2 were the training data, and 105 samples of set 1 were selected as testing data. Thus, in the second scenario, set 1 was chosen as training data, and set 2 was the testing data for the cross-validation.

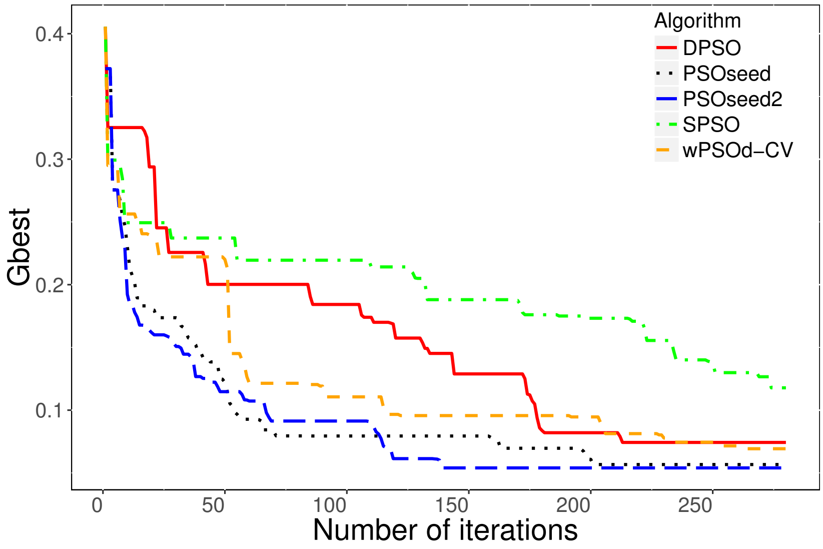


Figure 5.13: Reduction of *Gbest* with Iris dataset 105 training samples, 45 testing samples

The learning curve of the NN trained five PSO algorithms, or the reduction of the learning error *Gbest*, is presented in Fig. 5.13. The configurations of the PSO algorithms were 20 particles, 280 iterations. In this scenario, the learning error of the NN trained by the PSOseed2 algorithm decreased to the lowest value at iteration 280. As presented in Table 5.5, the final *Gbest* of the NN trained by the PSOseed2 algorithm was 0.0540 while the final *Gbests* of the NN trained by other PSO algorithms were 0.0567 (PSOseed algorithm), 0.0693 (wPSOd_CV algorithm), 0.0743 (DPSO algorithm), and 0.1178 (SPSO algorithm).

Concerning the recognition rate, the NN trained by all improved PSO versions (DPSO, wPSOd_CV algorithm, PSOseed, and PSOseed2) may obtain 100% while the NN trained by the standard version of PSO has only 88.89% of the recognition rate.

Table 5.5: Results of Iris dataset, 105 training samples, 45 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1178	0.0743	0.0693	0.0567	0.0540
Recognition rate	88.89%	100.00%	100.00%	100.00	100.00%

The results of the cross-validation show that the FPGA-based NN trained by the software-based PSOseed2 implemented by the NIOS II processor obtained the highest recognition rate and the lowest learning error not only with the small number of training samples but also with the large number of training samples.

5.3.2 Operations of the proposed co-design architecture using NIOS II processor

5.3.2.1 Operating Speed

The operating speed of the co-design architecture using NIOS II processor was also investigated. In this experiment, the testing speed of the proposed co-design architecture was compared with the testing speed of the software-only architecture. Thus, the software-only architecture was also created. The details of the software-only architecture could be seen in appendix A. This software-based NN was implemented by the C language by the Intel core i3 2.4 GHz. The hardware-based NN in the co-design approach is still implemented by the SystemVerilog language. The frequency of the components in the co-design approach could be modified using a phase-locked loop [7]. Thus, the clock frequency of the co-design approach was also increased from 50 MHz (default frequency of the DE1-SoC) to 100 MHz.

To measure the time, the performance counter core was used in the FPGA-based NN of the co-design apporach [7]. The performance counter core can be observed in Fig. 5.7. On the other hand, the `QueryPerformanceCounter()` function was employed in the Intel-based NN approach [11].

In the experiment with the proposed co-design architecture, the sending of the input data to the NN, the receiving of the results from the NN, the printing of the results to the NIOS II console were executed by the NIOS II processor. Therefore, the speed of the NIOS II processor still has the influence to our experimental results.

Table 5.6 shows the testing speed of both two approaches when testing with the heart disease dataset 100 training samples and 80 testing samples. Results showed that even the NIOS II processor operates at the lower clock frequencies than the Intel processor, the hardware-based NN obtained a higher operating speed than the conventional software-based NN.

Table 5.6: The testing time in second

frequency Approach	50 MHz	100 MHz	2.4 GHz
Co-design using NIOS II processor	0.1529	0.0764	-
Intel processor	-	-	0.1815

5.3.2.2 Required resources

To investigate the resource advantage of the proposed co-design when compared to the hardware-only architecture, the required resources in each type of design was also observed. Hence, the hardware-only design was also developed. The details of the hardware-only architecture could be seen in the appendix B.

In this experiment, the FPGA resource is the percentage which can be calculated from the number of the adaptive logic modules (ALMs) used in the design and the total number of ALMs available in the FPGA chip (the Cyclone V).

In our experiments, the required resources for the hardware-only architecture were excessive. Both the NN used to test with the iris dataset and the heart disease dataset cannot be fitted in the FPGA device using the hardware-only approach. In these situations, the compilations were failed, the required resources were over 100% of the Cyclone V.

To observe the required element with other cases when the compilation of the hardware-only architecture was successful, the smaller NN was created. The experimental results were shown in Table 5.7. However, the required resources for the hardware- only were still excessive. The 2-6-4 NN (two input nodes, six hidden nodes, and four output nodes) cannot be implemented in the Cyclone V. Only a smaller NN of size 2-2-2 can be fitted in this FPGA chip (70% of the resources). On the other hand, the co-design architecture with 2-6-4 NN required only 27% of the resources.

Table 5.7: The logic utilization

Approach	Logic utilization
2-6-4 NN hardware-only	>100% (compiles failed)
2-2-2 NN hardware-only	70%
2-6-4 NN hardware-software	27%

The reduction of the logic utilization is explained by the using of the NIOS II processor. In the hardware-only architecture, the PSO module implemented in FPGA requires many resources. On the other hand, the co-design architecture moves the PSO module to the NIOS II processor, and the FPGA resources reserved for this PSO module are not used. Our proposed co-design could address the problem of the limited resources of the hardware-only architecture.

5.4 Discussion

This chapter proposed our first co-design architecture for the training of the NN by the PSO algorithms. In this architecture, the NIOS II processor is used as the processor. All PSO algorithms are executed by the NIOS II processor.

Using the NIOS II processor, all components of the co-design architecture could be implemented in a single FPGA chip which is portable. In addition, the NIOS II processor is already optimized by Altera for the FPGA design.

Experimental results confirmed that the co-design architecture using NIOS II processor was successfully implemented. The results also showed that the NN trained by the proposed PSO algorithms, especially the PSOseed2 algorithms, obtained higher recognition rates and lower learning errors than the NN trained by the PSO algorithms presented in previous studies (SPSO algorithm, DPSO algorithm). The speed advantage of the co-design approach when compared with the software-only approach and the resource advantage of the co-design approach when compared with the hardware-only approach were also demonstrated in our experiments.

However, the using of the NIOS II processor, the softcore processor, has the drawback related the speed because of the constraints with the FPGA fabric. In addition, the NIOS II is created using available resources from the FPGA device which could be the resource-intensive processor, especially, if a high computing power of the NIOS II is required.

Thus, the next chapter of this dissertation proposes the second co-design architecture using another processor to overcome these drawbacks of the NIOS II processor.

References

[1] T. L. Dang, T. Cao, and Y. Hoshino. Hybrid hardware-software architecture for neural networks trained by improved pso algorithm. *ICIC Experss Letters*, 11(3): 565–574, Mar 2017.

[2] T. L. Dang and Y. Hoshino. An-fpga based classification system by using a neural network and an improved particle swarm optimization algorithm. In *2016 Joint 8th International Conference on Soft Computing and Intelligent Systems (SCIS) and 17th International Symposium on Advanced Intelligent Systems (ISIS)*, pages 97–102, Aug 2016.

[3] Altera Corporation. Introduction to the Altera Nios II Soft Processor. 2011.

[4] Altera Corporation. Nios II Classic Processor Reference Guide. 2015.

[5] Altera Corporation. Avalon Interface Specifications . 2017.

[6] Altera Corporation. Floating-Point IP Cores User Guide. 2016.

[7] Altera Corporation. Embedded Peripherals IP User Guide. 2016.

[8] Terasic Technologies Inc. *DE1-SoC User Manual 1 www.terasic.com April 8, 2015.* 2015. ISBN 8863575088.

[9] X. F. Xie, W. J. Zhang, and Z. L. Yang. Dissipative particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC '02*, volume 2, pages 1456–1461, 2002.

[10] M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

[11] Queryperformancecounter function (windows). `https://msdn.microsoft.com`. (Accessed on 06/06/2017).

Chapter 6

Proposed co-design architecture using the ARM processor

Chapter 6 deals with the performances of the NN trained by proposed PSO algorithms in the second co-design architecture that uses the ARM processor. The second co-design architecture is introduced to overcome the drawbacks of the first architecture using the NIOS II processor presented in chapter 5. Several parts of this chapter have been published in our research article [1].

6.1 Introduction

As discussed in chapter 4, the FPGA-based NN and the software-based PSO algorithms are used in the proposed partitioning methodology. The hardware-based NN is implemented using SystemVerilog programming language. The PSO algorithms executed by a processor.

In chapter 5 of this thesis, first co-design architecture using the NIOS II processor is introduced. However, the NIOS II is a softcore processor which is implemented using the available resources concerning the logic elements and the memory bits of the FPGA device. In addition, the softcore processor also has the speed limit because of the constraints with the FPGA fabric. Recently, the hardcore processor also has been developed. Unlike the softcore processor, the hardcore processor is physically implemented as a structure on the FPGA board. Compared to the softcore processor, the hardcore processor has advantages in terms of:

1. Speed: the hardcore processor may operate at a higher clock frequency than the softcore processor.

2. Required logic elements and memory bits: the resources previously reserved for the softcore processor such as the NIOS II processor will not be used in case of using the hardcore processor.

The ARM processor is one of the most famous hardcore processors used with the FPGA device. Even operating in the low-power consumption mode, the ARM processor could still get high performance. In addition, the NIOS II approach presented in chapter 5 is designed to use for the Altera device family while the ARM processor can be freely used with the device from other companies.

The main contribution of this chapter is to introduce the co-design approach for the training of the NN by PSO algorithms using ARM as the processor.

To evaluate the operation of the co-design and to confirm the efficiency of three proposed PSO algorithms. Five different PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed, and PSOseed2) were used to train the NN.

This chapter is presented as follows. Section 6.2 introduces the co-design architecture using the ARM processor. The experimental results are given in section 6.3. Section 6.4 is dedicated to the conclusion of this chapter.

6.2 Proposed architecture

6.2.1 Overview

The overview of the co-design approach using ARM processor is shown in Fig. 6.1. The ARM processor is inside a hard processor system (HPS). This research uses the Cyclone V SoC hard processor system provided by Altera [2]. A Linux operating system is installed on the ARM processor to implement the PSO algorithms.

Inside the HPS used in this research, there are three main components as follows.

1. SD card: the Linux file system is stored on an SD card. The FPGA board will boot Linux from this SD card. In addition, the configuration data of the FPGA components are also kept in the form of the raw binary file (.rbf) in the SD card.

2. ARM processor: the PSO algorithms are implemented by this processor.

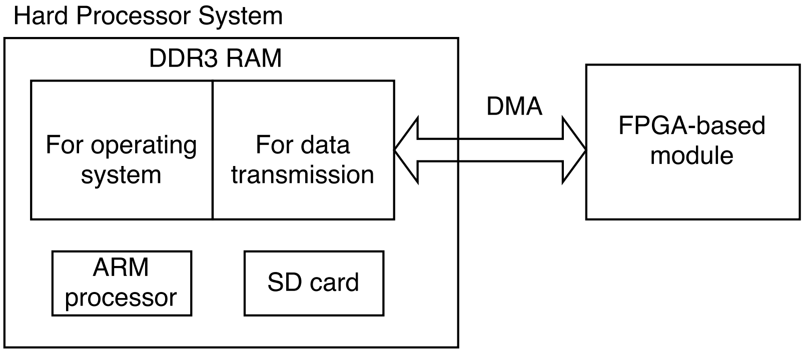


Figure 6.1: ARM apporach for the NN trained by PSO algorithms

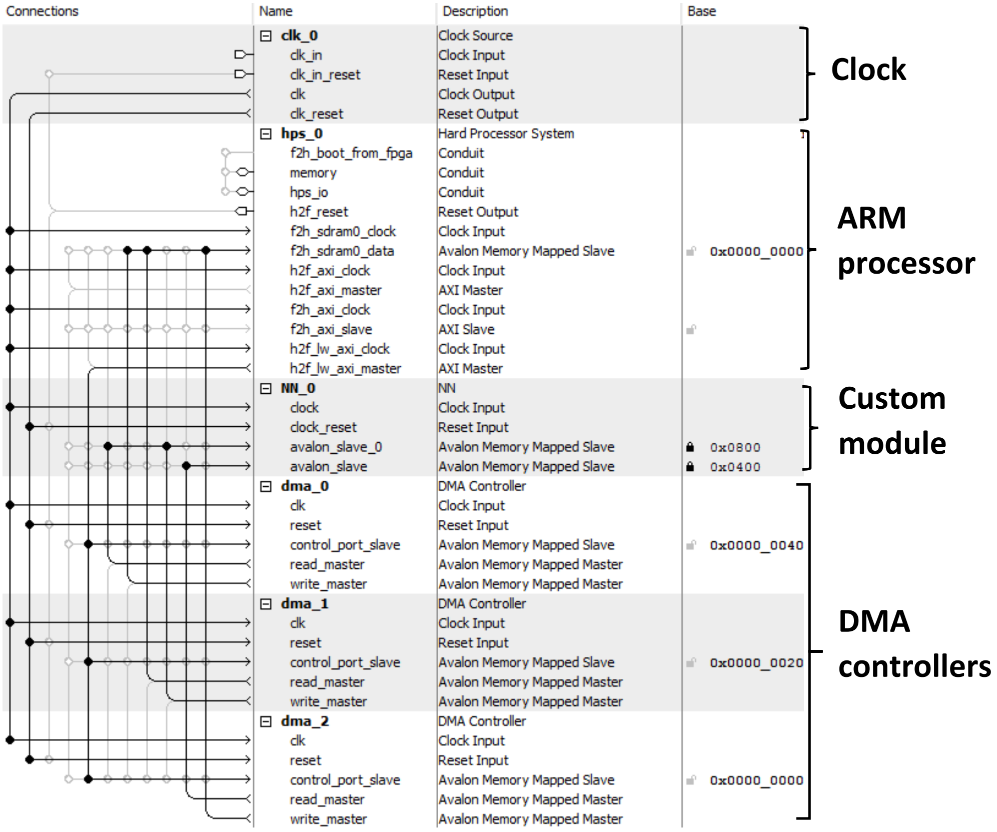
3. DDR3 random-access memory (RAM): to reduce the required memory bits, this approach uses 1 GB DDR3 RAM which is divided into two equal sections. The first section which has 512 MB is reserved for the Linux operating system. The other 512 MB section is used for the data transmission between the hardware side and the software side. All weights, biases of the NN, input data sent to the NN, and the output data received from the NN are put in this section.

The co-design architecture has three main components called the PSO training in software, the FPGA-based module, and the connections between software side and hardware side. The PSO training in software has been discussed in section 4.3 of chapter 4. Each step of the training is also described in Fig. 4.2 of this chapter. The FPGA-based module has been introduced in section 5.2.2 of chapter 5. This hardware-based module has not only the FPGA-based NN but also the floating-point submodule. The NN and the floating-point submodule work in parallel. The synchronization between the NN and the floating-point submodule is already shown in Fig. 5.3 of chapter 5.
Hence, this chapter focuses on the connection between the FPGA-based module and the ARM processor.

6.2.2 Connections between the software side and the hardware side

To increase the connection speed between the software side and the hardware side, this approach employs three direct memory access (DMA) controllers [3]. The DMA controllers are configured by the ARM processor. In this architecture, the data were sent from the FPGA-based module to the data transmission section of the DDR3 RAM and vice versa. Thus, to start the DMA transfer, the ARM processor will send the address of the data transmission section in RAM and the address of the FPGA-based module to the DMA controller.

To connect with the DMA, an Avalon memory-mapped slave interface [4] is created in the FPGA-based module. The whole system with the connections can be observed in Fig.6.2.



Connections	Name	Description	Base
	⊟ **clk_0**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **hps_0**	Hard Processor System	
	f2h_boot_from_fpga	Conduit	
	memory	Conduit	
	hps_io	Conduit	
	h2f_reset	Reset Output	
	f2h_sdram0_clock	Clock Input	
	f2h_sdram0_data	Avalon Memory Mapped Slave	0x0000_0000
	h2f_axi_clock	Clock Input	
	h2f_axi_master	AXI Master	
	f2h_axi_clock	Clock Input	
	f2h_axi_slave	AXI Slave	
	h2f_lw_axi_clock	Clock Input	
	h2f_lw_axi_master	AXI Master	
	⊟ **NN_0**	NN	
	clock	Clock Input	
	clock_reset	Reset Input	
	avalon_slave_0	Avalon Memory Mapped Slave	0x0800
	avalon_slave	Avalon Memory Mapped Slave	0x0400
	⊟ **dma_0**	DMA Controller	
	clk	Clock Input	
	reset	Reset Input	
	control_port_slave	Avalon Memory Mapped Slave	0x0000_0040
	read_master	Avalon Memory Mapped Master	
	write_master	Avalon Memory Mapped Master	
	⊟ **dma_1**	DMA Controller	
	clk	Clock Input	
	reset	Reset Input	
	control_port_slave	Avalon Memory Mapped Slave	0x0000_0020
	read_master	Avalon Memory Mapped Master	
	write_master	Avalon Memory Mapped Master	
	⊟ **dma_2**	DMA Controller	
	clk	Clock Input	
	reset	Reset Input	
	control_port_slave	Avalon Memory Mapped Slave	0x0000_0000
	read_master	Avalon Memory Mapped Master	
	write_master	Avalon Memory Mapped Master	

Figure 6.2: Whole system in ARM approach

The operations of the DMAs can be seen in Fig. 6.3. The DMA0 and DMA1 are used for the transmission of the data between the hardware side and the software side. The DMA2 is reserved for the $flag$ signal. This DMA always sends the current state of the FPGA-based module to the ARM processor. During the operation, if the ARM processor needs to conduct the data transmission with the FPGA-based module, the $flag$ signal will be investigated by the ARM processor. The $flag$ signal sent from the FPGA-based module to the software side using DMA2 is kept at the address 4096 of the DDR3 RAM.

All DMAs in this architecture use the Avalon-MM interface. As shown in Fig. 6.2, The DMA0 and DMA1 connect to the *avalon_slave_0* port, and the DMA2 connects to the *avalon_slave* port. The connections between the RAM inside the HPS and the DMA use the $f2h\_sdram$ port.

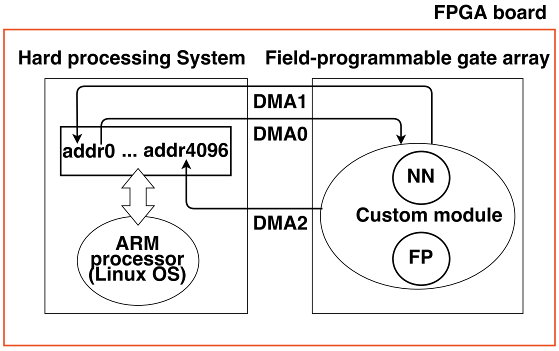


Figure 6.3: Using of Direct Memory Acces (DMA)

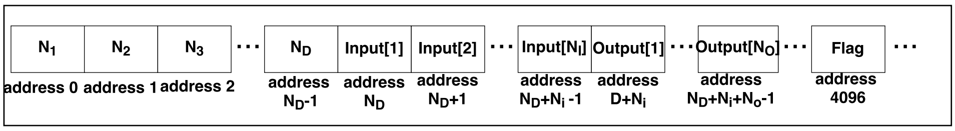


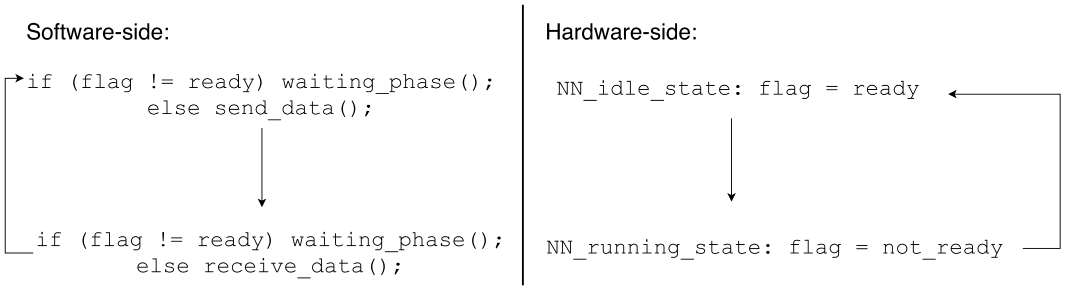
Figure 6.4: The data transmission section

N_D parameters (weights and biases) and the input data of the NN are stored in the data transmission section of the RAM with the corresponding addresses as can be seen in Fig. 6.4. As introduced in chapter 2, $N_D = N_{D1}$ if one-hidden-layer NN is used, $N_D = N_{D2}$ if two-hidden-layer NN is used. The calculations of D_1 and D_2 are already presented in Eqs. (2.3) and (2.4).

The first parameter of the NN (N_1) is put at address 0, the last parameter N_D is stored at address N_{D-1}. In this case, N_1 to N_D is the N_D components of the N_D-dimensional vector which is presented in Fig. 2.4 of chapter 2. These N_D parameters are the N_D weights and biases of the NN.

In a similar way, the first input is stored at address N_D, the last input is put at address $N_D + N_I - 1$ (N_I is the number of nodes in the input layer). The output data from the NN are stored from address $N_D + N_I$ to address $N_D + N_I + N_O - 1$ (N_O is the number of nodes in the output layer. Two vectors $\overrightarrow{input}$ and $\overrightarrow{output}$ are the N_I input data sent to the NN and the N_O output data received from the NN, respectively. The $flag$ is used for timing control. The transmission of the $flag$ uses the DMA2. The operation of the $flag$ is illustrated in Fig. 6.5.

On the software side: when the ARM processor (the software-side) needs to use the FPGA-based NN module (the hardware side), it will check the $flag$ stored at address



```
if (flag != ready) waiting_phase();
        else send_data();

if (flag != ready) waiting_phase();
        else receive_data();
```

```
NN_idle_state: flag = ready

NN_running_state: flag = not_ready
```

Figure 6.5: Timing control by using *flag*

4096 of the RAM. If the *flag* is not ready ($flag \neq ready$), the ARM processor is put in the waiting phase. Otherwise, the ARM processor will send the data to hardware side. After sending the data, the ARM processor changes the value of the *flag* ($flag = not\_ready$), and continues to recheck this flag. If the *flag* is still not ready ($flag = not\_ready$), the ARM will wait for the data from the hardware side. Otherwise, the RAM processor will collect the output data in the data transmission section of the RAM.
On the hardware side: The operation of the FPGA-based NN is based on a finite-state machine which has two main states. In the initial phase, the finite-state machine of the NN is in the NN_ilde_state. In this state, the *flag* is set to ready ($flag = ready$). After receiving the data from ARM processor, the finite-state machine of the NN changes to $NN\_running\_state$. In this state, the *flag* is set to not ready ($flag = not\_ready$). The NN will operate in this $NN\_running\_state$. This state has smaller states to calculate the output of each node in each layer of the NN according to Eqs. (2.1) and (2.2). Finishing the calculation, the output data are also sent to the RAM using DMA. Finishing the data transmission, the finite-state machine of the NN returns to NN_ilde_state and sets the *flag* to ready ($flag = ready$).

6.3 Experiments

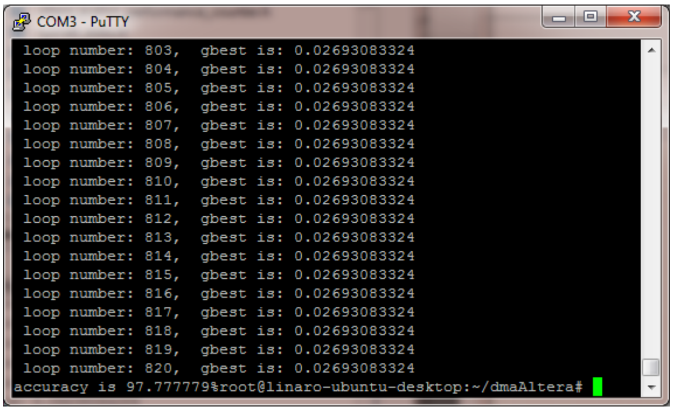
The experiments presented in this chapter were conducted with the Cyclone V SoC chip that has an ARM-based hard processor system.
In the experiments, the FPGA-based NN was trained with five different PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed, and PSOseed2).
Two different types of experiments were conducted. The first type was used to investigate the performances of the PGA-based NN trained by five different software-based PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed, and PSOseed2) using the ARM

processor. The second experiment type was employed to observe the operations of the co-design architecture when compared with the software-only architecture concerning the operating speed, and hardware-only architecture in terms of the required resources. In our experiments, the parameters for the PSO algorithms were similar to the parameters of the experiments presented in chapter 5. The details of the parameters were as follows.

- $w = 0.9$, $c_1 = c_2 = 0.5$ in the SPSO algorithm.
- w reduces from 0.9 to 0.00001, $c_1 = c_2 = 0.5$, $c_3 = 0.00001$ in the wPSOd_CV algorithm.
- $w = 0.9$, $c_1 = c_2 = 0.5$, $c_3 = 0.3$ in the PSOseed and the PSOseed2 algorithm
- $c_v = 0.0$, $c_l = 0.001$ in the DPSO algorithm (similar to previous research [5])

The number of particles and the number of iterations were varied to observe the operation of the NN trained by PSO algorithms in different situations.

In the experiments, the PSO algorithms were coded by C programming language by ARM processor, and the NN was implemented in FPGA by SystemVerilog programming language. The output data can be observed using Putty software as shown in Fig. 6.6.



```
 loop number: 803,  gbest is: 0.02693083324
 loop number: 804,  gbest is: 0.02693083324
 loop number: 805,  gbest is: 0.02693083324
 loop number: 806,  gbest is: 0.02693083324
 loop number: 807,  gbest is: 0.02693083324
 loop number: 808,  gbest is: 0.02693083324
 loop number: 809,  gbest is: 0.02693083324
 loop number: 810,  gbest is: 0.02693083324
 loop number: 811,  gbest is: 0.02693083324
 loop number: 812,  gbest is: 0.02693083324
 loop number: 813,  gbest is: 0.02693083324
 loop number: 814,  gbest is: 0.02693083324
 loop number: 815,  gbest is: 0.02693083324
 loop number: 816,  gbest is: 0.02693083324
 loop number: 817,  gbest is: 0.02693083324
 loop number: 818,  gbest is: 0.02693083324
 loop number: 819,  gbest is: 0.02693083324
 loop number: 820,  gbest is: 0.02693083324
accuracy is 97.777779%root@linaro-ubuntu-desktop:~/dmaAltera#
```

Figure 6.6: Putty software

6.3.1 Performances of the NN trained by PSO algorithms

Three different publicly recognized databases were used to investigate the operations of the NN trained by five PSO algorithms. The cross-validation was tested for each database. The full list of the databases can be seen in Table 6.1.

Table 6.1: List of experiments conducted with ARM approach

Dataset	Features
Wine dataset	13 attributes, 3 classes Cross validation data: 120 samples in set 1, 58 samples in set 2 (presented in section 6.3.1.1)
Australian credit dataset	14 attributes, 2 classes Cross validation data: 490 samples in set 1, 200 samples in set 2 (presented in section 6.3.1.2)
Iris dataset	4 attributes. 3 classses Cross validation data: 45 samples in set 1, 105 samples in set 2 (presented in section 6.3.1.3)

6.3.1.1 Wine dataset

The wine dataset has 178 samples of three different Italian wines. Each sample consists of thirteen attributes which come from the chemical analysis [6]. The cross-validation was done by dividing the data randomly into two different sets. The first set had 120 samples, and other set had 58 samples.

To conduct the experiments with this dataset, the NN had 13 input nodes (for 13 attributes), 22 hidden nodes, and 3 output nodes (for 3 classes of wines).

Scenario 1: Fig. 6.7 and Table 6.2 show the experimental results when 120 samples of set 1 were chosen as the training data, and 58 samples of set 2 were considered as the testing data. The configurations of the PSO algorithms were 15 particles and 800 iterations. The learning error *Gbest* of the NN trained by the PSOseed2 reduced to the lowest value 0.0034 at iteration 800.

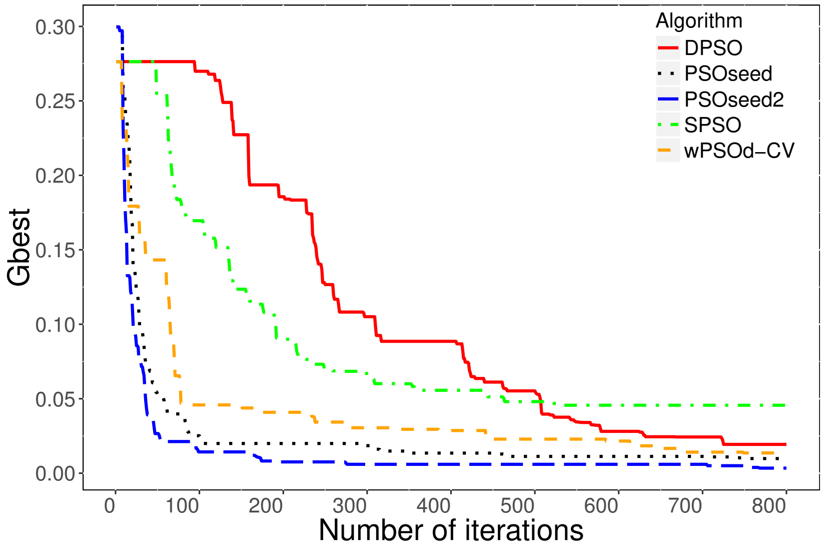


Figure 6.7: Reduction of *Gbest* with wine dataset, 120 training samples, 58 testing samples

Concerning the recognition rates of the NN trained by five PSO algorithms, Table 6.2 showed that at the minimum value of learning error (0.0034), the NN trained by the PSOseed2 algorithm also obtained the highest recognition rate at 98.28%.

On the other hand, as illustrated in Table 6.2, the NN trained by the SPSO algorithm had the highest learning error (0.0456) and the lowest recognition rate (89.66%).

Table 6.2: Results of FPGA-based NN with wine dataset, 120 training samples, 58 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.0456	0.0193	0.0136	0.0098	0.0034
Recognition rate	89.66%	94.83%	96.55%	96.55%	98.28%

Scenario 2: The cross-validation was carried out in this scenario. Set 2 which had 58 samples became the training set, and the 120-samples set was selected as the testing data. The PSO parameters had the configurations of 40 particles and 400 iterations.

Fig. 6.8 illustrates the reduction of the learning error, the *Gbest* value, in each iteration of the training phase. The list of all experimental results concerning the recognition rates and the learning errors of the NN trained by five PSO algorithms is expressed in Table 6.3. Similar to the results in the previous scenario when set 1 was considered as the training data, the high accuracy in terms of learning error and recognition rate of the NN trained by the PSOseed2 algorithm was confirmed by these experimental results.

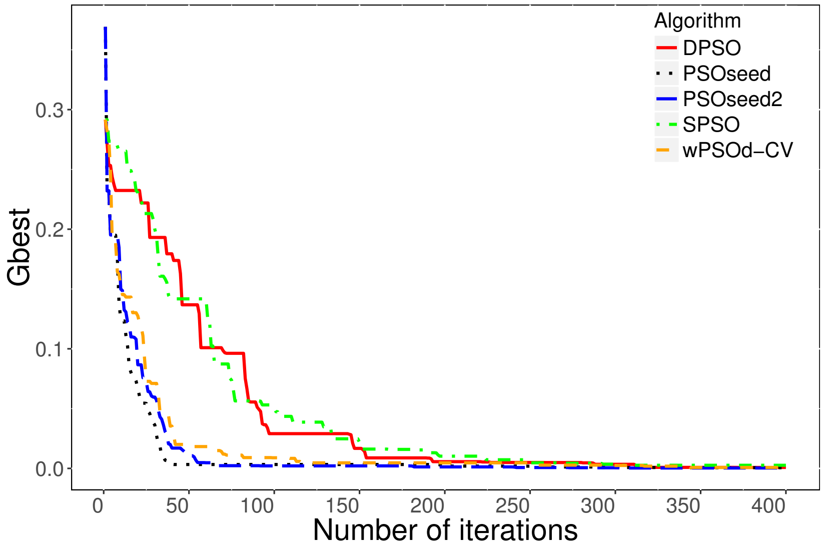


Figure 6.8: Reduction of *Gbest* with Wine dataset with 58 training samples, 120 testing samples

Table 6.3: Results with wine dataset, 58 training samples, 120 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	262×10^{-5}	79×10^{-5}	76×10^{-5}	58×10^{-5}	32×10^{-5}
Recognition rate	86.67%	90.00%	90.00%	90.00%	93.33%

6.3.1.2 Australian credit dataset

The Australian credit dataset was also employed in our experiments to investigate the operations of the hardware-based NN trained by software-based PSO algorithms in a different situation. This dataset comes from the credit card application of the Australian which has two classes called rejection and approval. Each sample of this dataset has fourteen attributes which were modified to protect the confidentiality [6]. With this dataset, 490 samples were included randomly in the set 1, and 200 samples were also selected randomly as set 2.

The cross-validation was conducted with the 14-24-2 NN. In this case, the NN had 14 input nodes corresponding to 14 attributes, 24 hidden nodes, and 2 output nodes corresponding to 2 classes.

Scenario 1: The experimental results when 490-sample set was chosen as training set, and 200-sample set was considered as the testing set are given in both Fig. 6.9 and Table 6.4. The number of particles was 10, and the number of iterations was 250.

Fig. 6.9 shows the reduction curves of the learning errors *Gbest* in this scenario. Among five PSO algorithms, the NN trained by the PSOseed2 algorithm continued to achieve the lowest learning error at iteration 250.

Table 6.4 shows the learning errors and the recognition rates after 250 training iterations. Similar to the learning error, the recognition rate of the NN trained by PSOseed2 algorithm obtained the highest number at 95.50% when compared with the NN trained by other four other PSO algorithms.

Table 6.4: Results with Australian credit dataset, 490 training samples, 200 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1582	0.1533	0.1282	0.1282	0.1244
Recognition rate	90.00%	93.00%	94.50%	94.50%	95.50%

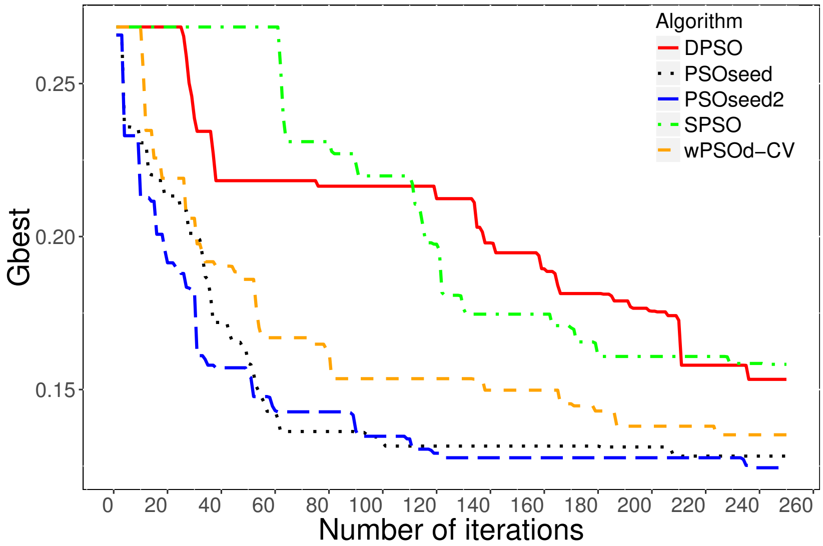


Figure 6.9: Reduction of *Gbest* with Australian credit dataset, 490 training samples, 200 testing samples

Scenario 2: the second scenario of the cross-validation focused on the small number of training data, and the high number of testing data. Therefore, 200 samples of set 2 became the training data while 490 samples of set 1 were the testing data.

Fig. 6.10 and Table 6.5 demonstrate the results of the experiment with 5 particles, 850 iterations. Compared with other PSO algorithms, the NN trained by PSOseed2 algorithm got the smallest learning error (0.0135) and obtained the best recognition rate (82.86%).

These results once again expressed that the PSOseed2 algorithm could be a potential solution to improve the SPSO algorithm in the training of the FPGA-based NN.

Table 6.5: Results with Australian credit dataset, 200 training samples, 490 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.0635	0.0286	0.0184	0.0164	0.0135
Recognition rate	80.82%	81.84%	82.65%	82.65%	82.86%

With the cross-validation, the NN trained by the PSOseed2 algorithm obtained the highest recognition rate and the lowest *Gbest* not only with the high number of training samples (490 samples) but also with the small number of the training samples (200 samples). In addition, the high recognition rate and the low *Gbest* of the NN trained by the PSOseed2 algorithm were also observed even with the small number of the particles (5 particles).

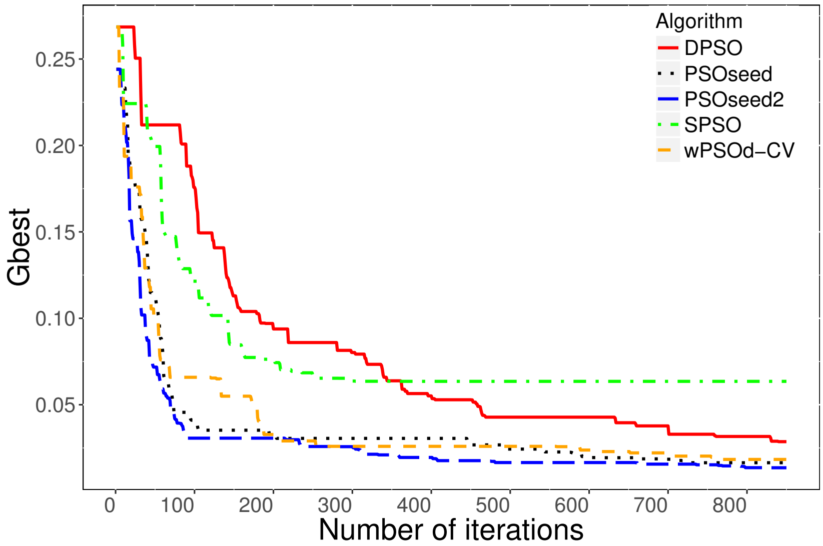


Figure 6.10: Reduction of *Gbest* with Australian credit dataset, 200 training samples, 490 testing samples

6.3.1.3 Iris dataset

The iris dataset which has four attributes, three classes was also evaluated [6]. In this situation, the configuration of the NN was four input node, ten hidden nodes in each hidden layer, two hidden layers, three output nodes. The cross-validation between 45 samples randomly selected as set 1 and 105 samples randomly chosen as set 2 was done.

Scenario 1: When set 1 used as training data, set 2 used as testing data, the PSO algorithms had 25 particles and 500 iterations.

The learning curves of the NN trained by five PSO algorithm are presented in Fig. 6.11. The final values of the learning errors and the recognition rates are illustrated in Table 6.6. As can be seen, the NN trained by all three proposed PSO algorithms (wPSOd_CV, PSOseed, PSOseed2) had higher recognition rates when compared with the NN trained by two other PSO algorithms presented in previous studies (94.29% with the DPSO algorithm, and 91.43% with the SPSO algorithm).

Concerning the errors of the learning phase, the NN trained by the PSOseed2 algorithm got the lowest learning error (0.0123) while the NN trained by the SPSO algorithm had the highest learning error (0.0702).

Scenario 2: For the cross-validation, 105 samples of set 2 were chosen as the training data, 45 samples of set 1 were selected as the testing data in this scenario.

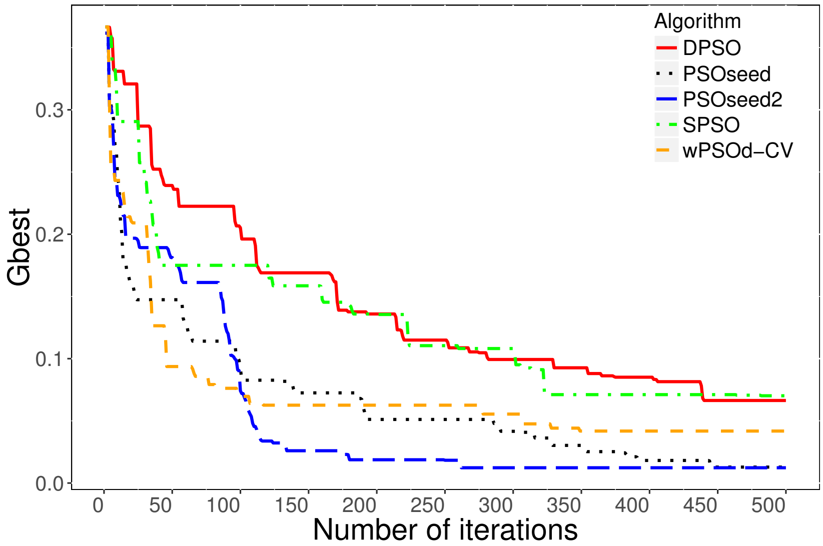


Figure 6.11: Results with Iris dataset, 45 training samples, 105 testing samples

Table 6.6: Results with Iris dataset, 45 training samples, 105 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.0702	0.0664	0.0418	0.0128	0.0123
Recognition rate	91.43%	94.29%	96.19%	96.19%	96.19%

Fig. 6.12 shows the reduction of *Gbest*, the learning errors, of the FPGA-based NN trained by five different algorithms executed by the ARM processor when the 105-samples dataset was used as the training data, and the 45-samples dataset was considered as the testing data. The configurations of the PSO algorithms were 20 particles and 820 iterations. As observed, the recognition rate of the NN trained by both the proposed PSOseed and the proposed PSOseed2 obtained the highest recognition rates. In addition, the learning error *Gbest* of the NN trained by the PSOseed2 algorithm also decreased to the lowest values among five PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed, and PSOseed2).

Table 6.7: Results of FPGA-based NN with iris dataset, 45 training samples, 105 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.0745	0.0450	0.0448	0.0353	0.0269
Recognition rate	95.56%	95.56%	95.56%	97.78%	97.78%

Results presented in this section confirmed the efficiency of the proposed PSO algorithms for the NN training, especially the PSOseed2 algorithm, when compared with the PSO algorithms presented in previous studies.

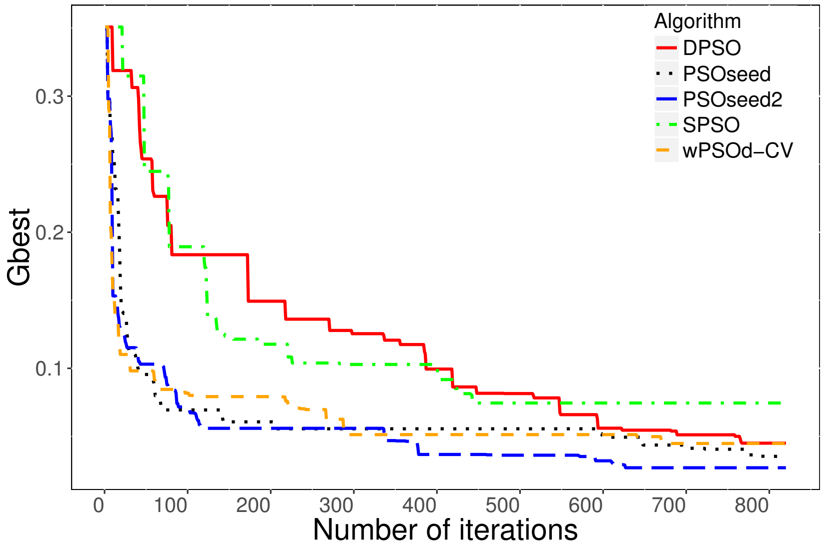


Figure 6.12: Results with iris dataset 105 training samples, 45 testing samples

6.3.2 Operations of the co-design architecture using the ARM processor

6.3.2.1 Operating speed

In this experiment, the testing speed of the FPGA-based NN in the co-design architecture was compared to the testing speed of the software-based NN in the software-only architecture. In this situation, the software-only architecture was also developed in this research. The details of the software-only architecture are shown in appendix A.

There are two different environments for this experiment. The first one is a conventional computer which implements the software-based NN. The second one is the DE1-SoC board which can investigate the operating speed of both software-based NN and hardware-based NN.

To investigate the speed advantage of the co-design architecture using the ARM processor when compared to the co-design architecture using the NIOS II processor presented in chapter 5 of this dissertation. The FPGA-based NN used with NIOS II processor was also included in this experiment.

The list of the experiments can be observed as follows.

1. The conventional computers: both the NN and the PSO algorithms were coded by the C programming language, and the executions of the NN and the PSO were done by an Intel processor. The programs were compiled using Visual Studio 2013 on two different computers. The first computer had Intel core i3 (2 cores, 2.4 GHz)

processor. The second computer was equipped with Intel core i7 (4 cores, 4.0 GHz) processor. The time was measured by $QueryPerformanceCounter()$ function [7].

2. The DE1-SoC board: The frequency of the system could be modified using the Altera phased-locked loop IP cores [3]. Therefore, this experiment observed the operating speed of the system in two different clock frequencies that were 50 MHz (the default clock of the DE1-SoC), and 100 MHz. There are three different scenarios for the DE1-SoC as follows.

   (a) The software-based NN in the NIOS II processor: both the NN and the PSO were coded by the C programming language, the operations of the NN and PSO were executed by the NIOS II processor. The measurement of the time can be investigated using performance counter [3].

   (b) The FPGA-based NN with NIOS II approach: the NN was implemented in the hardware by SystemVerilog programming language, the PSO algorithms which were coded by the C programming language were implemented by the NIOS II processor. This approach also used the performance counter core to investigate the execution time [3].

   (c) The FPGA-based NN with ARM approach: the NN was implemented in the hardware by SystemVerilog programming language, the PSO algorithms which were coded by the C programming language were implemented by the ARM processor. In this situation, the time was measured by *clock_gettime*() function [8].

The experimental results were shown in Table 6.8, and 6.9. The results for the Intel approaches were the average execution time after 15 tests. As can be seen, the speeds of the testing phase of the core i3 approach and core i7 approach were 0.0776 and 0.4523, respectively. The operating speed of the software-based NN using NN processor was 2.1595 seconds (at 50 MHz of the clock frequency of the NIOS II processor) and 1.4681 seconds (at 100 MHz of the NIOS II clock frequency). The operating speed of the software-based NN in NIOS II approach was higher than the software-based NN in Intel approach because the Intel processor could operate at a higher clock frequency than the NIOS II process.

In our experiments, the co-design architectures still need to use the NIOS II processor or the ARM processor to send the testing data to the input of the NN, to receive the processed data from the output of the NN, and to print the results. Therefore, the performance of the co-design architectures was still affected by the NIOS II processor or the ARM processor. However, experimental results confirmed that the FPGA-based NN achieved the lower operating speeds than the software-based NN even the clock frequencies of the NIOS II processor or the ARM processor were much lower than the clock frequencies of the Intel processors.

In the comparison between two FPGA-based NNs, the FPGA-based NN used with the ARM processor obtained the higher performance concerning the testing speed. These results can be explained because the frequency of the ARM processor did not depend on the system clock, and the ARM processor could operate at a higher clock frequency than the NIOS II processor. Another reason is the using the DMA for the data transmission between ARM processor and the FPGA-based module.

Table 6.8: The testing time in second by using DE1-SoC

frequency Approach	50 MHz	100 MHz
Software-based NN implemented in NIOS II	2.1695 seconds	1.4681 seconds
FPGA-based NN with NIOS II approach	0.4350 seconds	0.1714 seconds
FPGA-based NN with ARM approach	0.0406 seconds	0.0278 seconds

Table 6.9: The testing time in second by using Intel processor

frequency Approach	2.4 GHz	4.0 GHz
Software-based NN implemented in Intel core i3	0.4523 seconds	-
Software-based NN implemented in Intel core i7	-	0.0776 seconds

6.3.2.2 Required resources

Another aspect needs to be investigated in the co-design architecture is the required resources. With the goal of comparing the required resources between the co-design architecture and the hardware-only architecture, the hardware-only architecture was developed. The details of the hardware-only architecture can be seen in appendix B.

In the hardware implementation, the implementation of the floating-point intellectual property cores in the floating-point module required a lot of resources. Therefore, two different approaches of the hardware-only were investigated. In the first approach, each of hardware-based NN and hardware-based PSO had its own floating-point module.

Thus, two floating-point modules were implemented. On the other hand, in the second approach, the FPGA-based NN and the FPGA-based PSO shared one common floating-point module using a FPGA-based multiplexer (MUX).

This experiment also checked the required resources of the co-design using NIOS II processor to compare with the required resources of the co-design using ARM processor. Table 6.10 shows the experimental results when the NN was used to test with the iris dataset. In this situation, the size of the NN was four input nodes, ten hidden nodes, and three output nodes.

Experimental results showed that the required resources in hardware-only approach were excessive. The compilations in both scenarios of the hardware-only architecture were failed (more than 100% of the logic utilization). However, the using of the MUX reduced the required digital signal processing (DSP) blocks. This results can be explained because the floating-point intellectual property cores needed to used a lot of logic elements and DSP blocks as described in floating-point IP cores user guide [9].

Results also confirmed that co-design architecture reduced the required logic resources concerning the logic elements and the memory bits compared to the hardware-only architecture. In this situation, the co-design architecture could overcome the limited resources of the hardware-only architecture.

The required resources in the co-design architecture with the ARM processor was also lower than the resources required in the co-design architecture with NIOS II processor. This results can be explained because the resources previously resources for the NIOS II processor were not used in the ARM approach. As presented, the NIOS II processor was implemented using the available FPGA resources while the ARM processor was physically implemented on the FPGA board. Furthermore, the required memory bits of the NIOS II approach was high because this approach used the on-chip memory while the ARM approach employed the DDR3 RAM. In addition, the ARM processor already had the Linux operating system while the operating system did not be implemented in NIOS II processor. In case of using Linux, the required resources for the NIOS II processor will increase with the using of the resources reserved for the memory management unit (MMU).

Table 6.10: Resources Utilization in percentage

Resources / Approach	Logic elements	Memory bits	DSP blocks
Hardware-only without MUX	280%	2%	44%
Hardware-only with MUX	279%	2%	22%
Co-Design with NIOS II	51%	77%	22%
Co-Design with ARM	46%	<1%	22%

6.4 Discussion

This chapter presented the second co-design architecture using the ARM processor. In this architecture, the operations of the PSO algorithms and the FPGA-based module were similar to the operations in the architecture using the NIOS II processor. However, PSO algorithms were executed by the ARM processor which can operate at a higher clock frequency than the NIOS II processor. In addition, the ARM processor is physically implemented on the FPGA board while the NIOS II processor is implemented using the available resources of the FPGA device. Thus, the ARM approach may reduce the required resources of the program.

The co-design architecture using the ARM processors also has two improvements when compared to the NIOS II is the appearances of the DMA and the DDR3 RAM. The DMA may increase the operating speed of the program. On the other hand, the using of the RAM can reduce the required resources of the design.

Experimental results have demonstrated the speed advantage of the co-design with ARM processor when compared with the software-only approach or the co-design with the NIOS II processor. The results also confirmed the resource advantage of the co-design with ARM processors when compared with the hardware-only approach and the co-design using the NIOS II processor.

The operations of the proposed PSO algorithms for the training of the NN in the co-design architecture with ARM processor were also observed. Results showed that the NN trained by the proposed PSO algorithms, especially the PSOseed2 algorithm, got the higher accuracy concerning the learning errors and the recognition rates when compared with the NN trained by other PSO algorithms presented in previous research (SPSO, DPSO).

However, the resource was still an important issue in the proposed architecture. Therefore, the next chapter proposes the third co-design architecture which gives a solution to reduce the required resources.

References

[1] T. L. Dang, T. Cao, and Y. Hoshino. A proposed psoseed2 algorithm for training hardware-based and software-based neural networks. *International Journal of Innovative Computing, Information and Control*, 13(4), Aug 2017.

[2] Altera Corporation. Cyclone V Device Handbook. 2016.

[3] Altera Corporation. Embedded Peripherals IP User Guide. 2016.

[4] Altera Corporation. Avalon Interface Specifications . 2017.

[5] X. F. Xie, W. J. Zhang, and Z. L. Yang. Dissipative particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC '02*, volume 2, pages 1456–1461, 2002.

[6] M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

[7] Queryperformancecounter function (windows). `https://msdn.microsoft.com`. (Accessed on 06/06/2017).

[8] clock_gettime(3): clock/time functions - linux man page. `https://linux.die.net/man/3/clock_gettime`. (Accessed on 06/06/2017).

[9] Altera Corporation. Floating-Point IP Cores User Guide. 2016.

Chapter 7

Proposed NN-PCA architecture using the ARM processor

This chapter presents a method to improve performances of the NN trained by proposed PSO algorithms in the proposed co-design architectures. To reduce the required resources while maintaining the high recognition rate, this chapter proposes the third co-design architecture which is a combination between the co-design architecture using ARM processor introduced in chapter 6 and the FPGA-based principal component analysis (PCA). The performances of the NN trained by the proposed PSO algorithm in the third co-design architecture are also discussed in this chapter.

7.1 Introduction

The co-design architecture using the ARM processor presented in the previous chapter had obtained a higher operating speed and required fewer resources regarding the logic elements and the memory bits than the NIOS II approach. However, the required resources for the FPGA-based NN is still very high. It is beneficial to study the reduction of the size of the hardware-based NN. In this situation, the resource utilization of the design could be reduced.

Our research investigates the PCA to minimize the size of the NN. In the NN-PSO, the PCA is used not only in the training phase but also in the testing phase. Hence, to maintain the operating speed of the testing phase, the PCA needs to be hardware implemented on the FPGA side. However, the conventional PCA algorithm using the matrix calculation will require many resources [1]. This approach is unsuitable for the hardware device.

In previous research, the Generalized Hebbian Algorithm (GHA) was presented [2, 3]. Several researchers have mentioned that the GHA is considered to be one of the best PCA algorithms for the hardware implementation because this algorithm does not use the covariance matrix [4, 5]. For this reason, the GHA is used in our research as the PCA algorithm.

The main contribution in this chapter is to introduce an architecture for the co-design NN trained by PSO algorithms. This architecture is the combination between the architecture presented in chapter 6 and the hardware-based PCA (NN-PCA).

This chapter is presented as follows. Section 7.2 introduces the GHA algorithm. The proposed architecture is presented in section 7.3. The experimental results are shown in section 7.5. Section 7.5 concludes this chapter.

7.2 Generalized Hebbian Algorithm

The concept of the GHA is a single layer NN using the Hebbian learning algorithm. During the training phase, the weights of this NN are modified in each iteration. After a number of iterations, the NN becomes a filter for the principal components, and the i^{th} output of this NN corresponds to the i^{th} principal component of the input data. The GHA is the unsupervised learning. The details and the proof of this assumption have already been presented in previous studies [2, 3].

The outputs of the GHA-based NN can be calculated according to Eq.(7.1).

$$\mathrm{y_j(t)} = \sum_{i=1}^{N_I} w_{ji}(t) x_i(t) \tag{7.1}$$

where N_I is the number of input nodes, N_O is the number of output nodes, $y_j(t)$ is the j^{th} component of the output vector $\vec{y}$ at time t $(0 \leqslant j \leqslant N_O)$, $x_i(t)$ is the i^{th} component of the input vector $\vec{x}$ at time t $(0 \leqslant i \leqslant N_I)$, w_{ji} is the connection weight between i^{th} input and j^{th} output.

The weight update function is based on the Hebbian rule as can be seen in Eq. (7.2) [2, 3]

$$\mathrm{w_{ji}(t+1)} = w_{ji}(t) + \eta \left[y_j(t)x_i(t) - y_j(t) \sum_{k=1}^{j} w_{ki}(t)y_k(t) \right] \tag{7.2}$$

where η is the learning rate.

The weight update function can be rewritten to reduce the calculation task as can be seen in Eq. (7.3).

$$\mathrm{w_{ji}(t+1)} = w_{ji}(t) + \eta y_j(t) \left[x_i(t) - \sum_{k=1}^{j} w_{ki}(t)y_k(t) \right] \tag{7.3}$$

7.3 Proposed architecture

7.3.1 Overview

The proposed NN-PCA architecture still maintains the concept of the co-design discussed in chapter 4 of this thesis. In this architecture, the PSO algorithms are implemented on the software side by the ARM processor. On the other hand, the NN and the PCA are hardware implemented.

Compared with the hardware-only architecture, this partitioning strategy could not only maintain the testing speed of the FPGA-based program but also reduce the FPGA resources previously reserved for the PSO implementation. Compared with the software-only architecture, this co-design could not only maintain the flexibility of the software-based program in the training phase but may also obtain a higher operating speed in the testing phase. In this case, the flexibility relates to the easiness to modify the PSO parameters or even change the PSO algorithms without the need of rebuilding the FPGA part.

Fig 7.1 shows the proposed architecture with both FPGA side and software side. The FPGA side has two main components called the PCA module and the neural network module. The PSO algorithms are implemented by the ARM processor on the software side. There are four connections between these two sides. The *control_signal* and the *input_data* are sent from the software side to the hardware side. On the contrary, the *status_signal* and the *output_data* are sent from the hardware side to the software side. The data on the software side are stored in the DDR3 RAM. When the ARM needs to conduct the operation, these data will be collected from the RAM.

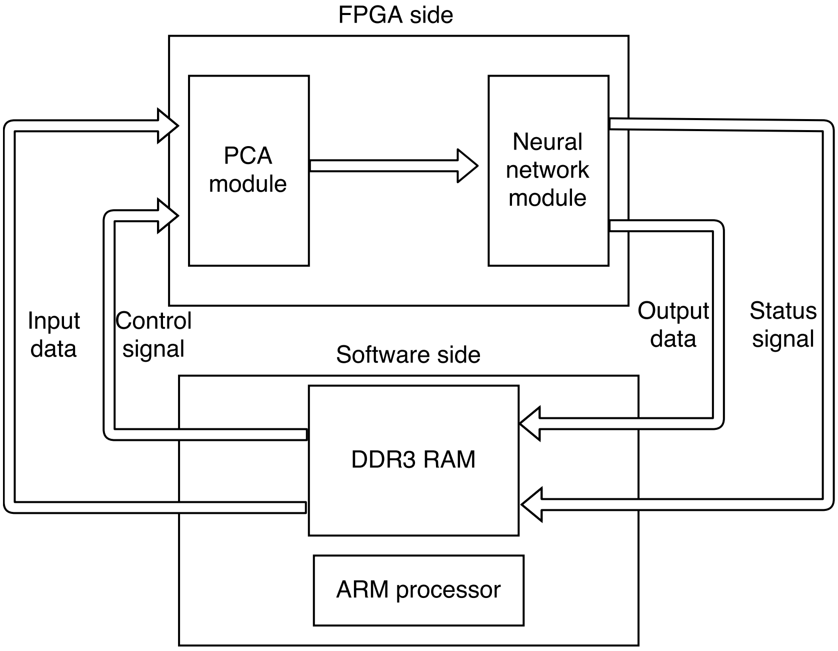


Figure 7.1: Proposed NN-PCA architecture

The operation of this proposed architecture is as follows.

1. The training phase of the PCA: the software side sends the $control\_signal = 1$ to the hardware side along with the trained data. This training phase is the unsupervised learning which does not depend on the output data. The trained PCA weights $\vec{w}$ are stored.

2. The training phase of the NN: the software side sends the $control\_signal = 0$ to the hardware side along with the trained data. In this case, the trained weights are loaded into the PCA module. This is the supervised learning. The output data from the NN are evaluated on the software side using PSO algorithms. Finishing this training phase, the trained weights are kept.

3. The testing phase of the system: the software side sends the $control\_signal = 0$ to the hardware side along with the test data.

The details each step will be discussed in section 7.3.3 of this chapter.

On the hardware side, the processing of the data in each node of the hardware-based NN is based on Eq. (2.1) shown in chapter 2 of this dissertation. To demonstrate, the operation of one node in the NN according to Eq. (2.1) can be expressed as can be seen in Fig. 7.2.

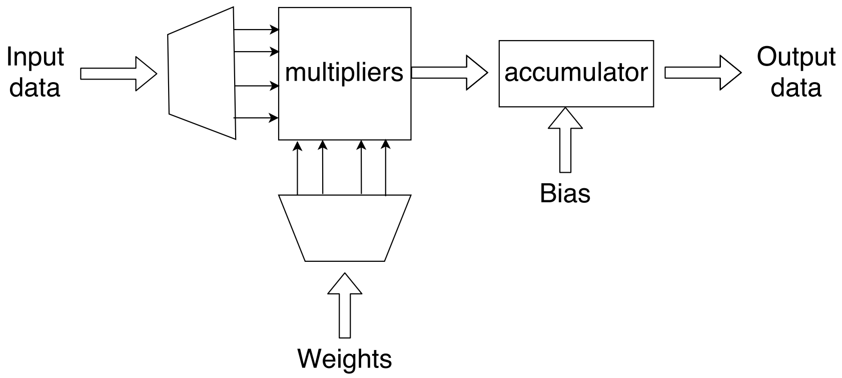


Figure 7.2: Operation of one node in the NN

The NN module and the PCA module are implemented in hardware to take the speed advantage of the FPGA. To conduct the arithmetic operators, both NN module and PCA module connects to another module called ALU which implements the floating-point IP cores. The output latency of each operator is presented in the floating-point IP cores user guide [6]. These latencies are used for the synchronization between the ALU module and other hardware modules (NN and PCA). The synchronization is shown in Fig. 7.3 and is described as follows.

1. The request signals from the FPGA-based NN or FPGA-based PCA are sent to the ALU module. For example, if the subtraction is needed, the *request_subtraction* is used.

2. The data signal are also forwarded to the ALU module.

3. The *calculation_counter* in the ALU module is configured based on the latency of the requested operation. For example, if the request signal is *request_subtraction*. The *calculation_counter* = 7 because the latency of the subtraction is seven clock cycles according to the user guide. The FPGA-based NN or the FPGA-based PCA always checks the value of the *calculation_counter*. The value of the *calculation_counter* reduces one unit on each clock cycle.

4. if *calculation_counter* = 0, the FPGA-based NN or FPGA-based PCA will receive the output of the floating-point calculation.

7.3.2 Connections between FPGA side and software side

The connection between hardware side and software side is illustrated in Fig. 7.4 which can be explained as follows.

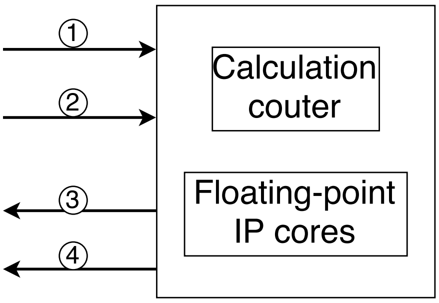


Figure 7.3: Connection with ALU module

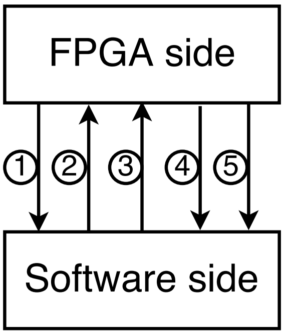


Figure 7.4: Connections between hardware side and software side

1. When the software side wants to conduct the data transmission with the FPGA side. The ARM processor will check the current state of the FPGA side by using $status\_signal$.

2. If the FPGA is in the busy state ($status\_signal = 0$), the software side will wait. Otherwise, the software side sends the $control\_signal$ to the FPGA side. The $control\_signal$ has two different states. If the program is in the training phase of the PCA, the $control\_signal = 1$. On the other hand, the $control\_signal = 0$ if the program is in the testing phase of the PCA.

3. The data from PSO module are also forwarded to the FPGA side by using the $input\_data$.

4. The software rechecks $status\_signal$ to determine whether the data processing have been finished on the FPGA side or not.

5. If the data processing is done ($status\_signal = 1$), the software side received the output data using the $output\_data$.

To take advantage of the FPGA architecture and to increase the operating speed, all connections between software side and hardware side using four signals $status\_signal$, *control_signal*, *input_data*, *output_data* are conducted by DMA. The DMA controllers are configured by the ARM processor.

7.3.3 Operations of the NN-PCA

The training phase of the proposed architecture contains two different tasks called the PCA training and the NN training which will be presented in this section.

7.3.3.1 PCA training phase

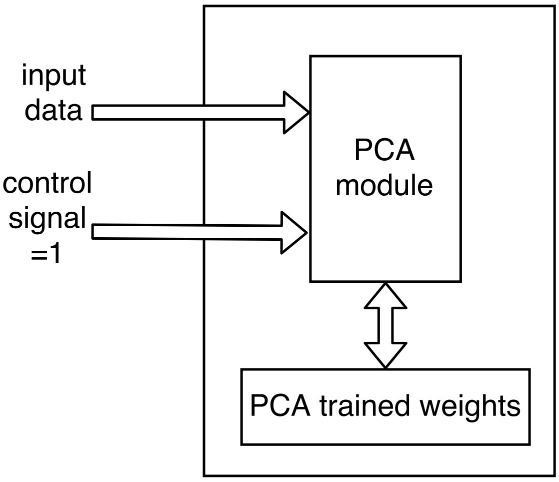


Figure 7.5: PCA training task

Fig. 7.5 shows the training of the PCA. In this situation, the *control_signal* received from the ARM processor has value one ($control\_signal = 1$). The PCA training is the unsupervised learning which is trained based only on the input data so that the software module does not need to receive the output data. At any given training time t, the trained weights $\vec{w}(t)$ are stored. At the time $(i + 1)$, these weights are used to calculate the new weights $\vec{w}(t + 1)$ in the weight update function as shown in Eq. (7.2).
As discussed in section 7.2, the weight update function operates based on the GHA algorithm. The operation of this function could be divided into two main subtasks. The first subtask is the calculation of the output $\vec{y}$ which is expressed in Eq. (7.1). The second subtask is to conduct the weight update function which is shown in Eq. (7.3). The operation of the proposed PCA module can be demonstrated in Fig. 7.6.

1. The input data from the software module are processed in the *Calculate_outputs* submodule. In this submodule, the output data are processed according to Eq. (7.1) using a finite-state machine. When this submodule needs to use the floating-point

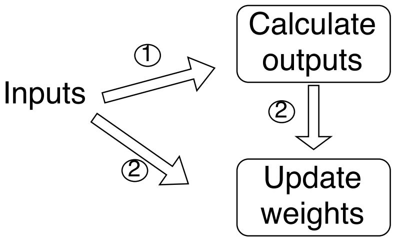


Figure 7.6: Operation of the PCA module

calculation, it will establish a data transmission session with the ALU module. The operation of the data transmission is already described in Fig. 7.3 of section 7.3.1.

2. The calculated outputs from the *Calculate_outputs* submodule and the inputs are used in the *Update_weights* submodule. The weights of the PCA are estimated in this module based on Eq. (7.3). The operation of this submodule is also based on a finite-state machine. This submodule also has the connections to the ALU module to conduct the floating-point calculations.

7.3.3.2 NN training phase

After finishing the PCA training task, the NN training task will be carried. Fig. 7.7 illustrates the training of the NN.

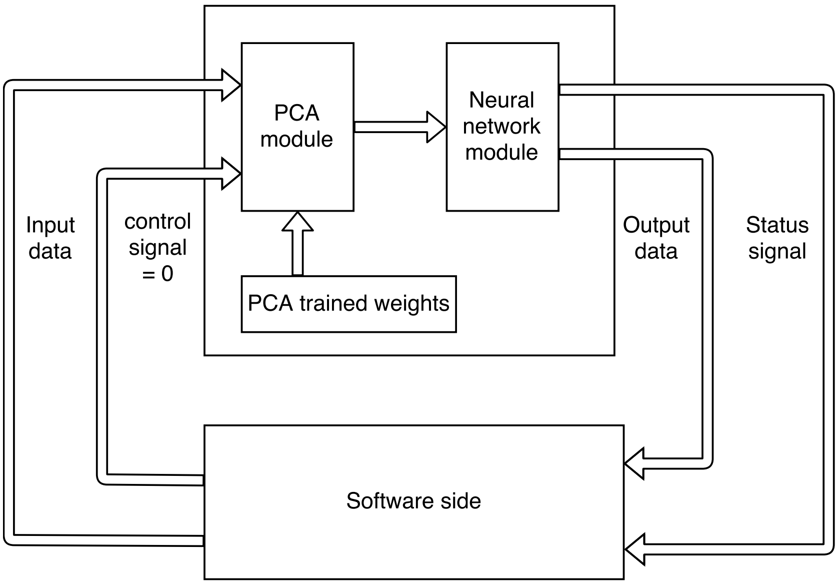


Figure 7.7: NN training task

To conduct the PCA training phase, the software side sends the *control_signal* which has value zero (*control_signal* $= 0$) to the FPGA-based module.

During the NN training phase, the trained PCA weights from the PCA training task will be loaded into the PCA module.

The output data from PCA module become the input of the NN. These data are processed in the NN module. The NN training is the supervised learning. Hence, in any given time t, the actual outputs from the NN are compared with the labeled outputs of the training data based on the fitness function on the software side. In our program, the mean square error function which is presented in Eq. (2.11) of chapter 2 is employed. The results of fitness function are used to update the new $Pbest_p(t+1)$ of particle p, the new $Gbest(t+1)$ of all particles, the new position at time $(t+1)$ of particle p, and the new velocity $\vec{v_p}(t+1)$ at time $(t+1)$ of particle p. The position update function, the new $Pbest_p(t+1)$ function, the new $Gbest(t+1)$ function are shown in Eqs. (2.5), (2.7), and (2.8), respectively. The velocity update function depends on the PSO algorithm. For example, if the PSOseed algorithm is used, Eq. (3.4) is applied. The details in each step of the training on the software side can be seen in Fig. 4.2 of chapter 4. As presented, all operations of the PSO algorithms in this architecture are executed by the ARM processor.

Finishing the training phase, the PCA trained weights and the NN trained weights will be used in the testing phase.

7.4 Experiment

All experiments presented in this section used the DE1-SoC development kit which had the Cyclone V SoC chip. The accuracy concerning the recognition rates and the learning errors of the NN trained by five PSO algorithms (SPSO, DPSO. wPSOd_CV, PSOseed, and PSOseed2) were observed. The parameters for PSO algorithms were similar to the parameters presented in chapter 5, and 6. The details of the parameters are as follows.

- $w = 0.9$, $c_1 = c_2 = 0.5$ in the SPSO algorithm.
- w reduces from 0.9 to 0.00001, $c_1 = c_2 = 0.5$, $c_3 = 0.00001$ in the wPSOd_CV algorithm.
- $w = 0.9$, $c_1 = c_2 = 0.5$, $c_3 = 0.3$ in the PSOseed algorithm and the PSOseed2 algorithm.
- $c_v = 0.0$, $c_l = 0.001$ in the DPSO algorithm (similar to previous research [7]).

All experiments were conducted using the NN-PCA architecture. In these experiments, the particle numbers and the iteration numbers of the NN learning, and the iteration numbers of the PCA learning were varied to investigate different situations.

In the experiments, the PSO algorithms were coded by C programming language by the ARM processor, and the NN was implemented in FPGA by SystemVerilog programming language.

7.4.1 Performances of NN trained by the PSO algorithms

In the experiments to investigate the performances of the PSO algorithms in NN-PCA architecture, two different databases were used which can be seen in Table 7.1.

Table 7.1: List of experiments conducted with NN-PCA approach

Dataset	Features
Diabetic retinopathy debrecen dataset :	19 attributes, 2 classes Cross-validation data: 450 samples in set 1, 300 samples in set 2 (presented in section 7.4.1.1)
Wine dataset:	13 attributes. 3 classes Cross validation data: 88 samples in set 1, 90 samples in set 2 (presented in section 7.4.1.2)

7.4.1.1 Diabetic retinopathy debrecen dataset

The diabetic retinopathy debrecen dataset comes from the feature extracted from the Messidor dataset. This dataset has nineteen features which are used to predict whether a sample of this dataset has any sign of diabetic retinopathy or not [8].

To conduct the experiments with this dataset, 450 samples were chosen randomly as set 1, and 300 samples were considered as set 2. The FPGA-based PCA was used to reduce the input data from nineteen to four. Thus, the configurations of the NN were four input nodes corresponding to four inputs, twenty-five hidden nodes, and two output nodes corresponding to two classes.

Scenario 1: When set 1 was used as the training set, set 2 was selected as the testing set. The settings of the PSO algorithms were 10 particles and 150 iterations. The iteration numbers for the unsupervised learning was 2000 iterations. Two measurements for the NN trained by five PSO algorithms were investigated.

The first measurement is the learning error in the training phase. The reduction of the learning error *Gbest* in this scenario is shown in Fig. 7.8. As can be seen, the reduction curve of the NN trained by the PSOseed2 algorithm declined to the lowest value at 0.1288 which confirmed the performance of the NN trained by PSOseed2 algorithm concerning the learning error.

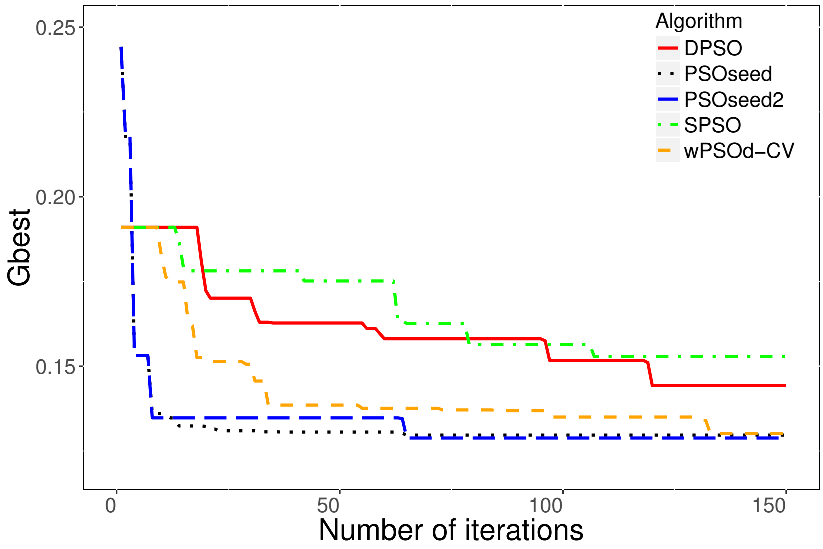


Figure 7.8: Reduction of *Gbest* with diabetic retinopathy debrecen dataset, 450 training samples, 300 testing samples

The second measurement is the recognition rate in the testing phase. The recognition rates of the NN trained by all five PSO algorithms after 150 training iterations are shown in Table 7.2. These results expressed the accuracy of the NN trained by the PSOseed2 algorithm in terms of the recognition rate.

Table 7.2: Results of FPGA-based NN with diabetic retinopathy debrecen dataset, 450 training samples, 300 training samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1529	0.1443	0.1302	0.1297	0.1288
Recognition rate	90.67%	91.00%	91.33%	92.67%	93.67%

Scenario 2: To conduct the cross-validation, set 2 which had 300 samples was also considered as the training set in the second scenario. In this case, 450 samples of set 1 were chosen as the testing data.

In this scenario, the PSO training had 20 particles and 500 iterations. The training iterations for the FPGA-based PCA module was 4200 iterations.

Fig. 7.9 presents the reduction of *Gbest* when the NN trained by five PSO algorithms. In this figure, the PSOseed2 approach obtained the lowest learning error while the SPSO approach had the highest learning error.

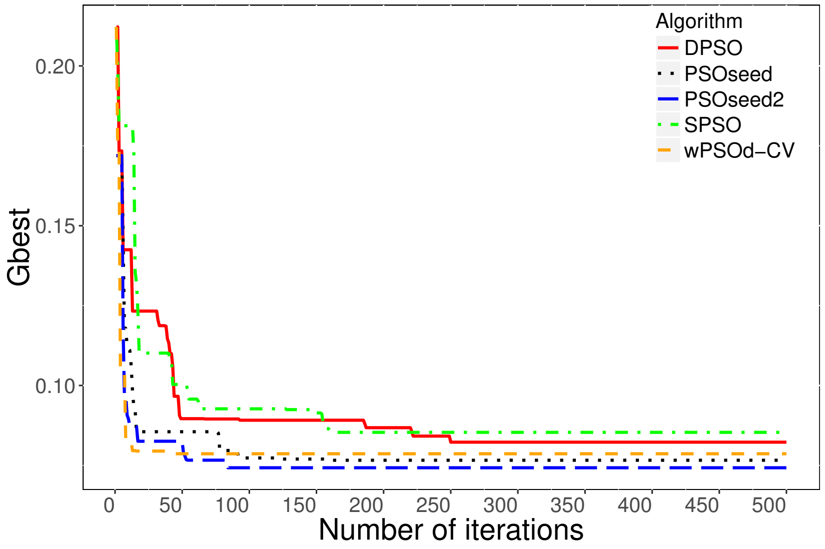


Figure 7.9: Reduction of *Gbest* with diabetic retinopathy debrecen dataset, 300 training samples, 450 testing samples

Experimental results shown in Table 7.3 also confirmed that the NN trained by the PSOseed2 algorithm in the NN-PCA architecture achieved the highest accuracy concerning the learning error when compared with the NN trained by other four PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed). At the minimum learning error (0.0743), the NN trained by the PSOseed2 algorithm also got the highest recognition rate at 85.33%.

Table 7.3: Results with diabetic retinopathy debrecen dataset, 300 training samples, 450 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.0853	0.0823	0.0786	0.0766	0.0743
Recognition rate	81.78%	82.45%	84.22%	85.11%	85.33%

7.4.1.2 Wine dataset

The wine dataset which has thirteen attributes and three classes was also used in the experiments with the NN-PCA architecture [9]. In these experiments, the FPGA-based PCA module reduces the size of the input data from thirteen to three. The configurations of the NN were three input nodes, seven hidden nodes in each hidden layer, two hidden layers, and three output nodes. This 178-sample dataset was divided randomly into two sets for the cross-validation, the first set has 88 samples and the remaining 90 samples were belong to set 2.

Scenario 1: Set 1 was used as the training data and set 2 was selected as the testing data, The configurations of the PSO algorithms were 25 particles and 355 iterations. The hardware-based PCA had 2000 iterations for the unsupervised learning.

Fig. 7.10 illustrates the reduction of the learning error, *Gbest*, of the NN trained by five different PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed, and PSOseed2). As can be observed, the *Gbest* in the PSOseed2 approach decreased to the lowest value while the SPSO approach had the highest learning error.

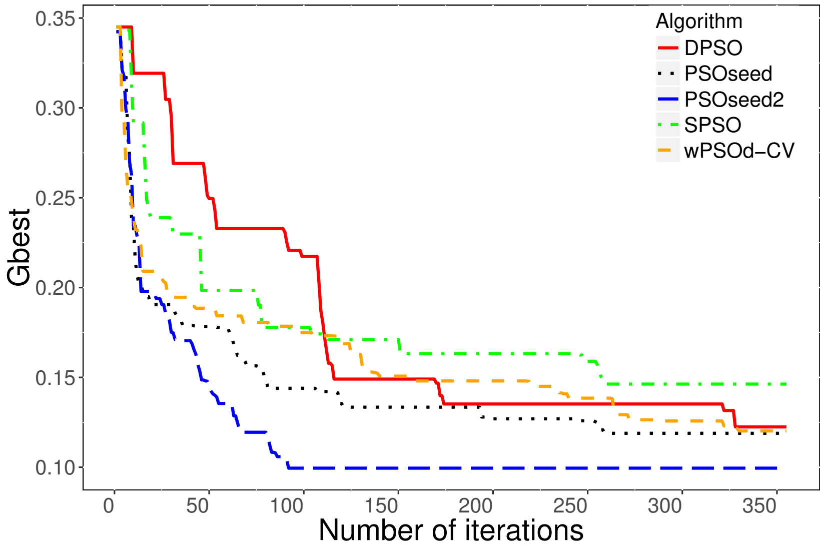


Figure 7.10: Reduction of *Gbest* with wine dataset, 88 training samples, 90 testing samples

Concerning the recognition rates of the NN, Table 7.4 gives information that at the minimum value of *Gbest* (0.0995), the NN trained by the PSOseed2 algorithm also obtained the highest percentage of correct recognition at 91.11%%. On the other hand, the NN trained by the SPSO algorithm still had the lowest recognition rate at 78.89% and the highest learning error *Gbest* at 0.1463 among five algorithms.

Table 7.4: Results with wine dataset, 88 training samples, 90 testing samples

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1463	0.1224	0.1202	0.1189	0.0995
Recognition rate	78.89%	84.44%	87.78%	88.89%	91.11%

Scenario 2: Another scenario was conducted when set 1 became the testing data, and set 2 was used as the training data. In this scenario, the number of particles was 30,

the number of iterations for the NN training was 225. The training of the FPGA-based PCA had 10000 iterations.

The reduction curves of the fitness values *Gbest* of the NN train by five PSO algorithms are shown in Fig. 7.11, and all experimental results are presented in Table 7.5. Similar to other experiments, the results demonstrated the efficiency regarding the recognition rate of the NN trained by three proposed algorithm (wPSOd_CV, PSOseed, and PSOseed2) when compared with the NN trained by PSO algorithms introduced in previous studies (SPSO, DPSO). Among three proposed PSO algorithms, the NN trained by PSOseed2 algorithm had the lowest error in the training phase at 0.0975.

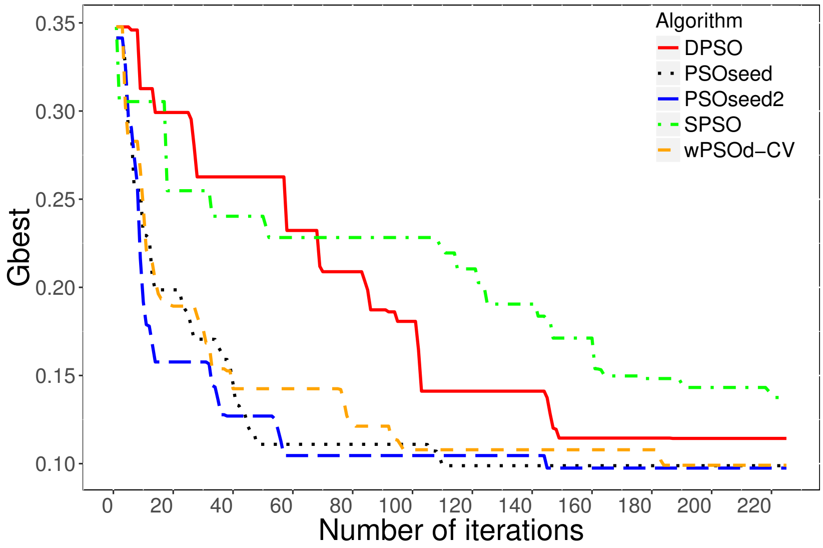


Figure 7.11: Reduction of *Gbest* with wine dataset 90 training samples, 88 testing samples

Table 7.5: Results wine dataset, 90 training samples, 88 training

	SPSO	DPSO	wPSOd_CV	PSOseed	PSOseed2
Final *Gbest* (Learning error)	0.1375	0.1143	0.0991	0.0988	0.0975
Recognition rate	87.50%	92.05%	93.18%	93.18%	93.18%

7.4.2 Operations of the NN-PCA architecture

The proposed NN-PCA architecture needs to be investigated in two main aspects. The first aspect is the required resources concerning the logic elements and the memory bits. The second aspect is the speed when compared to the conventional software-based program.

7.4.2.1 Required resources

The goal of this experiment is to investigate the advantage of the using of FPGA-based PCA concerning the resources. The resources concerning the memory bits of the NN-PCA architecture is very low because our proposed NN-PCA uses the DDR3 RAM instead of on-chip memory, all weights and biases of the NN are stored in the DDR3 RAM. Hence, the resources investigated in this situation are the logic elements. Two different architectures were used in the experiment. The first one is the co-design using ARM processor presented in chapter 6 of this dissertation (without-PCA approach). The second one is the NN-PCA architecture introduced in this chapter. The experiment was conducted with two different datasets presented in section 7.4.1. Our research also tried to implement the hardware-only version of the NN-PCA approach for using these two databases. However, the required resources of the hardware-only architecture were excessive, the Cyclone V did not have enough required resources. Thus, this section does not focus on the hardware-only architecture.

Diabetic retinopathy debrecen dataset: this scenario used one-hidden-layer NN to test with the diabetic retinopathy debrecen dataset (300 samples chosen randomly as training data, 350 samples selected randomly as testing data). Each architecture had three different configurations of the hidden nodes which were 8 hidden nodes, 20 hidden nodes, and 25 hidden nodes. The input nodes were 19, and the output nodes were 2. In the PCA approach, the input size was reduced from 19 to 4. The number of PCA iteration was 2000. The PSO parameters were 20 particles, 150 iterations.

Table 7.6: Results with diabetic retinopathy debrecen

Approach	Hidden nodes	Resources	*Gbet*	Recognition rate
PCA	8	58%	0.0862	96.86%
	20	69%	0.0848	96.86%
	25	75%	0.0829	96.86%
Without PCA	8	35%	0.0904	93.14%
	20	76%	0.0889	95.71%
	25	>100% (compile failed)	-	-

Table 7.6 presents the experimental results. Because the PCA is implemented in hardware using SystemVerilog programming language, this module also consumes the FPGA resources. Thus, when the NN had a small number of hidden nodes, the required resources for the PCA approach could be higher than the required resources for the

without-PCA approach. However, using a small number of nodes in hidden layers reduced the recognition rate and increased the learning error. In the case of using the NN which had a significant number of hidden nodes, the required resources of the PCA approach decreased significantly when compared with the required resources of the without-PCA approach. For example, with 25 hidden nodes, the number of weights and biases for the without-PCA are $(19+1)\times 25+(25+1)\times 2=552$ while the number of weights and biases for the PCA approach after conducting the PCA process from 19 to 4 are $(4+1)\times 25+(25+1)\times 2=175$. Experimental results also demonstrated that the PCA approach with 8 hidden nodes got the higher recognition rate and required fewer resources than the 20 hidden nodes of the without-PCA approach.

Wine datase: The second scenario investigated the wine dataset that has thirteen attributes and three classes (90 samples were chosen randomly as the training data, 88 samples were selected randomly as the testing data). The PCA which trained by 5500 iterations was used to reduce the input data from 13 to 3.

Two different configurations of the PSO parameters were chosen to investigate the operation in different situations. The first configuration used the smaller number of particles numbers and iteration numbers than the second configuration. In both configurations, the PCA iteration was 5500 iterations.

In the first configuration of the PSO parameters, two different two-hidden-layer NNs were used. The first NN had 5 hidden nodes in each layer and the second NN had 10 hidden nodes in each layer. The number of particles was 5, the number of iterations was 300. Experimental results are shown in table 7.7. Two different two-hidden-layer NNs were also employed in the second configuration of the PSO parameters which had 5 hidden nodes and 15 hidden nodes in each layer. In this case, the particle numbers and the iteration numbers were increased to 15 particles and 500 iterations. The results are shown in Table 7.8.

Table 7.7: Results with wine dataset, 5 particles, 300 iterations, 5500 PCA

Approach	Hidden nodes	Resources	*Gbet*	Recognition rate
PCA	5	43%	0.1535	94.318%
	10	63%	0.0582	95.45%
Without PCA	5	28%	0.1954	73.863%
	10	65%	0.0776	94.56%

Table 7.8: Results with wine dataset,15 particles, 500 iterations, 5500 PCA

Approach	Hidden nodes	Resources	*Gbet*	Recognition rate
PCA	5	43%	0.1293	94.318%
	15	71%	0.0501	96.54%
Without PCA	5	28%	0.1732	86.363%
	15	81%	0.0734	95.45%

Similar to the experimental results in the previous scenario, with a small number of hidden nodes, the required resources of the without-PCA approach was lower than the PCA approach because of the resources reserved for the implementation of the PCA. However, the recognition rates with the small number of hidden nodes were low (73.86% with 5 particles 300 iterations, and 86.33% with 15 particles, 500 iterations). Using bigger hidden nodes, the recognition rates of both approaches were increased. In this situation, results showed that the PCA approach obtained the higher recognition rates and reduced the required resources when compared with the without-PCA approach. This results can be explained because the PCA approach not only reduced the required resources previously reserved for the weights and biases of the NN but also reduced the correlated information.

7.4.2.2 Operating speed

Another aspect that needs to be considered is the operating speed. Experiments were conducted to compare the operating speed between the proposed co-design architecture and the software-only architecture. In the software-only architecture, both NN and PCA were implemented by Intel processors. Two Intel processors were employed in this study (Intel core i3 and Intel core i7). The frequency of the FPGA architecture could be modified using a phased-locked loop IP cores provided by Altera [10]. Thus, two different frequencies of the hardware module were used in the experiments. There were four different scenarios of the experiment as follows.

- Software-based NN-PCA in the Intel core i3 (2 cores, 2.4 GHz) processor.
- Software-based NN-PCA in the Intel core i7 (4 cores, 4.0 GHz) processor.
- FPGA-based NN-PCA with 50 MHz of the clock frequency.
- FPGA-based NN-PCA with 100 MHz of the clock frequency.

The Intel approach was coded by C programming language in Visual Studio 2013 on two different computers. The speed of the software-based program was measured using the *QueryPerformanceCounter*() function [11]. The average execution time after 20 tests were considered as the operating speed. In the FPGA-based program, the measurement of the time was conducted using the *clock_gettime*() function [12].
The execution time using two datasets presented in section 7.4.2.1 which were the diabetic retinopathy debrecen dataset and the wine dataset.

Diabetic retinopathy debrecen dataset: The number of the training samples (300 samples) and the testing samples (350 samples) were similar to the experiments of the required resources presented in section 7.4.2.1. The PCA reduced the input data from nineteen to four, and the NN had four input nodes, 8 hidden nodes, and two output nodes.

Table 7.9: The testing time with diabetic retinopathy debrecen dataset, 350 testing samples in second

Frequency / Approach	50 MHz	100 MHz	2.4 GHz	4.0 GHz
NN-PCA implemented in Intel core i3	-	-	0.022 seconds	-
NN-PCA implemented in Intel core i7	-	-	-	0.0208 seconds
NN-PCA implemented in FPGA	0.0175 seconds	0.0113 seconds	-	-

The experimental results can be seen in Table 7.9. These results showed that even operate at a lower clock frequency, the FPGA-based NN-PCA could obtain a higher operating speed than the software-based NN-PCA.

Wine dataset: In this scenario, the testing time with the wine dataset was investigated. Similar to the resources experiment presented in section 7.4.2.1, the number of training samples was 90, and the number of testing samples was 88. Therefore, 88 samples were used to measure the operating speed of the NN-PCA architecture. The size of the NN was three input nodes, ten hidden nodes in each layer, two hidden layers, and three output nodes. The PCA in this scenario reduced the input data from thirteen to three.
Experimental results are shown in Table 7.10. These results once again confirmed the speed advantage of the hardware-based PCA. As observed, the required times to process all 88 training samples of this approach were lower than the Intel approach.

Table 7.10: The testing time with wine dataset, 90 training samples, 88 testing samples in second

Frequency Approach	50 MHz	100 MHz	2.4 GHz	4.0 GHz
NN-PCA implemented in Intel core i3	-	-	0.0445 seconds	-
NN-PCA implemented in Intel core i7	-	-	-	0.0136 seconds
NN-PCA implemented in FPGA	0.0130 seconds	0.0085 seconds	-	-

7.5 Discussion

This chapter presents a co-design architecture for the training of the FPGA-based NN using the PSO algorithms. The proposed architecture called NN-PCA architecture contains not only the co-design architecture presented in chapter 6 of this thesis but also the hardware-based PCA. The FPGA-based PCA is based on the GHA which is suitable for the hardware implementation.

The operations of five PSO algorithms (SPSO, DPSO, wPSOd_CV, PSOseed, PSOseed2) for the training of the NN in this NN-PCA architecture were investigated in our experiments. The results once again demonstrated the performance of the NN trained by PSOseed2 algorithm concerning the learning errors and the recognition rates.

Results in our experiments also showed that the hardware-based PCA can reduce the required resources of the program while still got the high recognition rates when compared with the without-PCA approach with the same condition of the PSO parameters and the same number of the hidden nodes. In addition, this co-design architecture also had a higher testing speed than the software-only architecture.

References

[1] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[2] E. Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15(3):267–273, 1982.

[3] T. D. Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, 2(6):459 – 473, 1989.

[4] Y. Hirai and K. Nishizawa. Hardware implementation of a pca learning network by an asynchronous pdm digital circuit. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 65–70, 2000.

[5] Uwe Meyer-Baese. Digital signal processing with field programmable gate arrays, 2007.

[6] Altera Corporation. Floating-Point IP Cores User Guide. 2016.

[7] X. F. Xie, W. J. Zhang, and Z. L. Yang. Dissipative particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1456–1461, 2002.

[8] B. Antal and A. Hajdu. An ensemble-based system for automatic screening of diabetic retinopathy. *Knowledge-Based Systems*, 60:20–27, April 2014. ISSN 0950-7051.

[9] M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

[10] Altera Corporation. Embedded Peripherals IP User Guide. 2016.

[11] Queryperformancecounter function (windows). `https://msdn.microsoft.com`. (Accessed on 06/06/2017).

[12] clock_gettime(3): clock/time functions - linux man page. `https://linux.die.net/man/3/clock_gettime`. (Accessed on 06/06/2017).

Chapter 8

Conclusion

To achieve the research objectives, both the hardware aspect and the software aspect of the NN trained by the PSO algorithm were investigated.

The partitioning methodology for the co-design between hardware and software for the NN-PSO was proposed. The NN is implemented in hardware using SystemVerilog programming language, and the PSO algorithms are moved to the software side and executed by the processors. In the NN-PSO, the NN is only used in the testing phase while the PSO algorithms are employed in both training phase and testing phase. Compared to the hardware-only approach, the proposed co-design approach not only maintains the testing speed but also reduces the required FPGA resources concerning the logic elements and the memory bits previously reserved for the hardware implementation of the PSO algorithms. Compared to the software-only approach, the proposed co-design approach not only preserves the flexibility during in the training phase but also obtains the higher operating speed in the testing phase. The flexibility relates to the easiness to modify the PSO parameters or even change the PSO algorithms without redesigning or rebuilding the FPGA part.

Three different co-design architectures were presented based on the introduced partitioning methodology. The first architecture uses the NIOS II processor. In this case, both NIOS II processor and FPGA-based NN are implemented in a single FPGA device which is portable. The NIOS II processor is replaced by the ARM processor in the second architecture because the ARM processor could operate at a higher clock frequency than the NIOS II processor. In addition, the second architecture may reduce the required FPGA-resources previously reserved for the NIOS II processor because the

ARM processor is physically implemented on the FPGA board while the NIOS II processor is created using available resources from the FPGA device. The third architecture combines the ARM approach in the second architecture with the FPGA-based PCA to reduce the required resources but still maintain the accuracy of the NN trained by PSO algorithms concerning the recognition rates and the learning errors.

Regarding the software aspect, three different PSO algorithms were proposed in this thesis to solve the premature convergence of the SPSO without adding many compute-intensive tasks or functions to the SPSO algorithms. All three algorithms only modify the velocity update function of the SPSO algorithm. The first algorithm, called the wPSOd_CV algorithm, has the velocity control and the weight control mechanisms. This algorithm has a jump phase to prevent the situation when a particle sticks to a local minimum. However, even changing only the velocity update function, this algorithm still adds several tasks for the weight control and the velocity control. In addition, although the wPSOd_CV has the weight control to balance between the exploitation and the exploration, the exploration of the wPSOd_CV algorithm is significantly affected by the big jumps with a very high speed of the particles. To overcome the drawbacks of the wPSOd_CV algorithm, the second algorithm called the PSOseed algorithm was presented. Avoiding the jumping phases, the PSOseed algorithm uses a new variable called the seed position which is generated in the initial phase of the algorithm. In each iteration, each particle is attracted and pulled to the position of its seed. This seed mechanism has the possibility to keep the particles out of the local minimum. Compared to the wPSOd_CV algorithm, the PSOseed algorithm does not use any division operator and use fewer multiplication operators. However, the performance of the PSOseed algorithm highly depends on the seeds. If the seed positions are poorly generated, the accuracy of the NN trained by the PSOseed algorithm will reduce. The third algorithm called the PSOseed2 algorithm overcomes the seed problem of the PSOseed algorithm by using the reseed mechanism. In any given iteration, if the learning error does not reduce, the seed positions will be reseeded.

In the experiments, three proposed PSO algorithms (wPSOd_CV, PSOseed, PSOseed2) and two PSO algorithm presented in previous studies (SPSO, DPSO) were employed. The NN was trained with these five PSO algorithms in all three proposed co-design architectures. Experimental results demonstrated that the NN trained by PSOseed2 had the highest performances concerning the recognition rates and the learning errors in all three architectures. The results also confirmed both the speed advantage of the proposed co-design approach when compared with the software-only approach and the resource advantage of the proposed co-design approach when compared with the hardware-only approach. Results also confirmed that the NN-PCA approach had the reduction of the required resources while maintaining the high recognition rates when compared with the without-PCA approach.

The proposed NN-PSO system is suitable for studies and applications which require high-speed, low-power consumption, and portable program. An example of this system could be the real-time object recognition. The captured images from a camera will be pre-processed by the image processing filters such as the median filter or the Gaussian filter for the noise reduction. Previous research has presented that the hardware implementation of these filters is feasible. The features of pre-processed data will be extracted by using OpenCV which was built on ARM-based Linux system. The FPGA-based dimension reduction techniques may be used with the extracted features. After the dimension reduction step, the features are fed to the NN-PSO system. The whole system could be implemented in a small FPGA board.

The proposed PSOseed2 does not add many compute-intensive tasks to the SPSO. Therefore, this algorithm is suitable for the studies or applications which require the hardware implementation of PSO algorithm.

Our experiments were conducted with the Cyclone V which is the lowest system power and performance device from Atera. The Cyclone V also has the limited FPGA resources. Therefore, the target of these experiments is to investigate the operation of the proposed co-design architectures with small-scaled classification tasks which have the small size of the NN. A portable device which has low power consumption can be used to solve these tasks because all components of the proposed architecture such as the processor or the FPGA-based modules can be implemented in a single FPGA SoC chip, the Cyclone V in this situation. In future research, if a more powerful device could be used such as the Arria or Stratix, a more complex dataset and a bigger NN will be investigated. The target is to develop a portable and wearable system to solve the daily life problems related the classification.

Among three proposed PSO algorithms, the NN trained by the PSOseed2 obtained the highest recognition rate and the lowest learning error. Therefore, a possible avenue is to conduct more studies about this algorithm.

In the FPGA-based program, the issue related to the limited logic elements and memory bits needs to be investigated. Thus, another direction for the future research is the optimization task for the FPGA-based program. The goal of this task is to reduce the required resources for the proposed co-design.

Appendix A

Software-only architecture

This appendix has been published in our research article [1].

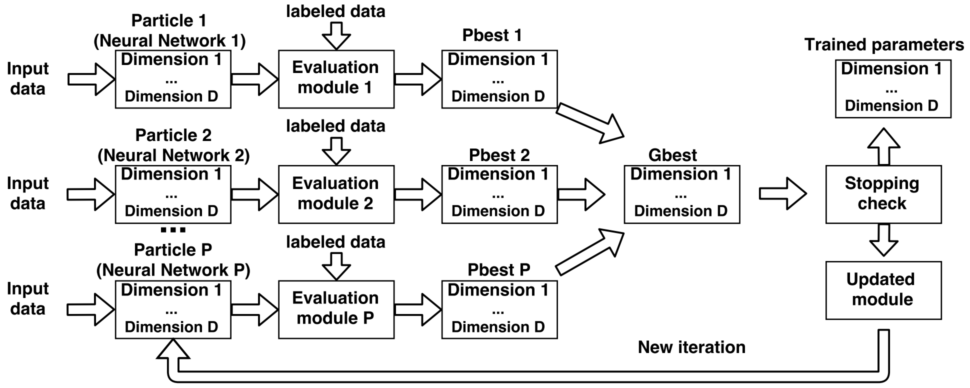

Input data
Particle 1 (Neural Network 1)
Dimension 1 ... Dimension D
labeled data
Evaluation module 1
Pbest 1
Dimension 1 ... Dimension D
Input data
Particle 2 (Neural Network 2)
Dimension 1 ... Dimension D
labeled data
Evaluation module 2
Pbest 2
Dimension 1 ... Dimension D
Gbest
Dimension 1 ... Dimension D
Input data
Particle P (Neural Network P)
Dimension 1 ... Dimension D
labeled data
Evaluation module P
Pbest P
Dimension 1 ... Dimension D
Trained parameters
Dimension 1 ... Dimension D
Stopping check
Updated module
New iteration

Figure A.1: The software-only architecture

Fig. A.1 presents the software-only approach. In this approach, all components of the NN-PSO are implemented on the software side by a processor such as the Intel processor. The training data are divided into two different parts called the input data and the labeled data, respectively. The labeled data are the desired output data. The input data are sent to the particle-block module, and the labeled data are forwarded to the evaluation modules. Each particle is a N_D-dimensional vector that corresponds to N_D parameters (weights and biases) of one NN. The output data of the NN are fed to the evaluation modules. In these modules, the fitness values are calculated by the mean square error function as presented in Eq. (2.11).

The calculated fitness values will be fed to the Pbest module to evaluate the new $Pbest_p$ according to Eq. (2.7). Results of the Pbest module are used to estimate the new $Gbest$ based on Eq. (2.8).

In each iteration, the stopping-check module investigates whether the stopping criterion is satisfied or not. If the stopping condition is not met, the new velocities and positions of all particles are calculated in the updated module, and the new iteration will be conducted. The velocity update function in the updated module based on the PSO algorithm. For demonstration, if the PSOseed2 is used, the velocity update function is calculated according to Eq. (3.4) and the reseed control. On the other hand, if the stopping condition is satisfied, the trained weights and biases will be stored to use in the testing phase.

Reference

[1] T. L. Dang, T.Cao, and Y. Hoshino. Data pre-processing for a neural network trained by an improved particle swarm optimization algorithm. *International Journal of Computer Applications*, 154(1):1–8, Nov 2016.

Appendix B

Hardware-only architecture

This appendix has been published in our research article [1].

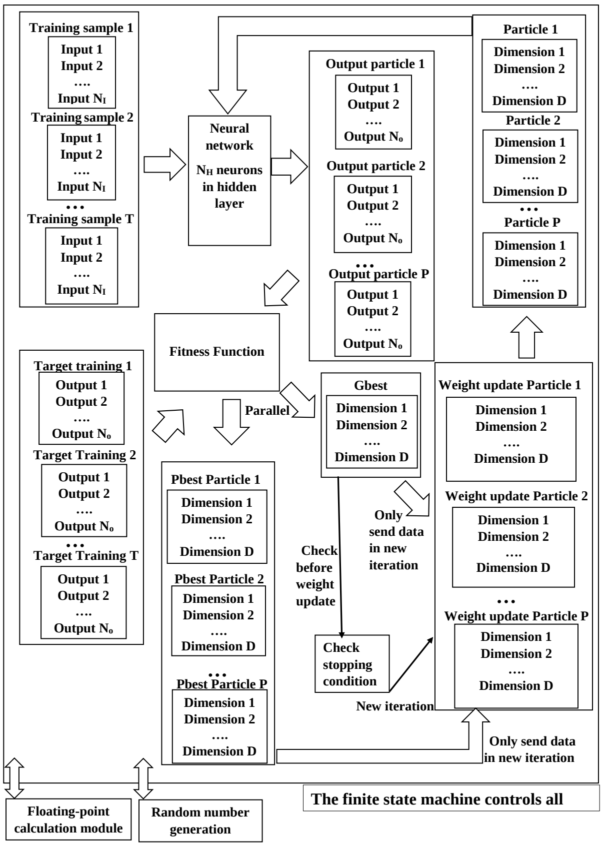


Figure B.1: Algorithm of hardware implementation of training step

Fig. B.1 shows the hardware-only architecture. The whole system is controlled by a global finite-state machine. In this architecture, module Gbest and module Pbest are processed in parallel to take advantage of the FPGA-based program. The components of this system are as follows.

1. Floating-point calculation (FPC): implements the floating-point IP cores provided by Altera. Every floating-point computation in other modules connects to this module. In the same way as the floating-point submodule presented in section 5.2.2 of chapter 5. This submodule has a *calculation_counter* for the synchronization with other modules.

2. Random number generator: This is a special submodule of the hardware-only architecture because the random number generator is software is not easy as the software. The random numbers are generated by linear feedback shifts register. Our program uses the IEEE 754 floating-point single precision so that the LFSR is the 32-bit register. To prevent some special cases of the IEEE 754 when all bits of the register are zero or all bits of the exponent are one, two bits are fixed. Two different *xor* functions are used for the LFSR. One is in the mantissa part, one is in the exponent part. Bit 29, 28, 26 in the exponent and bit 20, 17, 14 in the mantissa are selected as the taps bit. The content of the LFSR can be seen in Fig. B.2.

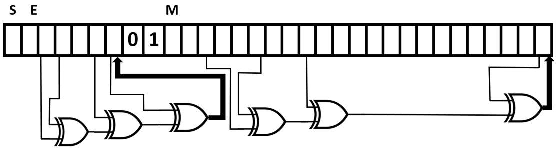


Figure B.2: Initial value LFSR: s-sign bit, e-exponent bit, and m-mantissa bit

3. Neural network (NN): the input of the NN is a two-dimensional array of weights called w which can be presented by SystemVerilog as `[31:0]w[0:`N_D` -1]` where the first dimension demonstrates the floating-point number (32 bits). The second dimension is reserved for the weights and the biases (N_D parameters). The activation function of the NN is the Sigmoid function shown in Eq. (2.2). The FPGA-based NN is based on another finite-state machine which has two states called *idle_state* and *running_state*, respectively. Normally, it is in the *idle_state*. When a computational request comes, the finite-state machine moves to *running_state* to conduct the processing based on Eqs. (2.1) and (2.2). Finishing the operation, the NN sends output data via output port and its finite-state machine returns to *idle_state*.

4. Training sample: processes the raw T training data and sends the processed data to the NN module.

5. Target training: contains T labeled training data (*target*) for the learning stage of the NN.

6. Particle: each component in this module is one particle which contains D weights of the NN. The values of these weights are calculated from weight update module. In the first iteration, each value of each particle is assigned the initial value from the LFSR.

7. Output particle: $Output\_particle_i$ ($0 \leqslant i < P$ where P is the number of particles) are the outputs of the NN when D weights of particle i are the parameters of the NN.

8. Fitness function: employs the mean squared error according to Eq. (2.11) implemented in hardware by using its finite-state machine.

9. Pbest: discovers the best fitness value in each particle based on results from the fitness function module. The weights correspond to each *Pbest* are kept.

10. Gbest: finds the best global fitness value in a particular iteration. The weights correspond to *Gbest* of the population are stored.

11. Weight update: updates the new velocities and the new positions of the weights and the biases based on the PSO_CV algorithm.

12. Check stopping condition: checks whether the stopping criterion is satisfied or not. If the stopping condition is not met, the next iteration is conducted.

The components on the FPGA board can be shown as Fig. B.3.

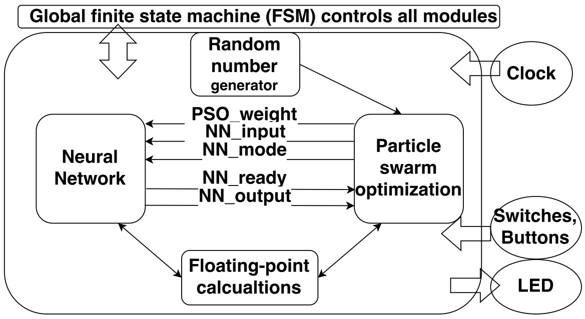

Global finite state machine (FSM) controls all modules
Random number generator
Clock
PSO_weight
NN_input
NN_mode
NN_ready
NN_output
Neural Network
Particle swarm optimization
Switches, Buttons
Floating-point calcualtions
LED

Figure B.3: The hardware-only architecture

Reference

[1] T. L. Dang and Y. Hoshino. A hardware implementation of particle swarm optimization with a control of velocity for training neural network. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1980–1985, Oct 2015.