An FPGA Implementation of Kak's Instantaneously-Trained, Fast-Classification Neural Networks

Jihan Zhu and Peter Sutton

School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane QLD 4072 Australia
{jihan, p.sutton@itee.uq.edu.au}

Abstract

Motivated by a biologically plausible short-memory sketchpad, Kak's Fast Classification (FC) neural networks are instantaneously trained by using a prescriptive training scheme. Both weights and the topology for an FC network are specified with only two presentations of the training samples. Compared with iterative learning algorithms such as Backpropagation (which may require many thousands of presentations of the training data), the training of FC networks is extremely fast and learning convergence is always guaranteed. Thus FC networks are suitable for applications where real-time classification and adaptive filtering are needed. In this paper we show that FC networks are "hardware friendly" for implementation on FPGAs. Their unique prescriptive learning scheme can be integrated with the hardware design of the FC network through parameterization and compile-time constant folding.

1. Introduction

There exist certain classes of real-time classification / adaptive control systems within which learning is a critical task which must be guaranteed to finish on time. Reconnaissance robots, and satellite sensory systems looking for interesting objects in unfamiliar environments are some of the examples of such real-time systems. In these types of systems, one cannot fully anticipant the full range of objects the classification systems may encounter. The classification system must learn to classify these in real-time and learn as they continue to explore. Neural networks have been shown to be powerful classification tools. However, neural networks which are based on iterative learning algorithms such as multilayer perceptrons, radial basis functions and support vector machines can suffer from training bottleneck [1]: that is learning may not converge or take too long to be useful in real-time applications even with hardware accelera-

tion. For this reason, iterative learning neural networks are not suitable for the real-time systems where learning is a critical task.

Kak's Fast Classification (FC) networks [2] overcome the learning bottleneck by employing instantaneous learning. The model of FC networks is motivated by a biologically plausible sketchpad mechanism for short-term memory in which learning occurs instantaneously. The learning in FC networks does not suffer from the learning bottleneck and is always guaranteed to converge. Both the weights and the topology of an FC network are determined by simple inspection of the training examples. Only two presentations of training samples are required to train an FC network, which is extremely efficient compared with iterative learning algorithms, such as backpropagation, where thousands of presentations of training samples are required.

In this paper, we show that Kak's FC networks with their prescriptive learning scheme are well suited for implementation on FPGA based reconfigurable hardware platforms by exploiting fine grained parallelism. We show that the prescriptive learning algorithm can be integrated into hardware design for the FC networks through parameterization and compile-time constant folding.

The remainder of this paper is organised as follows. Section 2 describes the algorithm framework for the FC networks. Operations in the training and execution phases of FC networks are formally presented. Section 3 presents the hardware design for FC networks. The overall system architecture is outlined first, followed by implementations for network components: hidden neurons, hidden layer rulebases and the output neurons. Section 4 discusses strategies for integrating prescriptive learning with the design of FC networks and Section 5 draws some conclusions.

2. Algorithmic Framework for FC Networks

The FC networks have a three layer feed-forward architecture which consists of a layer of inputs, a layer of dis-

tance based hidden neurons, a fuzzy rule base and an output layer as illustrated in Fig. 1.
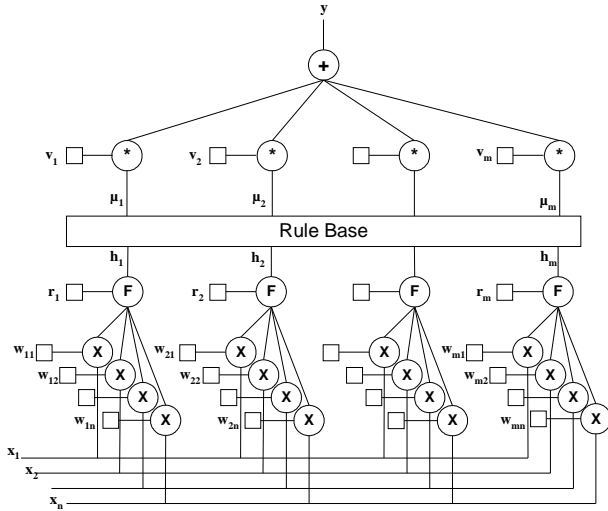

Figure 1: An Illustration of an FC Network

Input data presented to an FC network is a n element long continuous-valued vector $x = (x_1, x_2, ..., x_n)$, where n is the length of the input and is determined by the problem specification. Each hidden neuron i $(i = 1, 2, ..., m)$ stores an exemplar training sample faithfully as its weight vector $w_i = (w_{i, 1}, w_{i, 2}, ..., w_{i, n})$. A hidden neuron i first computes the distance d_i between the input vector x and its weight vector as shown in Fig. 2 (a).

(a)   (b)
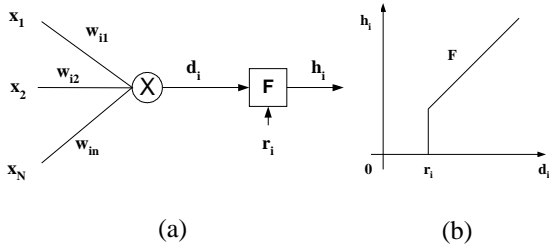
Figure 2: (a) Computation in a hidden neuron
(b) Activation function F

The distance d_i, a scalar, is then passed to the activation function F_i to produce an output h_i for the hidden neuron as illustrated Fig. 2 (b). The operation of the hidden unit activation function F_i can be specified mathematically as:

$$h_i = 0 \qquad d_i \leq r_i$$
$$h_i = d_i \qquad d_i > r_i \tag{1}$$

where r_i is defined as the radius of generalization for hidden neuron i. The effect of using this activation function is to make any test vector x within a certain distance of a stored exemplar training sample w indistinguishable from the training sample, and hence the test vector will be classi-

fied in the same output class as the training sample w. The generalization radius r hence "fuzzifys" the input space around a training sample w; any test vector that lies within this fuzzy region will be classified as the same as the training example.

All outputs from the hidden layer form a distance vector $h = (h_1, h_2, ..., h_m)$ which represents the similarity between the test vector x and each of the training samples stored in the hidden layer of the FC network. The distance vector is then presented to the fuzzy rule base which maps the distance vector h into a membership grade vector $\mu = (\mu_1, \mu_2, ..., \mu_m)$. The vector μ represents the degree of membership that the test vector x has to each of the output classes. The properties of μ are described below:

$$(0 < \mu_i < 1) \in \mu \qquad \sum_{1}^{m} \mu_i = 1 . \tag{2}$$

The output neuron then computes a dot product between the output weight vector $v = (v_1, v_2, ..., v_m)$ and the fuzzy membership vector μ to aggregate all fuzzy contributions from hidden neurons to produce the final network output y for the test vector x. The fuzzy aggregation is illustrated in Fig. 3. The fuzzy aggregation is described mathematically as:

$$y = \sum_{i=1}^{m} \mu_i v_i , \tag{3}$$

where output weight v_i is assigned to be the corresponding target outputs of each exemplar vector stored in the hidden neurons.
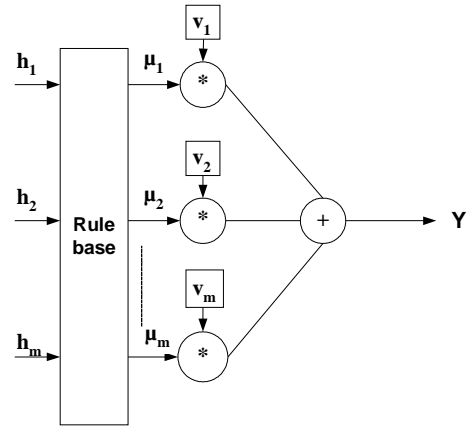

Figure 3: Fuzzy generalization and aggregation in the output layer.

2.1. Prescriptive Learning in FC Networks

The prescriptive learning scheme for training FC networks is very simple and only requires two presentations of

the training samples. The first presentation of training samples is used to prescribe the topology and weights while the second presentation is used to determine the radius of generalization r_i for each hidden neuron. These two processes are described briefly below.

2.1.1. Prescribing the Topology and Weights

An FC network faithfully represents all training samples in hardware by allocating one neuron for each training sample. Given a training sample set, the prescriptive algorithm determines directly the topology and weights of an FC network. The required number of input neurons is the length of the input vector. The number of hidden neurons equals the number of samples in the training set (i.e. each hidden neuron represents one training sample). The required number of output neurons equals the desired number of outputs (only one neuron is shown in Fig. 1 for simplicity, multi-way classification can always be partitioned into networks with one output neuron).

With the topology of the network specified, assigning weights for the network is done by simply inspecting the training samples. The first presentation of the training sample determines the input and output weights. Let T be the training set which contains m training samples and i is the index; let $\langle t_{i,j}, o_{i,k} \rangle$ be the input-target pair of the ith exemplar training sample to be stored at a hidden neuron, then the input weight for the hidden neuron is assigned to be $w_{i,j} = t_{i,j}$. Similarly, the corresponding output weight is assigned to be $v_{i,k} = o_{i,k}$. Here j, k are the length of the input vector and the target vector.

2.1.2. Determining the radius of generalization

The second presentation of the training samples is needed to determine the radius of generalization r_i for each of the hidden neurons i. Distances are calculated between the exemplar training sample t_i represented by the current hidden neuron and all other training samples. The smallest distance d_{min} is from the training sample t_i to its nearest neighbour. The radius of generalization r_i for the ith hidden neuron is then set to $d_{min}/2$. This is to ensure that the generalization regions of all hidden neuron never overlap.

2.1.3. Generalization with Fuzzy Rule Base

When a test vector falls in one of the generalization region of a hidden neuron, the fuzzy rule base merely acts as a gating function. That is if $h_j = 0$, the fuzzy membership grades are assigned according to the following rule:

$$
\begin{aligned}
\mu_i &= 1 \quad &for \quad & i = j \\
\mu_i &= 0 \quad &for \quad & i \neq j \quad . \\
& & i, j \in (1, 2, ..., m)
\end{aligned}
\tag{4}
$$

In this way the test vector is classified as belonging to the same output class that the exemplar training sample does.

The purpose of employing a fuzzy rule base in the FC network is to provide a generalization method to allow a test vector to have fuzzy membership grades in output classes of its k nearest neighbours when the test vector does not fall into the generalization region of any training sample. This is because, as mentioned above, the generalization regions for hidden neurons do not overlap. The fuzzy rule base provides a way to interpolate the final output for a test vector based on its distances from k nearest training examples. Although a variety of fuzzy membership functions can be used to map the distance vector h to membership vector μ, the simplest is the triangular membership function. For example, when $k = 2$, and d_1, d_2 are the distance between the test vector and its two nearest neighbour, the triangular membership function would be expressed as:

$$
\begin{aligned}
u_1 &= \frac{1}{d_1} \Big/ \left(\frac{1}{d_1} + \frac{1}{d_2} \right) \\
u_2 &= \frac{1}{d_2} \Big/ \left(\frac{1}{d_1} + \frac{1}{d_2} \right)
\end{aligned}
\tag{5}
$$

Fig. 4 gives an illustration for a triangular fuzzy membership function for the two nearest neighbour case. Other membership functions, for example the quadratic function S, are possible. However, Kak's experiments show that the performance of an FC network is not significantly effected by the choice of the fuzzy membership function.
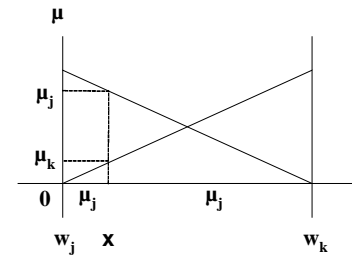

Figure 4: Fuzzy membership function for 2 nearest neighbour case.

3. Implementation

Like its biological counterpart, an FC network is also limited by its capacity to store training samples. By prescriptive learning definition, all training samples must be faithfully represented in the network. That is one hidden neuron in an FC network is needed to represent a training sample in the training set. Hence the size of hidden layer is of $O(m)$ where m is the total number of training samples. When m is large, the hardware realization of an FC network is also large. To overcome this problem, the implementation for a hidden neuron in a FC network must be space efficient.

Various strategies are used to minimise the resources used to implement the hidden neurons. Strategies are also used to simplify the implementation for the fuzzy rule base as the fuzzy rule base sorts the distance vector h_i to select k nearest neighbours of a test vector. The search space is related to the number of hidden neurons. Specifically, a m element full parallel bitonic sorter is $O(\ln^2(m))$ deep. These strategies are described below in detail.

3.1. An Overview of System Architecture

The implementation is targeted to a Celoxica RC2000 board with a Xilinx XC2V6000 Virtex-II chip. The design and simulation for the hardware implementation are carried out by using the JHDL hardware description language. As described later, the design of an FC network is parameterised and integrated with the prescriptive learning scheme to produce design cores for final synthesis on hardware.

3.2. Hidden Neuron Circuit

The hidden neuron circuit is a critical part of the implementation. As each of the training samples in the training data set is required to be represented by one hidden neuron, resources used by the hidden layer in an FC network are considerable when the training set is large. Strategies are use to make the hidden neuron circuit implementation as resource efficient as possible.

3.2.1. Implementation of Distance Function
As explained in section 2, hidden neurons in an FC network are distance based, and they compute the distance between the input vector x and every training sample stored in the hidden layer as the input weights. This distance calculated by the ith hidden layer can be expressed as:

$$d_i = \|x_i - w_i\|^p = \sum_{j=1}^{n} |x_{i,j} - w_{i,j}|^p \qquad (6)$$

where n is the length of the input vector x. In Kak's FC network proposal Euclidean distance is preferred because it is rotational invariant and it minimizes the within-class classification variance. Euclidean distance is a special case of general distance metric, i.e. $p = 2$, and

$$d_i = \|x_i - w_i\|^2 = \sum_{j=1}^{n} |x_{i,j} - w_{i,j}|^2 \qquad (7)$$

However, Euclidean distance is very expensive to implement on FPGAs as it requires a "squaring a number" operation. If Euclidean distance were used, a hidden layer of m neurons, each with a n element weight vector would require $O(mn)$ "squaring" operations. Fortunately, experiments conducted by Kak have shown that the FC network per-formance is robust and it is not seriously effected by the choice of distance metric. Two alternative distance metrics were investigated previously by [3] as replacements for the Euclidean distance:

• when $p = 1$, the city-block distance;

$$d_i = \|x_i - w_i\| = \sum_{j=1}^{n} |x_{i,j} - w_{i,j}| \quad , \text{and} \qquad (8)$$

• when $p = \infty$, the box-distance:

$$d_i = \|x_i - w_i\|^\infty = Max(|x_{i,j} - w_{i,j}|) \ . \qquad (9)$$

In the K-means image classification context, Estlick's experiments [3] show that these two alternative distance metrics gave acceptable performance and are more amenable to FPGA hardware implementation because the costly "squaring" operation is avoided.

The city-block distance, which requires n subtractions, n absolute values and $n - 1$ comparisons, is selected to implement the required distance computation in a hidden neuron. The distance circuit for a four input case is shown schematically in Fig. 5 (a).

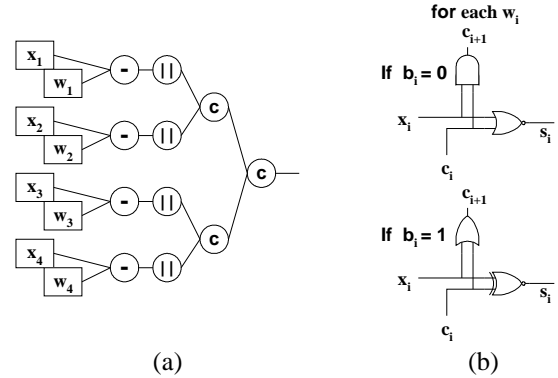(a)   (b)

Figure 5: (a) A four input distance circuit for a hidden neuron. (b) Two possible half-adders after folding of constant input weights into subtractors

Since the input weight w becomes known after the weights have been prescribed by training, significant real estate savings can be made by dynamically folding the now constant input weight w into the subtraction circuit. Hence, each of the subtractors is implemented by using a series of half-adders which take either of the configurations as illustrated in Fig. 5 (b). The circuits for absolute values and comparators in the comparator-tree are realised by using standard n bit adders and subtractors respectively.

3.2.2. Implementation of Activation Function
As the radius of generalization r_i also becomes a constant after training, the activation function is thus implemented as a n bit constant comparator KC as shown in Fig. 6. The constant

comparator is implemented by folding the constant r_i into a subtractor circuit as the subtractend in the same fashion as described above. The sign bit of the comparator is registered first then is connected to the MSB of bus h_i. This bit is set if $d_i < r_i$, else it is low. This signal is used in the final stage of the output layer circuit to select 1NN output as discussed in next subsection.

To simplify the control of the overall circuit, in addition to storing the radius of generalization constant, it was decided to also store the output weights v_i with the activation function circuit in a register bank. The b bit from the output weight v_i forms the next b bits of bus h_i. Storing and transmitting the output weights with the distance values is to avoid the need to track the indices for the k nearest neighbour hidden neurons and be able to load their corresponding output weights in the output layer stage. An alternative to this approach is to store and transmit the index encoding for each hidden neuron in the h_i bus. However, the index would occupy $\log(m)$ bits in the bus. When there are many hidden neurons (e.g. $m > 256$), the indices for hidden neurons will require more than 8 bits to represent, which is the width used to store and transmit the output weights v_i.

The distance d_i is also registered and then connected to the lower n bits of bus h_i. The comparator sign bit, the output weights v_i and the distance d_i are transmitted together through a $2b + 1$ bit wide bus h_i as illustrated in Fig. 6.
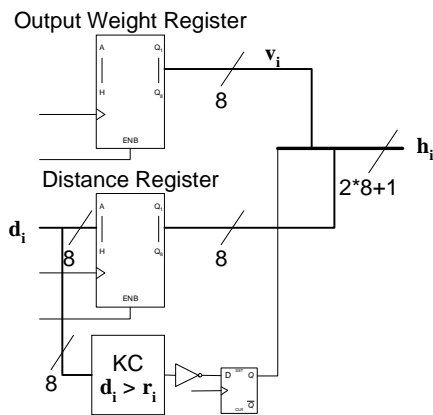

Figure 6: An eight bit example of the activation circuit.

A collection of buses $h = (h_1, h_2, ..., h_m)$ is connected to the fuzzy rule base for further processing.

3.2.3. Parameters of a Hidden Neuron The parameters of a hidden neuron are listed below.
Inputs to a hidden neuron:
- Input x: a vector of length n; each element has b bits.
- Input weight w_i: a constant vector of length n; each element has b bits;

- Radius of generalization r_i: a constant value which has b bits;
- Output weight v_i: a constant value which has b bits;
Output from a hidden neuron:
- bus h_i has $2b + 1$ bits with:
  bits [0:b-1] - distance d_i between x and w_i;
  bits [b:2b-1] - output weight v_i;
  MSB - set if $d_i < r_i$;

A hidden neuron is pipelined with b bit register banks after the constant subtractors, and absolute value units after each level of the comparator-tree. The depth of the pipeline is dependent on the size of the input layer. When there are n input neurons the depth of the pipeline is $3 + \lceil \log(n) \rceil$.

3.3. Implementation of the Fuzzy Rule Base

The primary function of the fuzzy rule base is to map the distances between a test vector and each of the training samples (stored in hidden neurons) into fuzzy membership grades. The fuzzy rule base accomplishes this function in two ways, 1NN and kNN, which are described below:
- 1NN: If and only if a test vector lies within the radius of generalization of a hidden neuron i (which implies $(d_i \le r_i)$ and $h^i_{MSB} = 1$), the hidden neuron i fires. In a FC network only one hidden neuron can fire at any given time because the distance regions of hidden neurons do not overlap. In this case, the fuzzy rule base functions as a 1NN. The fuzzy rule base acts as a gating function which assigns the fuzzy membership μ_i to 1 and assigns the rest $(\mu_j, j \ne i)$ to 0. The test vector will be classified as belonging to the same output class as the training sample stored at hidden neuron i.
- kNN: Further generalization is achieved by the fuzzy rule base if a test vector does not lay within any hidden neuron's radius of generalization, i.e. none of the $h^i_{MSB} = 1$. In this case the fuzzy rule base first selects k nearest neighbours of the test vector x from the distance vector and maps the distances between the test vector with each of its k nearest neighbours into a set of fuzzy membership grades $\mu_i, i = 1, 2, ..., k < m$. Through this process of "fuzzification", the decision of the FC network is further generalized.

Notice that with either the 1NN or kNN rule, the firing hidden neuron's index or the indices of hidden neurons corresponding to the k nearest neighbours are tracked and used to select their corresponding output weights. We have decided to simplify the fuzzy rule base because a faithful implementation of the above fuzzy rule base involves realizing complex controls and results in variable neuron firing rates. Recall from the above that 1NN rule fires instantaneously and selects its output immediately while kNN rule needs to

wait for the sorting circuit to select k nearest neighbours and then wait for the fuzzy computation of the output to finish. Specifically, the 1NN rule is now integrated into kNN, and the 1NN rule will not take effect until the final stage of the output layer. That is, regardless of whether a hidden neuron has fired, the kNN rule always applies. The appropriate 1NN or kNN output can always be selected at the last stage of the output layer by checking the most significant bit of bus h^i. If it is set, then a hidden neuron has fired and the 1NN output should be selected as illustrated in Fig. 11 in next subsection. In a sense 1NN is a special case of kNN: 1NN's output is the smallest value in the k nearest neighbour distance. Two benefits are achieved with this approach:

- no complex control circuitry is needed for output selection; and
- the output neuron firing rate is now constant.

3.3.1. Implementation of kNN circuit The kNN circuit performs two tasks:

- Task 1: given a test vector x, the kNN circuit selects the test vector's k nearest neighbours based on the distance vector h from the hidden layer.
- Task 2: it maps the distances between x and its k nearest neighbours into fuzzy membership grades μ.

The selection of k nearest neighbours is implemented using a bitonic selection network [4]. A bitonic selection network is used because its sorting components are very simple and the basic operations of the sorting algorithm are simple and highly parallel. Different stages in the sorting network are amenable to pipelining. Fig. 7 shows a schematic illustration of an eight-input bitonic sorting network using the notation and style used by Kumar et al. [5]. A bitonic sorting network takes an unsorted series of distances (transmitted by the lower n bits from bus h_i) and sorts them into a series with monotonically increasing order. Because we are only interested in selecting k nearest neighbours of a test vector, and we know that $k \ll m/2$, the bottom-left corner of the last stage of the bitonic sort network (shown in gray in Fig. 9) is not used.
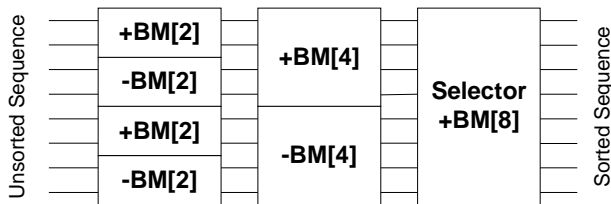
Figure 7: An eight input bitonic sorting network.

Given a bitonic sequence, a bitonic merging network, as shown in Fig. 9, is optimal and can be pipelined in $\log m$

stages. Each of the pipeline stages contains a total of $m/2$ two-input comparator-and-swap units labelled as either +BM[2] or -BM[2] as shown in Fig. 8. The two-input comparator-and-swap units +BM[2] or -BM[2] sort the two elements into a increasing and a decreasing order respectively, and can be expressed mathematically as follows:

$$+\text{BM[2]}: \begin{aligned} x &= min(l, r) \\ y &= max(i, r) \end{aligned} \qquad (10)$$

$$-\text{BM[2]}: \begin{aligned} x &= max(l, r) \\ y &= min(i, r) \end{aligned} \qquad (11)$$

As the FC network selects k nearest neighbours, we desire the k smallest distances to be sorted in monotonically increasing order, hence +BM[2] units are used.

Although bitonic sorting networks are optimal to sort bitonic sequences, overheads are incurred to convert an unsorted sequence into a bitonic sequence. For example, in the eight input bitonic sorting network shown in Fig. 7, the first two stages of the network convert an unsorted sequence into a bitonic sequence of length 8 so that the last stage of the eight-input bitonic sorting network can merge them into sorted sequence. The first two stages of the bitonic sequence converter unit are shown explicitly in Fig. 10.
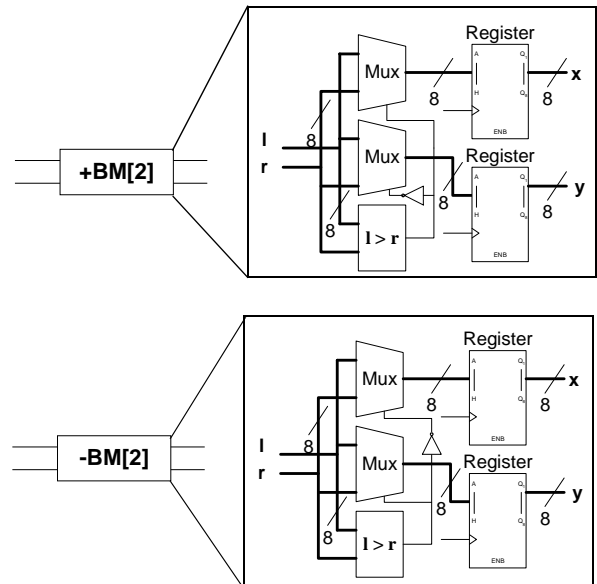
Figure 8: Two-input comparator-and-swap units: +BM[2] sorts two elements in increasing order, -BM[2] sorts two elements in decreasing order. An eight bits example is given.
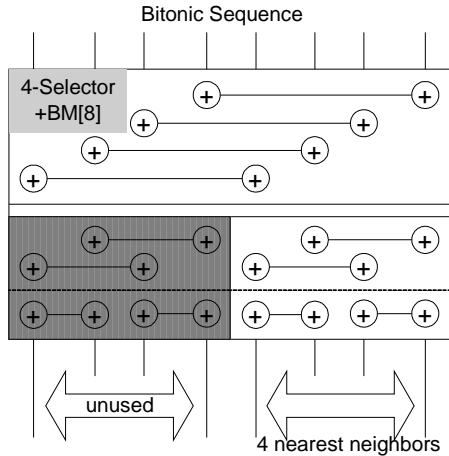
Figure 9: An eight bit example of bitonic sorting / selection network. The left half is unused, only first four outputs are selected.
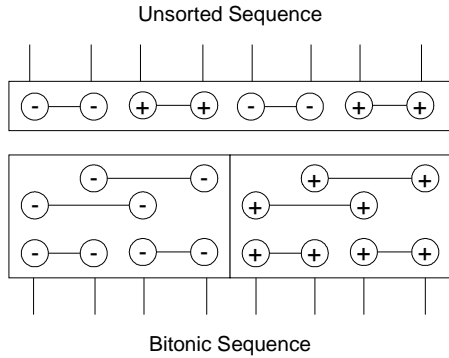

Figure 10: Bitonic converter network which merges an unsorted sequence into a bitonic sequence.

In our implementation, the unsorted sequences of distances are transmitted by h buses from each neuron in the hidden layer. The lowest n bits in each h bus represent the distance value. The distance values form an unsorted sequence. This unsorted sequence of distance values can be seen as a concatenation of bitonic sequences of size two. They are first converted by $\log n - 1$ stages of bitonic merging networks. Starting with pairs of +BM[2] and -BM[2] units, then pairs of +BM[4] and -BM[4] units, and so on, each stage merges a sequence which is twice as long as its input until a bitonic sequence of n is reached. This bitonic sequence of length n can then be merged by the n input bitonic sorting network and transformed into sorted sequences.

The final task of the fuzzy rule base is to map the selected k nearest neighbour distances into fuzzy membership grades. The triangular fuzzy membership function used in

Tang and Kak's original work for the two nearest neighbour case is shown in Fig. 4. Straightforward implementations of the triangular fuzzy membership functions would require a large number of division operations and would be very expensive to realize on FPGAs. Fortunately, their experiments also found that the FC network is robust with the choice of fuzzy membership functions. We have decided to use a linear approximation of the triangular fuzzy membership functions. In particular, depend on their ranking, each of the selected k nearest neighbours is assigned linear weights which approximate the triangle fuzzy membership function. For example, when $k = 4$, the fuzzy membership grades assigned to the four nearest neighbours are 24/32, 4/32, 2/32 and 2/32 respectively. The weights sum to 1 and monotonically decrease as ranking increases. Although other sets of weights are possible, the above weights are easy to implement in FPGAs as the weights are equivalent to a series of right shift operations as seen in the output neuron circuit (see Fig. 11).

3.3.2. Parameters of the Fuzzy Rule Base The parameters of the fuzzy rule base are listed below.

Inputs to the Fuzzy Rule Base:
- a set of m buses $h = (h_1, h_2, ..., h_m)$. Each bus is $2b + 1$ bits wide with:
  bits [0:b-1] - distance d_i between x and w_i;
  bits [b:2b-1] - output weight v_i;
  MSB - set if $d_i < r_i$;
- m a constant which is the number of hidden neurons;
- $k, (k \ll m)$ a constant which defines the number of nearest neighbours.

Output from the Fuzzy Rule Base:
- a selected set of k nearest neighbour buses $\mu = (\mu_i, \mu_2, ..., \mu_k)$ with the distance values $\mu_{[0:b-1]}$ sorted in monotonically increasing order. The width and content of bus μ is the same as bus h.

The fuzzy rule base circuit is pipelined with $2b + 1$ bit register banks after each stages of bitonic sorting network. Hence the depth of the pipeline is the depth of bitonic sorting network which is a variable which depends on m the number of hidden neurons: $\lceil (\log^2 m + \log m)/2 \rceil$.

3.4. Output Neuron Circuit

Buses belonging to sorted k nearest neighbours are connected to the output neuron circuit in a increasing order according to their ranking in the previous stage. The fuzzy membership grades are hardware encoded with respects to their ranks, and their outputs v_i the middle n bits of the bus are weighted by these fuzzy grades as illustrated in Fig. 11.

The contributions for all k nearest neighbours are then summed by an adder.
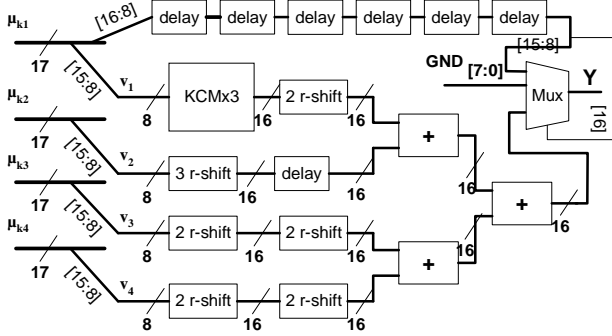

Figure 11: Output neuron circuit.

The final output is selected to be either 1NN or kNN depending on the MSB of the smallest neighbour of a test vector. If MSB is set, it means a hidden neuron has fired previously in the hidden layer and hence the corresponding output class of that hidden neuron is selected to be the final output. If MSB is not set, the kNN output is valid, and the final output is the kNN fuzzification output.

In our design, we have decided to fix $k = 4$ for convienence. The pipeline register banks of width b are inserted after each of parallel-right shift stages, and at each leaf of the addition tree. The pipeline depth in this case ($k = 4$) is 6. The final output is to be interpreted as a 16-bit signed fixed point two's complement number with 1 sign bit, 7 bit integer and 8 bit fraction.

4. Integrating Prescriptive Learning with the Design of FC Networks

The training for an FPGA based FC network is carried out offline because the training of an FC network is not computationally intensive. Offline training allows for compile-time folding of constants and hardwiring of constants into the circuit functionality. A JHDL package is written to fully parameterize the design for an FC network. The folding of the input weights w and the radius of generalization r into constant subtractors / comparators; the hardwiring of a set of fuzzy membership grades into constant multiplications and divisions in the output neuron circuit are carried out by the JHDL package at the compile-time. Given a set of training data and the required precision specified by the user, the JDHL package automatically defines the topology and weights for the FC network as well as designing the whole circuit for the appropriate FC network. Specifically, the JDHL package contains a main class and four circuit generator classes for instantiating the hidden neuron circuit, the activation function circuit, the bitonic sorting network

circuit and the output neuron circuit. The main class is responsible for instantiating the required circuit modules and connecting them together according to the problem specification, the training sample size and the number of those training samples to be stored. Each of the circuit generator classes is responsible for designing the corresponding submodules according to specific information such as the sizes, the constants and required stages of pipelines. As the final output, the main class generates an EDIF configuration for the fully instantiated circuit object as the solution for the required FC network.

5. Conclusions

We have shown that computational characteristics of FC networks are highly parallel, simple and modular. These characteristics are well suited for FPGA implementation exploiting fine-grained parallelism. We have also shown that the prescriptive learning scheme of FC networks can be integrated with the design of the FC network so that the implementation of an FC network can be fully parameterized. Strategies to reduce the resource cost by using compile-time constant folding techniques are also discussed. Given its fast training speed, and the ease of mapping into FPGA architectures, FC networks are a better alternative to other neural network models which are based on iterative learning.

References

[1] Chakrabarti, S., S. Roy, and M.V. Soundalgekar, "Fast and Accurate Text Classification via Multiple Linear Discriminant Projections", *in proceedings of International Conference on Very Large Data Bases*. 2001, Morgan Kaufmann: Hong Kong. p. 658-669.

[2] Tang, K.W. and S. Kak, "Fast Classification Networks for Signal Processing", *Circuits Systems Signal Processing*, 2002. 21(**2**): p. 207-224.

[3] Estlick, M., et al., "Algorithmic transformations in the implementation of K- means clustering on reconfigurable hardware", *in proceedings of the Ninth ACM International Symposium on Field-Programmable Gate Arrays*. 2001, ACM SIGDA: California. p. 103-110.

[4] Batcher, K.E., "Sorting Networks and Their Applications", *in Proceedings of American Federation of Information Processing Societies 1968 Spring Joint Computer Conference*. 1968, Thomson Book Company, Washington D.C.: Atlantic City, NJ, USA. p. 307-314.

[5] Kumar, V., et al., "Bitonic Sorting", *in Introduction to Parallel Computing - Design and Analysis of Algorithms*. 1994, The Benjamin/Cummings Publishing Company, Inc.: California. p. 214-224.