An Efficient Hardware Implementation of Competitive Learning Classifier for Gender Classification

James Mnatzaganian, *Student Member, IEEE,* and Qutaiba Saleh, *Student Member, IEEE*

***Abstract*—A Competitive Learning Classifier (CLC) is introduced as a suitable algorithm for performing real-time gender classification in embedded applications. CLC is a type of clustering algorithm, derived from competitive learning. The design was able to obtain an accuracy of up to 70 %. The hardware could perform training and testing operations at a rate of at least 25 MHz. The design footprint was 0.0176 mm^2 and consumed only 1.1 mW of power per epoch per 800 training and 1000 testing patterns. Due to the low power consumption, small footprint, and fast operation this design is suitable for real-time embedded applications.**

***Index Terms*—Competitive Learning Classifier (CLC), gender classification, ASIC, low power.**

GENDER classification is an important topic in social interactions. It can determine the greeting from one individual to another as well as the entire context of the conversation. Additionally, determining an individual's gender could be used to market products specifically for that gender. For example, a clothing store could have an automated advertising board to display products that would best suit the individual entering their store. If a gender detection system was being used in that manner, the system would need to be able to determine the gender classification in real-time, additionally it should consume as little power as possible.

This problem has been explored by many software approaches (e.g. [1]–[3]), but only a couple hardware implementations exist [4], [5]. For a real-time embedded system, the design should ideally be a low-power ASIC. None of the past designs created an ASIC and additionally none of those designs mentioned power consumption. This work developed a custom algorithm explicitly designed to be as efficient as possible while still achieving acceptable accuracies.

A. Dataset

The Labeled Faces in the Wild (LFW) [6] dataset was modified for use with gender classification. This dataset is comprised of one or more faces per person, with no labels to denote gender. To obtain the genders, the genderize.io [7] API was utilized. This API can be used to obtain the gender, with a confidence interval, for a given first name. Only results with over 90% accuracy were utilized.

The faces in the dataset are not all frontalized nor are they cropped to remove background noise. To remove noise, the frontalized images from Hassner et al. [8] were used as the dataset. This dataset frontalized and cropped all images from the LFW dataset, creating a cleaner dataset. The images were then resized to various sizes using bilinear interpolation. The raw pixels were then used as the features.

To ensure that the dataset would not be biased towards a particular individual, a single instance of each person was randomly chosen. The remaining images were then randomized and split by gender, to ensure an equal distribution of males and females. 400 of each gender were used for training and 200 of each gender were used for testing. All code for generating the dataset as well as the software and hardware models are available at [9].

J. Mnatzaganian, and Q. Saleh are with the NanoComputing Research Laboratory, Rochester Institute of Technology, Rochester, NY, 14623.

Manuscript received May 21, 2015.

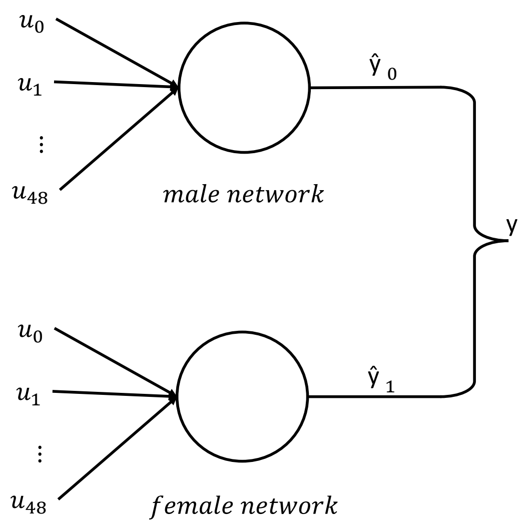
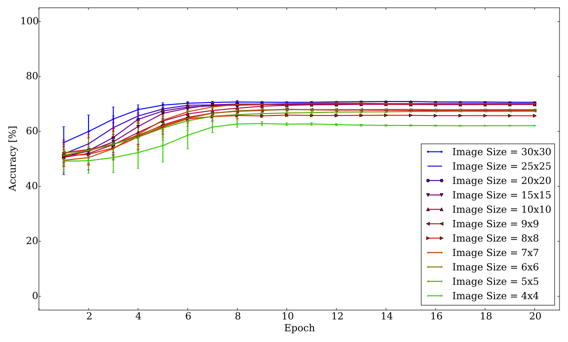


Fig. 1: A CLC consisting of two networks (male and female), 49 inputs per network, and a single output for each network.

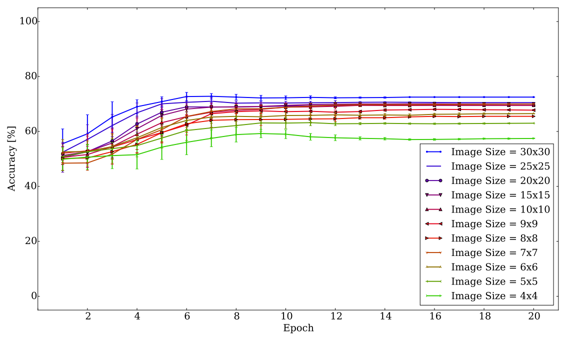
B. Design Approach

The primary design constraints were area and power. With this in mind, the system still had to perform its primary goal of gender classification. It was desired to obtain the highest possible accuracy; however, if a minor loss in accuracy resulted in major hardware improvements, favor was given to the hardware.

1) Algorithm: An algorithm similar to competitive learning was designed. This algorithm is coined Competitive Learning Classifier (CLC). The algorithm is comprised of N competitive



(a) Training results



(b) Testing results

Fig. 2: Training (2a) and testing (2b) results for various image sizes. Each network has a single output. The learning rate was set to 0.001.

learning networks, where N is the number of labels. Each network has I inputs and O outputs, where each output represents a cluster. The implementation of this algorithm used for this work is shown in Fig. 1. It consists of two competitive learning networks (one for males and one for females), 49 inputs per network (one for each feature), and a single output for each network.

$$\Delta w_{i,n} = \alpha \hat{y}_{o,n}^{(p)} (u_i^{(p)} - w_{i,n}) \tag{1}$$

$$\hat{y}_{o,n}^{(p)} = \sum_{i=1}^{I} (w_{i,n} - u_i^{(p)})^2 \tag{2}$$

$$J(\boldsymbol{w}_n) = \frac{1}{2} \sum_{p=1}^{P} \sum_{o=1}^{O} \hat{y}_{o,n}^{(p)} |\boldsymbol{w}_n - \boldsymbol{u}^{(p)}|^2 \tag{3}$$

In CLC, there is a randomly initialized weight for each input in each network. The weights are updated using the online update equation in (1), where $w_{i,n}$ is the weight of input i for network n, α is the learning rate, $\hat{y}_{o,n}^{(p)}$ is output o of network n for pattern p, and $u_i^{(p)}$ is input i for pattern p. Each output is computed by finding the squared Euclidean distance between the weight and the input, as shown in (2). The corresponding cost function is shown in (3), where P is the total number of patterns.

If $O > 1$ dead neurons may occur. To combat this, boosting is used, where frequently winning neurons are negatively boosted and infrequently winning neurons are positively boosted. Boosting is only updated during the learning phase, so after training the boost values are fixed.

Training occurs by selecting the network representing the current label and updating its weights. The other networks are left untouched. During testing, both networks receive the same input and compute their respective outputs. The neuron having the lowest value is the winning neuron and the network that it belongs to becomes the output of the network.

2) *Software:* The software design consisted of two phases. The first phase involved finding a set of parameters that would obtain suitable accuracies. Additionally, those parameters should be tweaked, if possible, to simplify the hardware. The second phase involved redesigning the CLC model to be identical to the hardware design. This includes any simplifications as well as performing calculations using the same precision that would be used in the hardware.

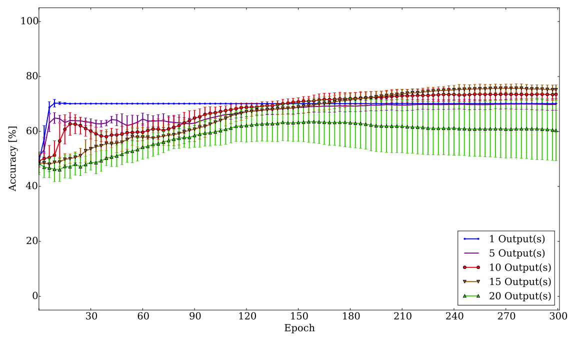
a) Parameter Optimization: A fully generic software model of CLC was created in Python. This model supports I inputs, N networks, and O outputs. For gender classification only two networks are required (male and female); however, the number of inputs as well as outputs for each network can both be set to any suitable value. In addition to those parameters, a suitable learning rate had to be determined.

To find suitable parameters, a simple search approach was taken, where one parameter was varied while all others were fixed. The best parameter from the current experiment was used in subsequent experiments. All simulations were performed 10 times for statistical purposes.

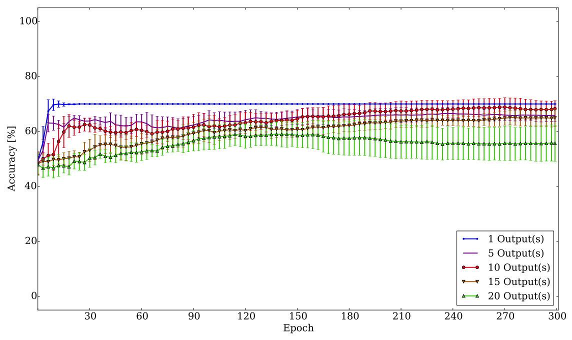
The first parameter varied was the number of inputs, i.e. the image size. The number of outputs for each network was fixed at one and the learning rate was set to 0.001. The training and testing results are shown in Fig. 2. The full image size (30x30) produced the highest accuracy, as expected; however, an image size of 7x7 produced a similar level of accuracy and would reduce the number of inputs per network by over 18x. For this reason, an image size of 7x7 was chosen.

The second parameter varied was the number of outputs. The image size was set to 7x7 and the learning rate was set to 0.001. The training and testing results are shown in Fig. 3. It is observed that a single output per network converges very quickly, while multiple outputs require hundreds of training epochs. Additionally, overfitting occurs once the number of epochs becomes too large. It is also observed that the variance increases drastically with multiple outputs. A final consideration is that once the number of outputs exceeds one, boosting must be performed. Boosting results in a substantial increase in HW complexity as it requires that a history of the past H outputs be stored per output. Additionally, an added computational strain is added for updating the boost values, calculating the network outputs, and selecting the network winner. For those reasons, a single output was chosen.

The final varied parameter was the learning rate. The image

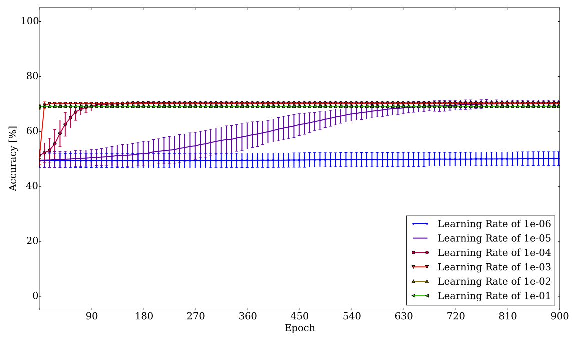


(a) Training results

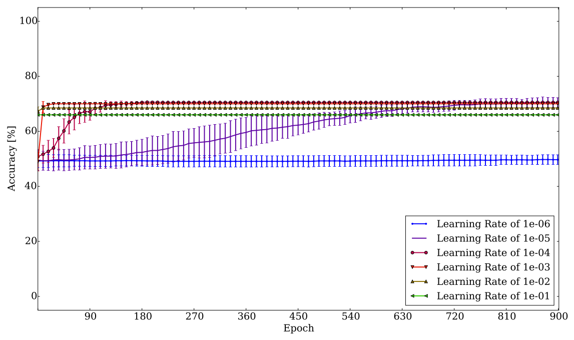


(b) Testing results

Fig. 3: Training (3a) and testing (3b) results for a various number of network outputs. The image size was 7x7. The learning rate was set to 0.001.



(a) Training results



(b) Testing results

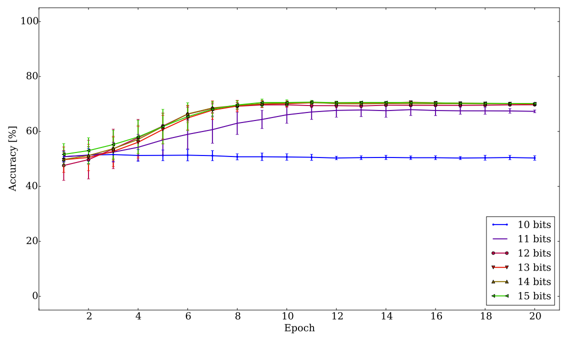
Fig. 4: Training (4a) and testing (4b) results for various learning rates. The image size was 7x7. Each network has a single output.

size was set to 7x7 and the number of outputs for each network was fixed at one. The training and testing results are shown in Fig. 4. When the learning rate is within a suitable range, the system converges on a similar output. If the learning rate is too small, the system is unable to learn. If the learning rate is too high, the system is unable to expand upon its learning. A learning rate of 0.001 was chosen, as this resulted in the best accuracies with the fastest convergence. This learning rate is additionally within a range suitable for representation within hardware.

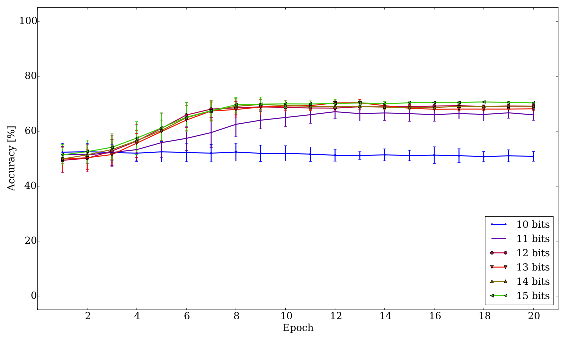
b) Hardware Model: The CLC model was redesigned to follow the hardware model as closely as possible. Boosting was removed from the model as it was assumed that there would only be a single output from each network. It was desired to use the Q fixed-point number format for representing all numbers. A Python datatype was created to allow all required arithmetic operations to be performed as a native Python datatype. This allowed for multiple benefits, namely ensuring that the arithmetic would resolve to the same exact values as in the hardware (including overflow and underflow conditions) and allowing for a near seamless transition between the pure software model and this new model.

In Q format it is important to know exactly what the bounds of the scalars will be. If they grow to be too large or too small, then they will no longer be able to be represented with the specified number of integer bits. Additionally, the level of precision must be noted. Each increase in decimal precision requires more fractional bits. With these concepts in mind, it is important to ensure that the numbers are all at a similar scale, as this will reduce the number of required bits. The chosen learning rate is 0.001, which is a relatively small number, and the magnitude of the weights and pixels are between zero and one. To keep the number of bits down, it was desired to ensure that the values did not exceed ± 1, because only a single integer bit would be required.

In (1), $\hat{y}_{o,n}^{(p)}$ is always going to be equal to one, as there is only a single network output and this output must always be chosen to be the winning cluster. The difference of $u_i^{(p)}$ and $w_{i,n}$ is able to be between -2 and 2, so with a single integer bit it is possible to have overflow; however, it is an unlikely scenario and should not drastically affect the overall output, as an overflow would result in the max positive value being returned. More importantly, because both $u_i^{(p)}$ and $w_{i,n}$ are of the same scale, the operation will result in a similar returned scale. This value is then going to be scaled by α, reducing the



(a) Training results



(b) Testing results

Fig. 5: Training (5a) and testing (5b) results for a various count of fractional bits.

resulting scale to be the same as α's. Since $I < \alpha^{-1}$ it is not possible for the resulting weight to overflow. This allows for (1) to be simply updated to (4).

In (2), the squared values of the difference of $u_i^{(p)}$ and $w_{i,n}$ are being summed. Squaring that difference will occur in an overflow in many cases. Additionally, summing those squared values will almost certainly result in an overflow. To address that problem, (2) is updated to be (5), where C is a scaling factor defined to be $I^{-1/2}$. This scaling factor ensures that the sum remains within a suitable range; additionally, performing the scaling operation before squaring eliminates overflow from that operation.

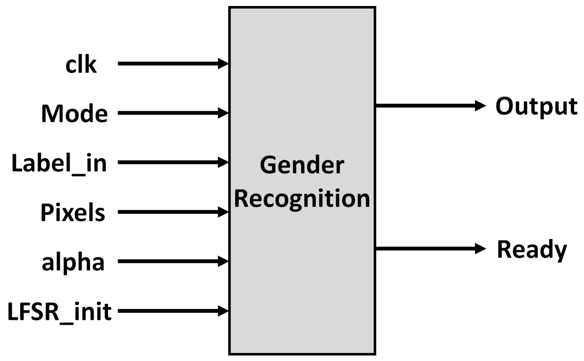
$$\Delta w_{i,n} = \alpha(u_i^{(p)} - w_{i,n}) \tag{4}$$

$$\hat{y}_{o,n}^{(p)} = \sum_{i=1}^{I}((w_{i,n} - u_i^{(p)}) * C)^2 \tag{5}$$

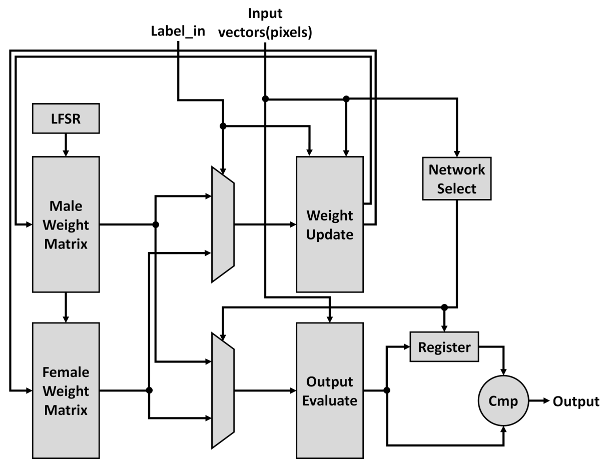
This hardware model introduces another parameter, the number of suitable fractional bits. To determine the minimum number of required bits, this new model was simulated across a range of count of fractional bits. The training and testing results are shown in Fig. 5. As seen, once the number of fractional bits drops below 11 the system is no longer able to function. At 11 bits or above, the accuracies are similar, thus 11 bits were chosen. This results in a total of 13 bits per number, which is still much smaller than the number of bits that would be required if floating point had been used.

3) Hardware: The hardware implementation (Fig. 6a) was designed to be identical to the software implementation. The system was designed structurally (Fig. 6b), allowing for modularity as well as flexibility in optimization. The system was divided into three main parts: two calculation blocks (one for the weight update during training and one for the output evaluation during testing) and a register block (used for storing the weights). These main components are controlled by a state machine.

The state machine, shown in Fig. 7, is the main control unit of the system. The input signal, *Mode*, controls the transition between the following states: *Idle*, *Initialization*, *Training*, and *Testing*. Based on those states the statuses of the output flags are changed. One external and three internal flags are driven by the state machine. The external flag is *Ready*. The internal flags are *Test*, *Train*, and *Reset*. Those flags are used to control the system. The system starts in *Idle* and waits for any changes in *Mode*. From *Idle* the state machine transits to all other states. The default use of the system is to transition to *Idle* between



(a) Entity



(b) Internal structure

Fig. 6: System block (6a) showing the hardware entity and the internal structure of the hardware (6b) showing the main components of the system: weight registers, weight update, output evaluate, LFSR, and comparator.

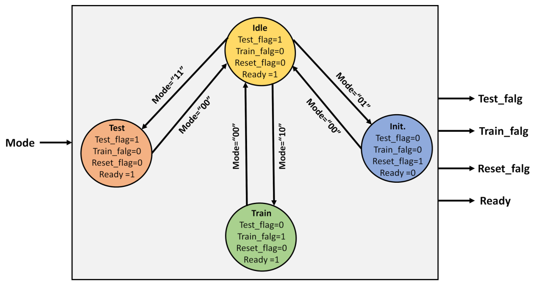


Fig. 7: State machine consisting of one input *Mode* and four ouputs. *Mode* controls the transition between the states (*Idle*, *Initialization*, *Training*, and *Testing*). The four output flags control the internal operation of the system.

all other states.

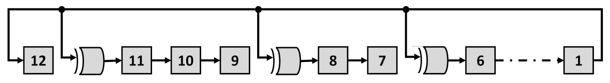
During *Initialization* the weights are randomly generated and stored in the male and female weight register banks. A 12-bit Linear Feedback Shift Register (LFSR) is used to generate the initial weights. The configuration of the LFSR is shown in Fig. 8a. Once the reset flag is active, the LFSR component takes the seed value from the input port *LFSR_init*. From that seed it generates the initial weight values at a rate of one value per clock cycle. Those values are then serially fed into the male and female weight register banks.

During *Training* the weights are updated by the weight update block. Each weight update takes two clock cycles. In the first clock cycle the appropriate weight block (male or female) sets its weights to be active. The weight update block takes the assigned weights and calculates the updated weights based on the input image. In the second clock cycle the new weight update values are saved into the associated weight's registers. The weight update block consists of a stack of identical smaller components. Each component is responsible for updating one weight value, as defined by (4). The internal structure of one weight update component is shown in Fig. 8b, where α is the learning rate as defined by the port *alpha*.

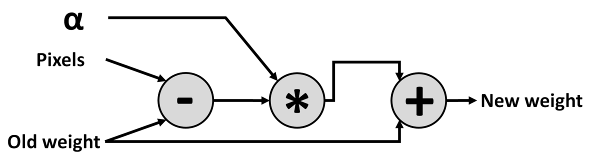
There is no limit on the number of training vectors that the system can learn. The user can supply new vectors as long as the system is in the *Training* state. Once the training process is completed, *Mode* should be set back to *Idle*. Returning back to *Idle* ensures that the training process will gracefully complete.

During *Testing* the output evaluate block (Fig. 8c) is used to calculate the output value of the network, as defined by (5). This block consists of a stack of one pixel's output evaluate component followed by an adder tree. The adder tree performs the summation of the individual weight-input distance calculations. The scale factor used in this component is saved in a register located in the output evaluate block and is distributed to all components in the block. This block is able to compute the output of a network every clock cycle.

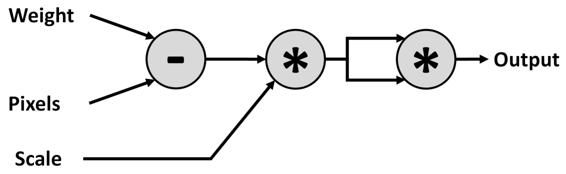
As previously explained, the system utilizes two identical networks (one for males and one for females). The hardware model exploits this to reduce the size and power of the design by almost 50 %. This was achieved by sharing the calculation blocks while storing the weight values separately. A drawback to this approach is that it reduces the testing performance



(a) LFSR



(b) Weight update component



(c) Output evaluate component

Fig. 8: Hardware computational elements, consisting of an LFSR (8a), weight update component (8b), and an output evaluate component (8c). The weight update block consists of stacked weight update components. The output evaluate blocks consists of output evaluate components connected to an adder tree.

by 50 %, but since the major design constraints were area and power, this was an acceptable loss. Additionally, sharing those blocks induced some overhead, as control signals were required to synchronize the pipeline between the networks. The network select block was designed for that purpose.

That network select block is a clock divider set to be half of the system clock frequency. Its output is used as a select signal for the multiplexer at the input of the output evaluate block, which is used to allow selection between the two networks. Additionally, that signal is used as an enable signal for the register at the output of the output evaluate block, which is used to store the output value of the male network. The output of that register and the output of the output evaluate block are connected to a comparator to perform the classification.

C. Results

The hardware design was initially tested in ModelSim. The same training and testing strategy used in the software were used for the hardware. Typical results are shown in Table I, where the final accuracies, along with their corresponding standard deviations, were obtained after 30 training epochs across 10 iterations. The accuracies are similar amongst the different implementations, with the ideal accuracy being 70 %. The hardware accuracies were able to reach that; however, they were typically lower and could be as much as 7 % lower, depending on the weight initialization. This high degree of variance is due to the loss of precision with only 11 fractional bits. This is loosely shown in Fig. 5b, where once the number of fractional bits reaches 12 the amount of variance decreases drastically. Based off these results, 12 or more fractional bits would have been a better alternative.

TABLE I: Accuracies for the Hardware and Software Models After 30 Epochs

Design	Train Accuracy	Test Accuracy
Ideal Software Model	70.125 ± 0.000 %	70.000 ± 0.000 %
Hardware Software Model	66.150 ± 0.378 %	64.200 ± 0.812 %
Hardware Implementation	68.363 ± 0.401 %	67.850 ± 0.950 %

The hardware design was synthesized with Synopsis' DC Compiler for TSMC018 process. The footprint of the design was 0.0176 mm^2. The total power of the system was measured to be 31.6 mW across all 30 epochs, resulting in a power consumption of 1.1 mW divided across 800 training and 1000 testing patterns (800 training + 200 testing). The area and power are both low, making this design suitable for embedded applications. Additional optimizations to both can be obtained. To improve the power the training and testing portions of the network could be disabled while they are not used. To improve the area a fully pipelined approach could be implemented, where only a single (or a few) functional components are used vs. an entire block of them.

The system was able to process a pattern in as few as two clock cycles (for both the training and testing phases). A clock period of 20 ns was used, allowing for a new pattern to be processed after only 40 ns. Additionally, the clock was not optimized, so it is possible that the system can operate at a faster rate. Additional performance was not necessary for performing real-time gender classification, so that level of optimization was not performed.

I. Conclusion

A CLC was implemented in both software and hardware to perform gender classification. A test accuracy of up to 70 % was obtained for the ideal software model. Due to the loss of precision in the hardware implementation, the accuracies were typically lower; however, with an optimal weight initialization similar accuracies could be obtained. The synthesized area of the hardware was 0.0176 mm^2. The hardware consumed 1.1 mW of power per epoch per 800 training and 1000 testing patterns. Both training and testing can operate at a rate of at least 25 MHz.

Given the extremely small size, low power consumption, and fast operation this design is suitable for real-time embedded applications. The performance is still many orders of magnitude faster than it would need to be for a simple marketing application. With that in mind, this design could be improved by performing power and area optimizations, such as disabling unused hardware and reducing the degree of parallelism. Additionally, the number of fractional bits could be increased to reduce the variance in the classification accuracy.

References

[1] E. Makinen and R. Raisamo, "Evaluation of gender classification methods with automatically detected and aligned faces," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 3, pp. 541–547, 2008.

[2] B. Moghaddam and M.-H. Yang, "Gender classification with support vector machines," in *Automatic Face and Gesture Recognition, 2000. Proceedings. Fourth IEEE International Conference on.* IEEE, 2000, pp. 306–311.

[3] G. Levi and T. Hassner, "Age and gender classification using convolutional neural networks."

[4] F. Smach, M. Atri, J. Mit, and M. Abid, "Design of a neural networks classifier for face detection," *Journal of computer Science*, vol. 2, no. 3, pp. 257–260, 2006.

[5] K. M. Irick, M. DeBole, V. Narayanan, and A. Gayasen, "A hardware efficient support vector machine architecture for fpga," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on.* IEEE, 2008, pp. 304–305.

[6] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments," University of Massachusetts, Amherst, Tech. Rep. 07-49, October 2007.

[7] "Genderize.io," Available at https://genderize.io, 2012, accessed on 2015-05-04.

[8] T. Hassner, S. Harel, E. Paz, and R. Enbar, "Effective face frontalization in unconstrained images," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[9] J. Mnatzaganian and Q. Saleh, "lfw_gender," Available at https://github.com/jwmqms/lfw_gender, 2015.