A HIGHLY SCALABLE RESTRICTED BOLTZMANN MACHINE FPGA IMPLEMENTATION

Sang Kyun Kim, Lawrence C. McAfee, Peter L. McMahon, Kunle Olukotun

Department of Electrical Engineering
Stanford University
353 Serra Mall Stanford, CA 94305 USA
email: {skkim38, lcmcafee, pmcmahon, kunle}@stanford.edu

ABSTRACT

Restricted Boltzmann Machines (RBMs) — the building block for newly popular Deep Belief Networks (DBNs) — are a promising new tool for machine learning practitioners. However, future research in applications of DBNs is hampered by the considerable computation that training requires. In this paper, we describe a novel architecture and FPGA implementation that accelerates the training of general RBMs in a scalable manner, with the goal of producing a system that machine learning researchers can use to investigate ever-larger networks.

Our design uses a highly efficient, fully-pipelined architecture based on 16-bit arithmetic for performing RBM training on an FPGA. We show that only 16-bit arithmetic precision is necessary, and we consequently use embedded hardware multiply-and-add (MADD) units. We present performance results to show that a speedup of 25-30X can be achieved over an optimized software implementation on a high-end CPU.

1. INTRODUCTION

A Deep Belief Network (DBN) [1] is a multilayer generative model where each layer encodes statistical dependencies between the units in the layer below it; it is trained to (approximately) maximize the likelihood of its training data. DBNs have been successfully used to learn high level structure in a wide variety of domains, including handwritten digits [1] and human motion capture data [2].

Although the use of DBNs to solve difficult AI problems looks promising, it is limited by the high computational cost of training the network on conventional processors. In this paper we address the problem of accelerating the training of DBNs using FPGAs.

Modern FPGAs contain a large amount of configurable logic elements, which allow custom designs for complicated algorithms to be built. Abundant logic resources and the customizable nature of FPGAs allow us to fully exploit the fine-grain parallelism in the DBN training algorithm. We describe an FPGA-based system that accelerates the training of DBN networks. Our implementation uses a single FPGA, but we have produced an architecture that we believe will be able to scale to many FPGAs, and thus will allow the training of larger DBNs. The size (number of neurons) of a DBN that can be trained is limited by the computational and memory resources on an FPGA board, so this scalability is important for future developments.

2. RELATED WORK

There has been considerable interest in accelerating the training of neural networks using FPGAs. In 1992, Cox and Blanz [3] demonstrated an FPGA implementation of a layered neural network for performing classification tasks. In 1994, Lysaght et al. [4] showed that dynamic reconfiguration of FPGAs could be used to train larger layered networks. Zhu and Sutton [5] provide a survey of FPGA implementations of neural networks, trained using backpropagation.

Since the introduction in 2006 of Hinton et al.'s fast learning algorithm [1] for DBNs (Deep Belief Nets), there has been renewed interest in neural networks. Ly and Chow [6] recently introduced an FPGA architecture for training DBNs. Our work improves on theirs in three significant ways. First, our approach generalizes the data representation of the network from binary numbers to fixed-point numbers. Modern FPGAs contain abundant multiplier resources, whether they are used or not — we exploit these resources so that a wider range of experiments can be conducted on the RBM accelerator. Second, major learning parameters such as the number of visible/hidden neurons are configurable at runtime — flexibility expands the exploration space for AI research even further. Third, instead of the deterministic threshold approach used in [6], our implementation operates as a true stochastic RBM which makes use of the sigmoid function and random number generators, which are explained in Section 4. Also, biases and momentum are included in the weight update phase for even faster convergence.

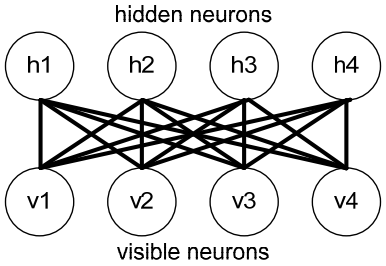

Fig. 1. Illustration of RBM Network.

-Visible neurons initially set to a batch of training examples, denoted vis_batch_0
-Repeat until convergence {
1) Sample hid_batch_0 from P(h|vis_batch_0)
a) *tmp_matrix_1* = *vis_batch_0* * *weights*
b) *tmp_matrix_2* = *tmp_matrix_1* + *hid_biases*
c) *tmp_matrix_3* = sigmoid(*tmp_matrix_2*)
d) *hid_batch_0* = *tmp_matrix_3* > rand()
2) Sample *vis_batch_1* from P(v|*hid_batch_0*)
3) Sample *hid_batch_1* from P(h|*vis_batch_1*)
4) Update parameters:
a) *weights* += α(*vis_batch_0*^{T}**hid_batch_0* - *vis_batch_1*^{T}**hid_batch_1*)
b) *vis_biases* += α(*vis_batch_0*^{T}***1** – *vis_batch_1*^{T}***1**)
c) *hid_biases* += α(*hid_batch_0*^{T}***1** – *hid_batch_1*^{T}***1**)
}

Fig. 2. RBM Training Algorithm Pseudo-code.

3. DEEP BELIEF NETWORK AND RESTRICTED BOLTZMANN MACHINE

RBMs, introduced in [1], are probabilistic generative models that are able to automatically extract features of their input data using a completely unsupervised learning algorithm. RBMs consist of a layer of hidden and a layer of visible neurons with connection strengths between hidden and visible neurons represented by an array of weights (see Fig. 1). To train an RBM, samples from a training set are used as input to the RBM through the visible neurons, and then the network alternatively samples back and forth between the visible and hidden neurons. The goal of training is to learn visible-hidden connection weights and neuron activation biases such that the RBM learns to reconstruct the input data during the phase where it samples the visible neurons from the hidden neurons.

Fig. 2 shows the pseudo-code for the RBM training algorithm. The sampling between the hidden and visible layers is followed by a slight modification in the parameters (controlled by the learning rate α) and repeated for each data batch in the training set, and for as many epochs as is necessary to reach convergence.

The motivation for using RBMs is that when stacked together in a hierarchical fashion, with the hidden units of one RBM used as the visible inputs to the next higher RBM — which describes the architecture of a DBN [1] — one can automatically learn "patterns-of-patterns" of the training set. Ideally, given enough layers, the user can learn very abstract features of the training set, with the intention of modeling the hierarchical learning structure of the brain. Some recent work using DBNs includes an application to classifying handwritten digits [1] and comparison of sparse DBN output to the V2 area of the visual cortex [7].

4. IMPLENTATION DETAILS OF AN RBM ON AN FPGA

It is clear from Fig. 2 that the RBM training algorithm is dominated by matrix multiplication. In a software implementation of an RBM running on a Sun UltraSparc T2 processor, the percentage of runtime consumed in matrix multiplication is between 90% for a small (512x512) network and 98% for a larger (2500x2500) network, Hence, the algorithm can run considerably faster by accelerating the matrix multiplication.

FPGAs are a promising technology with which to accelerate DBN training; the presence of fine-grain parallelism that can be exploited in the matrix multiplications is a promising attribute. Furthermore, we have found the use of fixed point arithmetic with just 18, or even fewer bits, does not adversely affect the results, which is explained in Section 4.1. This is fortuitous, since the built-in hardware multiplication units in modern FPGAs now support very similar precision calculations.

We have implemented a Restricted Boltzmann Machine on a development board that features an Altera Stratix III FPGA with a DDR2 SDRAM SODIMM interface. The Stratix III EP3SL340 has 135,000 ALMs (Adaptive Logic Modules)[1], 16,272 kbits of embedded RAM and 288 embedded 18x18 multipliers. With this number of multipliers, we are capable of processing approximately 256 neurons per clock cycle. The FPGA board can also be connected to up to 19 other boards in a stack via a high speed interface – this will be used in future work to scale up the size of the DBNs that can be processed. Although the current implementation of our system only supports a single board, the system was designed to be highly scalable as described in Section 6.

4.1. Overall System Architecture

Fig. 3 summarizes the structure of our FPGA implementation. The system consists of a soft processor, a DDR2 SDRAM controller, and an RBM module. The soft processor used in our design is the Altera Nios II operating at a clock frequency of 100 MHz. The processor functions as the interface between the user and the RBM module via JTAG-UART[2]. The CPU also initializes the weights, reads in the

[1] ALMs are essentially two 6-input ALUTs combined

[2] Future implementations will support faster interfaces for data transfer such as USB 2.0

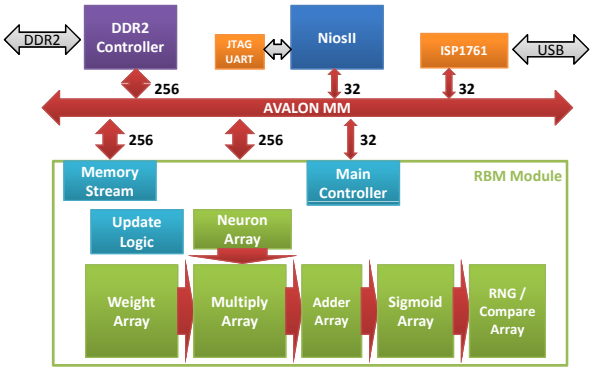

Fig. 3. Overall architecture.

visible neurons to SDRAM, initiates the algorithm, and returns the results to the user.

The RBM module, operating at 200MHz, is the key component that executes the algorithm with configurations chosen by the user. At a high level, the RBM module has an array of weights and neurons that are fed into an array of multipliers, and then into adders to perform the matrix multiplication. After that, the RBM computes the sigmoid to obtain the probability of firing a neuron and fires the neuron using a comparator and random number generator. After the positive and negative phases, the module continues to iterate until it meets the stopping condition given by the user. Every step is pipelined, which results in a throughput of approximately 256 multiply-and-add (MADD) per cycle.

The choice of the arithmetic precision to use in our design was a critical one, since the logic and multiplier resource utilization depends directly on the data width and format[3]. Neural networks exhibit soft computing [8] characteristics, which refers to a collection of software techniques that exploit the tolerance to noise for better performance and decision making. To determine the optimal precision, we simulated the DBN using the Fixed-Point MATLAB Toolbox for several fixed-point formats. From the simulation results, 16-bit fixed-point numbers were chosen to represent the weights and the training data set. Previous studies [9] also demonstrate that 16-bit precision is sufficient for a large range of neural network benchmarks.

As shown in Fig. 4, the RBM module is segmented into several groups, each consisting of an array of multipliers, adders, embedded RAM, and logic components. Weights and neuron data are distributed across the groups. Each group processes a different portion of the network. Nearly all computations take place in these groups. The rationale for such partitioning is that wire delay increases as semiconductor technology scales, so the wire delay becomes the performance bottleneck if the placement and routing is not performed efficiently. Localization of communication is an

[3]64-bit versus 16-bit, floating-point versus fixed-point

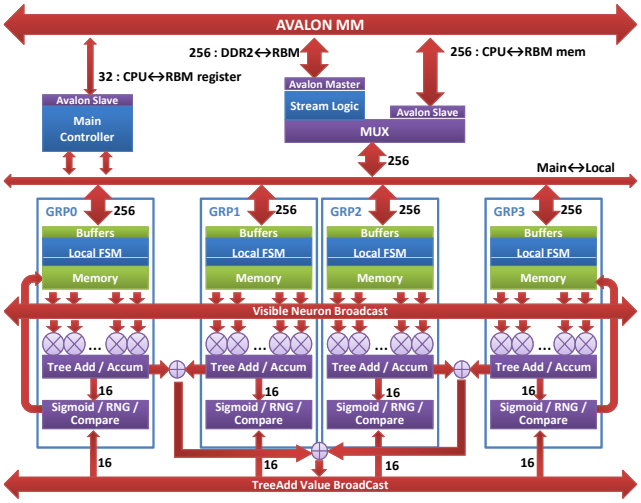

Fig. 4. RBM Module Detail.

efficient way, and possibly the only way, to fully exploit all the parallelism in modern FPGAs. Signals that must communicate with other groups are appropriately buffered.

Partitioning the design into multiple groups also makes the design scalable. Since most of the algorithm is performed within each group, the design can be migrated to a future device easily by instantiating more of these groups without having to worry about wire delays or routing. This is because most of the wiring is localized and the global signals are buffered. This also applies when extending the system to multiple boards. Each group was sized to match the DDR2 bus width of 256 bits, allowing for 16 multipliers per group with 16 bits of data precision.

A significant goal of this project is to facilitate AI research on large DBNs. To provide sufficient speedup, flexibility — relative to a software implementation — had to be sacrificed. Nonetheless, our system provides configurable parameters to allow wide-ranging experiments without the need to modify the FPGA design. The most significant parameter is one that allows the user to specify the number of neurons for each layer. This is in contrast to the implementation reported in ref. [6], which requires the network to be symmetric. However, for maximum performance, the number of neurons should either be a multiple of the number of multipliers, or vastly larger than the number of multipliers[4].

The system allows the user to specify the learning rate. The representation of neurons as either fixed-point numbers or binary numbers is also configurable. This widens the exploration space to non-binary numbers; non-binary numbers can already be found in the software RBM implementations for handwritten digits [1]. This generalization requires a multiplier as opposed to AND gates, which was

[4]This is only possible for our future multi-board implementation due to memory constraints.

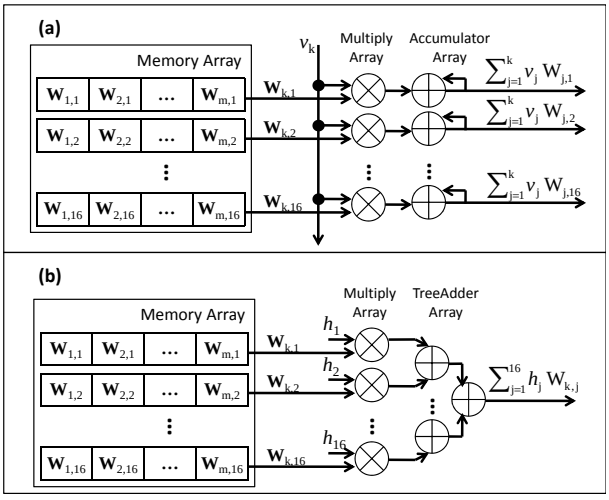

Fig. 5. Matrix Multiplication for Computing (a) hidden neurons (b) visible neurons

demonstrated in [6]. Currently, the visible neurons are hard-coded to use fixed-point representation, while hidden neurons make use of both. However, we can easily modify the design to make them configurable using multiplexers.

Due to the pipeline structure of our implementation, multipliers are only fully utilized when the number of neurons is a multiple of 256. Although this is not a significant limitation when our objective is to accelerate large DBNs, this considerably restricts the range of experiments for the current single board implementation since the on-chip memory only supports a weight matrix of size up to 512x512. However, as mentioned in Section 6, future implementations will be capable of accepting more nodes per board since the size of the network will no longer depend on the on-chip memory capacity.

4.2. Core Components

Matrix multiplication occurs in all three phases: the hidden and visible neuron sampling phases and the weight update phase. Thus, the input for the multiplication operations, which are the weights and the neurons, should reside in the embedded memories distributed across the FPGA. Although locating the inputs close to the multipliers is desirable, distribution of the weights is non-trivial due to a transpose operation that occurs during the visible neuron sampling phase.

The design in ref. [6] avoids the transpose problem by distributing the data such that no embedded RAM will simultaneously read out two or more elements from the same row with the same address, and no embedded RAM will contain two or more elements of the same column. Then, by using a carefully designed addressing scheme, a column or row of the matrix is read out from the memory each cycle and no communication is required for the transpose.

Although this approach eliminates communication for the transpose operation, it has two major drawbacks. One is that the weight and data matrices have to be shifted before being written into the on-chip memories, requiring a sophisticated routing scheme from each RAM to the appropriate multiplier since no row or column vector of the weight matrix contains the same index number. A more critical problem is that this approach assumes that the weight matrix fits on-chip and the number of RAM blocks for the weight matrix is equal to the number of neurons, or at least $\mathcal{O}(n)$. However, if the network size scales to a point that the weight matrix no longer fits on-chip, then the weight matrix has to stream in from off-chip memory. Since the embedded RAMs can no longer be equal to the number of neurons, a different weight matrix routing logic is required each time a portion of the weight matrix is streamed in, severely limiting the scalability. Although our current RBM implementation also assumes that the weight matrix fits on-chip, our approach solves the transpose problem in a way that scales to large networks where the weight matrix is stored on off-chip DRAM. Fig. 5 briefly illustrates our approach.

To understand how our module works, the key observation required is that matrix multiplication can be viewed in several different ways; a matrix multiplication $C = A \cdot B$ ($A \in \mathbf{R}^{m \times k}$, $B \in \mathbf{R}^{k \times n}$) can be considered as multiple linear combinations of vectors (1), multiple vector inner products (2), or as a sum of vector outer products (3).

$$\begin{bmatrix} C_{1,j} \\ C_{2,j} \\ \cdots \\ C_{m,j} \end{bmatrix} = \sum_{i=1}^{k} \left(B_{i,j} \begin{bmatrix} A_{1,i} \\ A_{2,i} \\ \cdots \\ A_{m,i} \end{bmatrix} \right) \tag{1}$$

$$C_{i,j} = \begin{bmatrix} A_{i,1} & A_{i,2} & \cdots & A_{i,k} \end{bmatrix} \cdot \begin{bmatrix} B_{1,j} \\ B_{2,j} \\ \cdots \\ B_{k,j} \end{bmatrix} \tag{2}$$

$$C = \sum_{i=1}^{k} \left(\begin{bmatrix} A_{1,i} \\ A_{2,i} \\ \cdots \\ A_{m,i} \end{bmatrix} \times \begin{bmatrix} B_{i,1} & B_{i,2} & \cdots & B_{i,n} \end{bmatrix} \right) \tag{3}$$

If the reconstruction phase ($HW^\top$) is viewed as vector inner products, then each row of H and each column of $W^\top$ is multiplied element-wise, followed by a sum reduction. This suggests that each row of $W^\top$ and each row of H should be placed in separate on-chip RAMs so that all of these elements can be read simultaneously, as shown in Fig. 5b. For the hidden computation phase (VW), consider the transposed matrix operation ($W^\top V^\top$), and view the operation as a linear summation of vectors. This requires that the *j*th column vector of $W^\top$ is multiplied by the *j*th element in a column vector of $V^\top$. This gives the structure of

Table 1. Resource utilization.

Resource		ALUT		Registers	Block Memory	18x18 DSP
		Combinational	Memory			
RBM	Segment	3893 (1.44%)	2205 (1.63%)	9552 (3.54%)	589824 (3.54%)	18 (6.25)
	Global	1564 (0.58%)	0 (0.00%)	2238 (0.83%)	0 (0.00%)	0 (0.00%)
SOPC		6670 (2.47%)	320 (0.24%)	5740 (2.13%)	123104 (0.74%)	0 (0.00%)
System Total		71494 (26.48%)	35600 (26.37%)	160996 (59.63%)	9560288 (57.38%)	288 (100.00%)

Fig. 5a, which computes multiple hidden neurons in parallel. Since at each cycle we only need to read a column vector of $W^\top$ for both cases, the memory layout for the weights can remain the same, and it requires no communication for a transpose as shown in Fig. 5.

Our approach requires more adders, since the adder requirements for each phase is different. The reconstruction phase uses an adder tree to compute reconstructed visible neurons each cycle[5]. The hidden neuron computation phase requires accumulators instead, holding 256 partial sums; this also computes (on average) one hidden neuron each cycle, assuming that the number of hidden units is equal to the number of multipliers. Using redundant hardware is acceptable since modern FPGAs have abundant logic resources. This approach provides a scalable method to perform matrix multiply operations without any movement of data in a transpose, at the cost of using additional hardware.

The update phase involves multiplying the transpose of the visible neuron matrix and the hidden neuron matrix, which is essentially the sum of outer products (3) between the visible and hidden neurons. Since the visible data already has a datapath that broadcasts as in Fig. 5a, and a large number of hidden neurons can be read into the multipliers as in Fig. 5b, the update phase multiplication can fit easily into the structure in Fig. 5a, where hidden neuron values take the place of weights.

The matrix multiplication results are provided to an activation function, which in our case is the widely used sigmoid function. Since the sigmoid function in hardware is an expensive operation — requiring exponentiation and division — we instead implemented an approximate sigmoid function design called PLAN (Piecewise Linear Approximation of Nonlinear function) [10] since it uses only a minimal number of addition and shift operations. In software simulations we found that the convergence properties were not degraded by the use of this approximate sigmoid function instead of the exact sigmoid function.

The stochastic characteristics of an RBM are also greatly influenced by the quality of the random number generator (RNG). We used the RNG described in ref. [11], which is a combination of an LFSR (Linear Feedback Shift Register) of 43 bits and a CASR (Cellular Automata Shift Register) of 37 bits, providing good statistical properties, along with a cycle length of 2^{80}, which is sufficient for our application.

[5]This is true if the number of hidden neurons is equal to 256. Larger networks can take several cycles to compute a visible neuron.

5. EXPERIMENTAL RESULTS

Table 1 summarizes the resource utilization of the RBM computation engine in our single FPGA design and for the entire system. It should be noted that partitioning the design may increase the total logic count. Thus, partitioning can be seen as a trade off of scalability and performance against silicon area.

5.1. Evaluation

The FPGA implementation was verified by comparing the results with the reference MATLAB implementation from Hinton et al. [1]. The MATLAB RBM code was modified to use fixed-point representation; the FPGA version of the RBM was only determined as correct when the results matched the MATLAB output stream, given the same input.

Performance measurement was done in comparison against an Intel Core 2 processor clocked at 2.4GHz running a single-threaded version of the RBM application. MATLAB was used for the comparison since MATLAB is highly optimized for matrix operations, which usually performs at least comparably to C implementations, if not better. For a fair comparison, both single and double precision versions of the MATLAB RBM were used. We did not consider a fixed-point version, since this would be very inefficient in MATLAB.

Three network sizes — 256x256, 512x512 and 256x1024 — were tested and compared to see how our system performs on small, large, and asymmetric networks. Performance measurement was done only on the execution of the algorithm itself; the time for data transfer between the host and the onboard DRAM was not taken into account. The current implementation uses a JTAG-UART interface to transfer the data. However, future implementations will have USB 2.0, which provides sufficient bandwidth that data transfer time will be negligible compared to the computation time.

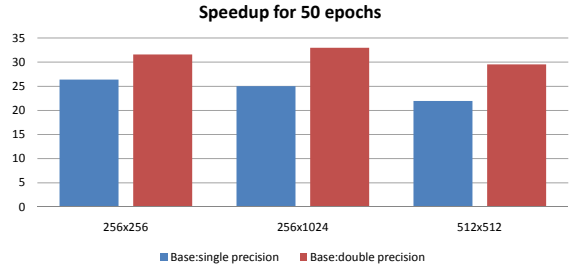

Fig. 6. Speedup for 256x256, 256x1024, 512x512

Fig. 6 shows the speedup achieved by our implementation. Our RBM system runs 25 times as fast as the single precision software implementation, and 30 times as fast as the double precision implementation. Based on these results, if we extend this single board RBM implementation to a system of 20 FPGA boards, we can get up to 600X speedup compared to the software implementations running on a modern processor.

6. FUTURE WORK

The current RBM system already provides the flexibility to adjust the network architecture and learning rates. Additional flexibility, such as sparse DBN computation or lateral connections within layers, may be included according to the requirements from a group of AI test users. In addition, our system will be extended to multiple boards and handle large networks. The weight matrix will no longer fit in on-chip memory since it scales as $O(n^2)$ with the number of neurons. Thus, weights will need to be streamed in from external storage such as DRAM. To tackle bandwidth issues, a batch size of at least 16 will be used. This enables weights to be reused for multiple data vectors within the batch to reduce bandwidth, at the cost of slightly increased number of iterations to converge. Our calculations show that for a batch size of 16, only 256 bits of weight data are needed every cycle, which is feasible with a DDR2 interface.

7. CONCLUSION

Deep Belief Nets are an emerging machine learning tool, which are based on Restricted Boltzmann Machines. FPGAs can effectively exploit the inherent fine-grained parallelism in RBMs to reduce the computational bottleneck for large scale DBN research. As a prototype of building a fast DBN research machine, we implented a high-speed, configurable RBM on a single FPGA. We have demonstrated a 25X speedup of the RBM implementation on the FPGA compared to a single precision software implementation running on a Intel Core 2 processor. Our future implementation using multiple FPGA boards is expected to provide enough speedup to attack large machine learning problems.

8. ACKNOWLEDGEMENTS

We acknowledge Andrew Ng, Honglak Lee, and Rajat Raina for their comments on computational needs for AI research. Special thanks to Nishant Patil for building a smaller version of RBM on FPGA. Sang Kyun Kim acknowledges funding support from the Kwanjeong Educational Foundation. Lawrence McAfee and Peter McMahon were supported by Stanford Graduate Fellowships.

9. REFERENCES

[1] G. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets." *Neural Computation*, vol. 18, pp. 1527–1554, 2006.

[2] G. W. Taylor, G. E. Hinton, and S. T. Roweis, "Modeling human motion using binary latent variables," in *Advances in Neural Information Processing Systems 19*. MIT Press, 2007, pp. 1345–1352.

[3] C. Cox and E. Blanz, "Ganglion - a fast field-progammable gate array implementation of a connectionist classifier," *IEEE J. Solid-State Circuits*, vol. 28, no. 3, pp. 288–299, 1992.

[4] P. Lysaght, J. Stockwood, J. Law, and D. Girma, "Artificial neural network implementation on a fine-grained fpga," *Field-Programmable Logic Architectures, Synthesis and Applications*, vol. 849, pp. 421–431, 1994.

[5] J. Zhu and P. Sutton, "Fpga implementations of neural networks: a survey of a decade of progress," in *Proc. 13th International Conference on Field-Programmable Logic and Applications*, Sept. 2003, pp. 1062–1066.

[6] D. Ly and P. Chow, "A high-performance fpga architecture for restricted boltzmann machines," in *Proc. of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2009, pp. 73–82.

[7] H. Lee, C. Ekanadham, and A. Ng, "Sparse deep belief net model for visual area v2," in *Advances in Neural Information Processing Systems 20*. MIT Press, 2008, pp. 873–880.

[8] L. Zadeh, "Fuzzy logic neural networks and soft computing," *Communications of the ACM*, vol. 37, no. 3, pp. 77–84, 1994.

[9] J. Holt and T. Baker, "Back propagation simulations using limited precision calculations," in *Proc. International Joint Conference on Neural Networks*, vol. 2, July 1991, pp. 121–126.

[10] H. Amin, K. Curtis, and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proc.-Circuits Devices Syst.*, vol. 144, no. 6, pp. 313–317, 1997.

[11] T. Tkacik, "A hardware random number generator," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. Lecture Notes in Computer Science, vol. 2523, Jan. 2003, pp. 450–453.