Restricted Boltzmann Machine in HLS

David Lau* and Ryan Kastner*
*University of California, San Diego

***Abstract*—With the rise in popularity of neural networks in machine learning applications, it is important to develop faster and more specialized hardware for different parts of the network. Most neural networks are implemented purely in software with the use of general purpose processors to execute computationally expensive algorithms. As a result, it is difficult to make neural networks scale to larger sizes with more features and more hidden nodes. In this paper, we make use of FPGAs to speed up the Restricted Boltzmann Machine (RBM) algorithm often used as a feature learning step in Deep Belief Networks. Restricted Boltzmann Machines are a natural aspect to optimize because it makes heavy use of matrix multiply operations, allowing FPGAs execute multiple calculations in parallel. The pipelined RBM results in an architecture with over 35 times speed up over the pure software approach at each epoch.**

***Keywords*—RBM, HLS, Feature Learning, Hardware, FPGA.**

I. Introduction

Artificial Neural Networks have become very popular in the machine learning world due to their ability to model many different types of problems including speech recognition and computer vision [1] [2]. One such neural network which is very popular in deep belief networks is the Restricted Boltzmann Machine. The goal of the Restricted Boltzmann Machine is to extract useful features from data in order to provide a better data representation than the original input [3]. Since larger neural networks can sometimes take several days to train [4], it is important to explore the available optimizations in order to increase performance and scalability.

To increase the performance and scalability of Restricted Boltzmann Machines, we must design an architecture for Field Programmable Gate Arrays which can take advantage of the innate parallelism of the algorithm. By creating a hardware-specific architecture, it is possible to scale the algorithm to a larger number of visible and hidden nodes. This new architecture will be possible to tackle more complex problems and learn more features about larger data sets.

In this paper, we present a Pipelined Architecture for Restricted Boltzmann Machines which reduces the latency of each epoch of weight update with respect to the full training set and scales to networks. To reduce the number of BRAMs needed to store connections between visible and hidden nodes, a 16 bit fixed point representation is used to store weight values. This smaller size allows for a larger number of visible nodes and hidden nodes to be present in the network. Each of the weight update stages of the RBM are pipelined in order to increase throughput and all matrix multiplication operations are executed in parallel to decrease latency.

The rest of this paper is outlined as follows: Section 2 provides some related works, Section 3 describes the background mathematics of the RBM algorithm, Section 4 describes the detailed RBM architecture, Section 5 presents the results and finally Section 6 contains the conclusion of the paper.

II. Related Works

Many papers have been published detailing a specific architecture for RBMs. In Ly and Chow [5], they introduce a RBM architecture that can be scaled to multiple FPGAs using an Alternate Gibbs Sampling approach for data and model sampling. Their method produces results which scale in O(n) resource usage and performance as the size of the network grows exponentially. They further develop this architecture to scale to thousands of visible and hidden nodes to show how a multi-FPGA architecture can support very large RBMs [6]. Kim et. al. [7] introduces a factored RBM architecture which uses 12 pipelined stages to implement the weight update routine. Due to the complexity of the factored RBM, their architecture does not get as much performance gain as the one introduced in Ly and Chow, however, it allows for a network which learns features between two input sets X and Y.

III. Restricted Boltzmann Machine Background

RBM is a neural network which is stochastic and generative. A stochastic network introduces random variations into the network which allow it to escape a local minima. A generative model is a probabilistic model of all features of the input set which can generate observable data values. RBM learns features about input data in order to provide a better representation which may speed up or improve learning in a larger neural network.

The RBM consists of visible and hidden nodes which have binary states, 1 or 0 (on or off). The number of visible nodes depends on the number of features in the original input data while the number of hidden nodes is a hyperparameter which can be chosen using heuristics. The visible layer consists of the input vector used to represent the data coming into the RBM while the hidden layer is used to capture the representation of the learned features of the data. There is a weighted connection between every visible node to every hidden node, these connections determine how much influence each visible node has on a hidden node.

RBMs learn features by using learning rules to iteratively update connection weights between visible and hidden nodes. The visible nodes consist of data from a data vector called the training set. The training set is generally obtained from data gathered from real-life applications. By repeatedly updating the connection weights on the given training set, the RBM can converge to a set of connection weights which represent the learned features.

The following notation will be used to represent nodes and weight connections: v_i and h_j will represent the binary state of visible node i and hidden node j, respectively; w_{ij} will represent the connection weight between visible node i and hidden node j. Figure 1 is an example of an RBM which shows the visible nodes, hidden nodes, and connection.

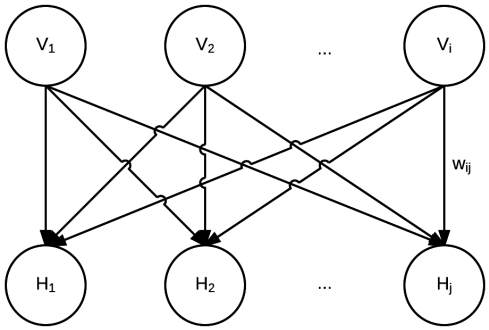

Fig. 1. Example of RBM with weighted connections

A. Sampling

To train the RBM, Gibbs Sampling is used to sample the data and model distributions. There are two phases to gibbs sampling, the generate and reconstruct phases. In the generate phase, the visible nodes are sampled with an input vector from the training set and the hidden nodes are turned on with probability proportional to the activation energy of the visible nodes. During the reconstruct phase, the hidden nodes are sampled and the visible nodes are reconstructed.

While nodes are sampled, their activation energies are calculated. The activation energy of a node can be thought of as a value which affects the probability that a node can become active. The RBM attempts to reduce the amount of global energy in the system while training by tuning the weights to minimize the total energy from the generate and reconstruct phases. The total energy of both hidden and visible nodes is given by the following equation:

$$E(v,h) = -\sum_{i,j} v_i h_j w_{ij} \tag{1}$$

The activation energy is scaled to a probability using the logistic function in Figure(2):

$$\sigma(t) = \frac{1}{1+e^{-t}} \tag{2}$$

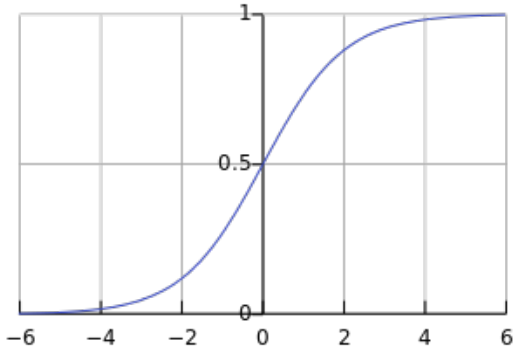

Fig. 2. Logistic function used to scale activation energies

In the two phases of the RBM, we can calculate the total activation energy by calculating the positive and the negative energies for each training vector. The network assigns a probability to every pair of visible and hidden nodes using the following energy function:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)} \tag{3}$$

Where Z is given by:

$$Z = \sum_{v,h} e^{-E(v,h)} \tag{4}$$

B. Weight Update

Weight update of the algorithm is done by a method called contrastive divergence, which is a form of approximate gradient descent. The gradient of the log probability is given by the following equation:

$$\frac{\partial log(p(v)))}{\partial w_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{5}$$

The positive association is given by the adding up the weights of every active visible node while the negative association is given by the reconstruction of the training vector given the weights and the probability that each hidden node is active. The positive and negative associations are calculated for the entire training set and weights are updated once each time the training set is parsed. The weight update is given by the following equation:

$$w_{ij}[k+1] = w_{ij}[k] + \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{6}$$

Where the brackets denote the expectation of the of visible node i and hidden node j given the distribution in the subscript and ϵ is the learning rate of the RBM. A smaller learning rate will update weights at a finer granularity, which may cause it to take more timer to reach a minima. Another hyperparameter of the RBM is the number of epochs or weight update iterations, however, this can be determined by checking the reconstruction error at each epoch.

The full training set is parsed at every epoch, however, this process may take too long in between each weight update. In order to alleviate the amount of time between each weight update, training can also be done in mini batches. Mini batches consist of a small subset of the full training set, generally 10 to 15 samples. Training is done by rotating mini batches until the entire training set is sampled.

Convergence of weights depends upon the initial weights and the learning rate of the RBM. The initial weights can be set to 0, however, many papers suggest that using a small random value between -0.01 and +0.01 will help the RBM converge more quickly. As a result, this architecture makes use of random weight initialization as this would be used in a real-world application.

IV. Restricted Boltzmann Machine Pipelined Architecture

The goal of this RBM architecture is to be scalable and perform much faster than the software implementation. The architecture is scalable in that resources and performance scales linearly with the number of visible nodes in the RBM. The current implementation of the architecture utilizes a single FPGA, however, it is possible to achieve multi-FPGA scalability by calculating partial energy activations in each FPGA and updating weights when all partial energies are calculated. In this section, we will discuss the data representation, memory architecture, random number generation, weight initialization, association calculation, and finally weight updates.

A. Data Representation

The data representation of the RBM is very regular in both the weights and the input data. Since we know that RBMs operating on binary input vectors, the visible nodes are stored in a one bit integers. This allows the RBM to use the number of bits proportional to the training set size and the number of visible nodes per training example. Additionally, weights are stored in 16 bit fixed point values. Fixed point values allow for faster arithmetic operations as well as smaller memory footprint as the size of the RBM grows. Another advantage of 16 bit fixed point values is that it allows for two weight values to be stored in a single BRAM, halving the number of required BRAMs. The fixed point representation uses 5 bits to represent the integer and 11 bits to represent the decimal as seen in Figure 3.

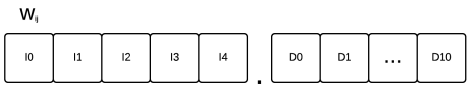

Fig. 3. Fixed Point representation of Weight value

B. Random Number Generation

The RBM is a stochastic algorithm, so it is important to use a random number generator with a high degree of randomness while still being efficient in hardware. In order to get efficient random number generation, a Tausworthe88 Random Number Generator is used because it is capable of generating one random floating point number between 0 and 1 in 10 cycles. The Tausworthe88 Random Number Generator has a high degree of randomness while still being implemented efficiently in hardware [8]. It can be predictable given an initial seed, making it easy to verify correctness of the output with respect to the golden output.

C. Logistic Function

The logistic function used in the algorithm poses a concern as it may take a long time to calculate an exponential as well as a division. While not implemented in this architecture, it is feasible to implement a piecewise approximation [9] of the logistic function for lower latency. Using an approximate algorithm is feasible because as long as the probability is close to the probability generated by the normal logistic function, the weights will converge to values close to the distribution generated by the original logistic function. The current implementation of the logistic function takes 25 cycles using exponentials and division operations. Since we need to utilize three logistic functions per training example, this can easily begin to dominate the run time of the algorithm. Using the piecewise approximation algorithm, the entire fixed point value can be calculated in 11 gate delay [9].

D. Weight Initialization

While weight initialization is an optional step in the RBM algorithm, it provides many benefits to the algorithm. It is possible to reach convergence of weights with all weights initially set at 0, however, weight initialization can be used to achieve faster convergence and better features. Our weight initialization architecture is developed to run in constant time for any sized network. For each weight, a two random number generators are used. The first random number generator calculates a value between 0 and 0.01 while the second random number generator determines the sign of the initial weight. The bias term is always initially set with a weight of 0. This step is done fully in parallel for every connection between visible and hidden nodes in order to reduce the amount of overhead

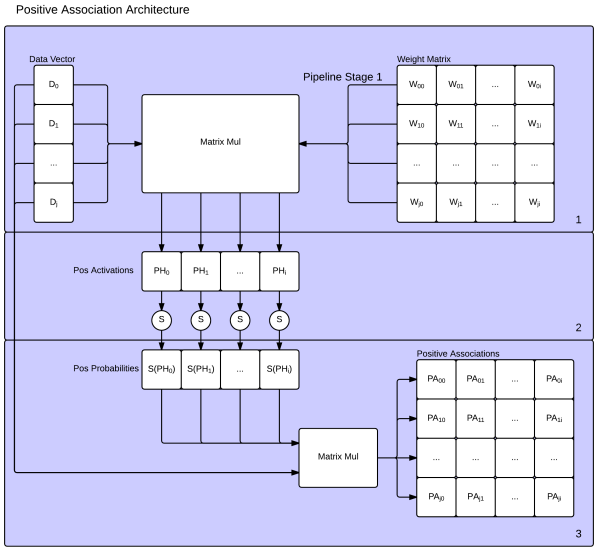

Fig. 4. Positive Association Architecture

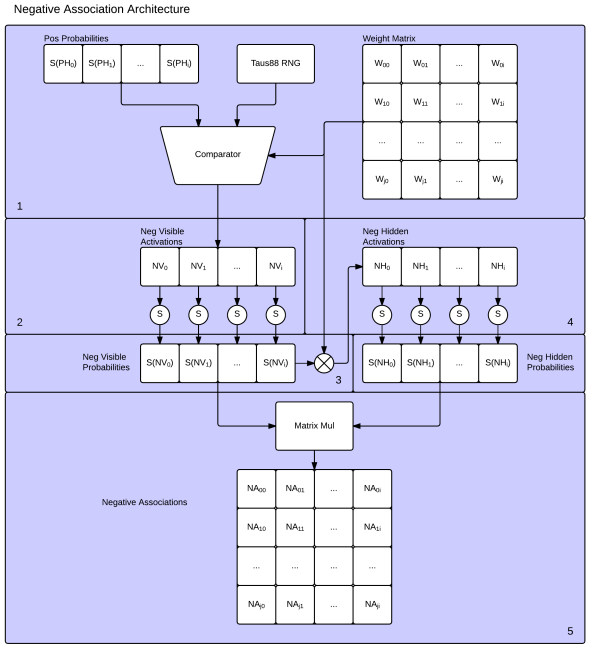

Fig. 5. Negative Association Architecture

time. Another possibility is to do weight initialization as a pre-processing step before using the RBM in order to reduce the amount of overhead on the FPGA.

E. Association Calculation

In order to process every training example, we calculate and accumulate associations before doing weight update. In order to calculate associations, we first gather positive associations (weights applied to data) then negative associations (reconstruction of data using model). The positive associations are calculated by applying the weights to each visible node. The negative associations are determined by turning on a hidden node with probability given by the logistic of the positive activation. Using the probability distribution, it attempts to reconstruct the original input vector using only weighted connections. Calculating both positive and negative associations require five matrix multiplications and three logistic functions per training example. Since the number of training examples may be high, the entire process is pipelined in order to increase throughput. All matrix multiplications are done fully in parallel into temporary registers and two accumulation registers are kept for the positive and negative associations.

Calculating the positive association is a relatively simple as we just calculate the activation energy of the weights applied to the input test vector, then calculate the probability of activating the hidden node by passing the calculated activation energy through the logistic function. Finally the association is equal to the sum of all probabilities multiplied by the input vector for each hidden node. The architecture used for calculating positive associations can be seen in Figure 4. Each matrix multiply stage occurs fully in parallel so that they can generate results in one cycle. The Logistic Function labeled S calculates one probability from a given activation energy fully in parallel with i hidden nodes. The last matrix multiply calculates all positive associations for all connections between visible and hidden nodes in one cycle. Each numbered box represents a pipeline stage in the specific calculation.

The negative association is a reconstruction of the input data given only the weights and positive probabilities. This reconstruction requires obtaining the activation energies for both the visible and hidden nodes then doing a matrix multiplication to obtain the negative association or reconstruction of the data given the model. Figure 5 shows this architecture and each pipeline stage required for the full calculation. As seen in the figure, the visible activations are calculated using the positive probability distribution and the hidden activations are calculated using the weights and the visible activations. The negative association is calculated using both previously calculated activation energies.

F. Weight Update

The weight update is pipelined in order to be more scalable. It takes the accumulation of positive and negative associations at the end of each training batch and update the weights according to Equation 6. The visible nodes are updated in a pipeline while the hidden nodes are updated all in parallel. This reduces the number of resources needed to update the weights down to only the amount needed to update all hidden nodes. The weight update architecture can be seen in Figure 6. Hidden nodes are updated in parallel while visible nodes are updated in a pipeline.

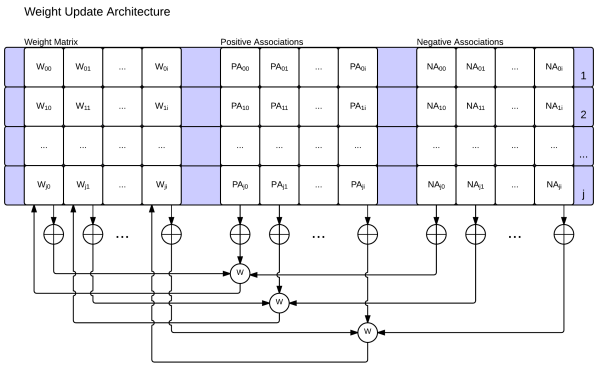

Fig. 6. Weight Update Architecture

G. Memory Architecture

In order to make full use of the inherent parallelism in weight updates and energy calculation, we design a memory architecture that can support all of the calculations occurring in parallel. To support the matrix multiplications occurring in parallel, all temporary values are stored in completely partitioned memory to allow reads and writes to all memory locations simultaneously. To support the weight update architecture pipeline, we partition the memory into Block RAMS with a factor equal to the number of hidden nodes. We do this in order to be able to update all hidden nodes at once then pipeline the loop for visible nodes. The input data is partitioned horizontally into Block RAMs with a factor equal to the number of visible nodes in order to calculate positive activations fully in parallel.

V. Results

To benchmark this FPGA implementation of RBM, we compare our results to a golden model developed in C and check to see the relative error per weight. The training set used in this example is the same example used in the blog post by Edwin Chen [10]. This training set was used due to its simplicity and easiness to evaluate. Additionally, we perform further experiments in order to evaluate the scalability of the system and the performance as the size of the RBM increases. To evaluate the performance of the RBM, we benchmark the amount of time required for one epoch of the RBM.

A. Training Results

The architecture used to test the RBM used 6 training examples, 6 hidden nodes, and 2 hidden nodes. The learning rate was set to 0.1 and the network was updated over 5000 epochs. Using the pipelined architecture, we were able to achieve an average error of 0.12 per weighted connection compared to the golden result. The small errors can be attributed to the fixed point notation used in this architecture because it does not represent values as accurately as floating point. It is possible to reduce the amount of error by increasing the number of bits used to represent the decimal if it is necessary for the application. This would retain the scalability of the architecture, however, the number of BRAMs may double if more than 18 bits are used to represent a single fixed point value. To measure performance of each epoch in the network, we compare the latency of the pipelined architecture to the pure software approach and find that we achieve over 34x reduction in latency for the RBM trained over 5000 epochs. The amount of resources used in this architecture is just over 2x over the pure software implementation. A summary of the results of the software approach and pipelined RBM is presented in Table I.

TABLE I. Architectural Results

Architecture	Total Latency (max)	Slice	BRAM	DSP48	FF	LUT
Software Architecture	48960374	3085	0	73	6185	7939
Pipelined RBM Architecture	1430302	7300	6	203	10760	20370

B. Scalability

We randomly generate a training set of 5 examples to find out how increasing the number of visible and hidden nodes scales with the architecture. One epoch of the RBM is synthesized to see how the area and latency are affected by size of the RBM. The results show that the amount of resources that the RBM utilizes scales in O(n) even though the size of RBM and number of weights scales in O(n^2). This is important because we want the FPGA to support larger sized networks while retaining the relative performance enhancement obtained in Table I. Figure 7 shows the scaling of resources with respect to the size of the RBM on a Virtex 7 series FPGA.

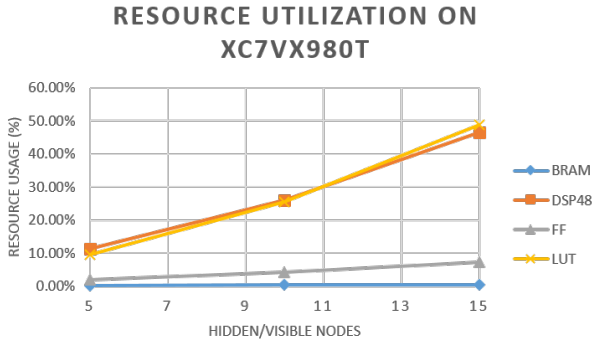

Fig. 7. Resource Utilization with respect to size of RBM

C. Performance

In addition to resource scaling, we also measure latency scaling to see if larger networks scale linearly in pipelined architecture.

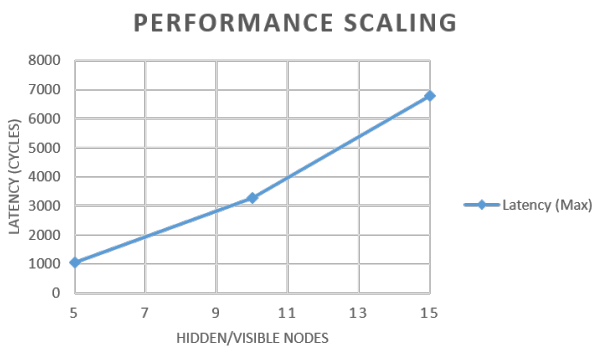

Fig. 8. Performance Scaling with respect to size of RBM

We find that the latency scales linearly even though the size of the network increases exponentially. This results tells us that the performance increase is much larger if the size of the network is larger. Since all hidden nodes are updated simultaneously, the algorithm scales linearly with the number of visible nodes in the network. In a symmetrical network where visible and hidden nodes are equal, the architecture will scale in O(n) in terms of performance. While our small test case only obtains a performance increase of 34x, a larger network of size 512x512 may achieve a performance increase of over 200x. Figure 8 shows the results of performance with respect to the size of the network.

VI. Conclusion

This paper presents a pipelined RBM architecture which achieves O(n) scaling in both resource usage and performance while the size of the network grows in O(n^2). It achieves a performance increase of over 34x over the pure software approach on a small training set and takes advantage of the inherent parallelism in the matrix multiply operations used in the RBM algorithm. As the size of the RBM grows exponentially, the latency of the architecture grows linearly. Very large networks will see performance gains of more than 34x using this architecture. The pipelined architecture allows for any number of visible and hidden nodes as well as training set size as long as it fits on the FPGA. Some hyper parameters which can be tuned are the learning rate and number of epochs of the network, allowing control over how fine grain the weight updates are and time to convergence respectively.

Some directions for future work include switching to a batch-based learning method in order to support larger training set sizes, optimization of the logistic function, and multi-FPGA support using partial activation energy calculation. By expanding the RBM to support larger networks, it will be possible to support more features on the input data as well as more learned features. Due to their popularity in Deep Belief Networks, another future work could be to implement a full Deep Belief Network architecture which can make predictions on labeled data. As neural networks become more prominent in machine learning, FPGA-based hardware architectures may provide the scalability and performance needed for large problems.

References

[1] S. Knerr, L. Personnaz, and G. Dreyfus, "Handwritten digit recognition by neural networks with single-layer training," *Neural Networks, IEEE Transactions on*, vol. 3, no. 6, pp. 962–968, 1992.

[2] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 6645–6649.

[3] H. Larochelle, M. Mandel, R. Pascanu, and Y. Bengio, "Learning algorithms for the classification restricted boltzmann machine," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 643–669, 2012.

[4] G. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[5] D. L. Ly and P. Chow, "A high-performance fpga architecture for restricted boltzmann machines," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2009, pp. 73–82.

[6] C. Lo and P. Chow, "Building a multi-fpga virtualized restricted boltzmann machine architecture using embedded mpi," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 189–198.

[7] L.-W. Kim, S. Asaad, and R. Linsker, "A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 1, p. 5, 2014.

[8] R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, vol. 19, no. 90, pp. 201–209, 1965.

[9] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," in *Circuits, Devices and Systems, IEE Proceedings-*, vol. 144, no. 6. IET, 1997, pp. 313–317.

[10] E. Chen. (2011) Introduction to restricted boltzmann machines. [Online]. Available: http://blog.echen.me/2011/07/18/introduction-to-restricted-boltzmann-machines/