Resource Efficient Arithmetic Effects on RBM Neural Network Solution Quality Using MNIST

Conference Paper · November 2011

DOI: 10.1109/ReConFig.2011.79 · Source: DBLP

2 authors, including:

Medhat Moussa
University of Guelph
67 PUBLICATIONS **597** CITATIONS

SEE PROFILE

2011 International Conference on Reconfigurable Computing and FPGAs

Resource Efficient Arithmetic Effects on RBM Neural Network Solution Quality Using MNIST

Antony W. Savich and Medhat Moussa

School of Engineering
University of Guelph
Guelph ON Canada N1G 2W1
{asavich,mmoussa}@uoguelph.ca

***Abstract*—This paper presents a case study on the impact of using reduced precision arithmetic on learning in Restricted Boltzmann Machine (RBM) deep belief networks. FPGAs provide a hardware accelerator framework to speed up many algorithms, including the learning and recognition tasks of ever growing neural network topologies and problem complexities. Current FPGAs include DSP blocks - hard blocks that allow designers to roll in hardware otherwise built using significant quantity of reconfigurable logic (slices) and increase clock performance of arithmetic operations. Accelerators on FPGAs can take advantage of, in some products, thousands DSP blocks on a single chip to scale up the parallelism of designs. Conversely, IEEE floating point representation cannot be fully implemented in single DSP slices and requires a significant amount of general logic thus reducing the amount of resources available to breadth of parallelism in an accelerator design. Reduced precision fixed point format arithmetic can fit within a single DSP slice without external logic. It has been used successfully for training MLP-BP neural networks on small problems. The merit of reduced precision computation in RBM networks for sizable problems has not been evaluated. In this work, a three layer RBM network linked to one classification layer (1.6M weights) is used to learn the classic MNIST dataset over a set of common limited precisions used in FPGA designs. Issues of parameter saturation and a method to overcome inherent training difficulties is discussed. The results demonstrate that RBM can be trained successfully using resource-efficient fixed point formats commonly found in current FPGA devices.**

***Index Terms*—FPGA, DSP block, Neural network, RBM, Arithmetic representation**

I. Introduction

Artificial Neural Networks (ANN) are popular for solving complex, irregular problem spaces. The most popular neural network algorithm has historically been the Multi-Layer Perceptron with Back Propagation learning (MLP-BP). In recent years, a new promising algorithm has gained attention. Restricted Boltzmann Machine (RBM) [1] has shown significant potential on a number of non-trivial machine learning problems, such as MNIST [2] and NORB [3]. But one of the main problems in using RBM is the extremely long training time that can react hours and even days on current computers. One approach to speed up the training of RBM is to design a hardware accelerator on a suitable chip. This technique is not new, since ANNs are inherently parallel architectures. There many examples of so called neurocomputers, hypercube and connection machines, and other supercomputers [4], [5], [6]. Other groups have designed and built parallel systems based on transputers [7], or digital signal processors (DSPs) [8]. Several companies have proposed Application Specific Integrated Circuits (ASICs) that act like ANN accelerators. The reader can refer to Nordstrom [9] or Dias [10] for an overview of historic and current efforts of such. Most of these designs require using special hardware boards or ASIC chips which limit their use on a large scale. The implementations are constrained by size and type of algorithm.

More recently, two platforms emerged as the most promising. GPUs and FPGAs can both be used to implement ANNs with potentially significant speed up (at least 10× and often more vs. a multicore general purpose CPU). GPUs tend to be used as a co-processor for applications that already require host PCs, while FPGAs are extensively used in embedded systems applications, where they are applied for a wide variety of tasks such as control and filtering. This paper will focus on the investigation of RBM implementation on FPGAs. Using FPGAs for ANN implementation has gained significant interest in the last 10 years with the advances in FPGA technology [11], [12], [13], [14], [15], [16], [17]. Commercial development of large, dense and configurable FPGA chips has allowed implementations with more flexibility of network size, type, topology, and other constraints while maintaining increased processing density using the natural parallel structure of ANNs.

However, using FPGAs to implement ANNs raises a number of design issues [18]. One of the main issues is the limited or rather not very area efficient support FPGAs have for floating point arithmetic. It if far too expensive, resources wise, to use IEEE double precision floating point arithmetic (FLP) on FPGAs. IEEE single precision arithmetic is manageable, but takes multiple DSP slices and much general logic to implement.

Every implementation is a trade-off between performance and resource utilization. Higher performance requires high degree of parallelism, which in turn requires more processing elements to be fit within an FPGA. Thus it is critical to use the most resource efficient arithmetic representation format while maintaining the importance of *adequate precision to achieve learning*. Fixed point arithmetic looks attractive for

978-0-7695-4551-6/11 $26.00 © 2011 IEEE
DOI 10.1109/ReConFig.2011.79

this purpose because it can fit entirely within a DSP block of an FPGA with no external logic use, thus allowing top clock rates and more parallelism to be used in any design.

Recent developments in RBM implementation on FPGA include [19], where binary arithmetic is used for high speed yet hindered quality of calculations, and [20], where 16 bit fixed point arithmetic is chosen, yet again running short of demonstrating the quality of the trained network, or presenting a method for selecting particular representations to suit a particular problem. Binary computations may not in fact generate an acceptable training outcome for complex problems, and current FPGAs accommodate efficient resources for arithmetic and storage wider than 16 bits thus potentially enhancing training accuracy.

The purpose of this paper is to investigate the impact of using various fixed point arithmetic formats typical in efficient FPGA implementation on learning using RBM neural networks. The goal is to find efficient formats and methods that can be used in RBM accelerators on FPGAs to aid both in real time learning for deep belief networks and generate results near the accuracy of equivalent floating point capable processors, such as CPUs and GPUs. Implementations using these formats can be implemented on wide range of FPGAs, from massive scale arrays to embedded low-power platforms.

II. Background

A. RBM

As a general overview, RBM is a Restricted subset of the more general Boltzmann Machine. It is very hard to find a training algorithm for a general Boltzmann Machine, where intra-layer connections are present. For this reason a connectivity reduction is made to enable a type of learning used similarly to how back propagation (BP) is used in MLP - Alternating Gibbs Sampling [21]. A representative diagram of RBM architecture is shown in Fig. 1.

In RBM, there are neuron nodes in two layers, the hidden and visible. They are fully and bidirectionally connected across layers with weighted synapses. The states of the hidden layer neurons can be either binary stochastic or continuous. The visible neuron states are continuous. The inputs during learning are applied in place of the values normally produced by the neurons in the visible layer. The values, according to the stochastic nature of the network, then represent the probability of a particular visible neuron being active within a sample of the input distribution. The weights are applied to each synapse and the combination of all weighed synapses for each hidden neuron is computed. The output of the hidden units, after being passed through a transfer function, then represents the probability of the reaction of a particular neuron to the input data distribution - a type of feature detector. The transfer function typically is sigmoid, often used in other neural nets for its "squashing" properties. This process repeats in the reverse direction, where the network regenerates the visible layer now based on the stimulus from the hidden neurons, producing a "recollection" of the stimulus through the same set of weights.

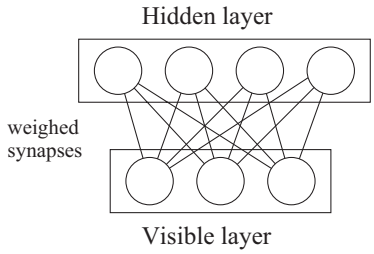

Fig. 1. Restricted Boltzmann Machine, one layer layout

The required computations are illustrated, where Equation (1) represents the operations performed to generate a hidden representation and (2) to recollect the visible layer:

$$p(h_k) = f\left(\sum_{j=1}^{N_{vis}} w_{kj} v_j + w_k\right) \tag{1}$$

where, for network hidden neurons h_k and visible neurons v_j:
$p(h_k)-$ stochastic probability of hidden neuron k activation
v_j- value of visible neuron j
$f-$ *transfer or squashing function*
$N_{vis}-$ number of visible neurons
$w_{kj}-$ *synaptic weight* associating hidden neuron k with visible neuron j
w_k- hidden neuron k's *bias weight*.

$$p(v_j) = f\left(\sum_{k=1}^{N_{hid}} w_{jk} h_k + w_j\right) \tag{2}$$

where, for network visible neurons v_j and hidden neurons h_k:
$p(v_j)-$ stochastic probability of reconstructed (or generated) value of visible neuron j
h_k- value of hidden neuron k
$f-$ *transfer or squashing function*
$N_{hid}-$ number of hidden neurons
$w_{jk}-$ *synaptic weight* associating visible neuron j with hidden neuron k
w_j- visible neuron j's *bias weight*.

In these operations, the synaptic weights are bidirectional. That is, w_{kj} is equivalent and equal to w_{jk}. A typically used transfer function is the log-sigmoid, Equation (3).

$$f(x) = \frac{1}{1+e^{-x}} \tag{3}$$

1) Alternating Gibbs Sampling (AGS): The first stage of the learning process is the *generate* stage, where the values of the visible layer are fixed and the hidden layer outputs are computed. The second stage of the process is the *reconstruction* phase, where hidden layer output is fixed and values on the visible layer are reproduced. In Alternating Gibbs Sampling, this process can be repeated several times. Each successive

iteration takes the state of the network away from reality (what the original input distribution looks like) to fantasy (what the network thinks the input should look like) based on the original stimulus and the weight state of the network. The goal is to minimize the difference between reality and the network's fantasy, thus making the fantasy a recollection of the real data being learned. This is accomplished by first considering a global energy level:

$$E = -\sum_{\forall i,j} w_{i,j} v_i h_j \tag{4}$$

where :
$v_i =$ output value of visible neuron i
$h_j =$ output value of hidden neuron j
$w_{i,j} =$ weight of the associated synapse between neurons v_i and h_j.

This is performed across all neurons in the network. Equation (4) effectively calculates the global state of the network, which is an inverse correlation between active hidden and visible neurons. Because in both stages of learning, the values of one layer are fixed, and neurons are not interconnected within each layer, a partial energy product can be derived for each of forward and backward processes, with the resulting probability densities of neuron states:

$$\begin{aligned} p(h_i) &= \frac{1}{1+e^{-\frac{1}{T}\sum_j w_{i,j} v_j}} \\ p(v_i) &= \frac{1}{1+e^{-\frac{1}{T}\sum_j w_{i,j} h_j}} \end{aligned} \tag{5}$$

where h_i and v_i are the hidden and visible neurons whose states are calculated. The parameter T has a similar effect to variable learning rate, and can be changed at start and at end to control the learning process. The distributions are then sampled to produce real values used in next stage calculations.

To adjust the weights for the network to learn a representation of the input data, a gradient descent approach on the global energy state is used. To do this, the energy gradient is found:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{1}{T}\left(\langle v_i h_j \rangle^1 - \langle v_i h_j \rangle^\infty\right) \tag{6}$$

where $\langle v_i h_j \rangle^1$ represents the value of correlation between the visible and hidden data at the initial stage of learning (first generation), and $\langle v_i h_j \rangle^\infty$ represents the same value at the final stage of learning (after an infinite number of iterations). When an infinite number of iterations are used over the entire data set, a local minimum occurs, which represents an ideal condition over the input set given the current energy state. A simplified process can be used with a limited number of iterations. The result sufficiently estimates the desired ideal value for a given step without reducing the rate of convergence. Thus, the energy gradient can be calculated by evaluating $\langle v_i h_j \rangle^3$. Epoch learning with limited size epochs can be used here as well. Only a number of patterns can be used to contribute to the calculation of the final estimate for $\langle v_i h_j \rangle^\infty$, and this calculation repeated a number of times to traverse a dataset. The weight update rule stemming from this is:

$$w_{i,j}[k+1] = w_{i,j}[k] + \frac{1}{T}(\langle v_i h_j \rangle^1 - \langle v_i h_j \rangle^A) \tag{7}$$

where
$A =$ number of forward and backward stages used $k =$ step number in the learning process

2) Multilayer RBM: In order to improve the generative power of the RBM architecture, it can be layered. In [1], [22], a multilayer RBM architecture is explored. Because of RBM's generative properties, a layer of hidden neurons interprets the input data into a number of underlying concepts or features. To add more dimensions, or depth, to the interpretation, layers of RBM neurons are stacked effectively creating features of features. As the first layer is trained to represent the input data by creating an interpretation of this data using simple features in the hidden layer, those lower level features become the inputs to higher layers. These layers learn a deeper degree of interpretation of the input domain by discovering correlations in the simpler concepts learned by previous layers. Deep networks can be trained as generative models for involved problems, where a single layer of interpretation cannot grasp the full complexity of the input space.

B. FPGA Hardware

Current Field Programmable Gate Arrays (FPGA) are more than just a collection of lookup tables (LUTs) and latches today. They host an array of specialized blocks to allow for improved clock performance and substantial resource availability for a multitude of implementations. These include hard memory blocks, clock generators, high speed transceivers, and in some cases multiple dedicated processor cores with own caches and a multitude of memory and peripheral controllers.

It is the availability of precision and range, based on the available resources, that determines whether a neural net solution will adequately function in hardware.

1) Fixed Point: In our case, it is the arithmetic modules that are of most interest. Typical adder configurations can be 48 bits wide without affecting overall clock rates or consuming significant resources. This suggests that there is little precision limitation in performing additions or subtractions, as resource consumption grows linearly with precision used. However, the integrated multiplier circuits[1] can be rather expensive in terms of area and power consumed due to a second order growth with accommodated bit widths. They typically appear in either 18×18 or 18×25 bit configurations and perform 2's complement arithmetic in fully pipelined mode. While the hardware itself does not keep track of the decimal point used in fixed point representations, the onus is on the designer to select, among a full 36 bit output for an 18×18 bit multiplier for example, the appropriate significant digits to keep. In the same case with 18 bit inputs, the bits representing the integer and fractional components must add up to that total width of 18. A typical representation fitting this constraint is 1-5-12, where one sign bit is used together with 5 integer bits, producing a range of

[1]Each vendor has their own name for these but they all perform the same basic functions.

± 32, and 12 fractional bits, producing a minimum $2^{-12} = 2.44 \times 10^{-4}$ precision or quantization step size.

Wider fixed point arithmetic is of course achievable in FPGA fabric, but at the cost of general configurable resources (LUTs), and hence reduced performance and increased power consumption vs. hard arithmetic blocks.

2) Floating Point: Floating point arithmetic provides more range in numeric representation over equivalent fixed point at the cost of precision. In 1-8-23 floating point representation (one used in IEEE single precision floating point system), 1 bit is used for the sign (s), 8 bits are used for the exponential component(e), and 23 significant fraction (f) bits are used for mantissa in a $1.f$ representation, where the mantissa bits represent the digits trailing a decimal point. The numeric values are represented, thus, as $(-1)^s(1.f)(2^e)$, with a least significant representable number 2^{-149} and the largest representable number $(1 - 2^{-23}) \times 2^{126}$.

Floating point arithmetic, however, cannot fully fit into available hard arithmetic blocks on FPGAs, even using reduced bit widths compatible to equivalent fixed point variants that *can* fit. For all operations, dedicated hardware is required to handle special numbers, overflow conditions and error correction. For multiplication, an extra adder is required in parallel to process the exponents. For addition, a dynamic shifter is required, which is typically implemented using a multiplier block, to align the mantissas of the two numbers depending on their exponents.

The question in this paper is whether fixed point, in a format that can be efficiently mapped to FPGA resources providing the highest number of functional units, is capable of providing sufficient representation to achieve good learning quality of RBM on a non-trivial problem. If it can, which will be demonstrated, higher performance can be achieved because of the increased number of functional units available to be exploited in parallel computations vs. hardware based on floating point that produces less functional units on the same chip.

III. Experimental setup

As the non-trivial problem, MNIST [2] is selected because its relative complexity and its popularity in previous studies where performance and quality data are available for a multitude of implementation platforms and methods. MNIST dataset [2] contains handwritten images of digits 0-9, and is used extensively as a benchmark in testing the performance of large scale neural networks and other algorithms, for both speed of computation and the quality of results. It contains 60,000 training and 10,000 testing samples of 28×28 grayscale pixel images of the digits. The current record for classification quality is 0.31%, which equals 31 misclassified digits out of 10,000 test images [23]. As special note to the experimental setup, the performance above, and similar results listed in [2], derive their accuracy from either pre-processing the original MNIST dataset, or using in-line distortion operations to produce a huge number of correlated input samples on the training set (from 60,000 to over 2 million). Specifically, in-line distortion operations help produce many more examples of digits that *could* exist based on the original test samples and typical noise sources involved in images of hand writing (muscular jitter, rotations, translations, various compressions, etc.). To generate the results, significant compute resources are allocated to generating the transformed test image variations vs. resourced required for the learning algorithm itself. Algorithm and parameter tuning, pretraining and other techniques are involved in generating the best results. The best result reported on the unmodified dataset with floating point precision and algorithm tuning is about 1%, or 100 misclassified digits. The focus of this work is the algorithm adaptation to implementation itself, while using the original unmodified MNIST dataset and varying arithmetic representation without algorithm tuning.

This work compares the results obtained full precision floating point arithmetic with ones achievable using limited precision arithmetic for efficient hardware implementation. The RBM topology is set to 784-500-500-2000 (input-hidden-hidden-output). The three layer network is first trained in an unsupervised way, one layer at a time, using the limited representations presented in this work, and is then coupled with a back end layer of 10 neurons using the 2000 wide output of the last layer for supervised classification. The training data set is broken up into 600 batches of 100 samples each. At each epoch, 600 weight updates are in fact performed, one for each batch of 100 digit images. In training, the network performs 3 stages of AGS, and uses $\langle v_i h_j \rangle^3$ for the weight update.

The experiments are done in Matlab 2010a, on a quad core 2Ghz Xeon machine. Each arithmetic operation is converted into a bit-representative FPGA form. The mean squared error (MSE) itself is computed without arithmetic limitations, while using as input the limited output of the network, to give an accurate view of converge. To produce a repeatable and correlating result, a number of random weight sets are generated and saved. This weight set is used in each of experiments, so that the random starting point of the initialized network does not significantly influence convergence results presented. An average of 5 runs across all representations is produced using the same sets of weights. It is evident that the starting position of the network does not indeed significantly influence the results.

IV. Training, bits and saturation

The first stage of our experiments discovers which fixed point representations are adequate and which are not by looking at the convergence of an individual RBM layer with a variety of available representations that may fit into a common 18 bit format available in a variety of FPGA products. Not all sample points in these experiments are optimal, but they are selected to demonstrate the effects of MSE convergence based on variation of either the integer or the fractional component of the representation; several points are chosen to compare fixed point with floating point performance as a benchmark.

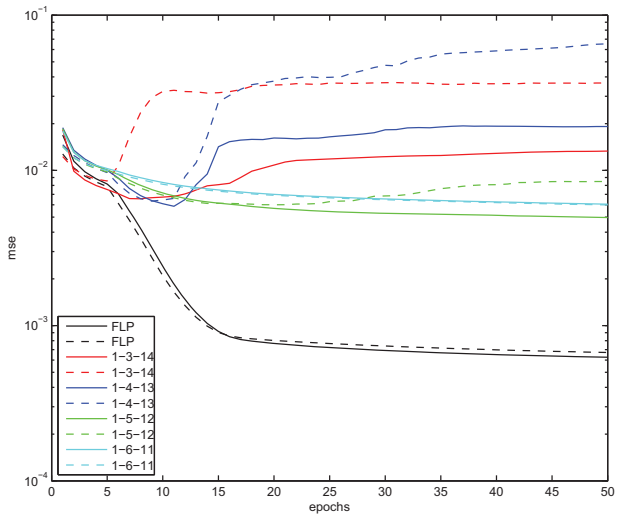

Fig. 2. Training performance using 18 bit arithmetic

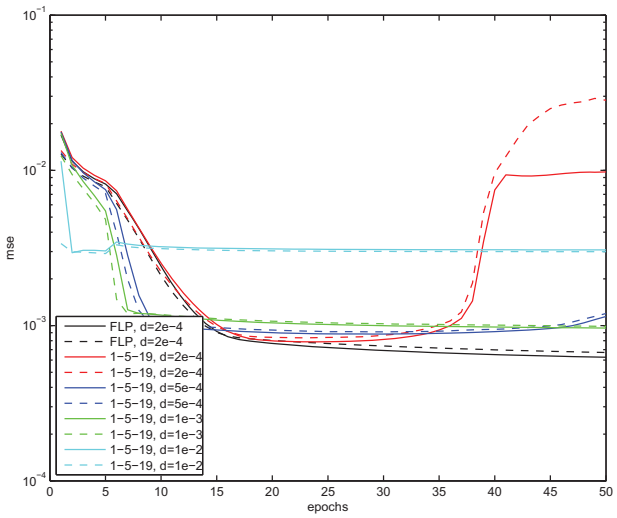

Fig. 3. Training performance with varying weight decay

Fig. 2 demonstrates that limited 18 bit fixed point representation clearly produces more output noise than its floating point counterpart. Here, 18 bits are chosen as the total numeric width, and the balance between range and precision are varied. Solid lines represent training set error, and dashed lines are test set error.

It is evident from Fig. 2 that smaller range (1-3-14) starts off with excellent train and test set results, but the representation of weights quickly saturates and results for both sets diverge - error worsens very quickly. As the arithmetic range increases, divergence in the test set is still evident at 1-5-12, while the training set produces the best performance for all limited representations. At 1-6-11, the numeric range is sufficient to keep the error rate consistent without divergence, but the 11 bit precision undercuts finals performance.

Newer FPGA chips are capable of efficiently performing calculations with 25 bit fixed point arithmetic. Using Fig. 2, it is clear that precision plays a significant role in final convergence of each layer of RBM. There, 6 bits of range were demonstrated sufficient for 50 epochs with a learning rate of 0.1, and weight decay of 2×10^{-4}. By adjusting the weight decay to contain range overflow from occurring, 5 bits of range can be effectively used in implementation. Fig. 3 shows results for various weight decay values, 5 bits of range, with the remaining 19 bits allocated to fractional precision.

The data demonstrates that large weight decay will produce faster learning results, but will also eventually hinder final convergence. Weight decay at the start prevents the significant weights from saturation by weight space scaling. When errors grow small, so do the weight update increments. At one point, the error increments become smaller than the proportionate continuous weight decay. In limited precision training, it is useful to provide a time-varying weight decay constant similar to time-decaying learning rates in full precision training. Note a difference, however, when the 1-5-19 representation, with a low 2×10^{-4} decay value, saturates starting at epoch 38. The weight decay constant can start high to expedite learning, decrease through the run to improve the accuracy of training during smaller error rates, and increase again when saturation is evident.

V. Testing and Classification

The true measure of performance in a classification task is the resulting classification rate. Here, we validate MSE results obtained so far with the overall usefulness goal - final classification rates under arithmetic limitations.

Table I relates the poor convergence results in Fig. 2 to classification rates as a percentage of patterns recognized. The numbers produced using limited representation are substantially worse than the floating point benchmark. Interesting to note that 1-3-14 produces better results than other limited representations. This is due to expanded precision (14 bits), and the correlation of the weight decay and the magnitude of intermediate parameters in the learning process. The MSE results in Fig. 2 show that 1-3-14 has a lower final classification rate than 1-4-12 on both the training and test set. Even still, 3.43% in the context of this problem is rather poor performance.

TABLE I
18 bit arithmetic, varying precision

Representation	Train set error	Test set error
1-3-14	0.98%	3.43%
1-4-13	6.26%	7.86%
1-5-12	4.95%	5.26%
1-6-11	5.24%	5.51%
float	0.02%	1.58%

Moving to a more robust representation, Table II demonstrates results for using 25 significant bits and a single 1-5-19

representation. Five integer bits proves best in experiments at this bit width, and the remaining bits are used for precision. This representation shows excellent classification results for the unmodified MNIST data, except with weight decay of 2×10^{-4}. As seen in Fig. 3, this particular configuration results in early saturation. This is concurred with a poor 2.9% classification rate.

While the results for 1×10^{-2} weight decay show the network stuck in a parameter minimum where MSE performance is poor, the final convergence results using this configuration are different. The training set classification rate is the worst among presented. The test set rate, however, is marginally better than even the benchmark floating point configuration. A more aggressive weight decay rate keeps weights from growing significantly, thus limiting the weight space away from solutions giving best reconstruction error performance. At the same time, with good precision (19 bits here), it is able to enhance the position of more significant weights by aggressively decaying the rest. A high decay rate stimulates the reduction in erroneous positives comparatively more than it does to improve the results that generate the correct outputs. The net result is improved classification on the test set.

TABLE II
25 BIT ARITHMETIC, VARYING WEIGHT DECAY

Representation	weight decay	Train set error	Test set error
1-5-19	0.0002	0.18%	2.9%
1-5-19	0.0005	0.07%	1.53%
1-5-19	0.001	0.03%	1.58%
1-5-19	0.01	0.19%	1.49%
float	0.0002	0.02%	1.58%

VI. CONCLUSION

Using limited arithmetic representation appropriately is not trivial when implementing RBM on FPGAs. Results have demonstrated that there are several design parameters on the algorithm level that need to be carefully handled. Limited numeric range, while providing extra noise that often helps generalization, typically results in parameter saturation early in training. Limited precision produces poor empirical convergence when looking at differences between presented and reconstructed stimuli. When considering classification results, there is a visible but indirect correspondence between MSE performance during training and classification ability of the resulting network. In fact representations that showed relatively poor MSE performance in training, demonstrated superior classification performance even over the floating point benchmark.

Nevertheless, by using an arithmetic representation that can maximize the parallelism of computation given available resources, FPGA implementations of RBM can harness good solution quality while providing enhanced performance for a particular chip family. While being in strong competition with GPUs in the large scale computing paradigm, by incorporating methods for dealing with early saturation, such as appropriately adjusting weight decay, RBM networks can be effectively implemented using limited precision arithmetic on FPGAs in low power mobile or personal use environments.

REFERENCES

[1] R. Salakhutdinov and G. E. Hinton, "Deep boltzmann machines." *Artificial Intelligence and Statistics AISTATS09*, vol. 5, no. 2, pp. 448–455, 2009.

[2] Y. LeCun and C. Cortes, "The MNIST database of handwritten digits," 2011. [Online]. Available: yann.lecun.com/exdb/mnist/

[3] Y. LeCun, F. Huang, and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.

[4] E. Kerckhoffs et al., "Speeding up BP Training on a Hypercube Computer," *Neurocomputing*, vol. 4, no. 1-2, pp. 43–63, 1992.

[5] X. Liu et al., "Benchmarking of the CM 5 and the Cray Machines with a Very Large BP Neural Network," in *ICNN*, vol. 1, Piscataway, NJ, 1994, pp. 22–27.

[6] M. B. Q. Malluhi and T. Rao, "Tree-Based Special Purpose Array Architectures for Neural Computing," *Journal of VLSI Signal Processing*, vol. 11, pp. 245–262, 1995.

[7] A. Petrowski, G. Dreyfus, and C. Girault, "Performance analysis of a pipelined backpropagation parallel algorithm," *Neural Networks, IEEE Transactions on*, vol. 4, no. 6, pp. 970 – 981, Nov 1993.

[8] U. A. Muller et al., "Fast neural net simulation with a dsp processor array," *IEEE Transactions on Neural Networks*, vol. 6, no. 1, pp. 203–213, January 1995.

[9] T. Nordstrom and B. Svensson, "Using and designing massively parallel computers for artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 260–285, March 1992.

[10] F. Dias et al., "Artificial Neural Networks: A Review of Commercial Hardware," *Engineering Applications of Artificial Intelligence*, vol. 17, pp. 945–952, 2004.

[11] A. R. Omondi and J. C. Rajapakse, *FPGA Implementations of Neural Networks*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[12] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: theory, algorithm, and FPGA implementation," *Neural Networks, IEEE Transactions on*, vol. 14, no. 5, pp. 993–1009, 2003.

[13] H. Ng and K. Lam, "Analog and digital FPGA implementation of BRIN for optimization problems," *Neural Networks, IEEE Transactions on*, vol. 14, no. 5, pp. 1413–1425, Sept 2003.

[14] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its FPGA implementation," *Neural Networks, IEEE Transactions on*, vol. 16, no. 6, pp. 1664–1672, 2005.

[15] J. B. R. Gadea, R. Palero and A. Cortes, "Fpga implementation of a pipelined on-line backpropagation," *Journal of VLSI Signal Processing*, vol. 40, pp. 189–213, Sept 2005.

[16] R. J. Aliaga, R. Gadea, R. J. Colom, J. Cerdá, N. Ferrando, and V. Herrero, "A mixed hardware-software approach to flexible artificial neural network training on fpga," in *SAMOS'09: Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1–8.

[17] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *PACT*, 2010, pp. 273–283.

[18] J. Zhu and P. Sutton, "FPGA Implementations of Neural Networks - A Survey of a Decade of Progress," in *Conference on Field Programmable Logic*, Lisbon, Portugal, 2003, pp. 1062–1066.

[19] D. L. Ly and P. Chow, "A high-performance, reconfigurable hardware architecture for restricted boltzmann machines," *IEEE Transactions on Neural Networks*, vol. 21, no. 11, pp. 1780–1792, Nov 2010.

[20] S. K. Kim, L. C. McAfee, P. L. McMahon, and K. Olukotum, "A highly scalable restricted boltzmann machine FPGA implementation," in *FPL*, 2009.

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

[22] G. E. Hinton, "Learning multiple layers of representation," *Trends in Cognitive Sciences*, vol. 11, pp. 428–434, 2007.

[23] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Handwritten digit recognition with a committee of deep neural nets on gpus," *arxiv.org*, Mar 2011.