Towards a Low Power Hardware Accelerator for Deep Neural Networks

Biplab Deka
Department of Electrical and Computer Engineering
*U*niversity of Illinois at Urbana Champaign, USA
deka2@illinois.edu

***Abstract*—In this project, we take a first step towards building a low power hardware accelerator for deep learning. We focus on RBM based pretraing of deep neural networks and show that there is significant robustness to random errors in the pre-training, training and testing phase of using such neural networks. We propose to leverage such robustness to build accelerators using low power but possibly unrelaible hardware substrate.**

I. Introduction

Deep Neural Networks have recently been shown to provide good performance on several AI tasks. Krizhevsky et al present a Convolutional Neural Network with five convolutional layers to classify the images in the ImageNet database into ten classes [1]. Mohammed et al present a Deep Belief Network (DBN) for for phone recognition that outperforms all other techniques on the TIMIT corpus [2].

All these applications of deep neural networks have been made possible by recent advances in training such networks. Classical methods that are very effective on shallow architectures generally do not exhibit good performance on deep architectures. For example, gradient descent based training of deep networks frequently gets stuck in local minima or plateaus [3].

Recent methods solve this issue by introducing a layer-wise unsupervised pretraining stage for deep architectures. During pretraining, each layer is treated separately and trained in a greedy manner. After pretraining, a supervised training stage is used to fine tune the weights assigned by pretraining. Several deep neural network models have been proposed that enable such unsupervised pretraining of the network. These include Deep Belief Networks (DBNs) [4], Stacked Auto-Encoders [5] and Convolutional Neural Networks [6]. A survey of these models and associated pretraining methods can be found in [7].

The long term objective of our work is to develop low power hardware accelerators for deep learning. Such accelerators could enable higher performance and better energy efficiency for AI tasks than is possible using platforms available today. To design very low power accelerators, we plan to use low power hardware devices that might be inherently unreliable. Such an approach has been shown to have significant power benefits when designing ASICs for several signal processing applications [8, 9].

For such implementations to be successful we plan on exploiting the error tolerance already present in the pre-training, training and testing algorithms for deep neural networks. In this project, we evaluate the robustness to errors of Restricted Boltzmann Machine (RBM) based pretraining of Deep Belief Networks. We perform evaluations for handwritten digit recognition on the MNIST dataset. Our results show that classification using Deep Belief Networks can be tolerant to random errors and has the potential for being able to produce acceptable outputs when implemented with low power (but unreliable) hardware substrates.

We also believe that the testing stage for AI applications might be implemented in mobile front-ends of systems and as such would need to be very energy efficient and might be implemented using a dedicated fixed point accelerator. As such, we evaluate the precision requirements of a fixed point implementation of the testing stage.

This work was done as part of the course ECE544NA: Pattern Recognition at University of Illinois at Urbana Champaign in Fall 2013.

II. Related Work

Previous work have shown promising speedups when deep learning is implemented on GPUs. Raina et al used RBMs and showed about 10x speedups over CPU implementations [10]. Farabet et al proposed an FPGA based accelerator architecture for convolutional neural networks for vision problems [11]. Their architecture is based on a dataflow model. Dean et al porposed another approach to enable deep learning on larger models that uses a distributed cluster of computers and adapts the learning algorithms accordingly [12]. Coates et al recently proposed a combination of the GPU and cluster approach [13].

Outside of deep learning, recent work in the area of image/video processing applications by Qadeer et al has shown that it is possible to build programmable accelerators that offer better energy and area efficiency than GPU-like architectures but at the same time are more flexible (in terms of number of potential applications they can support) than custom hardware ASICs [14].

III. Background

A. Training of Deep Belief Networks

This section provides a brief overview of training Deep Belief Networks (DBNs). For a detailed treatment please refer to [4].

Figure 1(a) shows a neural network (NN) with 2 hidden layers and 3 sets of weights that the training procedure aims to find. In the DBN setting, the pretraining phase treats the NN as two separate Restricted Boltzmann Machines (RBMs) as shown in Figure 1(b). Pretraining proceeds by performing unsupervised training one RBM at a time starting from the lowest RBM. For each RBM, it uses a procedure based on contrastive divergence [15].

Once pretraining is complete, the weights have some reasonable values (lets call it W_{PT}). This is followed by back-propagation based supervised training on the entire NN starting with weights W_{PT} and using the training data set. This fine tunes the weights to W_T. These weights (W_T) are then used during the testing phase to classify new input vectors. The overall picture is shown in Figure 2.

In our evaluations, both pretraining and training stages use minibatches where the weights are updated by looking at a number of input vectors (corresponding to the minibatch size) at a time. Pretraining and training are stopped when they have gone through the entire training set a fixed number of times (corresponding to the number of epochs). Note that pretraining uses only the training inputs and not the training outputs whereas training uses both. Also, the final metric that we care about in our evaluations is the classification accuracy of the neural network with weights W_T on a separate test input set.

In this work, we focus on evaluating the robustness of the pretraining, training and testing stages to random errors. We expect at least the pretraining stage to be error resilient as any errors during this stage would result in corruption of values in W_{PT} which have the potential of being corrected by the training stage.

We also believe that in the future, a scenario might exist where the deep neural networks are trained on clusters or servers but are actually used for classification tasks on mobile front-ends. In such a scenario, the testing phase would be carried out on mobile devices and as such we also evaluate the potential of implementing the testing stage using a low precision fixed point implementation.

B. Classification Task

In this work, we focus on the task of handwritten digit recognition. We use 60,000 training images and 10,000 test images from the MNIST database for our experiments [16]. A sample of the MNIST images are shown in Figure 3(a). Each image is of size 28x28 pixels.

The neural network architecture used for our experiments is shown in Figure 3(b). It has two hidden layers with 100 units each.

C. Classification Without Errors

In this section we look at the classification performance of neural networks (with and without pretraining) in classifying handwritten digits.

1) Neural Networks Without Pretraining: Figure 4 presents the classification error of a neural network with one hidden layer that was trained using back-propagation. The default parameters used were 4 epochs, minibatch size of 100 and 100 hidden units. We observe that increasing the number of epochs reduces the classification error on the test set but the benefits seem to slow down after 8 epochs. A minibatch size of 100 seems appropriate and increasing the number of hidden units beyond 100 seems to have a limited effect on the classification error on the test set.

Figure 5(a) and Figure 5(b) present the classification errors for neural networks with 2 hidden layers and 100 + 50 units and 100 + 100 units respectively. Both show similar decrease in classification error on the test set as was seen in the case of the neural network with one layer (Figure 4(a)). Figure 5(c) compares the classification errors on the test set for all three architectures (1 hidden layer, 2 hidden layers with 100 and 50 units, and 2 hidden layers with 100 and 100 units).

2) Effect of Pretraining: In this section, we look at the effect of pretraining on the weights and the final classification errors of the neural network with 2 layers with 100 and 100 units.

Figure 6 presents a visual representation of the weights of the 100 hidden units of the first hidden layer. Each image there has 28x28 pixels each of which represent the weight of the connection of that unit to the corresponding pixel in the input image. As can be seen in Figure 6(a), right after pretraining, the weights begin to detect specific shapes in the input images. Training refines these weights as shown in Figure 6(b) but the changes are small and can hardly be perceived by visual inspection.

Although, the changes made by the training stage is small, it has a significant impact on the final classification error on the test set. Figure 7 shows the change in classification error during testing using a neural network that used the pretraining weights for the hidden layers (training changed the weights of the output layer only) and using a neural network that updated the weights of the hidden layers during training. For example, training using 16 epochs reduces the classification error during testing by more than a factor of half.

Figure 8 presents the effect of increasing the number of epoch during pretraining and during training on the final test error rate. We observe that increasing the number of epoch during training is more beneficial as compared to increasing number of epochs during pretraining.

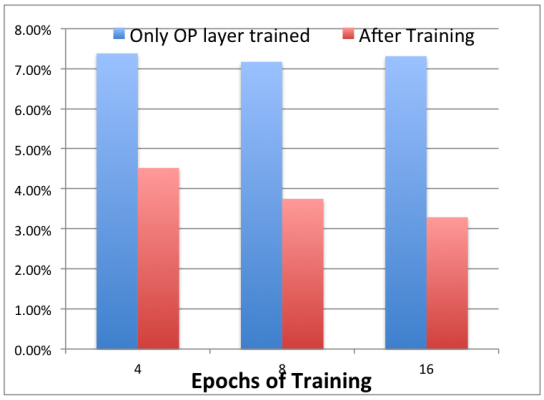


Fig. 7. The effect of supervised training on classification errors.

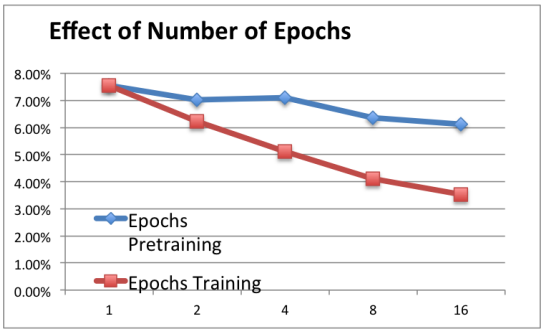
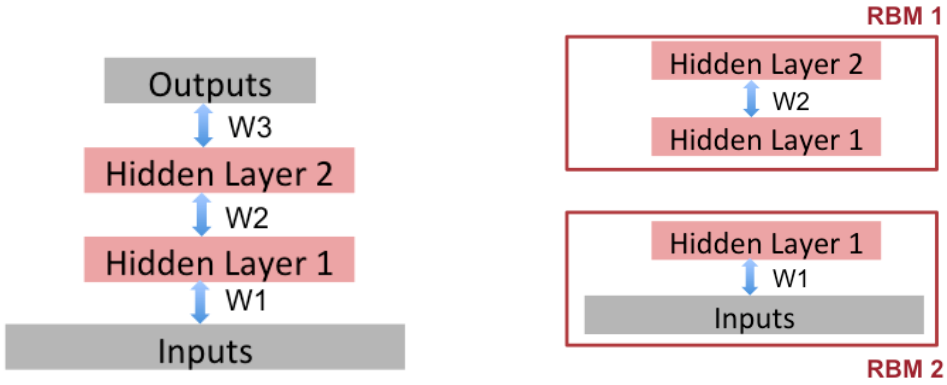


Fig. 8. The effect of increasing number of epochs of pretraining and training.

IV. Error Injection Methodology

In this section, we present our methodology for evaluating the robustness of the pretraining, training and testing stages to random errors. We also present our methodology for evaluating the precision requirement for a fixed point implementation of the testing stage.



(a) (b)

Fig. 1. (a) A neural network with two hidden layers (b) Pretraining in a DBN decomposes the neural network into Restricted Boltzmann Machines (RBMs) which are then trained one at a time from the bottom up.

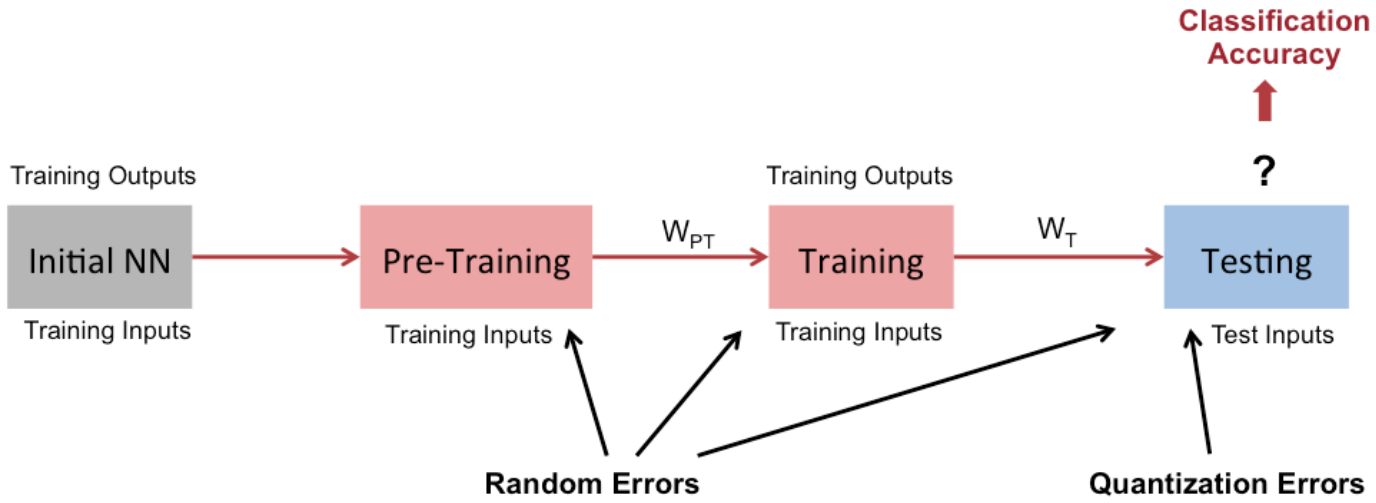


Fig. 2. Various steps in using a DBN for classification. In this work we study the effect of random errors on the pretraining, training and testing stages and that of quantization errors on the testing stage.

A. *Error in Pretraining*

To emulate errors in pretraining we corrupt the weights obtained after pretraining (W_{PT}) and let the subsequent stages (training and testing) continue without errors. The number of errors introduced in W_{PT} is decided by the fault rate. A fault rate of 1% means that on an average, 1 out of 100 weights in W_{PT} will be corrupted. We attempt to assign a reasonable value to the corrupted weights by choosing their values from a distribution that is close to the distribution of weights in W_{PT} without errors (shown in Figure 9(a)). We approximate this distribution by a normal distribution whose mean and variance we estimate to be μ and σ. Based on these estimates the erroneous weights are drawn as follows under three scenarios:

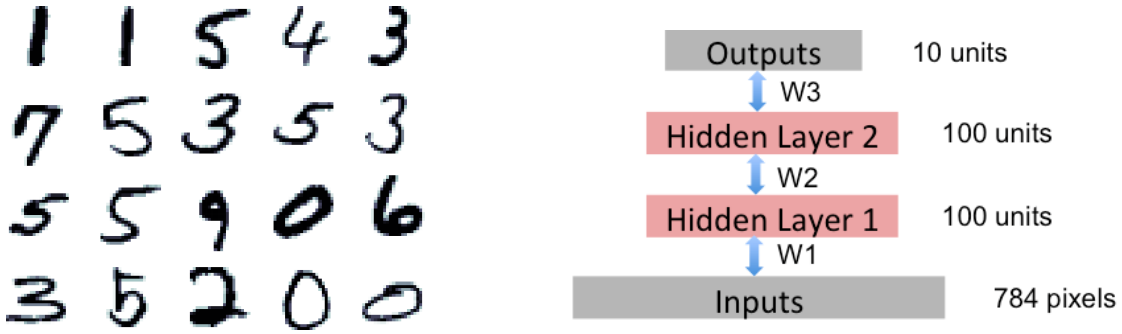
1) Nominal: In the nominal case, the erroneous weights are drawn from a normal distribution with parameters μ and σ.
2) Severe: In the severe case, the erroneous weights are drawn from a normal distribution with parameters μ and 10σ.
3) Corrected: In the corrected case, we look at the possibility of correcting erroneous weights by approximating them to be the average of the nearby weights. This of course depends on being able to detect when errors occur. Also, we only apply this to first layer weights as it is has a clear notion of nearby weights (weights from nearby pixels). To emulate this scenario, we corrupt weights in W_{PT} by replacing them with the average of their nearby weights.

B. *Error in Training*

To emulate errors in training we follow an approach very similar to the one for emulating error in pretraining (Section IV-A). We estimate the mean and variance (μ and σ) of the weights in W_T and use that to corrupt weights under three scenarios: nominal, severe and corrected.

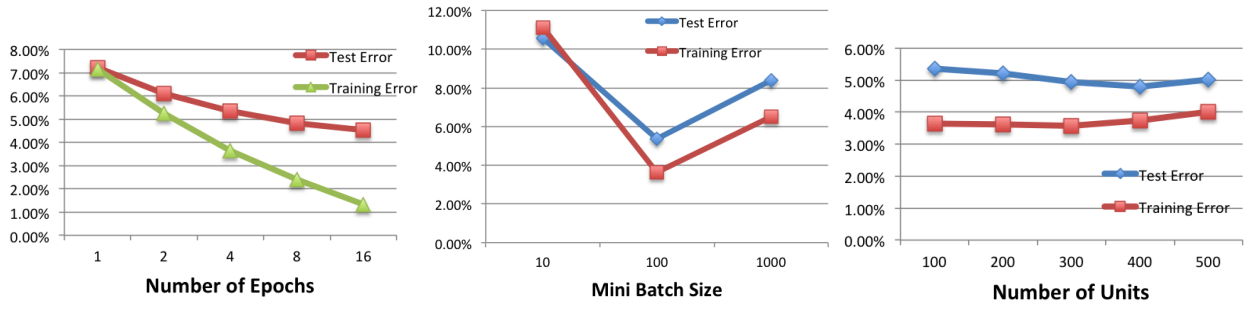
C. *Error in Testing*

To emulate errors in testing, we corrupt the output activations of the hidden layers at a given fault rate. To assign the corrupted output activations a reasonable value, we look at the distribution of output activations of the two layers (Shown in Figure 9(c) and Figure 9(d)). Since, the output units have a sigmoid non-linearity, nost values are either 0 or 1. To make things simpler, instead of accurately modeling these



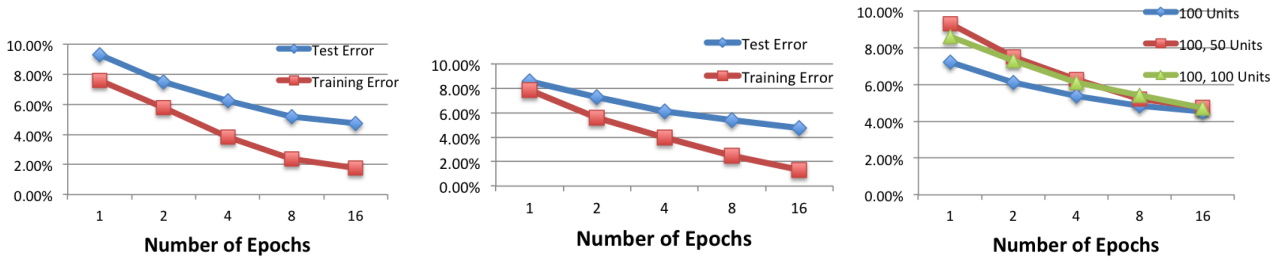
(a) (b)

Fig. 3. (a) Sample digits from the MNIST dataset of handwritten digits (b) The neural network architecture used for our digit recognition task.



(a) (b) (c)

Fig. 4. Effect of varying different parameters in the training of a NN with 1 hidden layer on its classification error (default parameters: 4 epochs, mini-batch size of 100 and 100 units) (a) Effect of varying the number of training epochs (b) Effect of varying the mini-batch size (c) Effect of varying the number of units in the hidden layer.



(a) (b) (c)

Fig. 5. Classification errors of NNs with two hidden layers (a) NN with two hidden layers with 100 and 50 units (b) NN with two hidden layers with 100 and 50 units (c) Comparison of the two layer networks with the one layer network.

distributions, we used a uniform distribution in the range [0,1] to draw the corrupted values.

V. Error Injection Results

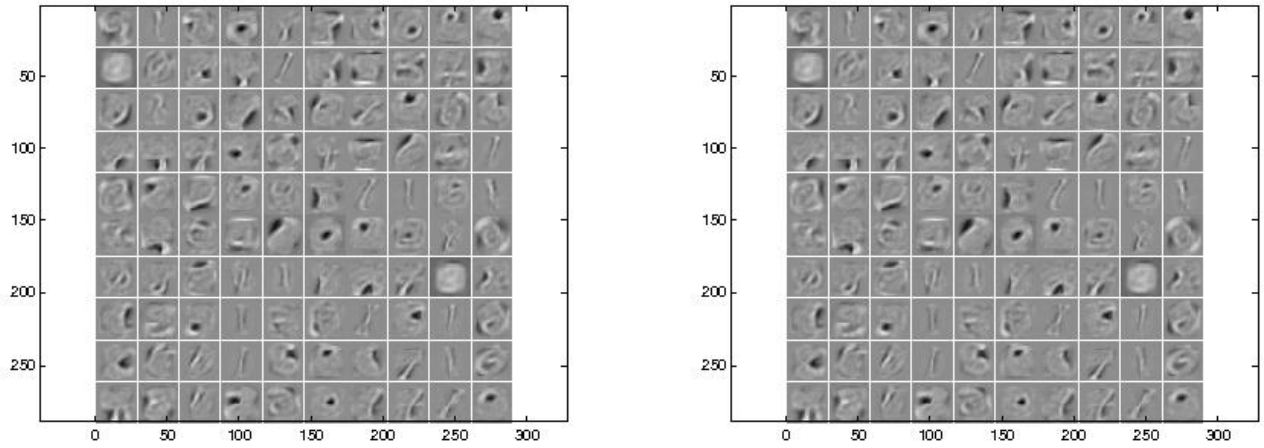
This section presents the results of our error injection experiments.

A. Pretraining

Figure 10 shows the classification error rate on the test set in presence of errors in pretraining under the three different error scenarios. We observe that for nominal errors an error rate as high as $10-20\%$ has classification accuracies very close to that of the error free case. For severe errors, an error rate of 1% has classification accuracies very close to that of the error free case. Even with 100% error rate, the classification works well compared to a completely random classification (for 10 classes a random classifier would have an error rate of 90%). We also observe that the correction scheme of replacing corrupted layer 1 weights with the average of their neighboring weights performs really well even at very high error rates (say 30%).

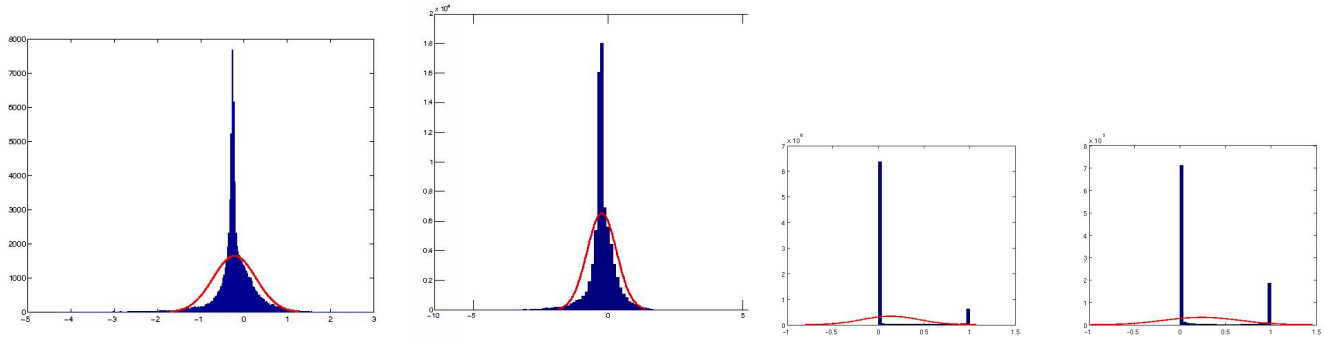
B. Training

Figure 11 shows the classification error rate on the test set in presence of errors in training under the three different error scenarios. We observe that for nominal errors an error rate as high as $10-20\%$ has classification accuracies very close to that of the error free case. For severe errors, an error rate of 1% has classification accuracies very close to that of the error free case. At higher error rates, the classification becomes almost random (it approaches an error rate of 90%). We also observe that the correction scheme of replacing corrupted layer 1 weights with the average of their neighboring weights performs really well even at very high error rates (say 30%).



(a) (b)

Fig. 6. First layer weights after (a) pretraining (1 epoch) (b) training (4 epochs).



(a) (b) (c) (d)

Fig. 9. (a) Distribution of weights after pretraining (W_{PT}) (b) Distribution of weights after training (W_T) (c) Distribution of output activations of hidden layer 1 (d) Distribution of output activations of hidden layer 2

C. Testing

Figure 12 shows the classification error rate on the test set when errors in testing are present in either of the two hidden layers or in both layers. We observe that for errors with error rate as high as $1 - 10\%$ the classifier still has acceptable classification accuracies.

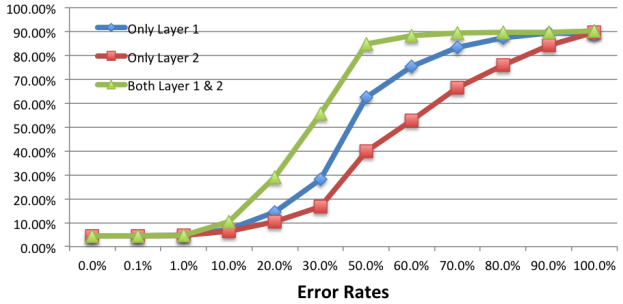


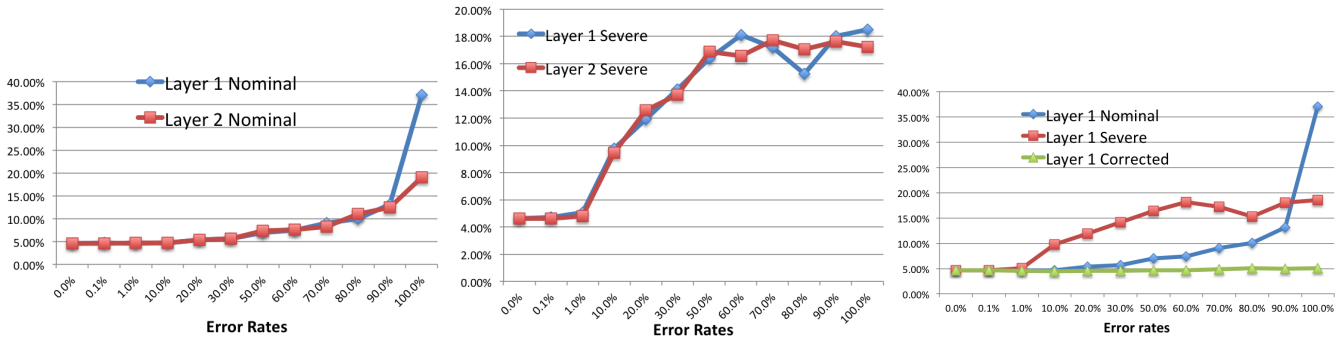
Fig. 12. Classification error rate on the test set with errors in testing.

VI. Precision Requirements for Testing

We imagine a scenario where the weights for the neural network are found by performing pretraining and training with double precision operations (possibly on a server) and then classification is performed on mobile devices using low precision fixed point operations. We performed evaluations to determine the number of bits required to represent the weight during testing.

To do so, we first start with double precision weights found after training and quantize them according to different fixed point representations. We then use these quantized weights during the testing stage which is still implemented in floating point. The fixed point representations used are shown in Figure 13. We used 1 sign bit and 5 integer bits. The number of fractional bits was varied and the effect on classification error on the test set was evaluated. The results are presented in **??**. We observe that 6 fractional bits exhibit the same accuracy as that of a double precision implementation.

This gave us an initial estimate of the precision required to represent the weights. We fixed our weights to have a fixed point representation with 12 total bits out of which 6 were



(a) (b) (c)

Fig. 10. Classification error rate on the test set with errors in pretraining (a) Nominal Errors (b) Severe Errors (c) Layer 1 results showing Corrected Errors

(a) (b) (c)

Fig. 11. Classification error rate on the test set with errors in training (a) Nominal Errors (b) Severe Errors (c) Layer 1 results showing Corrected Errors

fractional bits. We then performed detailed fixed point simulations of the testing stage with all operations implemented in fixed point. We experimented with different bit widths for the input, output and activation of each layer and evaluated the effect on the classification error on the test set. The architecture presented in Figure **??** with 10 total bits (and 8 fractional bits) for input, output and activations was found to have the same accuracy as that of a double precision floating point implementation.

Fig. 13. Fixed point representation with variable number of bits to represent the fractional part.

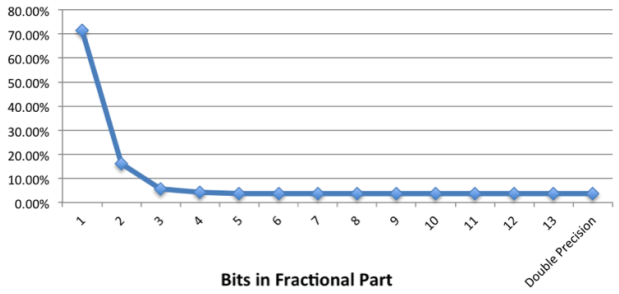


Fig. 14. Classification error on the test set for different number of bits in the fractional part of the fixed point representation.

VII. Conclusion

In this work, we evaluated the effect of random errors on the pretraining, training and testing stages of using a deep neural network based on training it as a DBN. Our results show that for nominal errors in both pretraining and training, the classification accuracy at $10-20\%$ error rate is similar to that of the error free case. For severe errors in both pretraining and training, the classification accuracy at 1% error rate is similar to that of the error free case. We also showed that one possible correction strategy to correct corrupted weights of the first layer (either in pretraining or training) is to replace it with the average of the neighboring weights. In such scenarios, the classification accuracy at $30-50\%$ error rate is similar to that of the error free case.

We also performed fixed point simulations to determine the precision required for representing various quantities (weights, input, output, hidden layer activations) in fixed point for the case when the testing stage is implemented using a fixed point implementation. We found that it is indeed possible to have low precision implementations of the testing stage in fixed point. We presented a low precision fixed point implementation that had the same accuracy as a double precision implementation. Our fixed point implementation represented the weights with 12 bits and the input, output, and hidden layer activations 10 bits.

This high tolerance to errors indicates that it might indeed be possible to implement accelerators for deep neural networks using low power but unreliable hardware substrates.

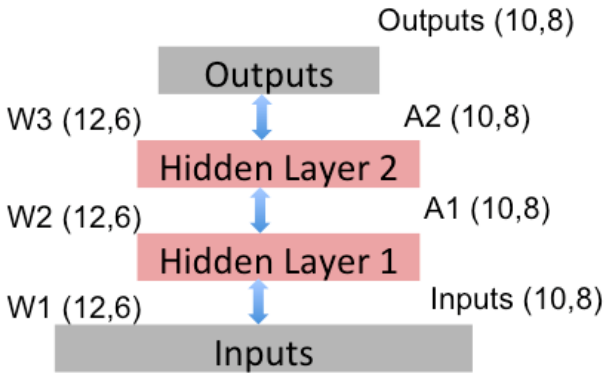


Fig. 15. A fixed point architecture that has the same classification accuracy as the double precision implementation. (m,n) means the number is represented using m total bits and n fractional bits in the fixed point implementation.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1106–1114.

[2] A.-R. Mohamed, T. N. Sainath, G. Dahl, B. Ramabhadran, G. E. Hinton, and M. A. Picheny, "Deep belief networks using discriminative features for phone recognition," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 5060–5063.

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," 2007.

[4] G. E. 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] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 1096–1103.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] L. Arnold, S. Rebecchi, S. Chevallier, and H. Paugam-Moisy, "An introduction to deep learning," in *ESANN*, 2011.

[8] J. Choi, E. P. Kim, R. A. Rutenbar, and N. R. Shanbhag, "Error resilient mrf message passing architecture for stereo matching," in *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, 2013, pp. 348–353.

[9] E. Kim, D. Baker, S. Narayanan, D. Jones, and N. Shanbhag, "Low power and error resilient pn code acquisition filter via statistical error compensation," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, 2011, pp. 1–4.

[10] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors."

[11] C. Farabet, Y. LeCun, K. Kavukcuoglu, and E. Culurciello, *Large-scale FPGA-based convolutional networks*, 2011.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012, pp. 1232–1240.

[13] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, "Deep learning with cots hpc systems," in *ICML (3)*, 2013, pp. 1337–1345.

[14] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 24–35.

[15] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.