LOW PRECISION ARITHMETIC FOR DEEP LEARNING

Matthieu Courbariaux & Jean-Pierre David
Department of Electrical Engineering
École Polytechnique de Montréal
Montréal, QC H3T 1J4, Canada
{matthieu.courbariaux,jean-pierre.david}@polymtl.ca

Yoshua Bengio *
Department of Computer Science and Operations Research
Université de Montréal
Montréal, QC H3T 1J4, Canada
yoshua.bengio@gmail.com

ABSTRACT

We simulate the training of a set of state of the art neural networks, the Maxout networks (Goodfellow *et al.*, 2013a), on three benchmark datasets: the MNIST, CIFAR10 and SVHN, with three distinct arithmetics: floating point, fixed point and dynamic fixed point. For each of those datasets and for each of those arithmetics, we assess the impact of the precision of the computations on the final error of the training. We find that *very low precision computation is sufficient* not just for running trained networks but *also for training them*. For example, almost state-of-the-art results were obtained on most datasets with **10** bits for computing activations and gradients, and **12** bits for storing updated parameters.

1 INTRODUCTION

Deep learning is very often limited by memory and computational power. Lots of previous works address the best exploitation of general-purpose hardware, typically CPU clusters (Dean *et al.*, 2012) and GPUs (Coates *et al.*, 2009; Krizhevsky *et al.*, 2012a). Faster implementations usually lead to state of the art results (Dean *et al.*, 2012; Krizhevsky *et al.*, 2012a; Sutskever *et al.*, 2014).

Actually, such approaches always consist in adapting the algorithm to best exploit state of the art hardware. Nevertheless, some dedicated deep learning hardware is appearing as well. FPGA implementations claim a better power efficiency than general-purpose hardware (Farabet *et al.*, 2011; Kim *et al.*, 2009). The corresponding ASIC implementations are even more efficient (Pham *et al.*, 2012). In contrast with general-purpose hardware, dedicated hardware such as ASIC and FPGA enables to build the hardware from the algorithm. In this context, it is important to know what is the minimum precision acceptable.

Actually, minimizing the size of the arithmetic operators and the size of the memories would lead to architectures with more operators and memories working in parallel. It would also drastically reduce the power consumption. For instance, using single precision (32 bits) instead of double precision (64 bits) for a floating point multiplier reduces its area by four on modern FPGAs (Govindu *et al.*, 2004; Underwood, 2004).

In this paper, we simulate the training of a set of state of the art neural networks, the Maxout networks (Goodfellow *et al.*, 2013a), on three benchmark datasets: the MNIST, CIFAR10 and SVHN, with three distinct arithmetics: floating point, fixed point and dynamic fixed point. For each of those datasets and for each of those arithmetics, we assess the impact of the precision of the computations on the final error of the training. We find that *very low precision computation is sufficient* not just for running trained networks but *also for training them*. For example, almost state-of-the-art results

*Yoshua Bengio is a CIFAR Senior Fellow.

were obtained on each dataset with **10** bits for computing activations and gradients, and **12** bits for storing updated parameters. We made our code available [1].

2 MAXOUT NETWORKS

A Maxout network is a multi-layer neural network that uses maxout units in its hidden layers. A maxout unit outputs the maximum of a set of k dot products between between k weight vectors and the input vector of the unit (e.g., the output of the previous layer):

$$h_i^l = \max_{j=1}^{k}(b_{i,j}^l + w_{i,j}^l \cdot h^{l-1})$$

where h^l is the vector of activations at layer l and weight vectors $w_{i,j}^l$ and biases $b_{i,j}^l$ are the parameters of the j-th filter of unit i on layer l.

A maxout unit can be seen as a generalization of the rectifying units (Jarrett *et al.*, 2009; Nair and Hinton, 2010; Glorot *et al.*, 2011; Krizhevsky *et al.*, 2012b) $h_i^l = \max(0, b_i^l + w_i^l \cdot h^{l-1})$ which correspond to a maxout unit when $k = 2$ and one of the filters is forced at 0 (Goodfellow *et al.*, 2013a). Combined with dropout, a very effective regularization method (Hinton *et al.*, 2012), maxout networks achieved state-of-the-art results on a number of benchmarks (Goodfellow *et al.*, 2013a), both as part of fully connected feedforward deep nets and as part of deep convolutional nets. The dropout technique is a good approximation of model averaging with shared parameters across an exponentially large number of networks that are formed by subsets of the units of the original noise-free deep network. In our article, we reproduce the experiments of Goodfellow *et al.* (2013a) on the MNIST, CIFAR10 and SVHN data sets while changing the arithmetic operations and the numerical precision of the computations.

3 FLOATING POINT

Format	Total bit-width	Exponent bit-width	Mantissa bit-width
Double precision floating point	64	11	52
Single precision floating point	32	8	23
Half precision floating point	16	5	10

Table 1: Definitions of double, single and half precision floating point formats.

Floating point formats are often used to represent real values. They consist in a sign, an exponent, and a mantissa. The exponent gives the floating point formats a wide range, and the mantissa gives them a good precision. You can compute the value of a floating point number using the following formula:

$$value = (-1)^{sign} \times \left(1 + \frac{mantissa}{2^{23}}\right) \times 2^{(exponent - 127)}$$

Table 1 shows the exponent and mantissa widths associated with each floating point format. In our experiments, we use single precision floating point format as our reference as it is the most widely used format in deep learning, especially for GPU computation. We show that the use of half precision floating point format has little to no impact on the training of neural networks. As of the moment of writing this article, no standard exists below the half precision floating point format.

4 FIXED POINT

Fixed point formats consist in a signed mantissa and a global scaling factor shared between all fixed point variables. The scaling factor can be seen as the position of the radix point. It is usually fixed, hence the name "fixed point". Reducing the scaling factor reduces the range and augments the

[1] `https://github.com/MatthieuCourbariaux/deep-learning-arithmetic`

precision of the format. The scaling factor is typically a power of two for computational efficiency (the scaling multiplications are then replaced with shifts). As a result, fixed point arithmetic can also be seen as a floating point arithmetic with a *shared fixed global exponent*. Fixed point arithmetic is commonly found on embedded systems with no FPU (Floating Point Unit). It relies on integer operations. It is hardware-wise cheaper than its floating point counterpart, as the exponent is fixed.

In Vanhoucke *et al.* (2011), the authors show that the use of 8 bits fixed point arithmetic speeds up over three times the *application* of a neural network on CPU, compared to floating point. This assumes that training has been previously performed with single precision floating point arithmetic. In our article, we go further by *actually training neural networks with fixed point arithmetic*. The application of a trained neural network (computing outputs given inputs) is required while training a neural network but the reciprocal is false. Training neural networks with fixed point arithmetic has already been done in older works (Holt and Baker, 1991; Presley and Haggard, 1994; Savich *et al.*, 2007). However, those works use very small models and datasets in today standards. One of our contributions is to update their results with some state of the art models, that is to say Maxout networks, and some of the latest benchmark datasets, that is to say MNIST, CIFAR10 and SVHN.

5 DYNAMIC FIXED POINT

Dynamic fixed point arithmetic (Williamson, 1991) is a variant of fixed point arithmetic in which there are several scaling factors instead of a single global one. Those scaling factors are not fixed. As such, it can be seen as a compromise between floating point arithmetic - where each scalar variable owns its scaling factor which is updated during each operations - and fixed point arithmetic - where there is only one global scaling factor which is never updated. In dynamic fixed point, a few grouped variables share a scaling factor which is updated from time to time to reflect the statistics of values in the group.

In practice, we associate each layer's weights, bias, weighted sum, outputs (post-nonlinearity) and the respective gradients vectors and matrices with a different scaling factor. Those scaling factors are initialized with a global value. They can also be found while training with a higher precision format. During the training, we monitor the overflow rates associated with the scaling factors and their halves. We update the scaling factors at a given frequency: if the overflow rate associated with a scaling factor is superior to a given maximum overflow rate, we multiply this scaling factor by two. If the overflow rate associated with the half of a scaling factor is inferior to the maximum overflow rate, we divide this scaling factor by two.

Dynamic fixed point arithmetic was introduced several decades ago (Williamson, 1991) and it was recently used for an SVM implementation (Wang *et al.*, 2010). However, as far as we know, this is the first paper reporting on the use of dynamic fixed point arithmetic to train deep neural networks, so this is one of the contributions of this paper.

6 TWO DIFFERENT BIT WIDTHS

We use two different bit widths in our fixed point and dynamic fixed point arithmetic experiments: one for the assignment of parameters and the other for the rest of the computations. The idea behind this is to be able to accumulate small changes in the parameters (which requires more precision) and thus spare a few bits for the computations, which can be done with lower precision because of the implicitly averaging performed via stochastic gradient descent during training:

$$\theta_{t+1} = \theta_t - \epsilon \frac{\partial C_t(\theta_t)}{\partial \theta_t}$$

where $C_t(\theta_t)$ is the cost to minimize over the minibatch visited at iteration t using θ_t as parameters and ϵ the learning rate. We see that the resulting parameter is the sum

$$\theta_T = \theta_0 - \epsilon \sum_{t=1}^{T-1} \frac{\partial C_t(\theta_t)}{\partial \theta_t}.$$

The terms of this sum are not statistically independent (because the value of θ_t depends on the value of θ_{t-1}) but the dominant variations come from the random sample of examples in the minibatch

(θ moves slowly) so that a strong averaging effect takes place, and each contribution in the sum is relatively small, hence the demand for sufficient precision (when adding a small number with a large number).

7 SIMULATION

Since we do not have access to actual low-precision hardware, all of the results reported in this paper are obtained by simulating low-precision computation. We use single floating point format for computation and storage as it allows us to use existing GPU libraries, for instance Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012). However, each time an activation, a gradient or a parameter is stored, we artificially reduce its precision.

The only limitation of our simulation is that it does not take into account the size of the accumulators. We somehow make the hypothesis that the accumulators are as precise as the single floating point format. That being said, in practice, accumulators are often more precise than single floats (Lee *et al.*, 2008).

8 BASELINE RESULTS

We reproduce the results of Goodfellow *et al.* (2013a) on the MNIST, CIFAR10 and SVHN, using single precision floating point arithmetic. In the next section, LOW-PRECISION RESULTS, we assess the impact of low-precision arithmetic on the models of this section, BASELINE RESULTS.

Data set	Dimension	Labels	Training set	Test set
MNIST	784 (28×28 grayscale)	10	60K	10K
CIFAR10	3072 (32×32 color)	10	50K	10K
SVHN	3072 (32×32 color)	10	604K	26K

Table 2: Overview of the data sets used in this paper.

Format	Comp.	Up.	PI MNIST	MNIST	CIFAR10	SVHN
Goodfellow *et al.* (2013a)	32	32	0.94%	0.45%	11.68%	2.47%
Single precision floating point	32	32	1.05%	0.51%	14.05%	2.71%
Half precision floating point	16	16	1.10%	0.51%	14.14%	3.02%
Fixed point	20	20	1.39%	0.57%	15.98%	2.97%
Dynamic fixed point	10	12	1.28%	0.59%	14.82%	4.95%

Table 3: Test set error rates of single and half floating point formats, fixed and dynamic fixed point formats on the permutation invariant (PI) MNIST, MNIST (with convolutions, no distortions), CIFAR10 and SVHN datasets. **Comp.** is the bit-width of the computations and **Up.** is the bit-width of the parameters updates. The single precision floating point line refers to the results of our experiments attempting to reproduce the results of Goodfellow *et al.* (2013a) without training on the validation samples. It serves as a baseline to evaluate the degradation brought by lower precision.

8.1 MNIST

The MNIST (LeCun *et al.*, 1998) data set is described in Table 2. We do not use any data-augmentation (e.g. distortions) nor any unsupervised pre-training. We simply use minibatch stochastic gradient descent (SGD). We use a linearly decaying learning rate and a linearly saturating momentum. We regularize the model with dropout and a constraint on the norm of each weight vector, as in (Srebro and Shraibman, 2005).

We train two different models on the MNIST. The first is a permutation invariant (PI) model which is unaware of the structure of the data. It consists in two fully connected maxout layers followed

by a softmax layer. The second model consists in three convolutional maxout hidden layers (with spatial max pooling on top of the maxout layers) followed by a densely connected softmax layer.

This is the same procedure as in Goodfellow *et al.* (2013a), except that we do not train our model on the validation examples because it requires to stop the training at an arbitrary epoch, which adds some randomness to the final test error, which blurs the impact of low-precision arithmetic. As a consequence, our test error is slightly bigger that the one reported in Goodfellow *et al.* (2013a). The final test error is in Table 3.

8.2 CIFAR10

The CIFAR10 (Krizhevsky and Hinton, 2009) data set is described in Table 2. We preprocess the data using global contrast normalization and ZCA whitening. The model consists in three convolutional maxout layers, a fully connected maxout layer, and a fully connected softmax layer. Otherwise, we follow a similar procedure as with the MNIST dataset. This is the same procedure as in Goodfellow *et al.* (2013a), except that we do not train our model on the validation examples. The final test error is in Table 3.

8.3 Street View House Numbers

The SVHN (Netzer *et al.*, 2011) data set is described in Table 2. We applied local contrast normalization preprocessing the same way as Zeiler and Fergus (2013). The model consists in three convolutional maxout layers, a fully connected maxout layer, and a fully connected softmax layer. Otherwise, we followed the same approach as on the MNIST dataset. This is the same procedure as in Goodfellow *et al.* (2013a). The final test error is in Table 3.

9 Low-precision results

We assess the impact of low-precision arithmetic on the models of the previous section, BASELINE RESULTS.

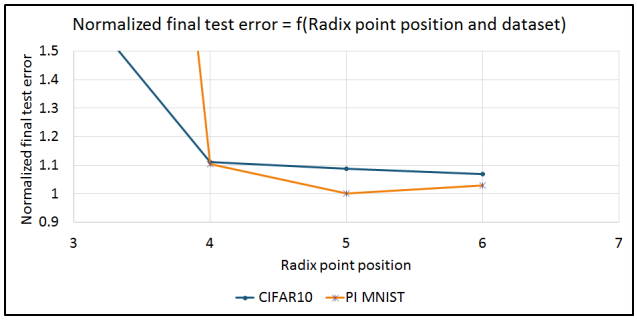

Figure 1: Final test error depending on the radix point position (5 means after the 5th most significant bit) and the dataset (permutation invariant MNIST and CIFAR10). The final test errors are normalized, that is to say divided by the dataset single float test error. The computations and parameters updates bit-widths are both set to 31 bits (32 with the sign).

9.1 Floating point

The use of half precision floating point format has little to no impact on the test set error rate, as shown in Table 3.

9.2 Fixed point

The optimal radix point position in fixed point is after the fifth most important bits, as illustrated in Figure 1. The corresponding range is approximately [-32,32]. The corresponding scaling factor

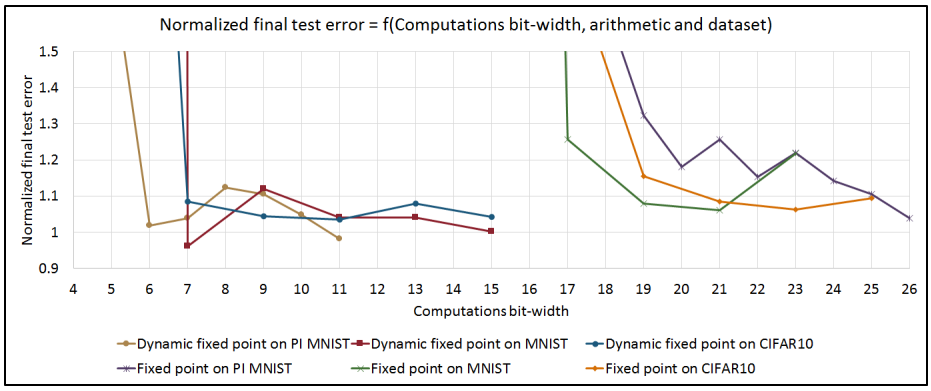

Figure 2: Final test error depending on the computations bit-width, the arithmetic (dynamic fixed or fixed point) and the dataset (permutation invariant MNIST, MNIST and CIFAR10). The final test errors are normalized, that is to say divided by the dataset single float test error. For both arithmetics, the parameters updates bit-width is set to 31 bits (32 with the sign). For fixed point arithmetic, the radix point is set after the fifth bit. For dynamic fixed point arithmetic, the maximum overflow rate is set to 0.01%.

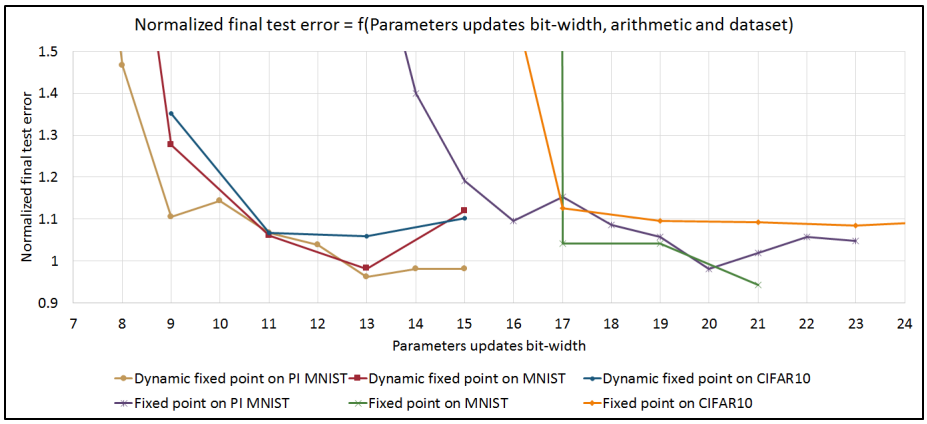

Figure 3: Final test error depending on the parameters updates bit-width, the arithmetic (dynamic fixed or fixed point) and the dataset (permutation invariant MNIST, MNIST and CIFAR10). The final test errors are normalized, that is to say divided by the dataset single float test error. For both arithmetics, the computations bit-width is set to 31 bits (32 with the sign). For fixed point arithmetic, the radix point is set after the fifth bit. For dynamic fixed point arithmetic, the maximum overflow rate is set to 0.01%.

depends on the bit-width we are using. The minimum bit-width for computations in fixed point is 19 (20 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 2. The minimum bit-width for parameters updates in fixed point is 19 (20 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 3. In the end, using 19 (20 with the sign) bits for both the computations and the parameters updates has little impact on the final test error, as shown in Table 3. Doubling the number of hidden units does not allow any further reduction of the bit-widths on the permutation invariant MNIST.

9.3 Dynamic fixed point

We find the initial scaling factors by training with a higher precision format. Once those scaling factors are found, we reinitialize the model parameters. We update the scaling factors once every

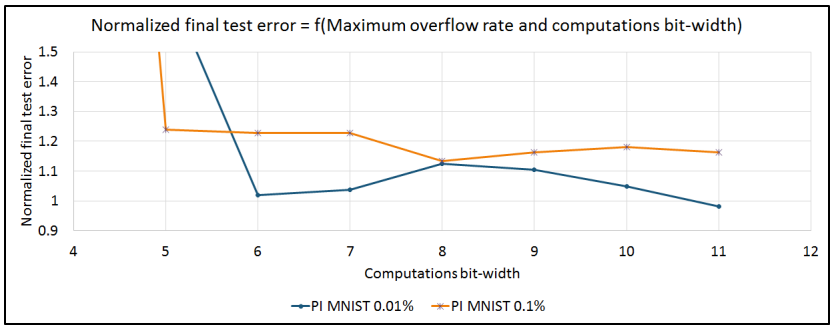

Figure 4: Final test error depending on the maximum overflow rate and the computations bit-width. The final test errors are normalized, that is to say divided by the dataset single float test error. The parameters updates bit-width is set to 31 bits (32 with the sign).

10000 examples. Augmenting the maximum overflow rate allows us to reduce the computations bit-width but it also significantly augments the final test error rate, as illustrated in Figure 4. As a consequence, we use a low maximum overflow rate of 0.01% for the rest of the experiments. The minimum bit-width for the computations in dynamic fixed point is 9 (10 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 2. The minimum bit-width for the parameters updates in dynamic fixed point is 11 (12 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 3. In the end, using 9 (10 with the sign) bits for the computations and 11 (12 with the sign) bits for the parameters updates has little impact on the final test error, with the exception of the SVHN data set, as shown in Table 3. Doubling the number of hidden units does not allow any further reduction of the bit-widths on the permutation invariant MNIST.

10 Discussion

We think there are two main arithmetic difficulties in the training of deep neural networks:

1. The activations, the gradients and the parameters typically have very different ranges
2. The gradients diminish during the training, so do their ranges

Half floating point format easily overcome those two difficulties at the cost of its exponent bit-width (5 bits). Fixed point format saves the exponent bit-width by sharing the exponent between all fixed point variables. However, the global exponent cannot simultaneously be adapted to the activations, the gradients, and the parameters. The resulting bit-width (20 bits) is wider than half precision floating point (16 bits). Dynamic fixed point format is an interesting compromise between floating and fixed point formats. The exponents are shared by groups of variables. The activations, the gradients, and the parameters have different exponents. The gradients exponents can be updated during the training. The resulting bit-width (12 bits) is narrower than half precision floating point (16 bits). Besides, using two bit-widths allows us to spare two additional bits on the computations (10 bits).

If very fast low-precision hardware is designed for taking advantage of these results, we conjecture that a high-precision fine-tuning could recover the small degradation brought by the low-precision training.

11 Conclusion

We have simulated the training of a set of state-of-the-art neural networks, the Maxout networks (Goodfellow *et al.*, 2013a), on three benchmark datasets: the MNIST, CIFAR10 and SVHN, with three distinct arithmetics: floating point, fixed point and dynamic fixed point. For each of those arithmetics, we have assessed the impact of the precision of the computations on the final error of

the training. We have found that very low precision computation is sufficient not just for running trained networks but also for training them. This opens the door to new memory optimizations of deep learning algorithms on general-purpose hardware, and most importantly, this opens the door to very power-efficient training of neural networks on dedicated hardware.

12 ACKNOWLEDGEMENT

We thank the developers of Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012), a Python library which allowed us to easily develop a fast and optimized code for GPU. We also thank the developers of Pylearn2 (Goodfellow *et al.*, 2013b), a Python library built on the top of Theano which allowed us to easily interface the data sets with our Theano code. We are also grateful for funding from NSERC, the Canada Research Chairs, Compute Canada, and CIFAR.

REFERENCES

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Coates, A., Baumstarck, P., Le, Q., and Ng, A. Y. (2009). Scalable learning for object detection with gpu hardware. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4287–4293. IEEE.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large scale distributed deep networks. In *NIPS'2012*.

Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., and LeCun, Y. (2011). Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *AISTATS'2011*.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013a). Maxout networks. Technical report, Université de Montréal.

Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013b). Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*.

Govindu, G., Zhuo, L., Choi, S., and Prasanna, V. (2004). Analysis of high-performance floating-point arithmetic on fpgas. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 149. IEEE.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580.

Holt, J. L. and Baker, T. E. (1991). Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 121–126. IEEE.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, pages 2146–2153. IEEE.

Kim, S. K., McAfee, L. C., McMahon, P. L., and Olukotun, K. (2009). A highly scalable restricted boltzmann machine fpga implementation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 367–372. IEEE.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012a). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'2012)*.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012b). ImageNet classification with deep convolutional neural networks. In *NIPS'2012*.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324.

Lee, D., Ryu, C., Park, J., Kwon, K., and Choi, W. (2008). Design and implementation of 16-bit fixed point digital signal processor. In *SoC Design Conference, 2008. ISOCC'08. International*, volume 2, pages II–61. IEEE.

Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. Deep Learning and Unsupervised Feature Learning Workshop, NIPS.

Pham, P.-H., Jelaca, D., Farabet, C., Martini, B., LeCun, Y., and Culurciello, E. (2012). Neuflow: dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 1044–1047. IEEE.

Presley, R. K. and Haggard, R. L. (1994). A fixed point implementation of the backpropagation learning algorithm. In *Southeastcon'94. Creative Technology Transfer-A Global Affair., Proceedings of the 1994 IEEE*, pages 136–138. IEEE.

Savich, A. W., Moussa, M., and Areibi, S. (2007). The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. *Neural Networks, IEEE Transactions on*, **18**(1), 240–252.

Srebro, N. and Shraibman, A. (2005). Rank, trace-norm and max-norm. In *Proceedings of the 18th Annual Conference on Learning Theory*, pages 545–560. Springer-Verlag.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. Technical report, arXiv preprint arXiv:1409.3215.

Underwood, K. (2004). Fpgas vs. cpus: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180. ACM.

Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*.

Wang, J.-F., Kuan, T.-W., Wang, J.-C., and Sun, T.-W. (2010). Dynamic fixed-point arithmetic design of embedded svm-based speaker identification system. volume 6064 LNCS, pages 524 – 531, Shanghai, China. Embedded environment;Fixed point arithmetic;Linear prediction cepstral coefficients;Point design;Sequential minimal optimization;Speaker identification;Speaker identification systems;.

Williamson, D. (1991//). Dynamically scaled fixed point arithmetic. pages 315 – 18, New York, NY, USA. dynamic scaling;iteration stages;digital filters;overflow probability;fixed point arithmetic;fixed-point filter;.

Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. In *International Conference on Learning Representations*.