Simultaneous Inference and Training using On-FPGA Weight Perturbation Techniques

Siddhartha*, Steven J.E. Wilton†, David Boland*, Barry Flower*, Perry Blackmore‡, Philip H.W. Leong*
* The University of Sydney, Email: {siddhartha.siddhartha, david.boland, barry.flower, philip.leong}@sydney.edu.au
† University of British Columbia, Vancouver, Canada, Email: stevew@ece.ubc.ca
‡Defence Science and Technology Group, Email: perry.blackmore@dst.defence.gov.au

***Abstract*—We present an FPGA-optimized implementation of online neural network training based on weight perturbation (WP) techniques. When compared to the classic backpropagation (BP) algorithm, WP is capable of delivering competitive performance while occupying minimal area resources. Perturbation-based methods have been demonstrated as viable training techniques and are suitable for on-line learning applications which adapt to changing conditions. The viability of applying WP-based on-chip training for low-precision fixed-point hardware is demonstrated on two distinct MLP benchmarks: the Iris dataset classification network and an RF anomaly detector. When synthesized to a Xilinx Kintex-7 XC7K410T FPGA, WP offers a $3-10\times$ area savings with <1% degradation in accuracy compared with backpropagation. Compared with an inference-only implementation the overhead of introducing on-chip learning is approximately 30%.**

I. Introduction

Recent machine learning research has seen the emergence of FPGA-optimized implementations for server, embedded and real-time applications, e.g. [1]. Order of magnitude lower latency and reduced power over CPU and GPU implementations has already been demonstrated. While most recent efforts have been focused on inference, supplementing inference with on-chip training improves the adaptability of such designs.

In this paper, we focus on on-chip training methods that enable on-line learning. We show that training can be accomplished using finite difference approximations, in which weights are perturbed, and the impact on the output is used to determine gradients, without the need to implement the full backpropagation algorithm. Compared to backpropagation, these techniques have two advantages: they can re-use much of the inference hardware, leading to low overhead, and are less sensitive to reduced precision arithmetic, leading to efficient FPGA implementations. Although weight perturbation has been described previously [2], to the best of our knowledge, this is the first paper that demonstrates the advantages of these techniques for on-line FPGA learning tasks.

We demonstrate our techniques using two target applications: (1) the Iris dataset, which is a multivariate dataset commonly used as a benchmark in classification neural network studies; and (2) an RF anomaly detector, which monitors an RF spectrum and determines whether the spectrum exhibits unusual characteristics, such as those that might occur if an adversary is interfering with a channel.

In these applications, we envisage that the network may initially be trained either in hardware or software. After training, during operation, on-line training can be performed in parallel (or interleaved) with inference, refining the values of weights as the environment (eg. input channel behaviour) changes. In addition to providing resilience to changes in the channel characteristics, the technique makes it less important to obtain a correct model of the correct behaviour of the channel at design time, since the parameters can be refined when the chip is put into the target system. The specific contributions of this paper include:

1) the first FPGA architecture for simultaneous on-chip training using weight perturbation techniques with minimal overhead compared to an inference-only design,
2) a demonstration of the feasibility of weight perturbation to multilayer perceptron (MLP) networks sizes of an order of magnitude larger than [2], and
3) a comparison of weight perturbation to backpropagation algorithms, in terms of convergence rate, accuracy, and FPGA on-chip area for two sample applications.

II. Background

A. Backpropagation

Backpropagation is a well-known algorithm for computing gradients during training to updating weights [3]–[5]. Weights are updated by the formula

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{1}$$

for all weights w_i, where Δw_i is the change to be applied to the weights. $\partial E/\partial w_i$ is the partial derivative of E with respect to w_i, and η is the learning rate. Backpropagation uses the following weight update formula:

$$\frac{\partial E(W)}{\partial W^i} = \delta^i (x^{i-1})^T \tag{2}$$

where

$$\delta^i = \begin{cases} (x^L - d) \circ f'_L(s^L), & \text{if } i = L-1 \\ (W^{i+1})^T \delta^{i+1} \circ f'_{i+1}(s^i) & \text{otherwise.} \end{cases} \tag{3}$$

for $i = [L-1, \ldots, 1]$, where T denotes the transpose operation and $\circ$ the Hadamard product formed by element-wise multiplication.

B. Weight Perturbation

Weight perturbation [2] implements gradient descent with direct approximation of the gradient. At each training step, a small perturbation Δ is applied to each parameter in the network one at a time. A feed-forward pass is then performed to determine the output with the perturbation applied. The difference between the perturbed output $E(w_i + \Delta)$ and the original output $E(w_i)$ can be used to estimate the derivative using

$$\frac{\partial E(W)}{\partial w_i} = \frac{E(w_i + \Delta) - E(w_i)}{\Delta} + O(\Delta) \quad (4)$$

where the error introduced by this approximation, $O(\Delta)$, is proportional to Δ. Unfortunately, reducing Δ does not necessarily reduce the error when finite precision arithmetic is used. It is important to note that a separate forward pass is required for each parameter in the network during each training iteration, leading to an $O(N^4)$ computational complexity where N is the number of nodes in the network. However, for some applications, on-chip training that adapts to slow changes in the real-time data can be advantageous. In such cases, since direct computation of gradients is not required, this technique has the distinct advantage of incurring only minimal additional hardware over an inference-only implementation.

C. Related Work on FPGAs

Several existing studies [6]–[8] have demonstrated that using fixed-point 16b weights is sufficient for implementing backpropagation for MLP networks on FPGAs. In [7], the authors report that 16b fixed-point implementation of backpropagation delivers 12× better speed and used 13× less area than a floating-point implementation. Other FPGA-based works on backpropagation have focused on efficiently using on-chip memory for weight storage and optimising precision so that more resources can be assigned to achieving parallelism.

III. Applications

Table I gives a summary of the network architecture of thee applications used in this paper to evaluate the potential of on-chip perturbation-based training.

Benchmark	Architecture	Parameters
Iris	4 – 7 – 12 – 3	170
Autoencoder	32 – 16 – 8 – 16 – 32	1352

TABLE I: Network architecture of Iris species detector and Autoencoder. Non-linear $tanh()$ activation after each layer output (except final output layer).

A. Iris

Iris is a multivariate dataset commonly used as a benchmark in classification neural network studies. The Iris dataset is a table of measurements on three species of *Iris* flowers. The dataset has a training/test set of length 150, four features and three classes. We avoid using the $softmax$ function at the output, as $softmax$ is highly susceptible to both under and overflow in fixed-point arithmetic. When trained in software with Tensorflow, the proposed Iris network achieves 95.3% accuracy on the test set.

B. RF Anomaly Detector

Anomaly detection on RF channels can be challenging due to the fast data rates of the physical layer. Recently, MLP-based techniques [9] have demonstrated promising results on real-time anomaly detection on RF signals. At the heart of the anomaly detector is an autoencoder MLP network. The autoencoder's first two layers encode the input signal by compressing it into fewer dimensions, while the final two layers reconstruct (or decode) the signal back to the original input. The mean-squared error loss is computed using the input into the autoencoder as the desired output. The intuition here is that, through training, the network learns the key hidden features of the input RF signal, which allows us to design anomaly detection systems based on simple techniques such as thresholding the loss observed at the output. In the scenario where there is an adversary interfering with the communication channel, the loss at the output would exceed the preset threshold and be flagged as an anomaly. While this system would suffice in stationary environments, in dynamically-changing environments in the field, the network needs to constantly train and adapt while performing real-time inference. This paper adds on-line training to the autoencoder network architecture described in [9].

IV. Hardware Architecture

Figure 1 shows the proposed system implementation. The prediction core carries out inference (predictions) on arriving inputs, based on the active network parameters. We use ping-pong buffering for the parameters, such that the training core and the prediction core can operate side-by-side. When a new set of parameters are ready after a training epoch, the select lines to the multiplexer (*mux2*) and demultiplexer (*demux2*) are flipped, such that the newly trained parameters are now inputs to the prediction core, while next training iteration parameters are written to the other buffer.

Figure 1 shows the backpropagation training core design for an example network with two hidden layers. Equations 2 and 3 are evaluated by the *Compute Gradients* blocks, while the new trained parameters are computed in and stored by the *Update Params* blocks. The *Training Controller* block manages the on-chip training process. At the end of each training epoch, the controller toggles the multiplexer/demultiplexer select lines that orchestrate the ping-pong buffering mechanism. We design each block carefully to support an initiation interval (II) of 1, such that latency of each training epoch is minimized (*i.e.* the network parameters can be updated in real-time as quickly as possible). This is an area-throughput tradeoff, where we unroll the network computations spatially on the FPGA to achieve a high-throughput fully-pipelined training core design.

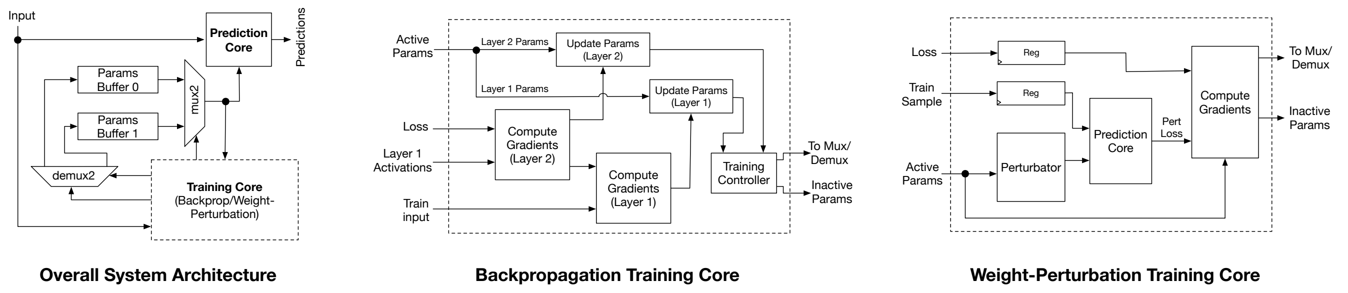


Fig. 1: Overall system design and the two variants of backpropagation and weight-perturbation training cores.

The final figure in Figure 1 shows the design of the weight-perturbation training core. At the heart of the training core is a prediction core that does continuous inference (II = 1) on the same input training sample. The *Perturbator* block adds perturbations to each parameter one at a time each cycle, which is fed into the prediction core. The size of the perturbation is fixed at compile time. The output from the prediction core is the $E(w_i + \Delta)$ term in Equation 4. The *Compute Gradients* block evaluates Equation 4, and commits the new network parameter to the inactive buffer. Once again, the design is fully-pipelined to support an II of 1 to maximize the training frequency.

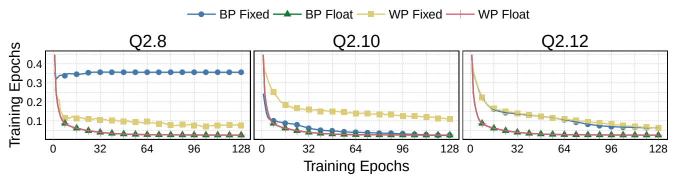
V. Results

All designs are written in VivadoHLS-C++ and synthesized with Vivado 2017.4 targeting the Kintex-7 XC7K410T FPGA. We wrote testbenches to quantify the training performance of each method across all benchmarks. The batch size is fixed to 1 to simulate stochastic gradient descent training. Our design space exploration consists of tuning the following hyper-parameters: (1) fixed-point precision of all weights, biases, inputs/outputs, and activations in the network; (2) learning rate (α); and (3) delta (Δ) by which to perturb each weight/bias (not applicable to backpropagation). To preserve numerical stability, we perform arithmetic operations (*e.g.* accumulations) in double the fixed-point precision used for storing network parameters and activations. For example, a Q2.12 implementation would have intermediate values of arithmetic operations computed in Q4.24, which would then be truncated back to Q2.12 after the computation.

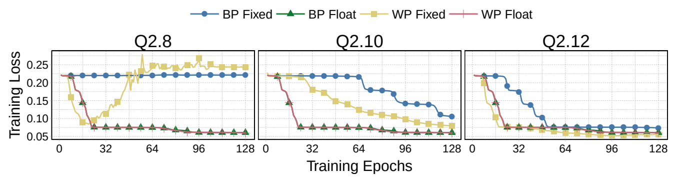
A. Fixed-Point Training

Figure 2 summarizes the fixed-point training capability of backpropagation (BP) and weight-perturbation (WP) methods on both applications.

Iris : The Iris classification network shows varying behaviour with different techniques at different fixed-point precisions. Interestingly, a trained Q2.8 WP implementation delivers 94.7% accuracy on the test set, while the competing Q2.8 BP implementation only achieves 52% accuracy on the test set. At Q2.10, the same comparison sits at 94.7% vs 85.3%, still in favor of WP. On closer inspection, the poor accuracy with BP is due to a 44% misclassification rate on a single class (Versicolour), vs. 10% error rate with



(a) Iris Network



(b) Autoencoder Network

Fig. 2: Training loss vs training epochs for both benchmarks, evaluated at three representative fixed-point precisions, and compared to their respective floating-point implementations.

WP. The classification probability at the output is relatively close (≈ 0.5) for Versicolour and Virginica classes, which is the cause for the high error rate. We hypothesize that the small Δ perturbations capture the dataset subtleties better than the BP gradients, where there is a higher chance of accummulating errors in the backward pass due to limited fixed-point precision.

RF Anomaly Detector : The RF anomaly detector is 8$\times$ bigger than the Iris network, which increases the complexity significantly. In Figure 2b at Q2.8, we observe that BP fails to train, while WP is unstable. The autoencoder requires at least 14b (Q2.12) of precision to match the floating-point training loss. To verify the performance of the anomaly detector, we create a test set in which we inject three different types of noise – bandpass, chirp, and complex sine – at known intervals. We then compute an F_1 score for anomaly detection with each implementation at the end of 128 training epochs. At Q2.12, both fixed-point implementations match the F_1 score (0.57) obtained with floating-point implementations.

B. Activation Function

Unlike floating-point numbers, low-precision fixed-point arithmetic has to manage the tradeoffs between numerical range (*i.e.* integer bits) and numerical precision (*i.e.* fractional bits). Hence, for training low-precision networks, we have to be careful when designing the network architecture, especially when choosing the non-linear activation function. A non-linear function such as $ReLu$ is not suitable for low-precision training, as the activations in the outputs can sum to large positive values at the output of an untrained network, resulting in unstable training due to saturation/overflow. Non-linear functions with closed/bounded intervals on the output are recommended for low-precision on-chip training – examples include $tanh$, $sigmoid$, and $softsign$.

C. FPGA Area and Performance

Table II shows a detailed resource utilization breakdown of the on-chip inference and training hardware of both benchmarks. In both cases, our design goal was to minimize training epoch frequency, followed by overall design size. To achieve these goals, each training core sub-blocks are designed to operate as close to an II of 1 as possible.

For small networks like the Iris network, the target FPGA is large enough such that the complete BP training circuit can be unrolled and spatially placed on the FPGA. This allows us train the network on every new data sample with the proposed architecture for backpropagation. Since weight-perturbation computes gradients of network parameters one at a time every cycle, it is not possible to achieve an II of 1 with the WP training core. Instead, the II is limited by the latency of the sub-blocks (247). Hence, the training frequency for BP is simply the achieved clock frequency of the design (*i.e.* 3.3×10^8 training epochs every second), while the WP achieves a slower training rate of one epoch every 0.6μs (*i.e.* 6×10^7 training epochs every second). This high-throughput design, however, comes at a severe resource cost. For example, the Iris classification network with BP takes up $\approx3\times$ more LUTs and $\approx10\times$ more FFs than the corresponding WP implementation.

As expected, the BP implementation is not able to achieve an II of 1 on the larger autoencoder network. Instead, the latency and II of the BP training core now degrades below the WP implementation at roughly the same resource cost. The training frequency of the WP core on the autoencoder is once every $\approx1.3\mu$s, while the BP training core achieves a rate of once every $\approx2.5\mu$s. For comparison, using the PAPI [10] profiler, we measure the BP training frequency of the autoencoder network on a commercial off-the-shelf Intel i7-6700 CPU to be on average $\approx2.7\mu$s per training epoch (batch size 1). The FPGA WP training core beats the CPU implementation by $\approx2\times$, at a fraction of the power cost.

VI. Conclusions

In this paper, we have demonstrated an effective and novel technique for FPGA implementations of artificial neural networks with on-line learning. Training is accomplished using weight-perturbation, which, compared to backpropagation, has two advantages: (1) it re-uses much of the inference hardware, leading to low overhead and (2) is less sensitive to reduced precision arithmetic. Overall, perturbation-based on-line training is a useful technique that can be performed in parallel (or interleaved) with inference, refining the values of weights as the statistics of the problem change over time.

TABLE II: Resource utilization and performance breakdown of both training methods across both benchmarks (at Q2.10)

Module	LUTs	FFs	BRAMs	DSPs	Latency	II
Iris Classification Network						
Prediction Core	15946	21931	30	302	142	1
Compute Layers	15623	21245	30	296	125	1
Compute Loss	323	597	0	6	15	1
Training Core (WP)	22189	34360	30	308	248	247
Prediction Core	15668	29483	30	296	142	1
Perturbator	3058	2759	0	12	48	1
Compute Gradients	2311	1792	0	0	54	1
Training Core (BP)	63650	333615	30	325	113	1
Compute Gradients	29594	39715	30	177	99	1
Update Params	33923	37529	0	148	13	1
RF Anomaly Detector						
Prediction Core	101817	110114	33	674	165	1
Compute Layers	98797	106458	33	610	139	1
Compute Loss	2992	2820	0	64	24	1
Training Core (WP)	215492	258227	60	1344	171	169
Prediction Core	195985	149284	33	1344	148	1
Perturbator	18011	110273	0	0	17	1
Compute Gradients	1128	1256	0	0	3	1
Training Core (BP)	199113	171057	9	917	721	721
Compute Gradients	74595	91915	9	269	438	256
Update Params	121527	86101	0	648	278	278

Acknowledgement

The authors gratefully acknowledge support from the Defence Science and Technology Group's Next Generation Technology Fund under the "High Speed Machine Learning using FPGAs" project, and the NSERC COHESA Strategic Network.

References

[1] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021745

[2] M. Jabri and B. Flower, "Weight Perturbation: An Optimal Architecture and Learning Technique for Analog VLSI Feedforward and Recurrent Multilayer Networks," *Trans. Neur. Netw.*, vol. 3, no. 1, pp. 154–157, Jan. 1992. [Online]. Available: http://dx.doi.org/10.1109/72.105429

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=104279.104293

[4] R. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, no. 2, pp. 4–22, Apr 1987.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[6] J. L. Holt and T. E. Baker, "Back propagation simulations using limited precision calculations," in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. ii, Jul 1991, pp. 121–126 vol.2.

[7] F. Ortega-Zamorano, J. M. Jerez, D. U. Muñoz, R. M. Luque-Baena, and L. Franco, "Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers," *IEEE Transactions on Neural Networks & Learning Systems*, vol. 27, no. 9, pp. 1840–1850, Sept 2016.

[8] M. Moussa, S. Areibi, and K. Nichols, "On the arithmetic precision for implementing back-propagation networks on FPGAs: A case study," in *FPGA Implementations of Neural Networks*, 2006, pp. 37–61.

[9] D. J. Moss, D. Boland, P. Pourbeik, and P. H. Leong, "Real-time FPGA-based Anomaly Detection for Radio Frequency Signals," in *International Symposium on circuits and systems (ISCAS)*. IEEE, 2018, pp. 1–5.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The international journal of high performance computing applications*, vol. 14, no. 3, pp. 189–204, 2000.