Wireless Modulation Classification Using Deep Learning on FPGAs

Cory Nezin

October 16, 2017

1 Abstract

Wireless modulation classification has been, and continues to be an important engineering problem. Sensing and classifying wireless signals is relevant to applications including government spectrum regulation, cognitive radio, and situational awareness in military/adversarial environments. [1] Deep neural networks have recently achieved impressive performance in classifying audio, images, and video. The application of neural networks to wireless communication has recently grown in the machine learning community. Applications include nonlinear channel modeling, learned data encoding, and modulation classification. [2] While promising results have been achieved, they have only been implemented on graphics processing units (GPUs) which have relatively large size, weight, power, and latency compared to FPGAs. [3] We propose a general framework for converting computational graphs (a more general term than neural network) bulit in TensorFlow [4] into synthesizable VHDL code for implementation on field programmable gate arrays (FPGAs).

In addition to the size, weight, power, and latency advantages offered by FPGA's, they have also drawn attention in deep learning applications for their reconfigurability from large companies like Microsoft. [5] Google has also recently developed specialized hardware for deep learning performance enhancement in the form of the "Tensor Processing Unit" (TPU). The TPU was originally planned to be an FPGA when "[Google] saw that the FPGAs of that time were not competitive in performance compared to the GPUs of that time." [6]

2 Literature Search

2.1 Operational Building Blocks

The multiply accumulate (MAC) operation is perhaps the most important and fundamental operation in signal processing and deep learning. It has the simple form:

$$a \leftarrow a + (b \times c)$$

This operation is fundamental in signal processing because it easily describes feedback. However it is even more important in general because it is the building block of matrix multiplication, which can be used to describe any finite linear transformation. Two of the most computationally intense operations in deep learning for signal analysis are filtering (convolutional layer) and matrix multiplication (fully connected layer). Since both operations are linear, one may think of a convolutional layer as a special case of a fully connected layer but having a special structure which makes its computation faster.

There are two primary architectures for high performance MAC computation: temproal and spatial. Temporal architectures use a central controller to aggregate the data of many arithmetic logic units (ALUs). Spatial architectures allow direct communication between many ALUs in what is called "dataflow processing." [7]

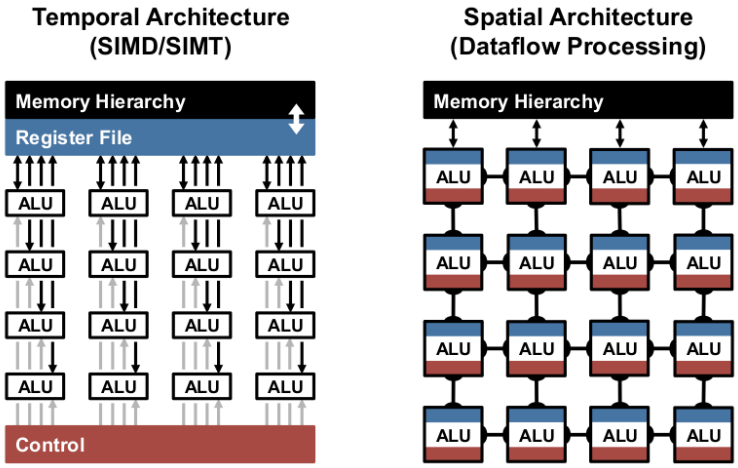

Figure 1: Temporal and Spatial architectures [7]

2.2 Operation Optimization

2.2.1 Theoretical Convolution

A standard (finite) circular convolution can be computed using the "flip and slide" method [FF] with exactly $N_s N_f$ MACs where N_s is the number of samples in the signal and N_f is the number of filter coefficients. However, it may be possible to do better. The well known convolution theorem states that "Convolution in the time (or spatial) domain is Hadamard (elementwise) multiplication in the frequency (temporal or spatial) domain." That is,

$$x[n] \circledast y[n] = z[n] \implies X[k] \odot Y[k] = Z[k]$$

So we may calculate

$$z[n] = \mathcal{F}^{-1}(\mathcal{F}(x[n]) \odot \mathcal{F}(y[n]))$$

It has been show that the lower bound on the number of multiplications for the fast fourier transform (for inputs of length 2^m) is $4N - 2\log_2^2(N) - 2\log_2 N - 4$, which has $O(N)$ complexity. And in fact, there are known algorithms [8] which achieve this lower bound, but they use too many additions to be practical on modern processors. [9] Practical algorithms such as the split-radix FFT achieve $4N\log_2 N - 6N + 8$ real additions and multiplications, which is $O(N\log(N))$ complexity. [10] Additionally, when dealing with purely real data, there are algorithms which are capable of roughly halving the number of operations. [11] Since our goal is to classify modulation using complex I-Q samples, it is unclear whether this optimization will be helpful, though previous work has suggested that complex neural networks offer only marginal imporvement. [12]

It is clear that direct convolution computation has a complexity of $O(N_s N_f)$ while the Fourier transform method has a complexity of $O(N_s \log N_s)$, assuming that $N_f \leq N_s$ and we don't take advantage of the relative sparsity of the filter. It is not immediately clear which of these is more practical. If $N_s \to \infty$ as N_f remains constant, the direct method is better. If $N_s \to \infty$ and $N_f \to \infty$, the Fourier transform method is better. We are also not accounting for the differing complexities and the fact that specialized hardware may exist for either. Therefore experiment and deeper analysis will be necessary to determine which method is more suited to our application.

2.2.2 Practical Convolution

Many popular algorithms for fast convolution or fast Fourier Transformation run very well on processors and GPUs, but are not necessarily optimal for usage on FPGA or for real time calculation. MIT Lincoln Laboratory has developed a systolic (spatial) FFT architecture which is designed specifically for real time operation on FPGAs and ASICs. [13]

One major practical consideration is that during inference of neural nets, the filter coefficients do not change, and the Fourier Transform of the input signal need only be generated once, no mater how many filters are used. When performing convolution directly in the time domain, there are several strategies for minimizing memory access (which as it turns out, is the primary bottle neck). [14] The primary strategies are similar to caching - the input signal or filter coefficients are stored locally (in static RAM) and reused in computations. The downside to this is of course increased hardware for storing values statically, and the increased complexity introduced. A descriptive diagram is shown in Figure 2.2.2

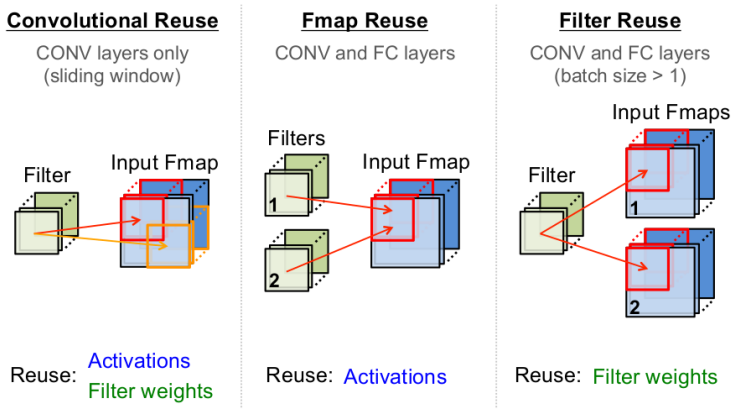

Figure 2: Types of reuse in time domain convolution (note that activations are just the signals to be filtered) [7]

2.2.3 Theoretical Matrix Multiplication

Fast matrix multiplication is very important in many numerical computation problems, and thus is well studied. Two well known methods for fast matrix multiplication are Strassen's algorithm and the Coppersmith-Winograd algorithm which have complexities $O(n^{2.807})$ and $O(n^{2.376})$ respectively. Unfortunately, the Coppersmith-Winograd algorithm is "wildly im-

practical for any conceivable applications." [15]. Strassen's algorithm encounters similar drawbacks, it is associated with "reduced numerical stability, increased storage requirements, and specialized processing". [7]

2.2.4 Practical Matrix Multiplication

In practice on GPUs, no single matrix multiplication algorithm is used for every case. Different algorithms are used depending on the shape and size and shape of the matrix. [14]. In a fully connected layer, it is almost always the case that the signal, whether it be audio, image, or video, is "unrolled" into a 1-D vector which is then multiplied on the left by a matrix. Thus, in designing specialized hardware for a neural network, it may not necessarily be important that multiplication of two matrices is fast, but that multiplying a matrix and a vector is fast. In this case there is no obvious opportunity for weight reuse since every weight in the matrix is used exactly once per vector multiplication. However, one may think of a matrix multiplying a vector as a series of inner products between the vector and a given row of the matrix. In this case, replicating the entire vector once for each row of the matrix may be desireable - though this comes with an $O(N_s N_r)$ (where N_r is the number of rows in the matrix) static memory complexity which may be prohibitively large in some cases.

2.3 Neural Net Optimization

Deep neural networks were not designed with hardware efficiency in mind. In order to tackle the difficult problem of both fitting a neural net on an FPGA and having it run fast, we should look at both optimization of the hardware itself and optimization of the neural net structure. Existing approaches to optimizing neural networks for hardware fall into one of two categories: precision reduction and operation reduction. [7] Precision reduction consists of reducing the number of bits used in computation, using fixed point arithmetic (GPUs use floating point), non-linear quantization, weight sharing, and Huffman coding. [16] Recent research has even taken this to extreme, constraining all weights to ± 1. [17] Since custom hardware was not being seriously considered for neural networks until relatively recently, many of the algorithms for precision reduction are recently discovered. However, operation

reduction is capable of significantly reducing size and increasing speed even on GPUs and CPUs and therefore many powerfull "pruning" algorithms already exist such Optimal Brain Damage (OBD) [18] and Optimal Brain Surgery. [19]

2.3.1 Quantization

Some FPGAs have specialized hardware for fast multiplication. In the case of Xilinx FPGAs, this fast multiplication hardware is only available for fixed point numbers, and a floating point implementation of a MAC would require large amount of hardware and would run slower. Thus at a minimum requirement for functionality, the neural nets must be quantized to fixed point, or linear precision. Obviously any amount of quantization introduces error, as an interesting aside, the signal to quantization error is given by roughly $6dB$ per bit which can be derived from assuming that quantization error is uniformly distributed. While it is fairly clear what this figure is in terms of quantizing the signal, it is not clear what exactly this implies for the quantization of filter coefficients. When optimizing our structure, we are not limited to fixed schemes for weight quantization. In particular, we know the exact weights we are using on every iteration and we can take advantage of that. This has led to research being primarily focused on optimal weight quantization rather than activation (signal) quantization. [7]

Before talking about advanced quantization techniques we give a brief review of typical number representations. Fixed point representation is a class of number representations where the precision does not change. That is, the number of digits before and after the binary point are constant. These number are often represented in digital logic in the Q format (Q is used since, roughly speaking, fixed point is to floating point as $\mathcal{Q}$ is to $\mathcal{R}$). There are two parameters to a Q representation, the number of digits before (n) and after (m) the binary point. Precisely, the binary sequence $\{b_i\}_{i=0}^{n+m-1}$ has the associated value

$$B = -b_0 \times 2^{n-1} + \sum_{i=1}^{n+m-1} b_i \times 2^{n-i-1}$$

for example, assuming a two's complement representation, $Q1.7$ would have a maximum value of $\frac{127}{128}$, a minimum value of -1, and a precision of $\frac{1}{128}$. The sequence 10010011 would be associated with the value $-1 + \frac{1}{8} + \frac{1}{64} + \frac{1}{128} = -\frac{109}{128}$ An IEEE standard 32-bit floating

point number is represented by

$$(-1)^s \times m \times 2^{(e-127)}$$

where s is the sign bit, m is the mantissa represented in fixed point, and e is the exponent, also represented in fixed point. Besides speed, a fixed point representation also offers imporvement in area and power. An 8-bit fixed point multiply consumes 15.5× less energy and 12.4× less area than a 32-bit fixed point multiply. An 8-bit fixed point add consumes 30× less energy and 116× less area than a 32-bit floating point add. [20] Recent specialized deep learning platforms such as Google's TPU [6] and Nvidia's PASCAL [21] have used 8-bit arithmetic.

Now we turn to non-linear quantization. Since weights tend to be distributed non-uniformly (like a Gaussian or Laplacian distribution), it makes sense that the quantization should not be uniform in order to capture the maximum information. One simple method is to use logarithmic quantization so that precision is more fine grained around zero where weights tend to cluster. Some recent methods use weight sharing which means that some filter coefficients or weights in a fully connected layer are forced to be the same. Obviously this has the advantage of reducing memory usage, but also leads to an increase in speed since the number of memory accesses is reduced. In order to increase the effectiveness of this procedure, one may encourage clustering during training so that we end up with many sharable values. [16]

2.3.2 Operation Reduction

In the naive implementation of most neural networks, there are many redundant operations. For instance a primary component of modern deep neural networks called the "Rectified Linear Unit" (ReLU) sets values less than 0 equal to 0. This means that operations following will needlessly multiply by 0 repeatedly - if the corresponding multiply accumulates are skipped, throughput can be increased. The zeros could also be more efficiently stored with a simple run length encoding to decrease memory usage. [7]

Regarding the pruning of neural networks, OBD and OBS are too difficult to apply to extremely deep networks because they are computationally expensive, and the cost scales

with the number of weights. A simpler method whereby weights with small values are eliminated was proposed in [22] and proved to be very effective, reducing the size of deep neural nets by up to 80%. There are also pruning methods which, instead of minimizing number of weights, attempt to minimize the energy usage of the neural net which is a strong function of DRAM access. [23] Note that all of these pruning methods introduce complications, storing arrays of values is simple because that is the natural structure of RAM and is easily indexed by two integers. When the weights are sparse, we must come up with a scheme for efficiently storing only the nonzero values. To combat this, structured pruning can be used which prunes entire groups of weights, resulting in a highly structured sparse matrix which is easy to work with. [24]

Instead of pruning after the fact, it is also possible to train neural networks in a constrained manner. For instance, cascading two 3x3 filters can create a 5x5 filter. Two 3x3 filters require 18 weights while a 5x5 would require 25. Note that the same benefit cannot be drawn from cascading two 1-Dimensional filters, however this method is conceptually similar to a singular value decomposition wherein a matrix multiply can be approximated as the sum of lower rank matrices which do not require as many weights together.

References

[1] S. Rajendran, R. Calvo-Palomino, M. Fuchs, B. V. den Bergh, H. Cordobés, D. Giustiniano, S. Pollin, and V. Lenders, "Electrosense: Open and big spectrum data," *CoRR*, vol. abs/1703.09989, 2017. [Online]. Available: http://arxiv.org/abs/1703.09989

[2] T. J. O'Shea and J. Hoydis, "An introduction to machine learning communications systems," *CoRR*, vol. abs/1702.00832, 2017. [Online]. Available: http://arxiv.org/abs/1702.00832

[3] N. et al., "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021740

[4] M. A. et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[5] P. et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678

[6] N. P. J. et al., "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[7] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017. [Online]. Available: http://arxiv.org/abs/1703.09039

[8] S. Winograd, "On computing the discrete fourier transform," *Math. Computation*, vol. 32, no. 1, pp. 175–199, Jan. 1978.

[9] P. Duhamel and M. Vetterli, "Fast fourier transforms: A tutorial review and a state of the art," *Signal Process.*, vol. 19, no. 4, pp. 259–299, Apr. 1990. [Online]. Available: http://dx.doi.org/10.1016/0165-1684(90)90158-U

[10] R. Yavne, "An economical method for calculating the discrete fourier transform," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 115–125. [Online]. Available: http://doi.acm.org/10.1145/1476589.1476610

[11] G. D. Bergland, "Numerical analysis: A fast fourier transform algorithm for real-valued series," *Commun. ACM*, vol. 11, no. 10, pp. 703–710, Oct. 1968. [Online]. Available: http://doi.acm.org/10.1145/364096.364118

[12] C. Trabelsi, O. Bilaniuk, D. Serdyuk, S. Subramanian, J. F. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. J. Pal, "Deep complex networks," *CoRR*, vol. abs/1705.09792, 2017. [Online]. Available: http://arxiv.org/abs/1705.09792

[13] P. Jackson, C. Chan, C. Rader, J. Scalera, and M. Vai, in *A Systollic FFT Architecture for Real Time FPGA Systems*, Sep. 2004.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[15] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/28395.28396

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[17] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[18] Y. L. Cun, J. S. Denker, and S. A. Solla, "Advances in neural information processing systems 2," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Optimal Brain Damage, pp. 598–605. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.109298

[19] B. Hassibi, D. G. Stork, G. Wolff, and T. Watanabe, "Optimal brain surgeon: Extensions and performance comparisons," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, ser. NIPS'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 263–270. [Online]. Available: http://dl.acm.org/citation.cfm?id=2987189.2987223

[20] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," *ISSCC*, vol. 57, pp. 10–14, 02 2014.

[21] T. P. Morgan, "Nvidia pushes deep learning inference with new pascal gpus," Sep. 2016.

[22] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015. [Online]. Available: http://arxiv.org/abs/1506.02626

[23] T. Yang, Y. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," *CoRR*, vol. abs/1611.05128, 2016. [Online]. Available: http://arxiv.org/abs/1611.05128

[24] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080215