Accuracy, Training Time and Hardware Efficiency Trade-Offs for Quantized Neural Networks on FPGAs*

Pascal Bacchus[1], Robert Stewart[1], and Ekaterina Komendantskaya[1]

Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK
{R.Stewart,E.Komendantskaya}@hw.ac.uk

Abstract. Neural networks have proven a successful AI approach in many application areas. Some neural network deployments require low inference latency and lower power requirements to be useful e.g. autonomous vehicles and smart drones. Whilst FPGAs meet these requirements, hardware needs of neural networks to execute often exceed FPGA resources.
Emerging industry led frameworks aim to solve this problem by compressing the topology and precision of neural networks, eliminating computations that require memory for execution. Compressing neural networks inevitably comes at the cost of reduced inference accuracy.
This paper uses Xilinx's FINN framework to systematically evaluate the trade-off between precision, inference accuracy, training time and hardware resources of 64 quantized neural networks that perform MNIST character recognition.
We identify sweet spots around 3 bit precision in the quantization design space after training with 40 epochs, minimising both hardware resources and accuracy loss. With enough training, using 2 bit weights achieves almost the same inference accuracy as 3-8 bit weights.

Keywords: Deep learning · Neural networks · Quantization · FPGA.

1 Introduction

Neural networks have proved successful for many domains including image recognition, autonomous systems and language processing. GPUs are often used to train and test neural networks, since GPUs offer highest peak performance compared with CPUs and FPGAs. Due to their specialised support for floating-point arithmetic operations, GPUs can deliver the highest arithmetic performance for 32 bit floating point neural network inference. However, the use of GPUs operating at 200+ Watts is becoming prohibitively expensive for energy use, e.g. the carbon footprint of state-of-the-art AI algorithms performed by GPUs is about five times the lifetime emissions of an average car [16].

Compared with GPUs and CPUs, peak performance of FPGAs becomes competitive for fixed-point representations [24], which are used in quantized neural

* Supported by EPSRC grant EP/N028201/1.

networks. Special purpose FPGA based accelerators significantly increase inference speed which is useful in domains like stock market trading and autonomous vehicles, and reduce energy requirements useful for remote computer vision on smart sensors and drones where access to power is scarce.

Models trained on a GPU have a memory footprint that is too large for FPGAs. This prohibits the use of FPGA accelerators for executing full precision neural networks because they have limited hardware resources for storage and computation. Worse still, neural networks for these domains have grown from single hidden layer topologies with several hundred weights, to deep models with more than one hundred hidden layers with millions of weights, meaning their memory resource requirements is increasing with state-of-the-art models. Quantized fixed point representation can significantly reduce power and resource costs, e.g. [4] reports $\times 0.164$ area and $\times 0.136$ power costs with a 16-bit multiplier compared with a 32-bit multiplier for neural networks.

Neural network performance is often measured by its inference accuracy on unseen inputs. Other performance metrics become important when resource constrained accelerators are to be used:

- **Throughput** How many achievable inferences per unit of time.
- **Latency** The time to infer.
- **Energy** The energy consumed by the processor.
- **Compression ratio** A ratio between an original model size before and after compression algorithms are applied.
- **Training time** The time needed to obtain a neural network with acceptable inference accuracy.
- **Robustness** The robustness of the network when confronted with adversarial perturbated inputs.

Two motivations for compressing neural networks is speed and energy efficiency:

1. **Speed** Many neural network layers are bandwidth bound. This introduces latency that dominates execution time because most time is spent to bring data to processors rather than performing computation. Effective pruning algorithms remove weights and layers entirely without adversly affecting accuracy, reducing memory access and thus reducing latency.
2. **Energy efficiency** It costs orders-of-magnitude more energy to access off-chip DDR memory compared to on-chip memory e.g. SRAM, BRAM and cache memory. Fitting weights into on-chip memories reduces frequency of energy inefficient off-chip memory accesses.

Neural network compression methods reduce both the memory size and computation time. The result of compression is neural networks with smaller spacial complexity, and low precision arithmetic operations that are cheaper to compute.

Several industry led neural network compression tools have recently emerged. Google's Tensorflow Lite is a framework that uses quantization-aware training [1] to quantize full precision models into 8 bit versions. Intel's Distiller [29] is a

Python framework for compressing neural networks specified in PyTorch, with support for pruning and quantization.

There are several neural network frameworks for FPGAs. FPGAConvNet [23] supports convolutional neural netwoks (CNNs) but does not support quantization. Caffeinated FPGA [7] is an extension of the Caffe framework for compiling convolutional neural networks (CNNs) to FPGAs and supports binarization but not quantization. ReBNet [10] is a more general framework, supporting neural network types other than just CNNs, but is restricted to binarized networks and is not actively maintained.

Xilinx's FINN is another general framework for neural networks of various types to FPGAs. FINN initially supported binarized neural networks [22], then was extended for quantized networks [3] and Long-Short Term Memory Neural Networks (LSTM) [20].

Neural network throughput improvements can be achieved by reducing FPGA resource use e.g. [28]. Our paper relies on this assumption about throughput speedups, and instead focuses on the trade-off between accuracy and FPGA resource requirements. Rather than pruning, it explores the design space granted by FINN's ability to independently quantize weights and activation functions of multilayer perceptron (MLP) neural networks, with arbitrary bit-width values between 1 and 8.

The most closely related work to our systematic evaluation in Section 3 is [21], which evaluates the trade-offs between accuracy, throughput and hardware efficiency for 1, 2, 4, 8, 16 bit quantized and 32 bit full precision neural networks with FINN. Our paper extends this work in two ways. 1) Measuring impact of *training time* on the accuracy of quantized neural networks with 10, 20, 30, 50 and 100 epochs. 2) A more fine grained evaluation for 1..8 bit precision varying activation function precision and weight precision independently of each other, i.e. measuring training time, accuracy and hardware efficiency trade-offs across 64 quantized neural networks.

Contributions The paper makes the following contributions:

- Evaluation of the inference accuracy for 64 quantized neural networks with 1-8 bit weights and activation function precision (Section 3.3).
- Evaluation of the effect that increased training time has on the accuracy of these quantized neural networks (Section 3.4).
- Evaluating the cost of LUTs, FFs and BRAMs for these quantized neural networks (Section 3.4).
- A relative performance comparison across diferent quantized neural networks showing the trade-off between required hardware resources and accuracy (Section 3.5).

2 Background: FINN

2.1 FINN Workflow

This section describes the workflow of the FINN framework. Our experiments in Section 3 assess very low precision neural networks, i.e. reducing precision from 32 bits to 1-8 bits to fit within the resource constraints of FPGAs. As such, quantization aware training is used.

Each experiment uses the following process:

1. **Prepare** Import training and testing data as numpy arrays and the Theano [2] model described in Section 3.1.
2. **Train** Set the precision for both parameters, chose the training time and training parameters. Once training is complete, we obtain a list of numpy arrays that correspond to the quantized trained weights. At this point we obtain the accuracy performance.
3. **Hardware generation** Numpy arrays are converted from floating point values into binary values and packed into binary files, and FINN compiles the quantized neural network to synthesisable C++.
4. **Synthesize hardware** Use HLS synthesis to obtain an estimation of the resources required.

2.2 Quantization

Quantization Quantization [12] shifts values from 32 bit floating point continuous values to reduced bit discrete values. In a neural network, weights between neurons and activiation functions can be quantized.

Binarization Binarization [6] is a special case of quantization that represents weights and/or activation function outputs with a single bit. These methods replace arithmetic operation with bit-wise operations, reducing the energy consumption and memory requirements.

Quantized neural networks can signifiantly outperform binarized neural networks and can compete with the accuracy of full precision models [12].

Another neural network compression approach is weight reduction. Weight reduction keeps a high bit precision for preserved weights while removing unimportant parameters in the network, examples are *pruning* [11] and *weight sharing* [5]. Pruning keeps only the most useful connections between nodes. Weight sharing packs groups of weights together given they have similar values. A complete study of approximating neural network approaches is in [25].

2.3 Quantization for Training in FINN

FINN trains a neural network at the Python level with Theano, before generating synthesisable C++ for hardware. The weights and activation functions during training in Python operate on floating point values but Python functions simulate quantization to limit weights and activation function outputs to

discrete values permitted by the chosen quantization configuration. When generating hardware, the arithmetic precision of weights and activation functions in the C++ match the quantized bit widths simulated during training.

Weight Binarization for Training Binarization transforms every weight into either a 1 or -1 value. Binarization is shown in Equation 1, which corresponds to the FINN implementation in Figure 1. The output of the approximate sigmoid function `hard_sigmoid` is rounded to the nearest integer to shift the range of values before binarization. This function is monotonic in the interval [0;1], thus preserving the order of its domain. The value is rounded to either 0 or 1 and then it set to either -1 or 1.

$$BinariseWeight(x) = \begin{cases} 1 & \text{if } \|(hard\_sigmoid(x))\| > 0.5 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

```
1  class QuantizationBinary(object):
2      def __init__(self, scale=1.0):
3          self.scale = scale
4          self.min = -scale
5          self.max = scale
6
7      def quantizeWeights(self, X):
8          Xa = hard_sigmoid(X / self.scale)
9          Xb = Theano.round(Xa)
10         return Theano.switch(Xb,self.scale,-self.scale)
```

The choice is either -1 or 1 for binarization

The range is converted to [0;1] and then to -1 or 1

Fig. 1: Weight Binarization in FINN for Training

Weight Quantization for Training FINN discretises the range of full precision values by rounding to a close neighbour using fixed point quantization. The *min* and *max* values for the quantization range are related to the quantization precision n, and they are defined by:

$$max = 2 - \frac{1}{2^{n-2}} \qquad\qquad min = -2 + \frac{1}{2^{n-2}}$$

The quantization formula for $x \in [min; max]$ is shown in Equation 2.

$$QuantiseWeights(x) = \frac{\lfloor x2^n + 2^{n-1} - 1 \rfloor}{2^{n-2}} - 2 + \frac{1}{2^{n-2}} \tag{2}$$

Table 1 shows examples of quantized values with $min = -2$ and $max = 2$ with $2^n - 1$ values in this interval. The values are all strictly positive but the

Value	Precision (bits)							
	1	2	3	4	5	6	7	8
0.136	1	0	0	0.25	0.125	0.125	0.125	0.140625
0.357	1	0	0.5	0.25	0.375	0.375	0.34375	0.359375
0.639	1	1	0.5	0.75	0.625	0.625	0.625	0.640625
1.135	1	1	1	1.25	1.125	1.125	1.125	1.140625
2	1	1	1.5	1.75	1.875	1.9375	1.96875	1.984375

Table 1: Example of quantized weights

quantization range is symmetric. The step between each quantized value is $\frac{1}{2^{n-2}}$. When n increases, the number of quantized values increase and we can obtain values close to the upper and lower bound of the interval.

Activation Function Quantization for Training The quantization of activation functions works similarly to weight quantization. Figure 2 shows the Python used to simluate a binarized *tanh* function. The *round3* function used on line 5 is from [6], which approximates standard activation functions for deep learning algorithms. All quantized functions are strictly increasing and differentiable.

```
1  def hard_sigmoid(x):
2      return Theano.clip((x+1.)/2.,0,1)
3
4  def binary_tanh_unit(x):
5      return 2.*round3(hard_sigmoid(x))-1
```

returns the result of the binarized tanh function

Fig. 2: Activation Function Binarization in FINN for Training

For the quantized hyperbolic tangent function $tanh(x) = \frac{e^x - e^{-x}}{e^x - + e^{-x}}$, the range of values in Table 1 is optimal because it has two asymptotes that goes towards -1 and 1, e.g. $tanh(2) = 0.964$. The saturation plateau of the activation function is almost attained. Figure 3 shows the shape of *tanh* for different quantization precisions.

3 Evaluation

For 64 neural network quantization weight and activiation function configurations, the evaluation in this section measures:

1. *Absolute* accuracy and hardware resource costs of the 64 quantized neural networks (Sections 3.3 and 3.4).
2. *Relative* performance comparison of accuracy and hardware resource costs of these neural networks (Section 3.5).

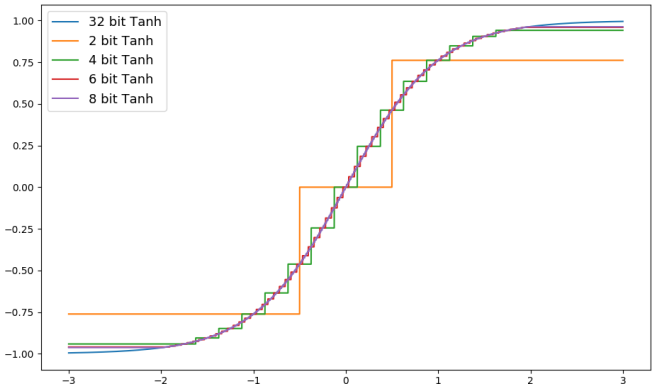


Fig. 3: Hyperbolic tangent with different quantization configuration

3.1 Neural Network Topology

Figure 4 shows the multilayer perceptron model used in the experiments. It is presented as Python code with Theano and Lasagne [8] libraries, to illustrate how quantized neural networks are constructed programmatically as input to the FINN framework. The input layer consists of 784 neurons that represent the 784 pixels from MNIST dataset [14] (28×28 grayscale images). The output layer has 10 neurons, one for each of the 10 possible classifications for recognised digits.

Between the input and output layers are three fully connected hidden layers that have all 1024 neurons, each using the same activation function, the hyperbolic tangent. The weight quantization is specified on line 10 and the activation function on line 16. The activation function quantization is defined in its constructor, elsewhere. Dropout layers (lines 6 and 22) are only used for training as a means of regularisation, and are not included in the generated hardware. This layer randomly removes some connections between two layers at each training batch to avoid overfitting and obtain a model that can generalize on new data.

Batch normalisation is used on the trained model, in the hidden layers (line 18) and as the last layer (line 31). Trained parameters from the `BatchNormaLayer` layers are included in synthesisable C++ for deployment to an FPGA. Batch normalisation is a stochastic operation that generally improves the speed and performance of a network. Lasagne's normalization is based on the following:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} * \gamma + \beta$$

where μ is the current batch input x, σ is the variance of the current batch input x, ϵ is a small variable to avoid numerical discontinuty, γ is the average statistic computed during training time and β is the average statistic computed during training time. On line 31, *alpha* is the exponential moving average factor which is calculated during training time but also initiated by the user. These values are used during testing, and are also included as constants in the C++ to deploy the quantized neural network to an FPGA.

```
1  mlp = InputLayer(shape=(None, 1, 28, 28),
2          input_var=input)
3
4
5
6  mlp = DropoutLayer(mlp,p=dropout_in)
7
8  for k in range(n_hidden_layers):
9      mlp = qn.DenseLayer(mlp,
10         quantization=weight_quant,
11         W_LR_scale=W_LR_scale,
12         nonlinearity=identity
13         num_units=1024)
14
15     mlp = NonlinearityLayer(mlp,
16     nonlinearity=activation_function)
17
18     mlp = BatchNormLayer(mlp,
19         epsilon=epsilon,
20         alpha=alpha)
21
22     mlp = DropoutLayer(mlp,
23             p=dropout_hidden)
24
25 mlp = qn.DenseLayer(mlp,
26         quantization=weight_quant,
27         W_LR_scale,
28         nonlinearity=identity,
29         num_units=num_outputs)
30
31 mlp = BatchNormLayer(mlp,epsilon,alpha)
```

Input size fits a 28*28 pixels image

Reduce overfitting during training with dropout regularization

Enhanced DenseLayer function from Lasagne that uses quantized weight

Quantized activation function a custom hyperbolic tangent

Improves speed and stability with normalized inputs

Regularization of the hidden layers

Output layer

Fig. 4: Quantized Neural Network Model Expressed in Python

3.2 Measurements Platform

The training is done using 50000 images from the MNIST dataset. A validation dataset of 10000 images is then used to minimise overfitting. Finally, accuracy is measured over a testing dataset. FINN's backend converts the model (specifically, numpy arrays) to a binary weight file and a synthesisable C++ implementation for hardware.

Our experiments target the mid-range Xilinx Zynq Z7020 device with 53k LUTs, 106k FFs and 4.9Mb BRAM. Of the 64 quantized neural networks, only four actually fit on this FPGA, validating the need for aggressive compression approaches such as quantization, on relatively small FPGA devices.

The software and library versions used in the experiments are Vivado 2018.3, Python 2.7.15 with Numpy 1.15, Scipy 0.19.1, Theano 0.9.0, and Lasagne 0.2.

3.3 Absolute Accuracy Performance

Each of the 64 neural networks is labelled with a quantized weight W-X and quantized activation function A-Y with $X, Y \in [1; 8]$. Accuracy is measured after 10, 20, 30, 50 and 100 epochs.

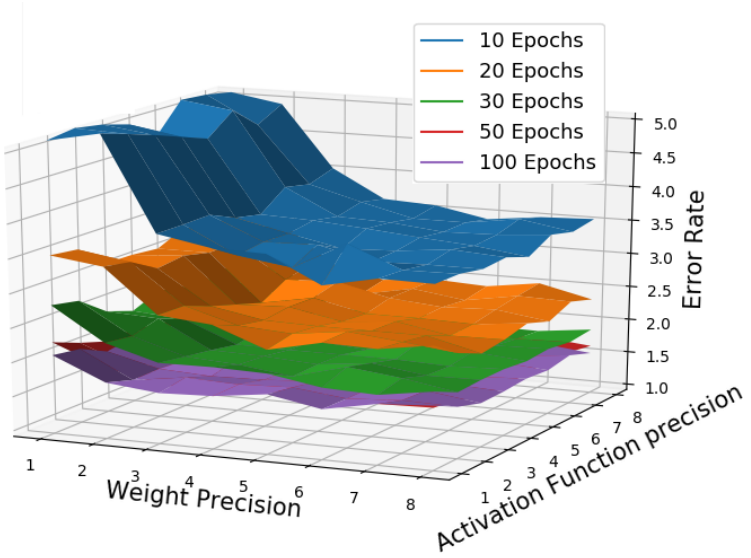


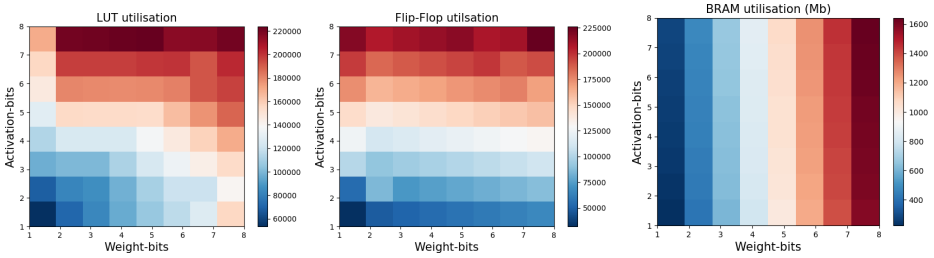
Fig. 5: Accuracy of QNNs with Increasing Training

Figure 5 plots the inference error rate for each of the 64 quantized neural networks after training with 10, 20, 30, 50 and 100 epochs. Using 1-3 bits weights has a noticeable effect on accuracy, i.e. between 3.9%-4.7% dropping down to below 3.7% using 4 bits or more. Training further with 40-100 epochs shifts the noticeable accuracy boundary to just 1 bit weight, meaning that with enough training, 2 bit weights achieves almost the same inference accuracy as 3-8 bit weights. The quantization of activation functions has a steady impact on accuracy, i.e. higher precision activation functions result in better accuracy, however, this is not as dramatic as the impact that quantized weight precision has on accuracy. With increased training time, the accuracy performance flattens, where absolute difference in accuracy between the best and worst quantization configuration greatly diminishes. Also, we observe a major gap between 1 and 2 bit weights versus 3-8 bit weights, especially for 10 and 20 epochs. Training beyond 40 epochs allows weights to be quantized from 3 to 2 bits without noticeable accuracy loss.

3.4 Absolute Resource Utilisation Performance

This section evaluates the trade-off between quantized precision and hardware resource use. The X axis is the number of bits for weights, the Y axis is the number of bits for the activation functions. The colour in the heat maps represents the relative measurement of the respective performance metric compared to the other 63 models.

Figure 6a shows that both weight precision and activation function precision contribute evenly to LUTs costs. Figure 6b shows that the precision of activation



(a) LUTs (b) FFs (c) BRAMs

Fig. 6: Hardware Resources Required for 64 Quantized Neural Networks

functions determines FF costs. While FFs and LUTs can store small amounts of data, BRAMs have greater storage capacity and are used by hardware synthesis tools for larger data structures such as arrays. Figure 6c shows that BRAM consumption is determined exclusively by weight precision.

3.5 Relative Quantization Performance

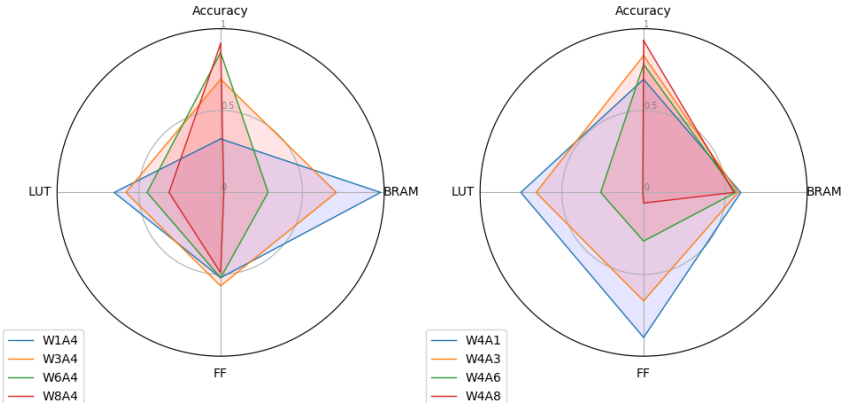
Sections 3.3 and 3.4 present *absolute* performance values. This section evaluates the *relative* performance trade-offs between inference accuracy, BRAM, FFs and LUTs by comparing selected quantized configurations from the 64 networks.

Metric	Relative performance worst	best
Classification error rate	2.07%	1.52%
BRAM	1643	224
Flip Flops	226282	31954
Look Up Tables	223910	53336

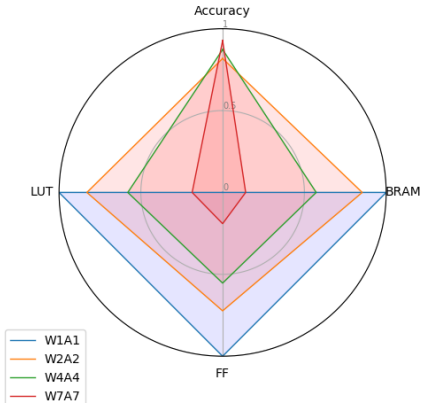
Table 2: Relative Performance for Radar Plots in Figure 7

Table 2 gives the best and worst relative performance numbers for the 64 quantized neural networks. The three radar plots in Figure 7 represents different quantized neural network configurations, comparing accuracy and resource use (LUTs, FFs and BRAMs) performance relative to the best and values in Table 2. Each metric defines one branch in a radar chart. The three precision variations in Figure 7 are:

1. Weight oriented distribution (Figure 7a) increases the weight precision and keeps the activation function constant at 4 bits, i.e. W1-A4, W3-A4, W6-A4 and W8-A4.



(a) Weight oriented distribution (b) Activation oriented distribution



(c) Linear distribution

Fig. 7: Radar charts for different quantization configurations

2. Activation oriented distribution (Figure 7b) increases the activation function precision and keeps the weight precision constant at 4 bits, i.e. W4-A1, W4-A3, W4-A6 and W4-A8.
3. Linear distribution (Figure 7c) increases both the weight and activation function precision across the diagonal from the heat maps in Figure 6, i.e. W1-A1, W2-A2, W4-A4 and W7-A7.

The radar plots compare the relative performance of these quantization configurations. Their scores are normalised between scores of 0 and 1. The model with the highest accuracy is plotted outermost in the radar plot in the *Accuracy* dimension whereas the models with lowest accuracy is plotted at the centre point. Likewise, the neural network using the fewest BRAMs is plotted outermost for the BRAM dimension, and the same for LUTs and FFs.

When activation functions are set to 3 bits, increasing weights from W1 to W3 causes the greatest relative accuracy score improvement (Figure 7a). When weights are fixed at 4 bits, all accuracy scores are in the top half, with increases of activation function precision costing significantly more LUT and FF resources, with BRAM costs largely the same (Figure 7b). Scaling both precision linearly has an equal impact on FF, LUT and BRAM scores, yet their accuracy score are all in the top quartile when weights are 2-8 bits (Figure 7c). In summary, if top-half relative accuracy performance is the goal, the most important constraint is 2+ bits for representing weights.

The importance of the trade-offs is highlighted by the fact that most of the neural networks do not fit on the target device (Xilinx Zynq Z7020). It has 280 BRAMs and only 7 of the networks meet this constraint, and 106400 FFs with 22 of the networks within this constraint.

3.6 Discussion

These experiments confirm observations in previous work [20] that beyond 3 bits there is no significant improvement to accuracy performance given sufficient training, but does have the undesirable effect of increasing the required hardware resources. The sweet spot in the quantization design space for the purpose of MNIST character recognition is about 3 bit weights and 3 bit activation functions. These experiments show:

- The amount of LUT and FF resources is mostly affected by activation functions.
- BRAM memory is determined by weight precision.
- Accuracy is highest with higher precision, i.e. least aggressive quantization. The biggest improvement step in accuracy is switching from 1 to 2 bits weight precision.
- With enough training beyond 50 epochs, 2 bit weights achieves almost the same inference accuracy as 3-8 bit weights.

Our experiment use the quantization scheme implemented in Xilinx's FINN framework. Developing compression algorithms for embedded devices is a research area of its own, e.g. a dynamic precision data quantization algorithm in [18], performed layer-by-layer from a corresponding floating point CNN, with the goal of improving bandwidth and resource utilisation. Other compression approaches are focused on specific goals e.g. reducing power consumption, or target specific hardware e.g. GPUs or FPGAs, or target specific domains or even specific application algorithms.

Target Specific Recent work explores the performance trade-offs between reduced precision of neural networks and their speed on GPUs, e.g. performance aware pruning can lead to $\times 3-10$ speedups [19]. Multi-precision FPGA hardware for neural networks significantly reduces model sizes, which in [28] enables an ImageNet network to fit entirely on-chip for the first time, significantly speeding up throughput. Another recent study [20] measures the hardware cost, power

consumption, and throughput for a High Level Synthesis extension of FINN that supports Long Short-Term memory (LSTM) models on FPGAs. [26] proposes a design flow for constructing low precision, low powered FPGA-based neural networks with a hybrid quantization scheme. [15] shows that resource-aware model analysis, data quantization and efficient use of hardware techniques can be combined to jointly map binarized neural networks to FPGAs with dramatically reduced resource requirements whilst maintaining acceptable accuracy.

Domain Specific The FPGA-based processor architecture in [27] achieves a clock frequency of 100MHz and supports acceleration of quantized CNNs for image processing. Refined still further, some quantization methods target specific algorithms, e.g. a resource-aware weight quantization framework for performing object detection on FPGAs [9]. Similarly, [17] shows that 3-bit weight quantization is required to fit an MNIST character recognition network entirely on-chip for the Xilinx XC7Z045 device, keeping power consumption to less than 5 Watts.

4 Conclusion

This paper evaluates the trade-off between inference accuracy, quantized precision, hardware resources and training time of a neural network performing character recognition. It identifies sweet spots around 3 bit precision in the quantization design space after training with 40 epochs, to minimise both hardware resources and accuracy loss.

Whilst this paper exhaustively measures resource use and accuracy design space between 1 and 8 bits for MNIST character recognition, to assess the reproducibility of these results at scale, the experiments should be repeated: i) on state-of-the-art networks comprising tens/hundreds of deep hidden layers, ii) on hardware-friendly activation functions such as ReLU and its variants, iii) with inference latency as an additional trade-off, and iv) on networks with tens/hundreds of output classes.

These results are timely, with technology for compressing neural networks evolving rapidly. FINN recently added support for PyTorch models and uses a new framework called Brevitas[1] for quantization-aware training. Intel's OpenVino toolkit [13] supports neural network quantization for computer vision applications, and targets CPUs, GPUs and FPGAs. Future work will compare this paper's results with the same experimental setup using Distiller, OpenVino and Brevitas across multiple neural network models on resource constrained FPGAs and embedded GPUs. We also plan to evaluate performance trade-offs with multi-precision neural networks, and explore how multi-precision quantization (e.g. [28]) can maximise compression whilst minimising accuracy loss and preserving robustness of neural networks to adversarial attacks.

[1] `https://xilinx.github.io/brevitas/`

References

1. Abadi, M., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), `https://www.tensorflow.org/lite`, software available from tensorflow.org
2. Al-Rfou, R., et al.: Theano: A python framework for fast computation of mathematical expressions. CoRR **abs/1605.02688** (2016), `http://arxiv.org/abs/1605.02688`
3. Blott, M., Preußer, T.B., Fraser, N.J., Gambardella, G., O'Brien, K., Umuroglu, Y., Leeser, M., Vissers, K.A.: FINN-*R*: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. TRETS **11**(3), 16:1–16:23 (2018)
4. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., Temam, O.: Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014. pp. 269–284. ACM (2014)
5. Cheng, Y., Yu, F.X., Feris, R.S., Kumar, S., Choudhary, A.N., Chang, S.: Fast neural networks with circulant projections. CoRR **abs/1502.03436** (2015)
6. Courbariaux, M., Bengio, Y.: BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. CoRR **abs/1602.02830** (2016)
7. DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G.W., Areibi, S.: Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks. In: FPT 2016, Xi'an, China, December 7-9, 2016. pp. 265–268. IEEE (2016)
8. Dieleman, S., et al.: Lasagne: First release. (Aug 2015). https://doi.org/10.5281/zenodo.27878
9. Ding, C., Wang, S., Liu, N., Xu, K., Wang, Y., Liang, Y.: REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs. In: FPGA 2019, Seaside, CA, USA, February 24-26, 2019. pp. 33–42. ACM (2019)
10. Ghasemzadeh, M., Samragh, M., Koushanfar, F.: ResBinNet: Residual Binary Neural Network. CoRR **abs/1711.01243** (2017)
11. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. pp. 1135–1143 (2015)
12. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. J. Mach. Learn. Res. **18**, 187:1–187:30 (2017)
13. Intel: Intel OpenVino Toolkit. `https://software.intel.com/en-us/openvino-toolkit`
14. LeCun, Y., Cortes, C.: The mnist database of handwritten digits (1998)
15. Liang, S., Yin, S., Liu, L., Luk, W., Wei, S.: FP-BNN: Binarized neural network on FPGA. Neurocomputing **275**, 1072–1086 (2018)
16. Lu, D.: Creating an ai can be five times worse for the planet than a car (Jun 2019), `https://www.newscientist.com/article/2205779-creating-an-ai-can-be-five-times-worse-for-the-planet-than-a-car/`, new Scientist
17. Park, J., Sung, W.: FPGA based implementation of deep neural networks using on-chip memory only. In: ICASSP 2016, Shanghai, China, March 20-25, 2016. pp. 1011–1015. IEEE (2016)

18. Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., Yang, H.: Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016. pp. 26–35. ACM (2016)
19. Radu, V., Kaszyk, K., Wen, Y., Turner, J., Cano, J., Crowley, E.J., Franke, B., Storkey, A., O'Boyle, M.: Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs. In: IISWC 2019. IEEE (October 2019)
20. Rybalkin, V., Pappalardo, A., Ghaffar, M.M., Gambardella, G., Wehn, N., Blott, M.: FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs. In: FPL 2018, Dublin, Ireland, August 27-31, 2018. pp. 89–96. IEEE Computer Society (2018)
21. Su, J., Fraser, N.J., Gambardella, G., Blott, M., Durelli, G., Thomas, D.B., Leong, P.H.W., Cheung, P.Y.K.: Accuracy to Throughput Trade-Offs for Reduced Precision Neural Networks on Reconfigurable Logic. In: ARC 2018, Santorini, Greece, May 2-4, 2018, Proceedings. pp. 29–42. Lecture Notes in Computer Science (2018)
22. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P.H.W., Jahre, M., Vissers, K.A.: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In: FPGA 2017, Monterey, CA, USA, February 22-24, 2017. pp. 65–74. ACM (2017)
23. Venieris, S.I., Bouganis, C.: fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. IEEE Trans. Neural Netw. Learning Syst. **30**(2), 326–342 (2019)
24. Véstias, M.P., Neto, H.C.: Trends of CPU, GPU and FPGA for high-performance computing. In: FPL 2014, Munich, Germany, 2-4 September, 2014. pp. 1–6. IEEE (2014)
25. Wang, E., Davis, J.J., Zhao, R., Ng, H., Niu, X., Luk, W., Cheung, P.Y.K., Constantinides, G.A.: Deep neural network approximation for custom hardware: Where we've been, where we're going. ACM Comput. Surv. **52**(2), 40:1–40:39 (2019)
26. Wang, J., Lou, Q., Zhang, X., Zhu, C., Lin, Y., Chen, D.: Design Flow of Accelerating Hybrid Extremely Low Bit-Width Neural Network in Embedded FPGA. In: FPL 2018, Dublin, Ireland, August 27-31. pp. 163–169. IEEE Computer Society (2018)
27. Zhang, Q., Cao, J., Zhang, Y., Zhang, S., Zhang, Q., Yu, D.: FPGA Implementation of Quantized Convolutional Neural Networks. In: ICCT 2019, Xi'an, China, October 16-19. pp. 1605–1610. IEEE (2019)
28. Zhao, Y., Gao, X., Guo, X., Liu, J., Wang, E., Mullins, R., Cheung, P.Y.K., Constantinides, G.A., Xu, C.: Automatic generation of multi-precision multi-arithmetic CNN accelerators for fpgas. In: ICFPT 2019, Tianjin, China, December 9-13, 2019. pp. 45–53. IEEE (2019)
29. Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., Novik, G.: Neural network distiller (Jun 2018). https://doi.org/10.5281/zenodo.1297430, `https://doi.org/10.5281/zenodo.1297430`