LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference

Erwei Wang, *Student Member, IEEE*, James J. Davis, *Member, IEEE*, Peter Y. K. Cheung, *Senior Member, IEEE*, and George A. Constantinides, *Senior Member, IEEE*

Abstract—Research has shown that deep neural networks contain significant redundancy, and thus that high classification accuracy can be achieved even when weights and activations are quantised down to binary values. Network binarisation on FPGAs greatly increases area efficiency by replacing resource-hungry multipliers with lightweight XNOR gates. However, an FPGA's fundamental building block, the K-LUT, is capable of implementing far more than an XNOR: it can perform any K-input Boolean operation. Inspired by this observation, we propose LUTNet, an end-to-end hardware-software framework for the construction of area-efficient FPGA-based neural network accelerators using the native LUTs as inference operators. We describe the realisation of both unrolled and tiled LUTNet architectures, with the latter facilitating smaller, less power-hungry deployment over the former while sacrificing area and energy efficiency along with throughput. For both varieties, we demonstrate that the exploitation of LUT flexibility allows for far heavier pruning than possible in prior works, resulting in significant area savings while achieving comparable accuracy. Against the state-of-the-art binarised neural network implementation, we achieve up to twice the area efficiency for several standard network models when inferencing popular datasets. We also demonstrate that even greater energy efficiency improvements are obtainable.

Index Terms—Deep neural network, hardware architecture, field-programmable gate array, lookup table.

1 Introduction and Motivation

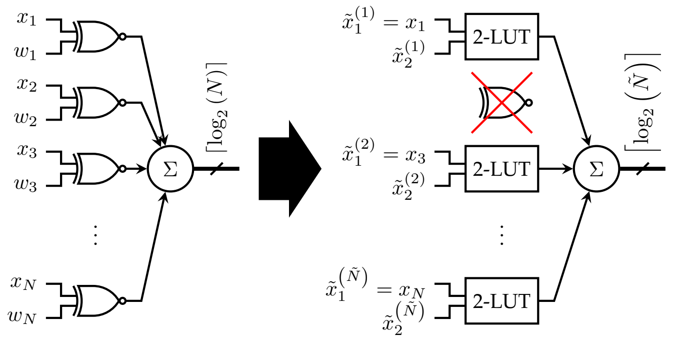
During inference, the most common—and expensive—computational node in a deep neural network (DNN) performs a function of the form in (1), calculating a channel output y. Each weight w_n is a constant determined during training, $\boldsymbol{x}$ a vector of N channel inputs and f an activation function such as the widely used rectified linear unit. In the extreme case where $\boldsymbol{w} \in \{-1,1\}^N$—so-called binarised neural networks (BNNs)—the multiplications become cheap or free to implement. With weight inputs left variable, multipliers become XNOR gates. When networks are unrolled, weights are fixed, and so the XNOR gates can be further simplified into buffers and inverters, all of which are usually subsumed into the downstream adder logic. Also beneficial for BNNs is the ability to use a population count (popcount) for summation: an operation that consumes half the LUTs of the otherwise-throughput-optimal balanced adder tree [1].

$$y = f\left(\sum_{n=1}^{N} w_n x_n\right) \tag{1}$$

No matter how simple these multiplications become, however, all of the products still need to be summed. In modern networks, N commonly reaches numbers in the thousands [2], [3]. To tackle this, we propose the replacement of (1) with the specifically FPGA-inspired function (2), wherein the activation function is unchanged but each product is replaced with an *arbitrary* term-specific Boolean function $g_n : \{-1,1\}^K \rightarrow \{-1,1\}$. The input to this function is a vector $\tilde{\boldsymbol{x}}^{(n)}$ whose elements are any K components of the original input vector $\boldsymbol{x}$, *i.e.* $\tilde{\boldsymbol{x}}^{(n)} = \boldsymbol{S}_n \boldsymbol{x}$ for some binary selection matrix $\boldsymbol{S}_n \in \{0,1\}^{K\times N}$ with $\|\boldsymbol{S}_n\|_\infty = 1$. Since its inputs and outputs are binary, each g_n maps directly to a single K-LUT. BNNs are a special case of this function: they are recoverable when $K = 1$ and $\tilde{N} = N$, with $\boldsymbol{S}_n$ being the row vector with the nth element equal to one and all others zero. An example of the resultant architectural transformation—excluding blocks for f, which are common to both approaches—is given in Fig. 1.

$$y = f\left(\sum_{n=1}^{\tilde{N}} g_n\left(\tilde{\boldsymbol{x}}^{(n)}\right)\right) \tag{2}$$

• *The authors are with the Department of Electrical and Electronic Engineering, Imperial College, London, SW7 2AZ, United Kingdom. E-mail: {erwei.wang13, james.davis, p.cheung, g.constantinides}@imperial.ac.uk*



x_1
w_1
x_2
w_2
x_3
w_3
x_N
w_N
Σ
$\lceil \log_2(N) \rceil$
$\tilde{x}_1^{(1)} = x_1$
$\tilde{x}_2^{(1)}$
2-LUT
$\tilde{x}_1^{(2)} = x_3$
$\tilde{x}_2^{(2)}$
2-LUT
$\tilde{x}_1^{(\tilde{N})} = x_N$
$\tilde{x}_2^{(\tilde{N})}$
2-LUT
Σ
$\lceil \log_2(\tilde{N}) \rceil$

Fig. 1. BNN to LUTNet architectural transformation for a single channel, mirroring the replacement of (1) with (2). Activation function blocks are not shown, but follow the adders. $\tilde{N}$ lookup tables (here, 2-LUTs) substitute N XNOR gates. $\tilde{N} \ll N$ is achieved via pre-substitution pruning, represented by the removal—*i.e.* lack of LUT substitution—of the second XNOR gate. LUT inputs $\tilde{x}_1^{(n)}\ \forall n$ are connected to preserve the pruned BNN's structure. In this case, LUTNet's weights are encoded in its LUT masks, thus they do not appear as inputs.

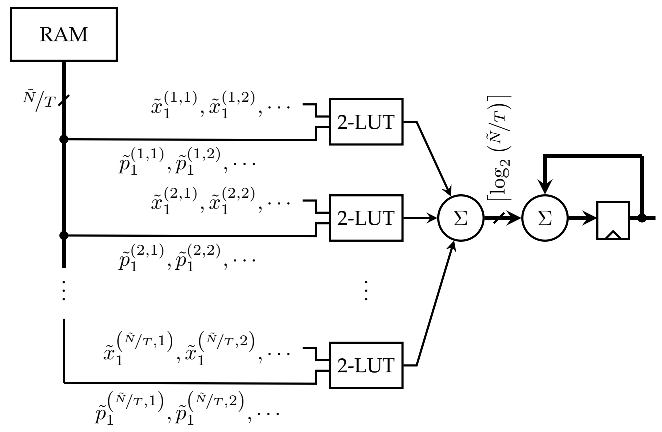

RAM
$\tilde{N}/T$
$\tilde{x}_1^{(1,1)}, \tilde{x}_1^{(1,2)}, \cdots$
$\tilde{p}_1^{(1,1)}, \tilde{p}_1^{(1,2)}, \cdots$
2-LUT
$\tilde{x}_1^{(2,1)}, \tilde{x}_1^{(2,2)}, \cdots$
$\tilde{p}_1^{(2,1)}, \tilde{p}_1^{(2,2)}, \cdots$
2-LUT
$\tilde{x}_1^{(\tilde{N}/T,1)}, \tilde{x}_1^{(\tilde{N}/T,2)}, \cdots$
$\tilde{p}_1^{(\tilde{N}/T,1)}, \tilde{p}_1^{(\tilde{N}/T,2)}, \cdots$
2-LUT
Σ
$\lceil \log_2(\tilde{N}/T) \rceil$
Σ

Fig. 2. Tiled version of the LUTNet architecture shown in Fig. 1. $\tilde{N}/T$ 2-LUTs substitute Fig. 1's $\tilde{N}$ 2-LUTs, with one of each of the former's inputs now connected to on-chip RAM. LUT masks and parameters stored in memory are both learnt during training.

Notice that, while in (1) each element of $\boldsymbol{x}$ only participates in a single summation term, in (2) each can participate in many terms. The intuition here is that inputs can be arranged such that $\tilde{N} \ll N$ for comparable accuracy via network pruning, dramatically reducing the sizes of the required popcount trees. Our experiments demonstrate that this is indeed the case.

The hardware realisation of (2) requires one-to-one $g_n \rightarrow$ LUT binding. Given the relative sizes of today's DNN models *vs* FPGAs, this either limits LUTNet's deployment to only a handful of layers or, in scenarios where throughput and energy efficiency are of paramount importance, *e.g.* for cloud-based computing [4], makes whole-network implementation expensive. By sacrificing a subset of LUT inputs and feeding them with supplementary parameters from RAM, we can trade off throughput and efficiency for additional accuracy. To achieve this, we transform (2) into (3), partitioning channel inputs over T non-overlapping *tiles*. The implementation of (3) is exemplified in Fig. 2.

$$y = f\left(\sum_{t=1}^{T} \sum_{m=1}^{\tilde{N}/T} g_m\left(\tilde{\boldsymbol{x}}^{(m,t)}, \tilde{\boldsymbol{p}}^{(m,t)}\right)\right) \tag{3}$$

In moving from (2) to (3), bivariate functions $g_m : \{-1,1\}^{K-P} \times \{-1,1\}^P \rightarrow \{-1,1\}$ replace every g_n, each of which is used T times per calculation of y. Now, $K-P$ inputs for some $P < K$ are available per node for connection to $\boldsymbol{x}$; the remaining P are used to receive additional learnt parameters, $\tilde{\boldsymbol{p}}$, streamed in from RAM. These parameters effectively allow runtime selection between 2^P candidate $K-P$: 1 Boolean operations per g_m. Note that (3) is a strict generalisation of (2): when $T=1$ and $P=0$, the former reverts to the latter. Comparing Fig. 2 with the equivalent unrolled implementation shown in Fig. 1, K-LUT requirements have been reduced by a factor of T and the popcount tree has been thinned, with small overheads introduced due to the need for RAM and an accumulator.

Our aim in proposing these new inference node functions is to play to the strengths of FPGA soft logic. While a LUT is capable of performing an arbitrary *nonlinear Boolean* function, traditional DNNs are based around *near-linear high-precision* functions: almost the exact opposite of the architecture's forte. Innovations such as BNNs have addressed one side of this weakness, by reducing precision [5]; we address both by also embracing the nonlinearity of the LUT.

Herein, we make the following novel contributions.

- We introduce LUTNet, the first neural network architecture featuring K-LUTs as inference operators. Since each K-LUT is capable of performing an arbitrary Boolean operation on up to K inputs, LUTNet's logic density is much greater than that of BNNs.
- We propose a training regime resulting in the conversion of a BNN architecture from a dense array of XNOR gates into a sparse network of arbitrary K-input functions directly mappable onto K-LUTs.
- We extend our training programme to natively support network tiling, allowing inference nodes to be shared between operations both within and across channels. This facilitates whole-network LUTNet deployment on current-generation FPGAs.
- We empirically demonstrate the effects of LUTNet's increased logic density on area efficiency and accuracy. We also experimentally explore the associated energy and training efficiency impacts. Our results for unrolled, 4-LUT-based inference operators reveal area compression of 2.08× and 1.90× for the CNV network [1] classifying the CIFAR-10 dataset [6] and AlexNet [3] classifying ImageNet [7], respectively, against an unrolled and losslessly pruned implementation of ReBNet [8], the state-of-the-art BNN, while achieving comparable accuracy.
- We comprehensively explore the (P, T) parameter space offered by our tiling-friendly architecture, finding that, while area and energy efficiency gains over ReBNet are less dramatic than in the unrolled case, we still achieve improvements in both metrics: up to 1.28× and 1.57×, respectively.
- We provide an open-source release[1]of LUTNet for the community to use and build upon.

A preliminary version of this work appeared in the proceedings of the 27th IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM) [9]. The early-stage implementation we described in that paper required the complete unrolling of network layers in order to realise the LUTNet architecture, limiting scalability. In this article, we describe how the sacrifice of K-LUT inputs can be used to enable tiling. This adds additional parameters—the number of weight inputs per LUT and tiling factors—to our design space which we empirically explore, finding favourable combinations while enabling whole-network deployment on today's FPGAs. Finally, we provide a documented, open-source release[1] of LUTNet's training and implementation code.

2 Related Work

2.1 Quantisation

The authors of early BNN publications, such as BinaryConnect [10] and BinaryNet [11], proposed network training with binary weights and activations (channel inputs

1. `https://github.com/awai54st/LUTNet`

and outputs) used for forward propagation. High-precision formats—most commonly IEEE-754 single-precision floating point, used to approximate reals $\mathbb{R}$—are always used for backward propagation; this is essential in order for stochastic gradient descent to work well [10]. Tang *et al.* showed that training from scratch with binarised forward propagation is significantly slower than through the consistent use of high-precision data, however; learning rates some $100\times$ lower are required than in the all-real case [12]. Furthermore, binary forward propagation results in the majority of real-valued weights being close to either -1 or 1, while a spread across $[-1, 1]$ is required to facilitate fine-grained pruning [13].

Improving upon BinaryNet's data representation, Rastegari *et al.*'s BWN features layer-wise trainable scaling factors $\boldsymbol{\alpha}$ used in order to increase BNN expressiveness [14]. During training, each $\alpha_l \in \mathbb{R}$ assumes the mean value of layer l's weights. When inferencing, this is multiplied with the layer's popcount results, compensating for some of the information lost to binarisation and increasing accuracy.

Tang *et al.* [12] and the authors of ABC-Net [15] and ReBNet demonstrated the alleviation of information loss from binarisation through the approximation of real-valued weights as linear combinations of multiple binary values. This is achieved via *residual binarisation*, a scheme in which each bit is the binarised residual error of its predecessor. Each bit b is associated with a trainable scaling factor $\gamma_b \in \mathbb{R}$, representing its relative importance. When quantising, each weight $\hat{w} \in \mathbb{R}$ is approximated as B binary weights $w_b = \text{sign}(\epsilon_b)$, as shown in (4), wherein ϵ_b is the b^{th} bit's residual error. During training, each γ_b is updated to minimise the total error. While accuracy was found to be positively correlated with B, diminishing returns were seen; little improvement was observed for $B > 2$.

$$\begin{aligned} \hat{w} &= \sum_{b=1}^{B} \gamma_b \, w_b \\ \epsilon_b &= \epsilon_{b-1} - \gamma_{b-1} \, \text{sign}(\epsilon_{b-1}) \end{aligned} \tag{4}$$

2.2 Pruning

Use of fine-grained pruning effectively adds zero to the set of possible binary weight values, resulting in a ternary representation. Ternarisation has been shown by the authors of many works to deliver significantly higher accuracy than binarisation [16], [17], [18]. Pruning also promotes regularisation, reducing overfitting [19]. The latter is particularly relevant to this work since the use of K-LUTs as inference operators greatly increases potential network complexity.

In order to promote pruning, Han *et al.* proposed training with the l_2 sparsification regulariser in (5) [13]. During backward propagation, Ω influences training loss, inducing weights carrying low significance to descend towards zero. λ, L and C are the regularisation factor, number of layers and number of channels per layer, respectively. $\hat{\boldsymbol{w}}^{(l,c)}$ denotes the real-valued weight vector of layer l's channel c.

$$\Omega = \lambda \sqrt{\sum_{l=1}^{L} \sum_{c=1}^{C} \left(\hat{\boldsymbol{w}}^{(l,c)}\right)^2} \tag{5}$$

2.3 Tiling

Rather than computing the outputs of entire network layers in parallel, many authors have explored the partitioning of weight matrices and corresponding input activations into non-overlapping tiles. Computing tiles sequentially typically reduces throughput and increases latency but increases opportunities for resource sharing, facilitating area and power consumption reductions. Zhang *et al.* proposed tiling along both input and output channels [20]. While authors including Chen *et al.* [21], Ma *et al.* [22] and Qiu *et al.* [23] have also proposed tiling across other dimensions, such as within convolutional windows, these have been shown to typically provide fewer opportunities for resource sharing than intra-channel strategies [22].

All of the aforementioned proposals are complementary to our approach, which uses K-LUTs in their full generality. We integrate this prior work through the use of high-precision training, fine-grained pruning, layer-wise scaling factors and residual binarisation, combining it with the key LUTNet novelty to achieve state-of-the-art performance significantly more cheaply than previously reported in the literature. We also support tiling over input and output channels in order to enable whole-network deployment.

2.4 Architectural Modification

Authors including Boutros *et al.* [24], [25] and Rasoulinezhad *et al.* [26] have proposed modifications to FPGA fabrics to suit the implementation of low-precision dot product operators, including additional carry chains and finer-grained multiplier and adder fracturability. We take the opposite approach: rather than changing the hardware to suit existing DNN arithmetic, we change the arithmetic to suit the FPGA platforms currently on the market.

Given that there is evidence showing that neural networks perform classification by simply memorising their training data, many find it surprising that they can generalise on unseen test data [27]. To explain this phenomenon, Chatterjee proposed a deep network architecture entirely constituting small memory blocks performing table lookups, showing that DNN generalisation can be achieved by memorisation alone [28]. In contrast, we approach LUT-based DNN inference from a hardware-oriented motivation: given existing FPGA LUTs, we seek to achieve the best area-accuracy tradeoffs possible. Unlike Chatterjee's software prototype, we present details of hardware implementations that beat state-of-the-art BNN inference designs. Complementarily to our LUT-based architecture, we also integrate commonly used DNN components including convolution and pooling to improve performance.

3 Network Construction and Training

LUTNet's initialisation comprises three successive stages: training, pruning and *"logic expansion"* (XNOR to K-LUT conversion), with each of the latter two including a retraining phase. All three phases were implemented with TensorFlow. While our training and pruning stages are fairly standard, the final phase—logic expansion—encompasses the key novelty of our approach.

3.1 Training

In order to both expedite learning and facilitate later pruning, our first step is to train the chosen network model using high-precision data during both forward and backward propagation. Layer-wise scaling factors $\boldsymbol{\alpha}$ are learnt during this stage along with weights, and sparsification is induced through the use of the l_2 regulariser in (5) with $\lambda = 5 \times 10^{-7}$ as suggested by Tang *et al.* [12].

3.2 Pruning

Following high-precision training, fine-grained pruning is conducted through the application of threshold θ on each weight $\hat{w}$, as shown in (6). θ exposes a continuum between area occupancy and accuracy: the higher its value, the more weights are pruned away.

$$\hat{w} \leftarrow \begin{cases} \hat{w} & \text{if } |\hat{w}| > \theta \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Once pruned, the network is binarised following the scheme shown in (4), after which it is retrained in order to recover some of the induced accuracy loss. Due to the diminishing returns previously found when applying residual binarisation [12], [15], [8], we used $B = 2$ (two-level binarisation) consistently.

3.3 Logic Expansion

At this point, we have obtained a residual-binarised ternary neural network with non-zero-weighted operators implemented as XNORs. It is from here that we depart from the standard BNN approach. Each XNOR gate is replaced with a K-LUT, whose first input $\tilde{x}_1^{(m,t)}$ is assigned to preserve the original connection, thereby retaining the pruned BNN's structure. If $K - P > 1$, the $K - P - 1$ subsequent inputs to each LUT are then randomly selected from channel inputs within the same convolutional window as $\tilde{x}_1^{(m,t)}$, ensuring that the window shape remains unchanged. We additionally constrain their selection such that each channel input is connected at most once to each LUT. Where $P > 0$, each LUT's final P inputs are connected to a P-bit-wide memory element, designated $\tilde{\boldsymbol{p}}^{(m,t)}$.

Given the aforementioned restrictions on the sources of additional LUT connections, it is possible that, with large K and low P, there will be insufficient (*i.e.* $< K - P - 1$) candidate signals with which to saturate the LUTs. If this does happen, $K - P$ should be reduced in order not to waste inputs. In practice, this scenario is unlikely to manifest: DNNs are complex and sensible choices of K are related to the size of the physical LUTs on the target device, which is typically small. We did not encounter this issue for any of the networks we experimented with.

The form of the inference function proposed in (3) is defined on the binary domain $\{-1,1\}^{\tilde{N}}$. In common with quantisation-inspired networks, such as BNNs, this causes difficulty for training algorithms designed to operate on real vectors $\mathbb{R}^{\tilde{N}}$, specifically in the backward propagation of derivatives. Our approach to this problem is to define an *interpolating extension* of the function $g_m : \{-1,1\}^{K-P} \times \{-1,1\}^{P} \rightarrow \{-1,1\}$, *i.e.* a function $\hat{g}_m : \mathbb{R}^{K-P} \times \mathbb{R}^{P} \rightarrow \mathbb{R}$ such that $\hat{g}_m\left(\tilde{\boldsymbol{x}}^{(m,t)}, \tilde{\boldsymbol{p}}^{(m,t)}\right) = g_m\left(\tilde{\boldsymbol{x}}^{(m,t)}, \tilde{\boldsymbol{p}}^{(m,t)}\right)$ for every $\tilde{\boldsymbol{x}}^{(m,t)}$ and $\tilde{\boldsymbol{p}}^{(m,t)}$ in the domain of g_m. There are many such functions. Of them, we prefer those that are as smooth as possible, allowing training optimisation methods to perform well, and that form a good interpolation in the sense that, if g_m remains constant when a Boolean input flips, so does $\hat{g}_m$. A natural choice for the extension is a Lagrange interpolating polynomial, leading to the form we use in (7).

$$\hat{g}_m\left(\hat{\tilde{\boldsymbol{x}}}^{(m,t)}, \hat{\tilde{\boldsymbol{p}}}^{(m,t)}\right) = \sum_{\boldsymbol{d} \in \{-1,1\}^K} \left(\hat{c}_{\boldsymbol{d}} \prod_{k=1}^{K} \left(\begin{bmatrix} \hat{\tilde{\boldsymbol{x}}}^{(m,t)} \\ \hat{\tilde{\boldsymbol{p}}}^{(m,t)} \end{bmatrix}_k - d_k \right) \right) \tag{7}$$

This expands as shown in (8) for $K > 0$ and $P \geq 0$, with each polynomial comprising 2^K trainable parameters $\hat{\boldsymbol{c}}$.

$$\hat{g}_m\left(\hat{\tilde{\boldsymbol{x}}}^{(m,t)}, \hat{\tilde{\boldsymbol{p}}}^{(m,t)}\right) =$$
$$\begin{cases} \hat{c}_{(-1)}\left(\hat{\tilde{x}}_1^{(m,t)} + 1\right) \\ \quad + \hat{c}_{(1)}\left(\hat{\tilde{x}}_1^{(m,t)} - 1\right) & \text{if } K = 1,\ P = 0 \\[2ex] \hat{c}_{(-1,-1)}\left(\hat{\tilde{x}}_1^{(m,t)} + 1\right)\left(\hat{\tilde{x}}_2^{(m,t)} + 1\right) \\ \quad + \hat{c}_{(-1,1)}\left(\hat{\tilde{x}}_1^{(m,t)} + 1\right)\left(\hat{\tilde{x}}_2^{(m,t)} - 1\right) \\ \quad + \hat{c}_{(1,-1)}\left(\hat{\tilde{x}}_1^{(m,t)} - 1\right)\left(\hat{\tilde{x}}_2^{(m,t)} + 1\right) & \text{if } K = 2,\ P = 0 \\ \quad + \hat{c}_{(1,1)}\left(\hat{\tilde{x}}_1^{(m,t)} - 1\right)\left(\hat{\tilde{x}}_2^{(m,t)} - 1\right) \\[2ex] \hat{c}_{(-1,-1)}\left(\hat{\tilde{x}}_1^{(m,t)} + 1\right)\left(\hat{\tilde{p}}_1^{(m,t)} + 1\right) \\ \quad + \hat{c}_{(-1,1)}\left(\hat{\tilde{x}}_1^{(m,t)} + 1\right)\left(\hat{\tilde{p}}_1^{(m,t)} - 1\right) \\ \quad + \hat{c}_{(1,-1)}\left(\hat{\tilde{x}}_1^{(m,t)} - 1\right)\left(\hat{\tilde{p}}_1^{(m,t)} + 1\right) & \text{if } K = 2,\ P = 1 \\ \quad + \hat{c}_{(1,1)}\left(\hat{\tilde{x}}_1^{(m,t)} - 1\right)\left(\hat{\tilde{p}}_1^{(m,t)} - 1\right) \\[2ex] \cdots & \cdots \end{cases} \tag{8}$$

Since connections are effectively remade from an unpruned BNN (Section 3.1), it makes sense to use those channel inputs' original weights, $\hat{\tilde{\boldsymbol{w}}}$, as a starting point for retraining. The goal of our initialisation process is to achieve the identity (9) holding true for all values of $\hat{\tilde{\boldsymbol{x}}}^{(m,t)}$, wherein $\{2, \cdots, K-P\}$ represents the set of reconnected channel inputs that were removed via pruning (Section 3.2), if any.

$$\hat{g}_m\left(\hat{\tilde{\boldsymbol{x}}}^{(m,t)}, \hat{\tilde{\boldsymbol{p}}}^{(m,t)}\right) = \sum_{i \in \{1, \cdots, K-P\}} \hat{\tilde{x}}_i^{(m,t)} \hat{w}_i^{(m,t)} \tag{9}$$

(9) can be solved by matching the monomial coefficients with respect to $\hat{\tilde{\boldsymbol{x}}}^{(m,t)}$ between the left- and right-hand sides of the equation. When $P = 0$, there are 2^K equations with 2^K unknowns, thus there is exactly one solution. If $P > 0$, however, we have 2^{K-P} equations and $P + 2^K$ unknowns, and so there are infinitely many solutions. For the latter scenario, we choose the most intuitive solution by initialising $\hat{\tilde{\boldsymbol{p}}}^{(m,t)}$ with values from $\hat{\tilde{\boldsymbol{w}}}$. When $P \leq K/2$, *i.e.* the number of memory connections does not exceed the number of input channel weights to initialise from, we

simply let $\hat{\tilde{p}}_i^{(m,t)} \leftarrow \hat{\tilde{w}}_i^{(m,t)} \; \forall i \in \{1, \cdots, K/2\}$. In the case where $P > K/2$, we treat the first $K/2$ elements of $\hat{\tilde{p}}_i^{(m,t)}$ in the same way, while the remainder are selected at random: $\hat{\tilde{p}}_i^{(m,t)} \sim \{-1, 1\} \; \forall i \in \{K/2+1, \cdots, P\}$.

Once all $\hat{g}_m$ are initialised, our second and final retraining phase is conducted, whereafter the binarised training parameters $\boldsymbol{c} = \text{sign}(\hat{\boldsymbol{c}})$ and $\tilde{\boldsymbol{p}}^{(m,t)} = \text{sign}\left(\hat{\tilde{\boldsymbol{p}}}^{(m,t)}\right)$ can be directly interpreted as the configuration mask of each K-LUT and contents of each memory element, respectively. It should be noted that, in the case where previously pruned connections remade in the solution of (9) are again found to be of little importance, this phase will drive those connections' respective $\hat{\boldsymbol{c}}$ parameters towards zero. As a result, the hardware synthesis that follows will reprune them.

We elected to follow the procedure detailed above rather than training with K-LUTs from scratch due to the exponential relationship between K and the number of trainable parameters $\hat{\boldsymbol{c}}$. Training these from the outset, particularly prior to network pruning, would cause both slow convergence and likely overfitting due to the large numbers of local minima in the search space. High-precision training followed by pruning not only ensures fast convergence, it also brings the starting point of K-LUT learning closer to global minima, reducing the likelihood of overfitting.

4 Network Implementation

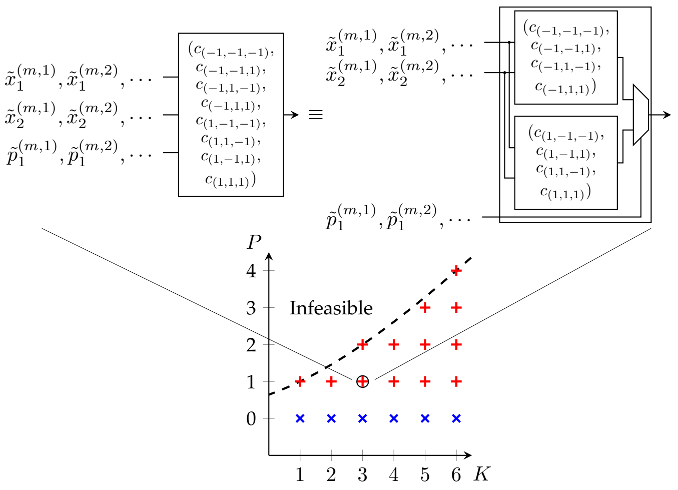
Each operator g_m produced by our training regime will be mapped to our FPGA-specific microarchitecture, which we denote as (K, P)-LUTNet. While "(K, P)-LUTs" are K-LUTs, each can more intuitively be viewed as 2^P internal $K-P$-LUTs, sharing all inputs, multiplexed between using P selection bits. Fig. 3 shows an example with $K = 3$ and $P = 1$. Here, two input ports are used for feeding input activations $\tilde{\boldsymbol{x}}$ and one for the multiplexer selection bitstream $\tilde{p}$, which comes from RAM. The eight c parameters form the configuration mask of the 3-LUT. While each arithmetic operation is associated with a unique $\tilde{p}$, $\boldsymbol{c}$ is shared across all activations computed with the same operator.

Fig. 3 also shows all feasible combinations of K and P for $K \leq 6$, the number of inputs per LUT in major vendors' current-generation FPGAs. We consider an architecture to be feasible only if the total number of expressible $K-P$-input functions is at least equal to the number of functions that can be selected via the P selection bits, *i.e.* if $2^{2^{K-P}} \geq 2^P$.

When tiled with $P = 0$, a network constructed with LUTs configured in this way would have significantly lower flexibility than an equivalent BNN, resulting in a low classification accuracy. We therefore only consider the case where $(K, 0)$-LUTNet implementations are unrolled.

A representation of the overall LUTNet software training and hardware implementation flow is shown in Fig. 4. As input, the user provides the desired network model, training dataset, activation precision and the required pruning level to our TensorFlow-based training software, which performs training and pruning. With user-supplied K, P and T, logic expansion is then performed on the chosen layers—also supplied as input—to construct the LUTNet architecture.

We chose to target Xilinx parts for this work, for which two parallel synthesis flows are required in order to convert the trained network into RTL. For ease of design and



$\tilde{x}_1^{(m,1)}, \tilde{x}_1^{(m,2)}, \cdots$
$\tilde{x}_2^{(m,1)}, \tilde{x}_2^{(m,2)}, \cdots$
$\tilde{p}_1^{(m,1)}, \tilde{p}_1^{(m,2)}, \cdots$
$(c_{(-1,-1,-1)}, c_{(-1,-1,1)}, c_{(-1,1,-1)}, c_{(-1,1,1)}, c_{(1,-1,-1)}, c_{(1,-1,1)}, c_{(1,1,-1)}, c_{(1,1,1)})$
$\equiv$
$(c_{(-1,-1,-1)}, c_{(-1,-1,1)}, c_{(-1,1,-1)}, c_{(-1,1,1)})$
$(c_{(1,-1,-1)}, c_{(1,-1,1)}, c_{(1,1,-1)}, c_{(1,1,1)})$
P
K
Infeasible

Fig. 3. Feasible choices of K and P values for (K, P)-LUTNet, with a demonstration of a $(3, 1)$-LUT microarchitecture implemented using a 3-LUT. The dashed line represents the frontier of feasible microarchitectures. We require those with $P = 0$ (×) to be fully unrolled, while we allow those with $P > 0$ (+) to be tiled across both input and output channels.

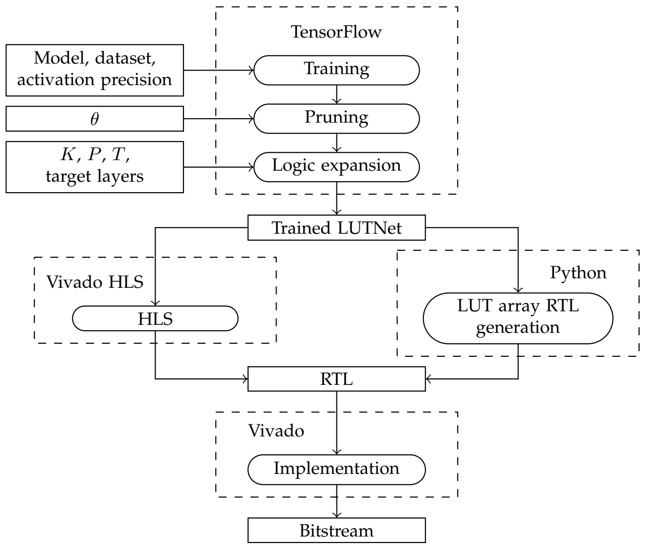


Fig. 4. LUTNet's fully automated training and FPGA implementation flow.

modification, all hardware apart from the inference K-LUTs is generated from C templates with Vivado HLS. LUT array generation is outsourced to a custom RTL generator written in Python, the output of which is combined with that from Vivado HLS after completion. Vivado is then used for implementation and bitstream generation.

A separate LUT array generator is required because, as a general-purpose C-to-RTL synthesis tool, Vivado HLS compulsorily performs code transformations and optimisations for the synthesis of efficient RTL. Given that LUT configurations are already learnt during training, it is unnecessary—and extremely time-consuming—for such optimisation to be performed on this logic at the C level. Optimisation of RTL LUT arrays at the netlist level during synthesis with Vivado is a lot more efficient, typically taking a few hours—rather than days or weeks—to complete for large designs.

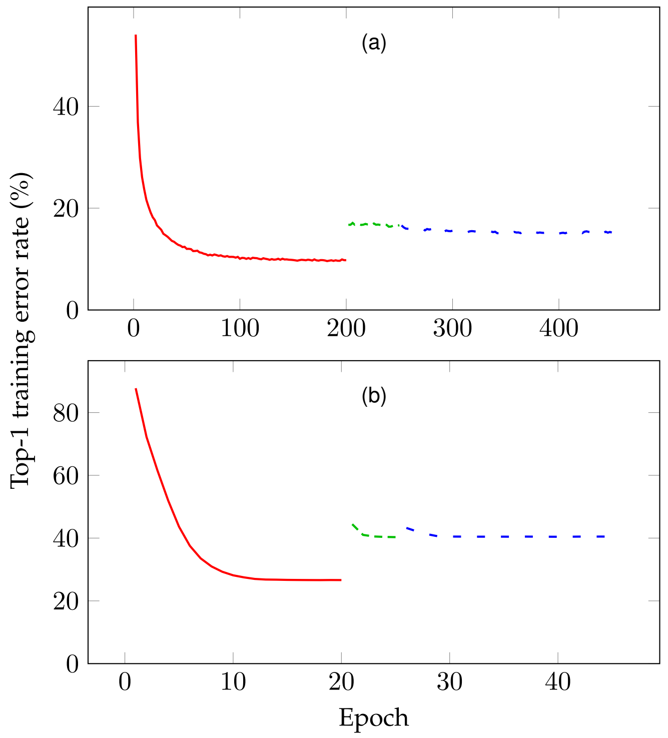


Fig. 5. Training loss for (a) the CNV network classifying the CIFAR-10 dataset and (b) AlexNet classifying ImageNet during high-precision training (——), high-precision post-pruning retraining (- - - -) and post-logic expansion retraining with binarised forward propagation (- -).

5 Evaluation

5.1 Benchmarks

For evaluation, we implemented end-to-end dataflow engines for the DNN models shown in Table 1, using them to classify the listed datasets. All hardware implementations targetted the Xilinx Kintex UltraScale XCKU115 FPGA and met timing at 200 MHz. Our baseline was the state-of-the-art BNN architecture, ReBNet. For fairness of comparison between LUTNet and ReBNet implementations, identical layer-wise tiling factors were always used.

5.2 Training Particulars

For our simpler datasets (MNIST [29], SVHN [30] and CIFAR-10), we performed the training, post-pruning retraining and post-logic expansion retraining described in Section 3 for 200, 50 and 200 epochs, respectively. For the more complex ImageNet dataset, these were performed for 20, 5 and 20 epochs instead. These periods were selected from our observations during training, the loss curves for which are shown in Fig. 5, demonstrating saturation at or before these epochs. Non-LUTNet implementations were identically trained, but the logic expansion phase (Section 3.3) was not performed. All training phases were executed in TensorFlow and accelerated using four Nvidia GTX 1080 Ti GPUs.

5.3 Area Efficiency

When evaluating our designs, we were primarily interested in *logic density*, which we define as the number of LUTs required to construct a network able to achieve a particular test accuracy for a given dataset. The fewer LUTs needed to reach the same accuracy, the higher the density and thus the more efficient the implementation.

5.3.1 *Unrolled Implementation*

In order to demonstrate the full capabilities of specialised LUTs, we initially unrolled a subset of each network such that each node within that subset was mapped to a distinct compute unit. We unrolled as many layers as the target device could accommodate and implemented those following the $(K, 0)$-LUTNet approach, leaving all other layers unchanged. Those selected for unrolled LUTNet implementation are marked in bold in Table 1. For fairness of comparison, the ReBNet baselines had the same layers unrolled, and fine-grained pruning was performed identically to that carried out for LUTNet on those layers. The remaining layers were left tiled, with identical tiling factors to those used for ReBNet's evaluation.

Fig. 6 shows the achieved whole-network area *vs* test accuracy points for ReBNet and $(K, 0)$-LUTNet implementations, each pruned to various densities (proportion of remaining pre-pruning parameters) via the tuning of θ, for CNV classifying the CIFAR-10 dataset. Each point marks the mean of five independent training runs, with an error bar indicating maxima and minima. LUTNet implementations used 2-, 4- and 6-LUT inference operators. For reference, the mean top-1 test error rate of ReBNet without pruning—again averaged over five training runs—is also shown. From this data, one can clearly observe that while the error rate increases as more aggressive pruning is applied, LUTNet demonstrates greater robustness to that pruning than ReBNet through its increased logic density. That several LUTNet points achieve greater test accuracy than the unpruned baseline speaks to LUTNet's increased expressiveness. For example, despite having a significantly lower (2.27×) area requirement, our 91.1%-pruned $(4, 0)$-LUTNet implementation achieved an accuracy 0.590 percentage points (pp) above that of the ReBNet implementation without pruning.

It is interesting to note from Fig. 6 that $(6, 0)$-LUTNet implementations tended to achieve lower logic densities than those of $(4, 0)$- and sometimes even $(2, 0)$-LUTNet. To explain this, we must consider area and accuracy separately.

Fig. 7a shows the test accuracy of ReBNet, $(2, 0)$-, $(4, 0)$- and $(6, 0)$-LUTNet pruned to two densities: 4.02% and 11.3%. These densities were selected for comparison since they represent a wide spread over those found to achieve accuracy reasonably close (±2.00 pp) to ReBNet when unpruned. Of particular pertinence is the difference in accuracy spread between the two: those at 11.3% are much tighter than their 4.02% parallels. These diminishing accuracy returns when adding LUT inputs at higher densities point to complexity saturation.

Turning now to area, Fig. 7b shows the LUT requirements of the same implementations. While $(K, 0)$-LUTNet designs for any K with equal density contain the same number of logical LUTs, this does not mean that they consume the same number of physical LUTs. The LUTs actually present in our target device are 6-LUTs, each capable of implementing either a single logical 6-LUT or two logical K-LUTs with at least five (for 5-LUTs), three (4-LUTs) or one (3-

TABLE 1
Network architectures for evaluated benchmarks. $\text{Conv}_{x,y,z}$ denotes a convolutional layer with x outputs, kernel size $y \times y$ and stride z. FConn_x is a fully connected layer with x outputs. MaxPool_x is an $x \times x$ maximum-pooling layer, and BatchNorm and SoftMax are batch normalisation and normalised exponential layers, respectively. For the experiments described in Sections 5.3–5.6, only the layers shown in bold were implemented following the LUTNet approach; the remainder used the ReBNet architecture. In Section 5.7, LUTNet was used for all Conv and FConn layers.

Dataset	Model	Network architecture
MNIST	LFC	FConn_{256}, BatchNorm, $\mathbf{FConn_{256}}$, BatchNorm, $\mathbf{FConn_{256}}$, BatchNorm, $\mathbf{FConn_{256}}$, BatchNorm, $\mathbf{FConn_{10}}$, BatchNorm, SoftMax
SVHN & CIFAR-10	CNV	$\text{Conv}_{64,3,1}$, BatchNorm, $\text{Conv}_{64,3,1}$, BatchNorm, MaxPool_2, $\text{Conv}_{128,3,1}$, BatchNorm, $\text{Conv}_{128,3,1}$, BatchNorm, MaxPool_2, $\text{Conv}_{256,3,1}$, BatchNorm, $\mathbf{Conv_{256,3,1}}$, BatchNorm, FConn_{512}, BatchNorm, FConn_{512}, BatchNorm, FConn_{10}, BatchNorm, SoftMax
ImageNet	AlexNet	$\text{Conv}_{96,11,4}$, BatchNorm, MaxPool_3, $\text{Conv}_{256,5,1}$, BatchNorm, MaxPool_3, $\text{Conv}_{384,3,1}$, BatchNorm, $\text{Conv}_{384,3,1}$, BatchNorm, $\mathbf{Conv_{256,3,1}}$, BatchNorm, MaxPool_3, FConn_{4096}, BatchNorm, FConn_{4096}, BatchNorm, FConn_{1000}, BatchNorm, SoftMax

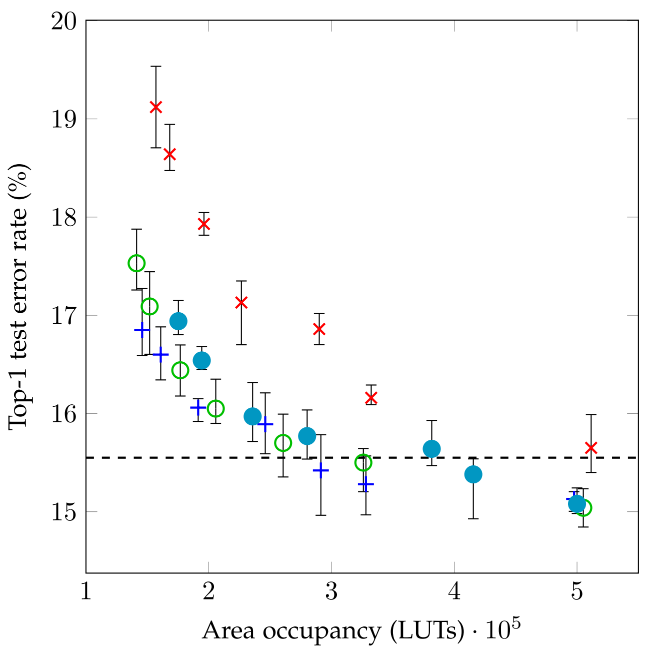


Fig. 6. Whole-network area *vs* CIFAR-10 accuracy tradeoffs for unrolled, pruned ReBNet [8] (×), $(2,0)$-LUTNet (○), $(4,0)$-LUTNet (+) and $(6,0)$-LUTNet (●) implementations of CNV's sixth convolutional layer. All other layers were tiled, unpruned ReBNet realisations. Each point is representative of a distinct pruning threshold. Error bars show the minimum, mean and maximum accuracy achieved over five independent training runs. The dashed line shows the baseline accuracy for an unrolled, unpruned ReBNet implementation (660196 LUTs).

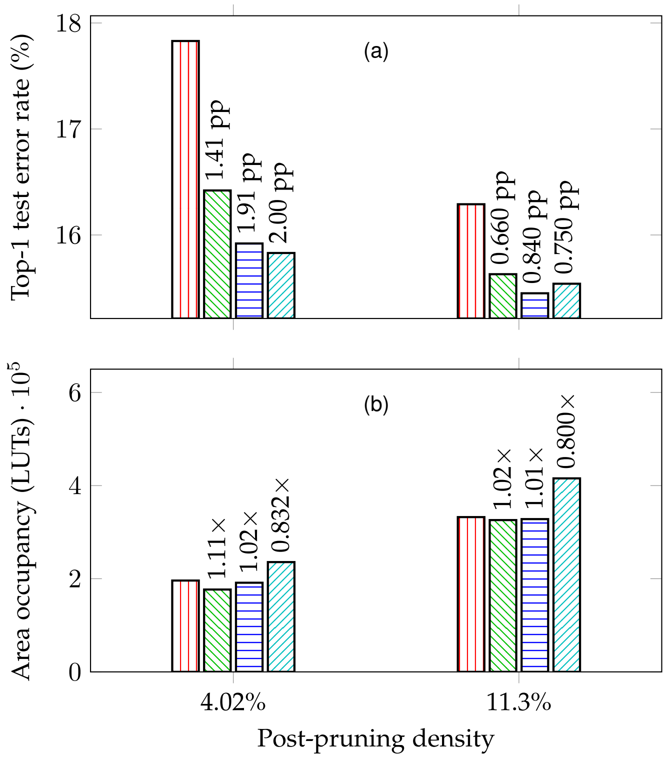


Fig. 7. (a) Accuracy and (b) area for ReBNet [8] (▥), $(2,0)$-LUTNet (▨), $(4,0)$-LUTNet (▤) and $(6,0)$-LUTNet (▨) with CNV, pruned to two densities, classifying CIFAR-10. Annotations denote decreases *vs* ReBNet.

LUTs) shared inputs. 1- and 2-LUTs are not required to share any inputs; two of these can always be packed together. For $(2,0)$- and $(4,0)$-LUTNet, in which each inference operator uses fewer than five inputs, Vivado can often (for $(4,0)$-LUTNet) or always ($(2,0)$-LUTNet) pack two logical K-LUTs into each physical 6-LUT, resulting in high logic density. Training-induced simplifications, *e.g.* inputs treated as don't-cares that are removed during synthesis, also lead to higher probabilities of additional packing when smaller logical LUTs are used. These optimisation phenomena are rarely seen for $(6,0)$-LUTNet, hence its greater area requirements at equal density.

When moving from 4- to 6-LUTs at the higher density, despite the >20% increase in physical LUTs, no accuracy benefit was obtained. Due to this, as was shown in Fig. 6, $(4,0)$-LUTNet almost always achieves a better area-accuracy tradeoff than $(6,0)$-LUTNet.

We also benchmarked LUTNet on other popular datasets and models: MNIST (on LFC), SVHN (on CNV) and ImageNet (on AlexNet). Fig. 8 shows the LUT requirements of each of these model-dataset combinations when implemented using both the ReBNet and $(4,0)$-LUTNet inference architectures. The same layers for all pairs of implementations were unrolled and pruned, with the pruning threshold tuned to achieve an accuracy degradation no more than ±0.300 pp *vs* ReBNet's without pruning.

For CNV and AlexNet, our use of arbitrary inference operators sees area reductions of around 2×. For the classification of SVHN, the CNV network used can be pruned more heavily than for CIFAR-10, hence the greater area saving for that dataset. For LFC classifying MNIST, however, more LUTs were consumed by $(4,0)$-LUTNet than its pruned

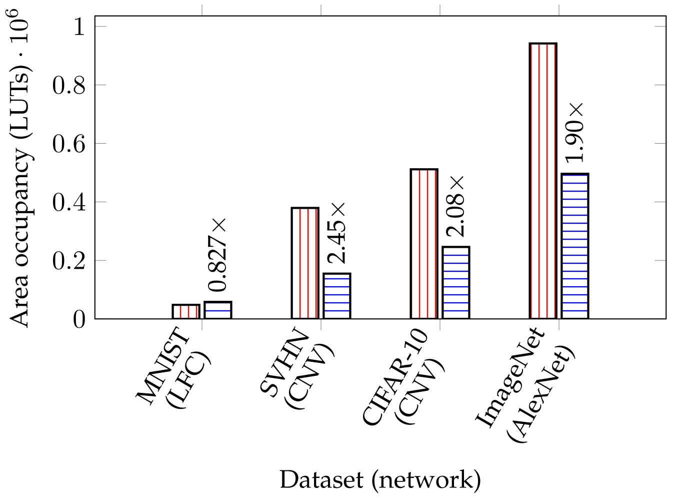


Fig. 8. Area occupancy for ReBNet [8] (▥) and $(4,0)$-LUTNet (▤) with various models and datasets. Via pruning, each implementation's test accuracy was kept within ±0.300 pp of that of the unpruned ReBNet baseline's. Annotations show the area decrease in each case.

ReBNet counterpart. While each of CNV's hidden layers has 2304 inputs per channel, LFC's channels each have only 256 inputs, presenting less opportunity for area reduction through popcount simplification. In this case, $(4,0)$-LUTNet's area savings through popcount tree thinning were unable to compensate for the inference node LUT incursion.

5.3.2 Tiled Implementation

Although we have shown that implementing just one network layer using the unrolled LUTNet architecture leads to significant area efficiency gains for a given modern DNN, their complexities make whole-network unrolled LUTNet implementation infeasible. Given a fixed-sized FPGA, tiling allows us to trade off throughput and efficiency for additional accuracy by enabling our architecture to be used to implement a greater proportion—including all—of the target network. Since tiling can be performed across both input and output channels, to facilitate their distinction we break tiling factor T into two dimensions whose sizes are denoted T_i and T_o for input- and output-wise tiling, respectively. The overall tiling factor $T = T_\mathrm{i}T_\mathrm{o}$.

In order to explore the tradeoffs between K, the number of RAM connections per node P, area and accuracy in the presence of tiling, we repeated the experiments performed for Fig. 6 but with the LUTNet layer implemented with $T_\mathrm{i} = T_\mathrm{o} = 8$. Fig. 9 shows the results of these for $K \in \{3, \cdots, 6\}$ and all feasible P for each K. Two conclusions can quickly be drawn from this data. Firstly, with the exception of the more heavily pruned $(5,3)$-LUTNet design points, tiled LUTNet implementations perform favourably compared to their ReBNet counterparts. For example, our 64.6%-pruned $(5,1)$-LUTNet implementation occupies 1.28× less area than the equivalently tiled, 16.5%-pruned ReBNet, with both delivering test accuracy comparable (within ±0.300 pp) to our unpruned ReBNet baseline. Secondly, and conversely, the gains in area and accuracy over ReBNet we see with tiling are smaller than those without. This is unsurprising; the LUTNet approach becomes less effective when LUTs trained to perform specialised inference functions are used repeatedly.

Understanding the relative area-accuracy behaviours shown across Fig. 9 requires knowledge of the relationships between K, P, T_i, T_o and the total number of trainable parameters. All else being equal, tiling *decreases* the total number of trainable LUT parameters by a factor of $T_\mathrm{i}T_\mathrm{o}$ *vs* an unrolled implementation. The number of parameters fed from RAM, however, *increases* linearly with P, T_i and T_o. With pruning, both quantities will be reduced by a factor of $1/\omega$, where ω is the post-pruning density. Overall, the total number of LUTNet-related parameters within our implementations is $9216\omega\left(2^K/T_\mathrm{i}T_\mathrm{i} + PT_\mathrm{i}T_\mathrm{o}\right)$, where 9216 is the number of inference nodes in our unpruned target layer. To facilitate analysis, we show these values for a selection of the design points in Fig. 9 in Fig. 10. Note the logarithmic x-axis, which we used due to the relative magnitudes of fixed K (Fig. 10a) and fixed P (Fig. 10b) implementations' parameter spaces as well as the decaying exponential relationship between the number of parameters and accuracy.

Comparing between the plots of Fig. 9 reveals a weak trend of worsening area-accuracy behaviour with increasing K, with curves moving slightly rightwards between Figs 9a and 9d. In common with unrolled LUTNet implementations, larger K limits opportunities for double-logical-to-physical LUT packing, decreasing area efficiency. These effects tend not to be outweighed by the slight accuracy improvements brought about through the expansion of our parameter space. Although the K-influenced component of the total number of parameters increases exponentially with K, its scaling by $1/T_\mathrm{i}T_\mathrm{o}$—in this case, $1/64$—means that the effects of increasing it are typically small for sensible choices of K. This is exemplified in Fig. 10b for $P = 1$. Relatively, these gaps become even tighter for $P > 1$, since the P component of the parameter space's size is, although only linearly related to P, scaled by a much larger factor: $T_\mathrm{i}T_\mathrm{o}$.

The significant increase in total parameters with P is demonstrated in Fig. 10a for $K = 6$. With a few exceptions for heavily pruned cases, we see a weak negative correlation between complexity and accuracy here, suggesting that increasing our training space via P is not particularly effective in improving performance. These effects tend to be more than outweighed, however, by decreases in area. Apart from the anomalous results for $(5,3)$-LUTNet, for which this relationship unexpectedly reverses, Fig. 9 consistently shows improved area-accuracy tradeoff with increasing P. We believe that this is due to the structured *vs* unstructured nature of routing connections made from RAM and activations, respectively. With fixed K, increased P enlarges the ratio of RAM-to-activation connections. Since signals from RAM are clustered into busses, while those from the previous network layer tend to be haphazard in structure, the regularity induced through higher P affords Vivado more opportunity for denser LUT packing, lowering area.

Finally, while complexity saturation is apparent for all (K,P) at higher pruning levels, there is little evidence of overfitting in either Fig. 9 or Fig. 10; non-negligible downturns in accuracy do not occur with greater numbers of parameters. We can therefore conclude that, even with significant complexity increases over ReBNet, LUTNet-based networks are amenable to effective training using standard

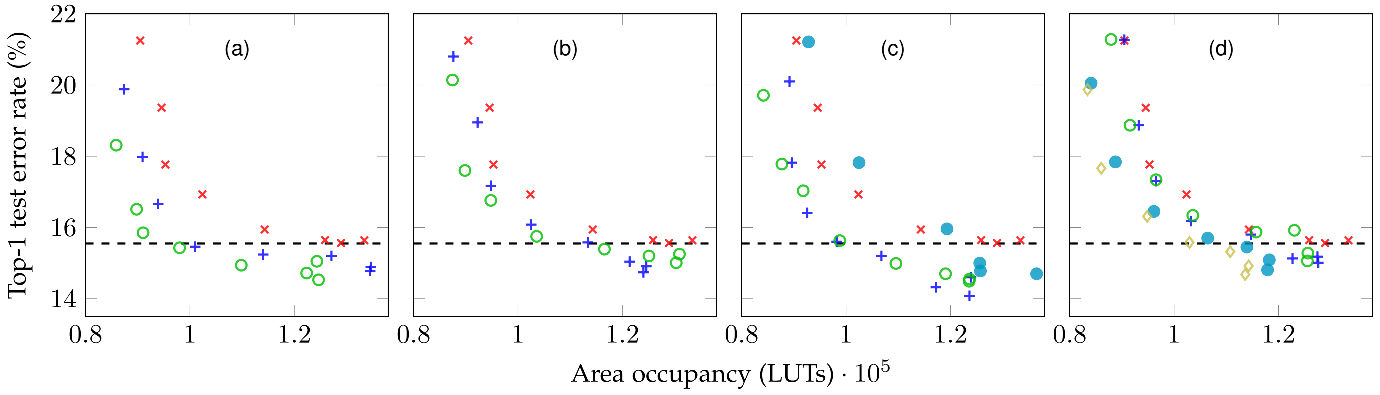


Fig. 9. Area-accuracy tradeoffs for pruned ReBNet [8] (×) and LUTNet implementations of CNV classifying CIFAR-10 with tiling factors $T_i = T_o = 8$. Across the subplots, we show results for (K, P)-LUTNet for feasible combinations of $K \in \{3, \cdots, 6\}$ and $P > 0$. In (a), these are $K = 3$ with $P = 1$ (+) and 2 (○); in (b), $K = 4$ with $P = 1$ (+) and 2 (○); in (c), $K = 5$ with $P = 1$ (+), 2 (○) and 3 (●); and, in (d), $K = 6$ with $P = 1$ (+), 2 (○), 3 (●) and 4 (◊). Each point represents a distinct pruning threshold. The dashed lines show the baseline accuracy for unpruned ReBNet (133418 LUTs).

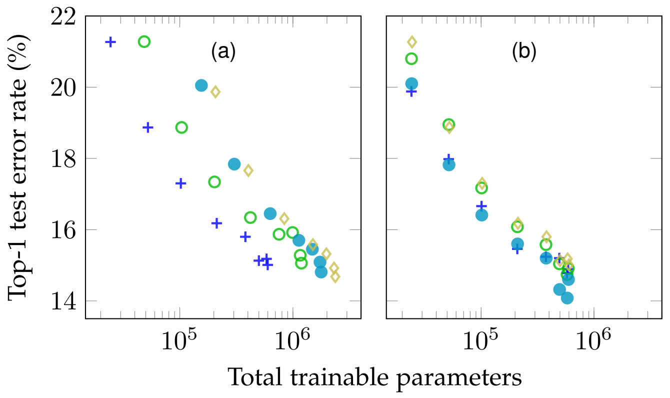


Fig. 10. Number of parameters *vs* accuracy for implementations shown in Fig. 9 with (a) fixed K and (b) fixed P. In (a), these feature $K = 6$ with $P = 1$ (+), 2 (○), 3 (●) and 4 (◊), while, in (b), we show designs with $P = 1$ and $K = 3$ (+), 4 (○), 5 (●) and 6 (◊).

backward propagation techniques.

5.4 Area Breakdown

As a crude method of verifying the source of LUTNet's area savings, we disabled design hierarchy optimisation in Vivado, preventing the synthesis engine from flattening across modules. By taking a slice of implementations shown in Figs 6 and 9 at the unpruned ReBNet error rate (84.5%) ±0.300 pp, we obtained ReBNet and LUTNet implementations for CNV all of comparable CIFAR-10 test accuracy. Fig. 11 shows the LUT requirements for each of these, with area split into that required by popcount operators, inference operators, multiplexing logic and everything else. The overall height of each bar is the whole design's area occupancy with hierarchy optimisation *enabled*, but the height of each stacked bar is relative to the proportional area obtained with hierarchy optimisation *disabled*.

We can see from Fig. 11a that, as more inputs are used per logical LUT, physical LUT requirements generally decrease, highlighting $(K, 0)$-LUTNet's increasing logic density with K. Each implementation's post-pruning density is also shown. From the breakdowns, it can be seen that the number of physical LUTs required for popcount operators drops dramatically with density. More aggressive pruning reduces the number of branches in the popcount trees, which consume the majority of the target device's area.

As was pointed out in Section 1, due to following a traditional BNN inference paradigm, unrolled ReBNet implementations—whether pruned or not—require zero LUTs for the realisation of their inference operators since their XNOR gates become free-to-implement buffers and inverters. For LUTNet, this is not the case: physical LUTs are always consumed by our logical K-LUTs. As shown in Fig. 11a, however, this is more than outweighed by significant popcount area reduction. This confirms the statement made in Section 1 regarding $\tilde{N} \ll N$.

Between $(2, 0)$- and $(6, 0)$-LUTNet, we can observe a general trend of decreasing inference operator LUT requirements with density. Looking more closely, some more interesting features emerge. The jump in total area between $(4, 0)$- and $(5, 0)$-LUTNet can be attributed to two factors: lack of density reduction and decreased opportunity for LUT sharing. Here, the increased expressiveness of 5-LUTs was not significant enough to enable increased pruning while remaining within the required accuracy bound. On top of this, the logical-to-physical LUT packing effects discussed in Section 5.3.1 were marked, pushing both inference operator and total LUT requirements for $(5, 0)$-LUTNet above those for $(4, 0)$-LUTNet. Thereafter, although a greater number of physical LUTs were occupied by the $(6, 0)$-LUTNet implementation, a decrease in density facilitated through increased network complexity caused a more-than-compensatory popcount area reduction.

In Fig. 11b, we show area information in the same form as Fig. 11a for the tiled LUTNet implementations in Fig. 9 with $K \in \{4, 5, 6\}$ and all feasible choices of P. As expected, tiled implementations' area savings are lower than those for unrolled designs due to their lower tolerance to pruning, although we still see gains over ReBNet in all cases.

It is important to note that, while unrolled LUTNet implementations can achieve significantly greater area savings over ReBNet than their tiled counterparts, they are still physically large. For example, while our unrolled $(5, 0)$-LUTNet design was some 1.87× more compact than unrolled ReBNet due to its 94.4% sparsity, it was 2.78× the size

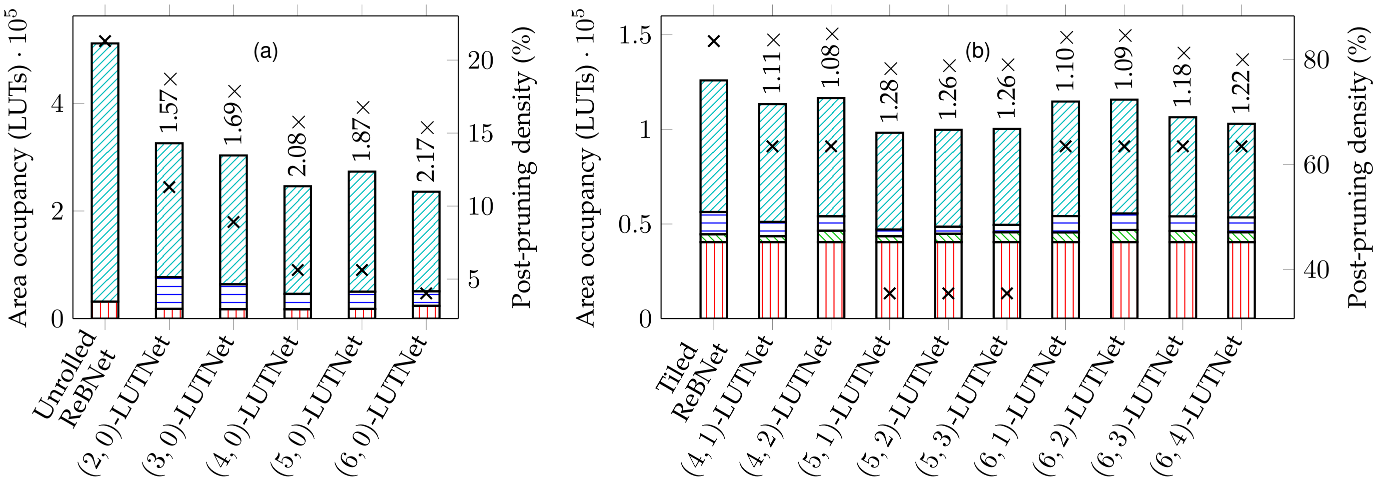


Fig. 11. LUT use breakdown, presented in terms of popcount operators (▨), inference operators (⊟), multiplexing logic (▧) and other layers (▥), for CNV implementations. In (a), designs were unrolled; those in (b) were tiled with $T_\mathrm{i} = T_\mathrm{o} = 8$. Annotations show decreases *vs* pruned ReBNet. Each implementation's test accuracy was within ±0.300 pp of that of the unpruned ReBNet baseline's [8]. Points (×) show post-pruning densities.

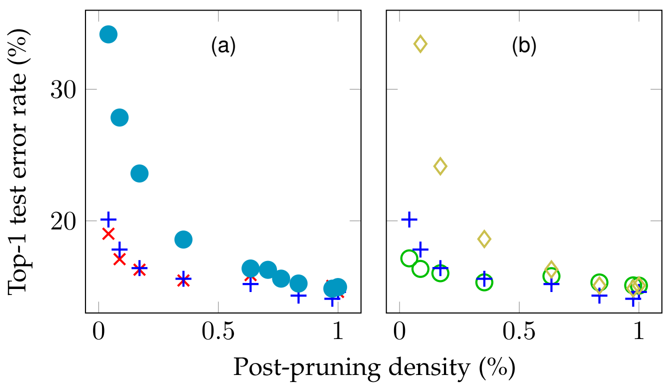


Fig. 12. Density *vs* CIFAR-10 accuracy for tiled $(5, 1)$-LUTNet CNV implementations. In (a), we show designs with $T_\mathrm{i} = 8$ and $T_\mathrm{o} = 4$ (×), 8 (+) and 16 (●), while (b) features $T_\mathrm{o} = 8$ and $T_\mathrm{i} = 4$ (○), 8 (+) and 16 (◇). Each point represents a distinct pruning threshold.

of $(5, 1)$-LUTNet with $T_\mathrm{i}T_\mathrm{o} = 64$, which reaches comparable accuracy, albeit at greatly reduced throughput. The choice of whether or not to tile—and to what level if so—depends on the number of LUTs one can afford to use. With one-to-one $g_n \rightarrow$ LUT binding, unrolled LUTNet makes the most efficient use of soft logic yet consumes significant numbers of resources, while tiled implementations sacrifice both area efficiency and speed in return for demanding lower area.

5.5 Implications of Tiling

While the relationships between tiling factors T_i and T_o and area and throughput are straightforward—area and throughput both scale with $1/T_\mathrm{i}T_\mathrm{o}$—those with accuracy are less so. Clearly, increased $T_\mathrm{i}T_\mathrm{o}$ will result in accuracy degradation. It is not obvious, however, whether tiling across either input or output channels, or some combination of the two, is preferable. In Fig. 12, we show results for the same experiment performed for Fig. 9, but with (K, P) fixed at $(5, 1)$ and the constraints on T_i and T_o relaxed.

While Figs 12a (fixed T_i) and 12b (fixed T_o) show similar trends, a slight preference towards increasing T_o is evident. This is reflected in the more heavily pruned cases, where both $(T_\mathrm{i}, T_\mathrm{o}) = (8, 4)$ and $(16, 8)$ jump up in error sooner than their respective counterparts, $(4, 8)$ and $(8, 16)$. When tiling across only inputs, weights are shared within units computing each output. With output-wise tiling, however, they are instead shared across units calculating different outputs; there is no intra-unit weight sharing in this case. When pruning to some predetermined level, the likelihood of entire output channels being thinned is therefore higher in the case of input-only tiling, leading to greater accuracy degradation. The same argument holds for cases where tiling across both inputs and outputs is applied, but with a higher degree of the former over the latter. Thus, when one is faced with a choice between increasing either T_i or T_o, we suggest that prioritisation should be given to T_o.

5.6 Energy Efficiency

We estimated LUTNet's energy efficiency using the Xilinx Power Analyzer (XPA) tool with default settings: vectorless mode and 12.5% primary input switching probability. The resultant power estimates, for the same implementations captured in Fig. 11, are shown in Fig. 13. All were obtained post-placement and -routing. Power consumption is equivalent to energy efficiency here since all implementations presented side-by-side have identical throughput. While vectorless power estimates are not particularly accurate—typically around ±10–20% from measured values [31]—they are sufficiently so for our purposes.

Since dynamic power consumption is directly related to area occupancy, Figs 11a and 13a show similar trends. Most of the fully unrolled networks' area consumption is attributable to popcount adder trees, whose carry chains are dominant with respect to switching activity. Popcount branch pruning shortens the chains, more than proportionately lowering their switching rates and thereby causing the large dynamic power drop. The reduction in static power

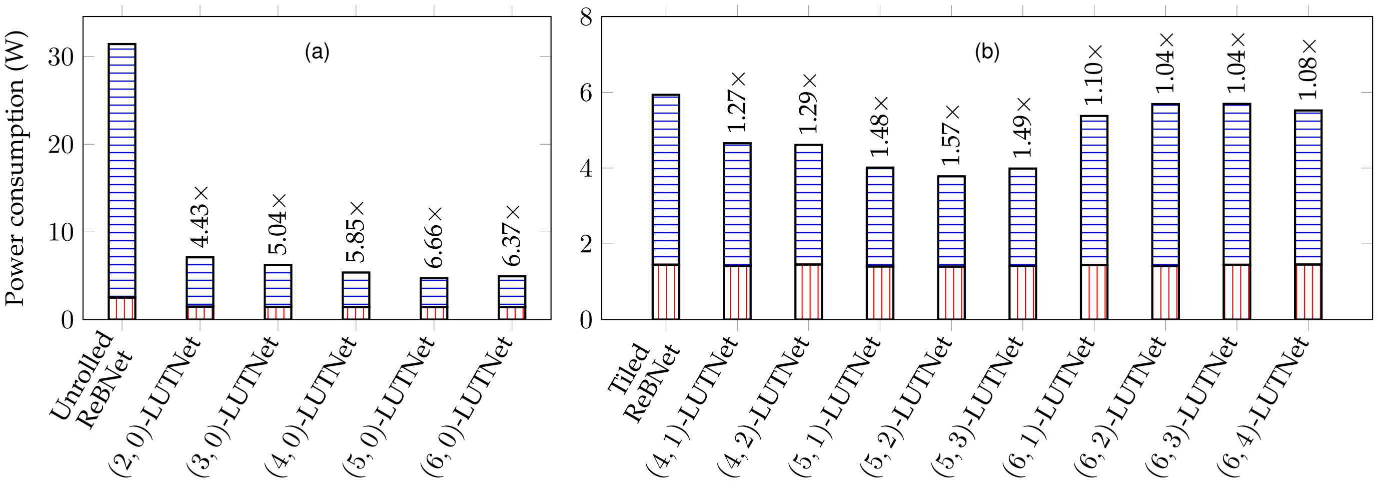


Fig. 13. Implementation power consumption estimates, broken into static (▥) and dynamic (▤) components, for the same CNV implementations shown in Fig. 11. In (a), designs were unrolled, while those in (b) were tiled with $T_\mathrm{i} = T_\mathrm{o} = 8$. Annotations show decreases *vs* ReBNet [8].

between the unrolled ReBNet and LUTNet implementations can also be linked to area, although indirectly. Between Pruned ReBNet and $(2, 0)$-LUTNet there was a drop in estimated junction temperature from 60.1° to 31.3°, leading to reduced leakage current and therefore static power draw. Such temperature decreases are also useful since they limit ageing, thereby increasing device lifetime [32]. Overall, we can conclude that unrolled LUTNet implementations' significant area reductions result in even greater energy efficiency improvements over their ReBNet counterparts.

In the same style as Fig. 13a, Fig. 13b shows power consumption estimates for the tiled implementations whose area breakdowns were reported in Fig. 11b. Comparing between Figs 13a and 13b, we can see large jumps in the ratio of static *vs* dynamic power for the latter over the former. This is a direct consequence of the tiled designs' greatly reduced resource requirements. Dynamic power reductions over ReBNet were modest compared to those for the unrolled implementations, which tended to be large: up to 8.76×. Despite this, however, we still achieved energy efficiency improvements over ReBNet in all cases. Notice also that the total power consumptions of tiled LUTNet implementations are similar to those of the unrolled alternatives. For example, unrolled $(5, 0)$-LUTNet is estimated to consume 4.72 W, while $(5, 1)$-LUTNet with $T_\mathrm{i} = T_\mathrm{o} = 8$ consumes 4.01 W. Although unrolled implementations occupy more area than their tiled counterparts, the former's greatly increased resilience to pruning—coupled with throughput-bottlenecking by other network layers—makes them similarly power-hungry, despite being faster.

5.7 Throughput Maximisation

In an effort to demonstrate the potential of LUTNet for whole-network realisation, we created implementations for each of the network-dataset pairs detailed in Table 1, using our proposed architecture for all convolutional and fully connected layers. Our aim was to, within the area constraints imposed by our target FPGA, maximise both throughput (in classifications per second, cl/s) and top-1 test accuracy. We combined these into a single metric, accuracy-throughput product, for comparison to the ReBNet implementations we constructed in the same way.

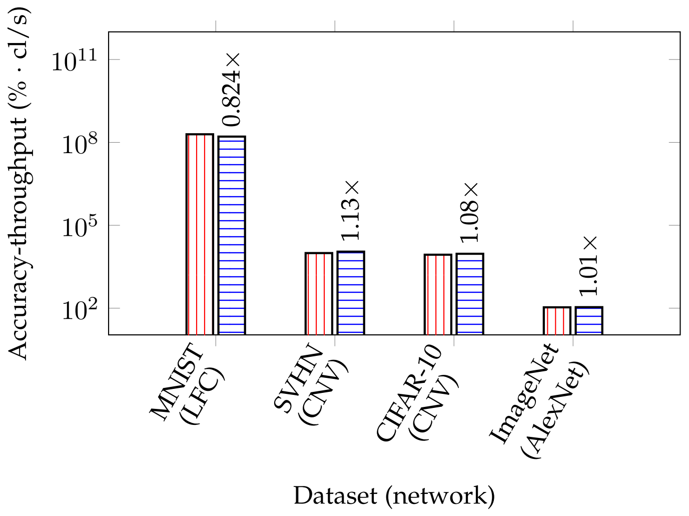


Fig. 14. Accuracy-throughput product for whole-network ReBNet [8] (▥) and $(5, 1)$-LUTNet (▤) implementations with various models and datasets. For each design, layer-wise pruning thresholds θ and tiling factors T_i and T_o were chosen to attempt to maximise accuracy per unit throughput by saturating our target FPGA's resources while keeping throughput balanced across layers. Annotations show increases.

The experiments reported in Fig. 9 revealed that, of the feasible combinations of (K, P) for tiled LUTNet, $(5, 1)$ behaved the most favourably in terms of area *vs* accuracy when pruned. For completeness, we performed the same experiments for $K = 2$ (not shown in Fig. 9) and confirmed this to be the case. Thus, for this section, we used $(K, P) = (5, 1)$ throughout. For each layer within each benchmark network, we hand-tuned pruning threshold θ and tiling factors T_i and T_o to maintain high accuracy and keep throughput balanced across layers while making as much use of the available resources as possible. We present

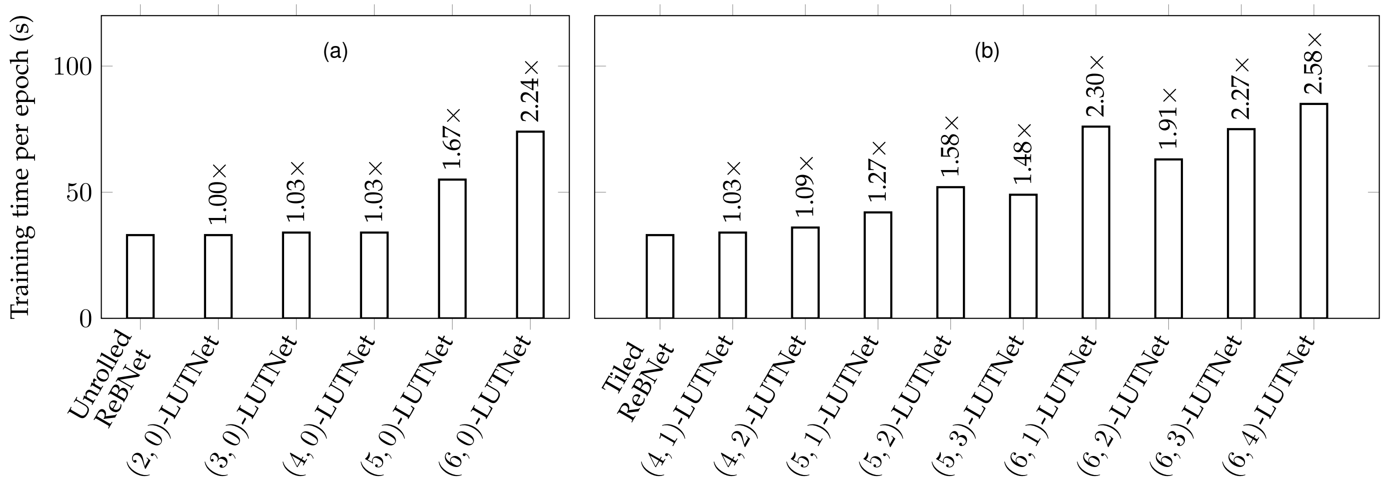


Fig. 15. Training time per epoch for the same CNV implementations shown in Figs 11 and 13. Annotations show increases *vs* ReBNet [8].

the results of these experiments in Fig. 14, which shows modest improvements over ReBNet implementations for all models except for LFC classifying MNIST, in common with the result for the same network-dataset pair seen in Fig. 8.

We stress that these results are not optimal. Due to the sheer size of our design space and the lengthy training times associated with whole-network implementation, it was not practicable for us to locate the best design points possible. Based on our findings from the parameter tuning of limited numbers of layers, we made educated guesses of sensible choices for K, P, θ and T_i *vs* T_o ratios. With time and compute power dedicated to design-space exploration, greater performance gains could be achieved.

5.8 Training Time

We finally sought to quantify the primary cost of LUTNet's deployment: that of training time. In Fig. 15, we present the whole-network training times of the implementations used to generate the results in Figs. 11 and 13.

Each of $(K, 0)$-LUTNet's inference K-LUTs consists of 2^K parameters: $2\times$ more than that for $(K-1, 0)$-LUTNet. Consequently, the number of training operations required per epoch increases exponentially with K. This does not necessarily translate to exponentially increasing training times over XNOR-based BNNs, however, since, as pointed out by Jouppi *et al.*, the majority of DNN training accelerators' speed is bounded by memory bandwidth, not compute power [33]. This is evident from Fig. 15a, which shows the per-epoch training times of ReBNet and LUTNet implementations for CNV with CIFAR-10. Implementations from ReBNet to $(4, 0)$-LUTNet all have approximately the same training rate, despite the number of parameters increasing by up to $16\times$. The training time did not increase because, for all of these implementations, progress was bottlenecked by high-precision activation transfer to and from GPU RAM. Increases of significance were seen for $(5, 0)$-LUTNet and beyond, for which the number of multiply-accumulate operations performed per activation transferred rose enough for the former to dominate.

Tiled implementations take longer to train than those with the same layers unrolled, as reflected in Fig. 15b. This is unsurprising due to tiling's effect on the number of trainable parameters within the network, the most significant component of which increases linearly with P, T_i and T_o, as discussed in Section 5.3.2. It is interesting to note that the increases in training time with P are not as significant as the parameter growth rate with $PT_\mathrm{i}T_\mathrm{o}$ might suggest. In every epoch, TensorFlow stores a new copy of input activations in RAM for each LUT's $K-P$ input connections. As P grows, $K-P$ reduces for the same K, thus increasing the number of operations to perform but decreasing the number of these expensive memory copies. We see that the latter thus dampens the slowdowns brought about by the former.

Recall that all of the designs featured in Fig. 15 only have a single layer implemented using LUTNet operators. Slowdowns in training increase significantly with, in particular, whole-network LUTNet deployment. For example, while a CNV implementation with only the largest convolutional layer implemented using the unrolled $(5, 0)$-LUTNet architecture takes $1.67\times$ longer to train than a fully ReBNet equivalent, this factor increases to 15.8 when the same network is wholly implemented using $(5, 1)$-LUTNet inference operators with $T_\mathrm{i} = T_\mathrm{o} = 8$. We do not consider this to be of significant concern, however: training times of around a day are not of dissimilar duration to compilation times for large FPGA designs, which will be needed as well, and are far shorter than their typical development cycles.

6 Limitations

Figs 6 and 9 show that while our expansion to 2-LUTs results in significant logic density gains over XNORs, returns for movement to K-LUTs for $K > 2$ are diminishing. We suspect that this is due to our current restriction on the form of the function g_m in (3), *i.e.* $\{-1,1\}^{K-P} \times \{-1,1\}^P \to \{-1,1\}$ rather than $\{-1,1\}^{K-P} \times \{-1,1\}^P \to \mathbb{N}$. This makes (9) insoluble when $\hat{c}_{\boldsymbol{d}}$ and $\hat{\tilde{p}}^{(m,t)}$ are restricted to binary values. We can overcome this, and potentially make even more efficient use of the underlying FPGA fabric, by learning the popcount circuitry along with our XNOR substitutes, replacing the summation as well as $w_n x_n$ in (1).

While the introduction of nonlinearity significantly increases the expressiveness of each inference operator, the experiments reported in Section 5.3 revealed that $(6, P)$-LUTNet showed early signs of overfitting. In the future, we will explore methods of throttling expressiveness during training guided by losses, *e.g.* switching to higher or lower values of K and P where appropriate.

Finally, LUTNet's software does not currently skip zeroes during training. As networks increase in size, GPU RAM will be increasingly inefficiently used, resulting in unnecessarily long training times. A future revision will therefore incorporate sparse matrix multiplication, preventing the storage of and multiplication by zero-valued weights.

7 Conclusion

In this article, we introduced LUTNet: the first DNN architecture featuring K-LUTs as inference operators specifically designed to suit FPGA implementation. Our novel training approach results in the construction of K-LUT-based networks robust to high levels of pruning with little or no accuracy degradation, enabling the achievement of significantly higher area and energy efficiencies than those of traditional BNNs.

We comprehensively evaluated both unrolled and tiled versions of the LUTNet architecture. For the former, our experiments with 4-LUT-based inference operators revealed that FPGA implementations following our proposals achieved a mean area reduction of $1.81\times$ *vs* the state-of-the-art BNN architecture with unrolling and pruning. These designs targetted a range of standard DNN models and datasets, required approximately the same training time and reached accuracy bounded within ± 0.300 pp in all cases. Due to their efficient use of soft logic, unrolled LUTNet implementations can exhibit energy efficiency up to $6.66\times$ greater than reported by the authors of related prior works. Thanks to its parameter hardening, our unrolled architecture also requires no use of block RAM: a common bottleneck for FPGA-deployed DNNs.

While unrolled LUTNet implementations have many advantages over their tiled counterparts—particularly higher throughput and compressibility—they typically consume large numbers of resources. When tiled, LUTNet designs can achieve similarly high accuracy while occupying much less area. For example, we found that an 8×8-tiled and 64.6%-pruned $(5, 1)$-LUTNet CNV implementation was $2.78\times$ smaller and $1.18\times$ less power-hungry than an unrolled and 91.1%-pruned $(4, 0)$-LUTNet equivalent, with both delivering comparable CIFAR-10 classification accuracy. The choice of whether or not to tile, and to what level if so, will depend on the amount of LUT and RAM resources one can afford to use. With increased tiling, throughput and area efficiency can be sacrificed to achieve smaller, lower-power designs.

The authors of existing works on low-precision DNN inference seem to have assumed that their forward-propagation functions must be good approximations of the linear dot product. With LUTNet, we argue for a tangential approach: through the embracement of nonlinearity, one can do more with less by unlocking the full potential of the LUT.

Acknowledgements

The authors are grateful for the support of the United Kingdom EPSRC (grant number EP/P010040/1), Imagination Technologies and the Royal Academy of Engineering.

Supporting data for this article are available online at `https://doi.org/10.5281/zenodo.3517151`.

References

[1] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2017.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Conference on Neural Information Processing Systems*, 2012.

[4] R. Zhao, S. Liu, H.-C. Ng, E. Wang, J. J. Davis, X. Niu, X. Wang, H. Shi, G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Hardware Compilation of Deep Neural Networks: An Overview," in *International Conference on Application-specific Systems, Architectures and Processors*, 2018.

[5] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going," *ACM Computing Surveys*, vol. 52, no. 2, 2019.

[6] A. Krizhevsky and G. Hinton, "Learning Multiple Layers of Features from Tiny Images," 2009.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-scale Hierarchical Image Database," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[8] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "ReBNet: Residual Binarized Neural Network," in *IEEE International Symposium on Field-programmable Custom Computing Machines*, 2018.

[9] E. Wang, J. J. Davis, P. Y. Cheung, and G. A. Constantinides, "LUTNet: Rethinking Inference in FPGA Soft Logic," in *IEEE International Symposium on Field-programmable Custom Computing Machines*, 2019.

[10] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," in *Conference on Neural Information Processing Systems*, 2015.

[11] M. Courbariaux and Y. Bengio, "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[12] W. Tang, G. Hua, and L. Wang, "How to Train a Compact Binary Neural Network with High Accuracy?" in *Association for the Advancement of Artificial Intelligence*, 2017.

[13] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning Both Weights and Connections for Efficient Neural Network," in *Conference on Neural Information Processing Systems*, 2015.

[14] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *European Conference on Computer Vision*, 2016.

[15] X. Lin, C. Zhao, and W. Pan, "Towards Accurate Binary Convolutional Neural Network," in *Conference on Neural Information Processing Systems*, 2017.

[16] F. Li and B. Liu, "Ternary Weight Networks," in *Conference on Neural Information Processing Systems*, 2016.

[17] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained Ternary Quantization," in *International Conference on Learning Representations*, 2017.

[18] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural Networks with Few Multiplications," in *International Conference on Learning Representations*, 2015.

[19] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning Structured Sparsity in Deep Neural Networks," in *Conference on Neural Information Processing Systems*, 2016.

[20] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2015.

[21] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-state Circuits*, vol. 52, no. 1, 2017.

[22] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2017.

[23] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, and S. Song, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2016.

[24] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing Diversity: Enhanced DSP Blocks for Low-precision Deep Learning on FPGAs," in *International Conference on Field-programmable Logic and Applications*, 2018.

[25] A. Boutros, M. Eldafrawy, S. Yazdanshenas, and V. Betz, "Math Doesn't Have to Be Hard: Logic Block Architectures to Enhance Low-precision Multiply-accumulate on FPGAs," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2019.

[26] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong, "PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks," in *International Symposium on Field-programmable Custom Computing Machines*, 2019.

[27] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding Deep Learning Requires Rethinking Generalization," in *International Conference on Learning Representations*, 2017.

[28] S. Chatterjee, "Learning and Memorization," in *International Conference on Machine Learning*, 2018.

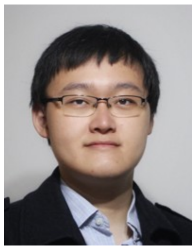
[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, 1998.

[30] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading Digits in Natural Images with Unsupervised Feature Learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[31] J. J. Davis, E. Hung, J. M. Levine, E. A. Stott, P. Y. K. Cheung, and G. A. Constantinides, "KAPow: High-accuracy, Low-overhead Online Per-module Power Estimation for FPGA Designs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 1, 2018.

[32] E. Stott, J. S. J. Wong, and P. Y. K. Cheung, "Degradation Analysis and Mitigation in FPGAs," in *International Conference on Field-programmable Logic and Applications*, 2010.

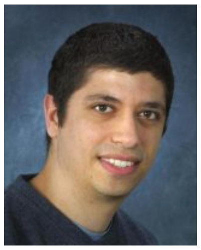
[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture*, 2017.



Erwei Wang is a PhD student in the Department of Electrical and Electronic Engineering's Circuits and Systems group at Imperial College London. His research interests include deep neural networks, computer vision systems and other massively parallel architectures with an emphasis on improving speed and energy efficiency for custom hardware implementation. He is a Student Member of the IEEE.

James J. Davis is a Research Fellow in the Department of Electrical and Electronic Engineering's Circuits and Systems group at Imperial College London. He received a PhD in Electrical and Electronic Engineering from Imperial College London in 2016. His research is focussed on the exploitation of FPGA features for cutting-edge applications, driving up performance, energy efficiency and reliability. Dr Davis serves on the technical programme committees of the four top-tier reconfigurable computing conferences (FPGA, FCCM, FPL and FPT) and is a multi-best paper award recipient. He is a Member of the IEEE and the ACM.

Peter Y. K. Cheung is Professor of Digital Systems in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include VLSI architectures for signal processing, asynchronous systems, reconfigurable computing and architectural synthesis. Prof. Cheung received a DSc from Imperial College London. He is a Senior Member of the IEEE and Fellow of the IET.



George A. Constantinides received the PhD degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Professor of Digital Computation and Head of the Circuits and Systems research group. He was the general chair of the ACM International Symposium on Field-Programmable Gate Arrays in 2015. He serves on several programme committees and has published over 200 research papers in peer-refereed journals and international conferences. Prof. Constantinides is a Senior Member of the IEEE and a Fellow of the BCS.