Dynamic pruning for parsimonious CNN inference on embedded systems

Paola Busia[1], Ilias Theodorakopoulos[2], Vasileios Pothos[2], Nikos Fragoulis[2], and Paolo Meloni[1]

[1] Università degli Studi di Cagliari, Italy
[2] Irida Labs, Greece

Abstract. As a consequence of the current edge-processing trend, Convolutional Neural Networks (CNNs) deployment has spread to a rich landscape of devices, highlighting the need to reduce the algorithm's complexity and exploit hardware-aided computing, as two prospective ways to improve performance on resource-constrained embedded systems. In this work, we refer to a compression method reducing a CNN computational workload based on the complexity of the data to be processed, by pruning unnecessary connections at runtime. To evaluate its efficiency when applied on edge processing platforms, we consider a keyword spotting (KWS) task executing on SensorTile, a low-power microcontroller platform by ST, and an image recognition task running on NEURAGHE, an FPGA-based inference accelerator. In the first case, we obtained a 51% average reduction of the computing workload, resulting in up to 44% inference speedup, and 15% energy-saving, while in the latter, a 36% speedup is achieved, thanks to a 44% workload reduction.

Keywords: Convolutional Neural Networks · Pruning · Hardware acceleration

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have reached outstanding levels of accuracy [1], favoring their success in multiple application fields, from natural language processing, to image classification, and object detection. A turning point was represented by the design of deeper and complex architectures [2], having pushing requirements in terms of storage and computing capabilities. Their deployment on edge resource-constrained systems, encouraged by bandwidth, security, and privacy concerns, poses many challenges and has been a prolific field of research. On the one hand, more efficient hardware architectures, specifically targeting neural networks, have been designed. Industry and academia have proposed multiple dedicated processors and accelerators [3] [4] [5] [6] [7] and embedded GPUs [8], and heterogeneous computing systems exploiting FPGAs and All-Programmable-SoCs to combine parallelism and flexibility [9] [10]. On the other hand, optimized software libraries, specifically targeting a class of devices, have been developed and released, such as CU-DNN for NVIDIA platforms [11], CMSIS [12], targeting class-M ARM low-power microprocessors, and ARM-NN [13], targeting more high-end architectures. Moreover, multiple approaches have been focusing on the simplification of the computing model, to reduce the footprint, or the computing workload of CNNs [14] [15].

In this work, we mean to leverage the combination of such approaches. We take as a reference a technique, proposed in [16], performing an online pruning of a CNN's connections, reducing the computational load associated with convolutional layers, based on run-time processing of the input data. We test its implementation on a resource-constrained commercial platform based on ARM Cortex-M, the ST SensorTile, considering a CNN for a keyword spotting (KWS) task, and evaluating the obtained improvement in terms of execution time and power consumption. We also present an image recognition use case, exploiting a custom network architecture built for CIFAR-10 [17], to evaluate the feasibility of dynamic pruning on a hardware-assisted computing platform, considering as a target NEURAghe, an FPGA-based inference accelerator.

2 Related Work

The efficient execution of CNNs on resource-constrained systems requires careful optimization, both of the computational workload and of the number of accesses to the off-chip memory. The community has addressed this matter by either designing shallower and optimized network architectures [18] [19], or by developing several compression techniques, reducing the number of network's parameters or the precision of their representation [20]. In Table 1 we list some of the most recent works on network compression. For each one, we define in Column 1 the dominant compression method resulting in most of the reported advantages, while in Column 2 we report whether the compression strategy is *static* or *dynamic*, thus evaluated at runtime based on the complexity of the input to be classified. In Column 3 we define the granularity and structure level of the pruning action. In Column 4 we report the performance metrics considered to evaluate and refine the CNN architecture, and finally in Column 5 we list the hardware architectures considered for the analysis.

In [21], the authors present an hybrid neural network, combining the advantages of Strassen representation for matrix multiplications [24] and of Bonsai decision trees [25]. Their proposed compression method exploits ternary representation for most of the weights in convolutional layers, keeping only a few full precision weights, which can be further quantized to 16 or 8-bit precision. The ST-HybridNet reaches 94.71% accuracy, using 2.4 MOPS and requiring a 41.8 kB memory footprint. The compression method exploited in this work is static, and its advantages are mainly due to compression through quantization rather than connections pruning, and the possibility to replace most of the resource-hungry multiplications with additions. Thus, it can be considered as an orthogonal technique to classical pruning methods, and especially to dynamic ones. Furthermore, the advantages of compression are only indirectly analyzed

Work	**Compression Strategy**	**Static/ Dynamic**	**Structure Level**	**Performance Metric**	**Hardware Platform**
[21]	Transformation, Quantization	Static	not applicable	Accuracy, OPS, Footprint	✗
[22]	Pruning, Quantization	Dynamic	Group of channels	Accuracy, OPS, Power	ASIC, GPU, CPU
[23]	Pruning	Dynamic	Group of channels	Accuracy, OPS, Power	GPU
this work	Pruning	Dynamic	Single channel	Accuracy, inference time, Power	FPGA, μC

Table 1: Comparison with state of the art works on CNN compression.

in terms of OPS and footprint reduction, which cannot always be translated into performance improvements, depending on the flexibility of the target library.

In [22], the authors explore channel gating as a dynamic pruning method, to reduce at runtime the network's complexity based on the input's content. Given a baseline network architecture, two different paths are identified, a *base path*, and a *conditional path*. The first one is always computed, and such a partial sum is exploited as an activation rule for the *conditional path*, which is either skipped or selectively executed. In the case of skipped computations, the partial sum is used as an approximation of the final value, thus the workload reduction is not inherited by the successive layers. Thus, in the resulting channel gating networks (CGNets) a structured pruning is enforced, which allows for efficient inference, even on a hardware accelerator. The authors report a 2.8× FLOPS reduction on ResNet-18, resulting in a 2.3× inference speedup on their CGNet accelerator.

In [23], the authors present a dynamic pruning strategy based on reinforcement learning, exploiting a decision network to evaluate the pruning actions on the convolutional layers of the network. For each layer, the output features are grouped into a certain number, k, of sets, and the decision establishes how many of such ordered sets are to be evaluated based on the desired trade-off between performance and accuracy. Thus, it is still possible to define a *base* and a *conditional path*, and the enforced pruning can be defined as structured.

In this work, we consider a dynamic pruning technique [16], which can be exploited concurrently with other compression methods, to further reduce the workload of an efficient network based on the content of the specific input to be processed. Compared to [22] and [23], we focus on a less structured pruning strategy, where the activation rule is applied independently to the single output features of a convolutional layer, and all the combinations can be in theory obtained. The compression effect also impacts the following layers, resulting in fewer valid input features to be computed in the successive convolutions. The work we reference [16] represents an extreme case of dynamic pruning, thus we mean to evaluate whether the resulting OPS reduction can still produce performance improvement. We consider an efficient state-of-the-art library, as CMSIS-NN, enabling it to support the selective evaluation of convolutional kernels, and finally evaluate the advantages of the dynamic pruning on on-hardware direct measurements of inference time and energy consumption. We also consider a convolution accelerator implemented on FPGA and evaluate how the workload reduction impacts its performance.

In the following, we show that significant performance improvement can be achieved by introducing the required support in the hardware architecture or software library performing the convolutions. Our purpose is to:

- test the pruning method in [16] on two use-cases, KWS and image recognition;
- evaluate its effect on two reference hardware platforms, the ST SensorTile and NEURAghe;
- present a method to estimate the Parsimonious Inference's (PI) impact on power consumption.

3 Reference methodology

The common inspiration for pruning methods is CNN computations are often redundant, and some of them can be skipped with little effect on the accuracy. [16] focuses on convolutional layers, which represent the main source of workload for several CNN architectures, and aims at adapting the complexity of the network to the particular item to be classified. The approach exploits a specialized training procedure, where the network model is changed into one enforcing PI through dynamic pruning. As shown in Figure 1, the network architecture is distorted through the insertion of a dedicated software module, the Learning Kernel Activation Module (LKAM), which can be associated with one or more convolutional layers along the network. The LKAM reproduces a simple network model, consisting of a 1x1 convolution, average pooling, and a sigmoid activation, followed by a threshold step. During the inference execution, this lightweight processing is applied to the convolutional layer's input features, resulting in a set of activation flags, provided as an additional input. In detail, given a layer with OF output features, the LKAM computes OF activation flags. At runtime, convolution is evaluated only on the *active* output features, thus skipping the computations associated with particular sets of weights. Since the LKAM output is data-dependent, different levels of deactivation can be obtained for different input items. The computation savings also involve the following layer, which will receive a reduced number of valid input features. The LKAM parameters are learned during the training procedure, aiming at preserving the network's accuracy while maximizing the sparsity of the activation flags. To that end, the chosen loss function, $L_t(w, b)$, needs to be modified through an additional term, indicated as $L_{aug}(sw)$, to consider the LKAM output.

$$L(w, b, sw) = L_{\mathrm{t}}(w, b) + L_{\mathrm{aug}}(sw)$$

Such new term is defined as:

$$L_{aug}(sw) = \frac{G_{\mathrm{i}}}{2m} \sum_{i} |sw_i|$$

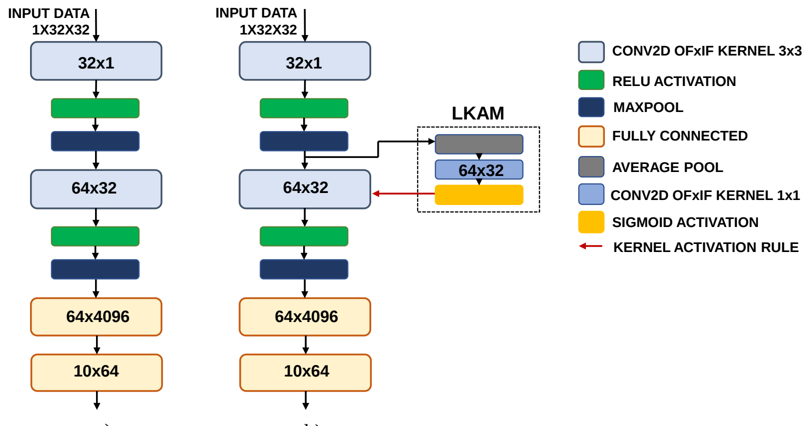


Fig. 1: a) Network architecture for the KWS use-case; b) Parsimonious model obtained through retraining with LKAMs.

where sw represents the LKAM output before the threshold is applied, m is the length of such vector and G_i is a gain factor, that can be tuned independently for each layer of the network. The tool output is a set of new models, integrating different numbers of trained LKAMs in different positions. When a target accuracy is set, the selected model is the one with the highest deactivation percentage within the target accuracy.

When trying to exploit this technique on hardware-aided or parallel computing devices, the main challenge is posed by the need to selectively perform computations, and keep track of the deactivated features when retrieving the appropriate kernels to evaluate across the various layers of the network. A possible drawback is represented by the additional storage space required by the LKAM's parameters. Loss of precision, due to the need for data quantization, is another issue to be considered in both the implementations evaluated in this paper.

4 Reference computing platforms

We present in the following the main features of the platforms considered in this paper, the ST SensorTile and NEURAGHE, and the modifications we introduced to support PI.

4.1 SensorTile

The SensorTile is an IoT module, developed by STMicroelectronics, embedding an 80 MHz ARM Cortex-M4 32-bit low-power micro-controller. The system architecture exploits a Real-Time lightweight Operating System (RTOS), supporting multi-threading and scheduling of the different application tasks on defined timings. To reduce power consumption, it can switch between two main operating states, *run mode* and *sleep mode*, exploited whenever possible in our application, through a specific *idle* task that is entered every time none is pending.

Support for PI. Given the real-time constraints of the KWS task on SensorTile, we exploited CMSIS, a library specifically targeting Cortex-M Processing Cores, including several NN utilities and designed to maximize performance [12]. To obtain the results presented in the last section, we used the basic version of the 8-bit square convolution function provided by the library, *"arm_convolve_HWC_q7_basic"* and customized it to make it able to receive and use the deactivation information produced by the LKAMs. As a first attempt at supporting PI, after scanning the activation flags we split the convolution execution into separate layers, of width given by the number of consecutive active output features.

As shown in Figure 2, where the execution time of layer Conv2 in Table 2 is plotted as a function of the kernels' deactivation percentage, such a solution is not very efficient, preventing the processor to take advantage of the optimized sequence of operations. As an alternative, we acted on the function itself, to introduce the selective evaluation of computational units, while preserving the computational efficiency. We replaced the weight tensor with an array of addresses, each pointing to the active filtering kernels. During convolution execution, the read pointer of weights is assigned a new value from the next location of the addresses tensor. As can be derived from Figure 2, introducing kernel deactivation inside a knowingly optimized function allows obtaining a linear speedup with the deactivation percentage.

4.2 NEURAghe

NEURAGHE is a CNN inference accelerator that can be ported with different parameters on different FPGA devices [10]. The results disclosed in this work come from its implementation on a Xilinx Z-7020 SoC mounted on a Zedboard by Digilent. It exploits a Convolution Engine (CE) embedding a matrix of multipliers, and a programmable micro-controller, efficiently scheduling convolutions and data transfers towards the local storage space accessed by the CE, dedicated to the convolutional weights and activations. In this work, we only refer to the hardware acceleration of convolutional layers, with kernel size 3x3 or 5x5. When receiving the offload command, the micro-controller is provided with the layer's parameters, and the memory addresses from which to read the network's parameters and write the computed results. According to the internal structure of the CE and the size of the local storage space, the micro-controller groups the layer's input and output features, and handles a task-level pipeline, made up of three main stages: 1) *load* of weights and input features; 2) setup and *run* of the CE; 3) *store* of the computed results. The disclosed results refer to a CE implementation embedding a single SoP module, processing 2 output pixels per cycle, and performing three convolutions with 3x3 filtering kernels, or one with 5x5 kernels, in 16-bit fixed-point precision.

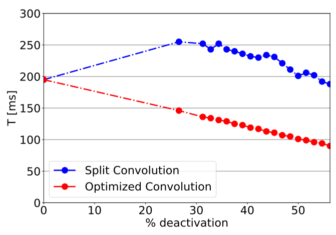


Fig. 2: Execution time of Conv2 layer in the KWS network model, considering different percentages of deactivated kernels and exploiting the two solutions developed to support kernel deactivation.

Support for PI. To efficiently support PI, the hardware/ software architecture needs to selectively load only the sets of weights corresponding to active kernels, while exploiting at best the hardware resources, by setting up the engine with the optimal number of inputs and weights, from the scheduling point of view. First, the layer's description provided to the micro-controller is enriched with two fields, carrying kernel activation information, and consisting of two arrays of flags, storing the output of the LKAM associated with the current layer, and with the preceding one. Given such information, the micro-controller needs to program the CE to only perform the computations corresponding to a valid input feature and active output feature. Furthermore, in the baseline architecture, weight transfers are handled in batches, while PI requires treating each kernel independently, to freely discard those that have been deactivated. To enable this, we changed weight transfer granularity and added the memory-mapped programmable registers needed to control the number of elements to be expected per transfer, and the position of the element that should be interpreted as bias. We also introduced some changes in the middleware executed by the micro-controller, to evaluate the activation flags before the data transfers are programmed. An outer loop scans the flags associated with the input and output features: only when both flags are set to true the address is evaluated, and the transfer is programmed. Finally, we introduced two inner loops on the activation flags, to keep track of the number of programmed transfers, until the necessary number of features, required to run the accelerator at full speed, has been reached.

5 EXPERIMENTAL RESULTS

5.1 KWS on SensorTile

To evaluate the advantages of PI in terms of power consumption, we selected a KWS use-case, deployed on ST SensorTile. Classification is performed through a simple CNN model architecture, trained on the Speech Commands dataset [26], and whose structure is described in Table 2. We refer to the model enriched

Conv1; Convolution	Input Size = 32×32 Input Features = 1	Kernel Size = 3×3 Output Features = 32
Conv2; Convolution	Input Size = 16×16 Input Features = 32	Kernel Size = 3×3 Output Features = 64
Fc1; Fully Connected	Input Size = $64 \times 8 \times 8$	Output Size = 64
Fc2; Fully Connected	Input Size = 64	Output Size = 10

Table 2: Architectural model of the reference CNN for the KWS task.

	Accuracy
Baseline	94,43%
PI	90,54%

Table 3: Accuracy of the baseline and PI version of the CNN for KWS.

Idle	Power: P_{idle} = 28,78 mW
Audio Processing	Energy Contribution: E_{pr} = 3,48 mJ
CNN Classifier	Energy Contribution: E_{cnn} = 8,67 mJ

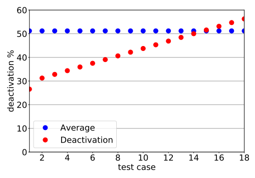
Table 4: On-hardware measurements of power consumption and energy contributions of the examined tasks, performed on the ST SensorTile.

with an LKAM associated with Conv2 as the PI model, resulting in the accuracy drop reported in Table 3. Figure 3a reports the different percentages of deactivation obtained over the dataset, showing an average deactivation of 51% of the convolutional kernels in Conv2, and evaluated on a 16-bit implementation. To assess how power consumption is affected, we refer to an 8-bit implementation, running on the reference platform. We exploited the CMSIS library [12], introducing the modifications described in Section 4.1 to support the selective evaluation of kernels. Figure 3b reports the inference time as a percentage of the baseline execution time, considering the kernels activation percentages in Figure 3a. To translate processing time savings into a reduction of power consumption, we considered a simple application model, involving three main tasks:

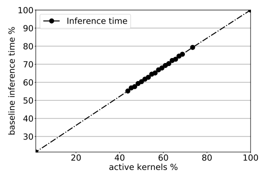
- *Get Data*, performing data acquisition with the desired sampling frequency;
- *Audio Processing*, evaluating the Mel-spectrogram of the sampled audio, provided as input to the CNN;
- *CNN Classifier*, executing the network model for recognition.

The audio processing is performed by evaluating 32 Mel features through 32 temporal frames, covering 1s of sampled audio. Considering real-time execution, processing and classification need to be performed periodically. Since the audio processing time was evaluated to be 104 ms at an 80 MHz working frequency, and the worst-case inference time, required by the baseline model without deactivation, is 290 ms, we considered a fixed execution period of 400 ms. When no software task is executing, the RTOS switches the operating state from *run* to *sleep* mode. As shown in Figure 4, the idle time grows with the percentage of kernel deactivation produced by the input. The effect on the system's power consumption can be estimated through a simple model [27]:

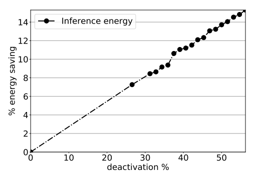
$$P = P_{\text{idle}} + E_{\text{gd}} \times f_{\text{gd}} + (E_{\text{pr}} + E_{\text{cnn}}) \times f_{\text{pr}}$$



(a)



(b)



(c)

Fig. 3: a)Kernel deactivation percentage through the Speech Commands dataset, and average deactivation; b)Percentage execution time over the baseline model execution time, for different kernel activation levels; c)Percentage energy saving over the baseline model power consumption, for different percentages of kernel deactivation.

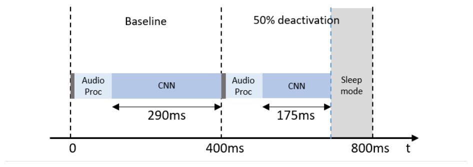


Fig. 4: Real-time inference cycle, considering the baseline model and a case of 50% deactivation in Conv2, allowing to save power by entering the sleep mode.

where f_{pr} is the execution frequency of processing and classification, while with E_{pr} and E_{cnn} we refer to the energy contribution of the different tasks, that add up to the idle power consumption of the system. Table 4 reports the idle power consumption and the energy contribution of the considered tasks, evaluated by measuring with an oscilloscope the current absorbed by the device in different working conditions. Figure 3c reports the energy consumption, as a percentage of E_{cnn} in Table 4, based on the fixed deactivation patterns in Figure 3a, and referring to a loop execution of the CNN classifier. The results in this section show that the examined pruning method can produce remarkable hardware performance improvements, despite the high level of granularity.

5.2 CIFAR-10 on NEURAghe

Here we explore the feasibility of applying PI on a hardware assisted architecture, considering an image recognition task executing on NEURAGHE, and exploiting a simple network model, LeNet, trained on CIFAR-10 [17]. The network structure is summarized in Table 5. The PI version of the model was obtained by associating LKAMs to the second and third convolutional layers, with an accuracy drop of around 3%. Table 6 reports three test cases, producing different deactivation patterns, and resulting in a different inference speedup, evaluated considering: 1) software execution on ARM Cortex A9 processor (667 MHz); 2)

Conv1; Convolution	Input Size = 32 × 32 Input Features = 3	Kernel Size = 5 × 5 Output Features = 32
Conv2; Convolution	Input Size = 16 × 16 Input Features = 32	Kernel Size = 5 × 5 Output Features = 32
Conv3; Convolution	Input Size = 8 × 8 Input Features = 32	Kernel Size = 5 × 5 Output Features = 64
Fc1; Fully Connected	Input Size = 64 × 4 × 4	Output Size = 64
Fc2; Fully Connected	Input Size = 64	Output Size = 10

Table 5: Architectural model of the considered LeNet for CIFAR-10.

	Test 1			**Test 2**			**Test 3**		
	Conv 2	*Conv 3*	*Tot*	*Conv 2*	*Conv 3*	*Tot*	*Conv 2*	*Conv 3*	*Tot*
Deactivation	31.3%	31.3%	30.3%	43.8%	48.1%	45.3%	40.6%	40.6%	39.4%
ARM speedup	30%	27%	29%	43%	44%	43%	40%	36%	39%
CE speedup	19%	24%	22%	33%	36%	35%	31%	30%	30%

Table 6: Kernel deactivation percentage in the three test cases, and PI execution speedup on ARM Cortex A9 (667 MHz) and NEURAghe Convolution Engine on Xilinx Zynq Z-7020.

hardware execution exploiting NEURAghe's CE. As shown in the Table, in the case of software execution the additional overhead introduced by dynamic pruning is completely compensated by the reduction of computations to be performed, producing significant advantages in the overall execution. When convolutions are handled on FPGA, kernel deactivation still results in performance improvement, but it is less effective, because of a programming overhead, whose burden does not depend on the size of the workload.

6 CONCLUSION

In this work, we presented the implementation of dynamic neural network pruning through data-driven kernel deactivation on two resource-constrained platforms, exploiting different computing units, SensorTile and NEURAghe. We referred to two common application fields of CNNs, such as image recognition and KWS, and considered custom network architectures. Referring to common datasets, we found that the method allows an average deactivation of 51% of the convolutional kernels in the KWS task. The experimental results show that the reduced computational load creates the possibility to reduce the system's power consumption, up to 15% of energy-saving, corresponding to a 44% speedup. The data on NEURAghe implementation show it is possible to exploit dynamic deactivation even when adopting FPGA acceleration, although with less effective improvements. In this case, a maximum 36% speedup due to a 44% deactivation is obtained.

References

1. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, p. 436–444, May 2015.

2. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.
3. N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *ISCA '17: Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, June 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080246
4. E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 420–434, Feb 2018.
5. G. Desoli *et al.*, "14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 238–239, 2017.
6. Movidius. (2020) Movidius neural compute stick: Accelerate deep learning development at the edge. [Online]. Available: https://developer.movidius.com/
7. Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.
8. NVIDIA. (2020) Nvidia deep learning accelerator. [Online]. Available: https://developer.nvidia.com/embedded/buy/tegra-k1-processor
9. M. Blott, T. Preusser, N. Fraser, G. Gambardella, K. O'Brien, and Y. Umuroglu, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Dec 2018. [Online]. Available: https://doi.org/10.1145/3242897
10. P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "Neuraghe : Exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Dec 2018. [Online]. Available: https://doi.org/10.1145/3284357
11. NVIDIA. (2020) cudnn. [Online]. Available: https://developer.nvidia.com/cudnn
12. L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *CoRR*, vol. abs/1801.06601, 2018. [Online]. Available: http://arxiv.org/abs/1801.06601
13. ARM-NN. (2020) Arm-nn. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/machine-learning/arm-nn
14. S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ISCA '16: Proceedings of the 43rd International Symposium on Computer Architecture*, p. 243–254, June 2016. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30
15. S. Han *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84, Feb 2017. [Online]. Available: https://doi.org/10.1145/3020078.3021745
16. I. Theodorakopoulos, V. Pothos, D. Kastaniotis, and N. Fragoulis, "Parsimonious inference on convolutional neural networks: Learning and applying on-line kernel activation rules," *CoRR*, vol. abs/1701.05221, 2017. [Online]. Available: https://arxiv.org/abs/1701.05221
17. A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf

18. F. N. Iandola *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360
19. Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. arXiv:1711.07128, 2017. [Online]. Available: https://arxiv.org/abs/1711.07128
20. S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *International Conference on Learning Representations '16*, Oct 2015. [Online]. Available: https://arxiv.org/abs/1510.00149
21. D. Gope, G. Dasika, and M. Mattina, "Ternary hybrid neural-tree networks for highly constrained iot applications," 2019.
22. W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, "Boosting the performance of cnn accelerators with dynamic fine-grained channel gating," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 139–150. [Online]. Available: https://doi.org/10.1145/3352460.3358283
23. J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/a51fb975227d6640e4fe47854476d133-Paper.pdf
24. M. Tschannen, A. Khanna, and A. Anandkumar, "StrassenNets: Deep learning with a multiplication budget," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 4985–4994. [Online]. Available: https://proceedings.mlr.press/v80/tschannen18a.html
25. A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the internet of things," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 1935–1944. [Online]. Available: https://proceedings.mlr.press/v70/kumar17a.html
26. P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *CoRR*, vol. arXiv:1804.03209, 2018. [Online]. Available: https://arxiv.org/abs/1804.03209
27. M. A. Scrugli, D. Loi, L. Raffo, and P. Meloni, "A runtime-adaptive cognitive iot node for healthcare monitoring," *Proceedings of the 16th conference on Computing Frontiers (CF '19)*, pp. 350–357, April 2019. [Online]. Available: https://doi.org/10.1145/3310273.3323160