Incremental Training and Group Convolution Pruning for Runtime DNN Performance Scaling on Heterogeneous Embedded Platforms

Lei Xun, Long Tran-Thanh, Bashir M Al-Hashimi, Geoff V. Merrett
School of Electronics and Computer Science
University of Southampton
Southampton, UK
{lx2u16, ltt08r, bmah, gvm}@ecs.soton.ac.uk

***Abstract*—Inference for Deep Neural Networks is increasingly being executed locally on mobile and embedded platforms due to its advantages in latency, privacy and connectivity. Since modern System on Chips typically execute a combination of different and dynamic workloads concurrently, it is challenging to consistently meet inference time/energy budget at runtime because of the local computing resources available to the DNNs vary considerably. To address this challenge, a variety of dynamic DNNs were proposed. However, these works have significant memory overhead, limited runtime recoverable compression rate and narrow dynamic ranges of performance scaling. In this paper, we present a dynamic DNN using incremental training and group convolution pruning. The channels of the DNN convolution layer are divided into groups, which are then trained incrementally. At runtime, following groups can be pruned for inference time/energy reduction or added back for accuracy recovery without model retraining. In addition, we combine task mapping and Dynamic Voltage Frequency Scaling (DVFS) with our dynamic DNN to deliver finer trade-off between accuracy and time/power/energy over a wider dynamic range. We illustrate the approach by modifying AlexNet for the CIFAR10 image dataset and evaluate our work on two heterogeneous hardware platforms: Odroid XU3 (ARM big.LITTLE CPUs) and Nvidia Jetson Nano (CPU and GPU). Compared to the existing works, our approach can provide up to 2.36x (energy) and 2.73x (time) wider dynamic range with a 2.4x smaller memory footprint at the same compression rate. It achieved 10.6x (energy) and 41.6x (time) wider dynamic range by combining with task mapping and DVFS.**

***Index Terms*—Embedded Deep Learning, Dynamic Deep Neural Network, Runtime Performance Trade-off**

I. Introduction

In the last few years, Deep Neural Networks (DNNs) [1] have gained lots of attention and been widely adopted in many computer vision tasks such as image classification [2], object detection [3] and face recognition [4] due to their ability to deliver near or super-human accuracy.

The execution of DNNs has two stages: training and inference. At the training stage, DNNs learn to perform a task. Millions of DNN parameters need to be adjusted during this process. Therefore training is usually executed on powerful desktop/server GPU(s). At the inference (also known as testing) stage, DNNs perform the task on unseen data using pre-trained parameters. Inference can also be executed on desktop/server GPU(s), but it is increasingly being executed locally on mobile and embedded platforms due to its advantages in latency, privacy and connectivity [5], [6].

TABLE I
Mean inference time for a single CIFAR10 image using AlexNet

Platform	Computing cores	Time (ms)	Accuracy (%)
Jetson Nano	GPU (614MHz)	7.21	71.2
	GPU (921MHz)	4.88	
	A57 CPU (921MHz)	70.4	
	A57 CPU (1.43GHz)	46.8	
Odroid XU3	A15 CPU (200MHz)	1020	
	A15 CPU (1.8GHz)	117	
	A7 CPU (200MHz)	1780	
	A7 CPU (1.3GHz)	280	

The performance of inference can be defined using platform-dependent metrics such as execution time and energy consumption, and platform-independent metrics such as classification accuracy and confidence. As shown in Table I, when the same image classification DNN is deployed on two different platforms, the inference time varies considerably, whereas the accuracy remains the same.

Modern mobile and embedded System on Chips (SoCs) typically execute a combination of different and dynamic workloads concurrently on heterogeneous computing cores (e.g. CPU, GPU, NPU). The use of runtime resource management techniques (e.g. task mapping and Dynamic Voltage Frequency Scaling (DVFS)) make these SoCs highly efficient. It is challenging to consistently meet inference time/energy budget at runtime because of the local computing resources available to the DNNs vary considerably. For example, the target computing cores might be unavailable due to other applications running on them, or are available at a lower voltage/frequency level because of other computing cores execute

978-1-7281-5758-0/19/$31.00 ©2019 IEEE

in the same voltage/frequency domain, or fixed power/thermal budgets. To address this challenge, a variety of dynamic DNNs were proposed to cover the performance variance at runtime. The computation workload of dynamic DNNs can be scaled through model compression at runtime to meet the performance budget. However, these works have significant memory overhead, limited runtime recoverable compression rate (RRCR) and narrow dynamic ranges of performance scaling. RRCR is defined as the percentage of filters/channels can be pruned and added during runtime without model retraining,

In this paper, we present a dynamic DNN using incremental training and group convolution pruning. The channels of the DNN convolution layer are divided into groups, which are then trained incrementally. At runtime, following groups can be pruned for inference time/energy reduction or added back for accuracy recovery without model retraining. Compared to the existing works, our approach can provide up to 2.36x (energy) and 2.73x (time) wider dynamic range with a 2.4x smaller memory footprint at the same compression rate. Moreover, previous works did not consider the runtime resource management techniques on SoCs. We combine our dynamic DNN with task mapping and DVFS to deliver a finer trade-off over performance metrics, and up to 10.6x (energy) and 41.6x (time) wider dynamic ranges.

The contributions of this paper are:

- Proposed an approach for building dynamic DNNs using group convolution with incremental training.
- The first implementation of dynamic DNN with task mapping and DVFS to achieve finer performance trade-offs and wider dynamic ranges of performance scaling than standalone dynamic DNN approaches.

II. Related Work

Modern SoCs typically execute a combination of different dynamic workloads concurrently, and hence the local resources available to the DNN vary considerably at runtime. Static DNN compression [5], [7] generates one DNN for a given performance budget at a pre-defined hardware setting (e.g. computing core and voltage/frequency level). This raises a significant problem since the performance budgets cannot be met when the pre-defined hardware setting is unavailable at runtime. Multiple DNNs are needed to cover all hardware settings, which result in significant memory storage overhead. Furthermore, the switching activities of these DNNs at runtime may cause significant delay and energy consumption [8].

To address this problem, a variety of dynamic DNNs were proposed. Dynamic DNNs can be partially executed at runtime to meet the performance budget using available resources. Xu *et al.* [6] proposed a two-step DNN compression scheme. At design time, static DNN compression similar to [5] is used to adapt DNN on target hardware. At runtime, filters in the static compressed DNN are pruned further to meet dynamic budgets, or are added back for accuracy recovery without model retraining. Although this approach allows DNNs to adapt to dynamic resource variance (e.g. cores, frequency), the limited RRCR leads to a limited dynamic range, at most 20% of the filters are pruned for a 25% time/energy reduction. Such a narrow dynamic range is not enough to cover the performance variance when DNN is mapped on computing cores with lower performance than the pre-defined one, and/or with lower voltage/frequency level as shown in Table I. Therefore, this approach still has significant memory storage overhead since multiple DNNs are needed to cover all hardware settings in modern SoCs.

In order to achieve a greater RRCR and wider dynamic range, DNNs can be initially designed to support runtime trade-off. Tann *et al.* [9] proposed a dynamic DNN using channel-wise incremental training. Unlike regular training which trains all channels and layers at the same time, channel-wise incremental training trains part of channels of all layers at a time. For example, For a four-increment dynamic DNN, 25% of the channels of all layers are trained first, then another 25% of the channels are trained while incorporates the pre-trained and frozen 25% of channels, and so on so forth. After the training is finished, four discrete DNN configurations with different size/accuracy/time/energy are generated and stored as a single model. At runtime, the DNN is partially or fully executed depending on performance budgets and available resources. However, because even a smallest DNN configuration (e.g. with 25% of channels) is required to perform the complete task (i.e. should have enough capacity for all data without under-fitting), this approach requires using an oversized model.

Although our work is similar to that in Tann *et al.* [9], there are two main differences: 1) our work uses group convolution (Fig 1(a)). Each channel-wise incremental training is encapsulated in a group, and there are no connections between groups. This makes our design 2.4x smaller at the same compression rate. 2) we combine our dynamic DNN with task mapping and DVFS to deliver finer performance trade-offs over wider dynamic ranges.

III. Incremental Training and Group Convolution Pruning

This section introduces our approach of building a dynamic DNN from existing state-of-the-art DNN architectures. The building procedure has two steps: model preparation and group-wise incremental training. Once the model is trained, it can then be scaled at runtime to meet performance budgets using dynamically available resources without further model retraining. Moreover, the dynamic DNN is combined with task mapping and DVFS to deliver finer performance trade-offs over wider dynamic ranges.

A. Model Preparation

Our dynamic DNN approach is based on group convolution, which was first introduced in AlexNet [2]. AlexNet has two groups that are deployed on two GPUs separately due to limited GPU memory at that time. DNNs with group convolution are smaller and faster than their original configuration, as the dense connections between groups are disconnected, the model can have significantly fewer parameters. Group convolution

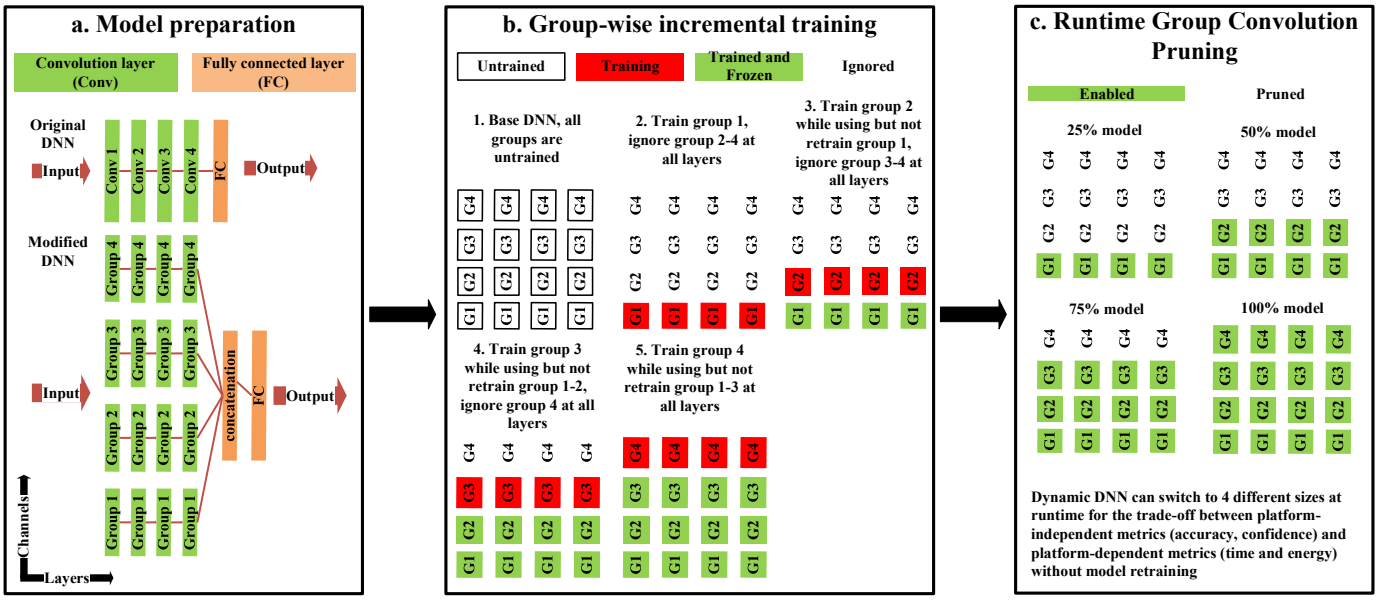

Fig. 1. Our dynamic DNN using incremental training and group convolution pruning. The channels of the DNN convolution layer are (a) divided into groups, which are then (b) trained incrementally. At runtime, (c) following groups can be pruned for inference time/energy reduction or added back for accuracy recovery without model retraining.

has been used as the foundation of many state-of-the-art DNN architectures [10], [11].

The concept of our model preparation approach is illustrated in Fig 1(a). The channels of the DNN convolution layers are divided into groups which are then concatenated before the fully connected layer. Each group has its own activation layer (e.g. rectified linear unit (ReLU)), and there are no connections between groups. For DNNs that initially have branches [12] or residual connections [13], the model preparation can be seen as capturing the topology of the model as a building block which is then copied and concatenated, and the size of each building block is also reduced accordingly. This is equivalent to group convolution as shown in Xie *et al.* [11]. However, unlike previous works which train all groups concurrently [10], [11], our work trains these groups incrementally.

B. Group-wise Incremental Training

Each group tends to learn very different features due to less feature sharing between groups [2]. Based on this observation, incremental training [9] is incorporated in our work, and we use four increments in this paper to illustrate our approach. As shown in Fig 1(b-1), a base DNN is obtained from model preparation step (Fig 1(a)) and all of its groups are untrained. The training in our approach has four steps:

- **Step 1:** Train group 1 of all layers, ignore all other groups through initialising all their parameters with zero value so that they do not affect the output (Fig 1(b-2)).
- **Step 2:** Train group 2 of all layers while incorporate pre-trained group 1, ignore group 3-4 (Fig 1(b-3)).
- **Step 3:** Train group 3 of all layers while incorporate pre-trained group 1-2, ignore group 4 (Fig 1(b-4)).
- **Step 4:** Train group 4 of all layers while incorporate pre-trained group 1-3 (Fig 1(b-5)).

Each training steps generate a model that uses a different number of groups. Therefore these four models have different accuracy, confidence and computation requirements. Unlike four separated models, these models can be seen as four different DNN configurations within a single model. In this paper, we refer them as the 25%, 50%, 75% and 100% models as shown in Fig 1(c). All parameters are initialised randomly during training except those are in the ignored groups. The pre-trained groups are frozen so they cannot be changed during later training. The learning rate of the FC layer is reduced with every increment so that following groups have less impact on the 25% model. This helps to make sure later training only add new knowledge to the previous knowledge rather than delete them. Training steps 3-5 are executed multiple times until a target accuracy improvement is met.

C. Runtime Group Convolution Pruning

Once all incremental training is finished, the model can be pruned progressively, one group at a time, as shown in Fig 1(c). For example, 25% model uses only using one group of DNN parameters. Therefore it is the least accurate model but requires minimum computation. The 100% model is the full model which is the most accurate and computationally expensive model. At runtime, the dynamic DNN can switch between these model configurations to explore the trade-off between platform-independent metrics (accuracy and confidence) and platform-dependent metrics (time and energy) without requiring model retraining.

TABLE II
MODIFIED ALEXNET FOR CIFAR10

Layer	Configuration	Output size
Input	N/A	[32*32*3]
Conv1 + ReLU	kernel=3, stride=1, pad=0	[30*30*16]*4
Norm1	Local=5, alpha=0.0001, beta=0.75	[30*30*16]*4
MaxPool1	kernel=4, stride=1	[27*27*16]*4
Conv2 + ReLU	kernel=5, stride=1, pad=2	[27*27*16]*4
Norm2	Local=5, alpha=0.0001, beta=0.75	[27*27*16]*4
MaxPool2	kernel=3, stride=2	[13*13*16]*4
Conv3 + ReLU	kernel=3, stride=1, pad=1	[13*13*16]*4
Conv4 + ReLU	kernel=3, stride=1, pad=1	[13*13*16]*4
Conv5 + ReLU	kernel=3, stride=1, pad=1	[13*13*16]*4
MaxPool5	kernel=3, stride=2	[6*6*16]*4
Concatention	N/A	[6*6*64]
FC6	N/A	10
Softmax	N/A	10

D. Task mapping and DVFS

One disadvantage of incremental training is that the generated DNN configurations have sparse trade-offs. Since our approach is not tied to specific hardware, it can be combined with task mapping and DVFS to deliver finer performance trade-offs.

The effectiveness of our incremental training is validated through accuracy and confidence tests over four model configurations. Furthermore, runtime group convolution pruning is deployed on two heterogeneous embedded platforms and validated through empirical measurements of time/energy. More details are covered in the next section.

IV. EMPIRICAL VALIDATION

A. Dynamic DNN Implementation Details

We illustrate our approach using AlexNet [2] for the CIFAR10 image classification dataset [14] with the Caffe framework [15]. AlexNet is originally designed for the ILSVRC 2012 dataset [16] which contains around 1.3 million training images and 50,000 test images (image size 256*256*3) over 1000 image classes, whereas CIFAR10 only contains 50,000 training images and 10,000 test images (image size 32*32*3) over 10 image classes. Using AlexNet for CIFAR10 directly is unnecessary, and will most likely result in overfitting. Therefore, in our design, the layer size in AlexNet is reduced, but the topology of AlexNet is kept (except the reduced number of FC layers to fit CIFAR10). The detailed architecture is listed in Table II. The model is trained incrementally, and around 50 epochs at each training step. In addition, intermediate models are saved during each training step. In training step 2, the model with the highest validation accuracy is selected as seed for the next increment. However, in training steps 3-5, a seed model is selected only when a target accuracy improvement is met. If no intermediate models meet the target, this step is then repeated.

B. Experimental Setup

Our model is validated on two heterogeneous embedded platforms: the Nvidia Jetson Nano and Odroid XU3. On the Jetson Nano, the ARM A57 CPU and 128-core Maxwell GPU are used; both are configured with two different frequency levels. On the XU3, the ARM A15 and A7 CPUs are used with 17 and 12 different frequency levels respectively. All measurements using CPUs use only a single core because the program is single-threaded, and the batch size for GPUs is 1.

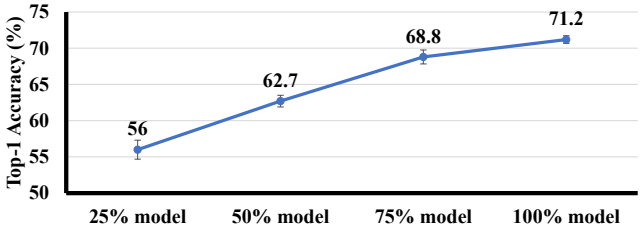

Fig. 2. Top-1 image classification accuracy on 10,000 CIFAR10 validation images. Our Dynamic DNN has four different model configurations which have four different accuracies. At runtime, dynamic DNN can switch to smaller configuration for time/energy reduction with accuracy loss, or switch back to larger models for the accuracy recovery once more computing resources become available.

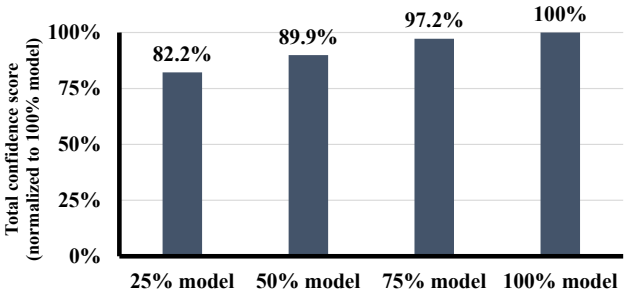

Fig. 3. The total confidence score of the correct DNN output over 10,000 CIFAR10 validation images. The value is normalised to the 100% model. The confidence is improved when more groups are added to dynamic DNN, this indicates different feature filters are learnt in following groups.

C. Inference Top-1 Accuracy and Confidence

To validate our work, accuracy and confidence are measured over four different DNN configurations. Accuracy measures how accurate a model is for a given validation image dataset, and it is defined as:

$$\text{Accuracy} = \frac{\text{Number of correctly classified images}}{\text{Total number of images}}$$

Each output of the DNN has a value, and the confidence score of an output is defined as:

$$\text{Confidence score} = \frac{\text{Value of the output}}{\text{Sum of the value of all outputs}}$$

The confidence score of an output is higher when more image filters are matched with the patterns of the image class in input images and the feature maps at following layers. Top-1 accuracy is defined as the percentage of images that are

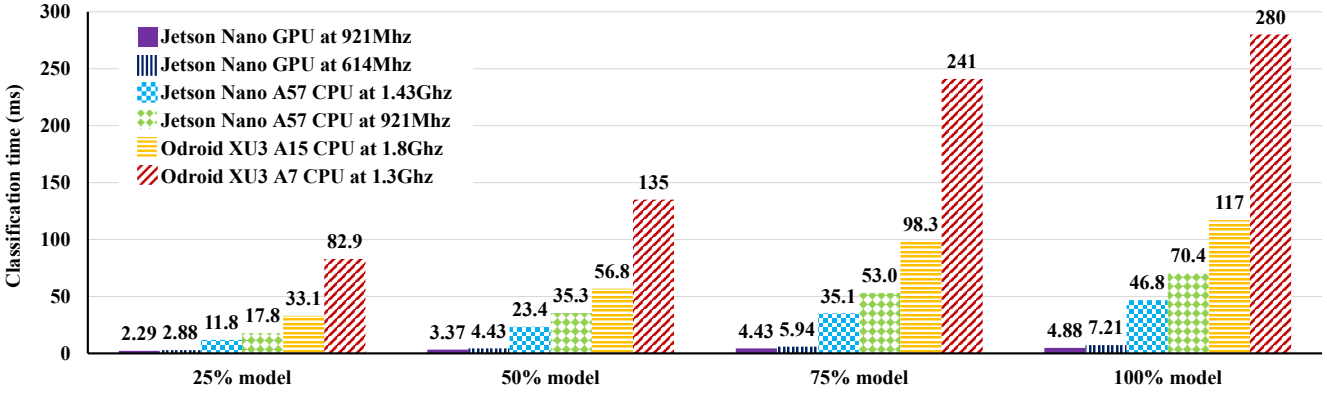

Fig. 4. Inference time on four heterogeneous cores of two hardware platforms. Our group convolution pruning is fully compatible with both CPU and GPU.

classified correctly (also with the highest confidence score) in the entire dataset.

Our dynamic DNN has four different model configurations which have four different accuracies, as shown in Fig 2. The error bar shows the variance over 10 image classes of CIFAR10. Although there is accuracy loss during configuration switching, these losses are fully recoverable without any retraining. Fig 3 shows the improvement in total confidence score over 10,000 CIFAR10 validation images. The improvement is normalised to the 100% model. Confidence is improved when more groups are added to the dynamic network. This matches our expectation since different groups tend to learn different feature filters, and the confidence score is improved when more filters are matched. At runtime, the dynamic DNN can switch to a smaller configuration using group convolution pruning for time/energy reduction with accuracy loss, or switch back to larger models for the accuracy recovery once more computing resources become available.

D. Adapting to Different Hardware Platforms and Heterogeneous Cores

In this experiment, runtime group convolution pruning and its dynamic range are tested by deploying the dynamic DNN on four heterogeneous cores of two hardware platforms. As shown in Fig 4, the inference time is platform-dependent, therefore using a single model to achieve consistent time budget is hard since different platforms/cores have considerably different computing capabilities. Our dynamic DNN has up to 2.5x and 4x dynamic range on GPU and CPU, respectively. These dynamic ranges support the same model to be deployed on different computing cores across hardware platforms while achieving the same time budget. For example, for a time budget of 33ms (30 fps), the dynamic DNN can be deployed on Odroid XU3 A15 CPU using the 25% model, or on the Jetson Nano GPU using the 100% model at a lower frequency setting or A57 CPU using a 50% model if the GPU is unavailable.

E. Combining Dynamic DNN with Task mapping and DVFS

This experiment further explores the combination of our dynamic DNN, task mapping and DVFS. The proposed dynamic DNN is deployed on both A15 and A7 CPUs of Odroid XU3 with 17 and 12 different frequency levels respectively. At runtime, combinations of these three adjustable "knobs" can be applied to meet considerably different energy (Fig 5a), power (Fig 5b) and time budgets. A standalone dynamic DNN with four increments can only provide four trade-off points over a limited dynamic range. With the combination of task mapping and DVFS, the trade-offs are finer and the dynamic ranges are wider.

F. Comparison against State-of-the-art Works

We compare our work with existing works, and the results are shown in Table III. Our approach provides large RRCR since the DNN is trained incrementally to support runtime pruning. Compared to Tann *et al.* [9], our work has the same 75% RRCR due to the same four increments DNN design. Our approach can provide up to 2.36x (energy) and 2.73x (time) wider dynamic ranges with a 2.4x smaller memory footprint due to the use of group convolution. Moreover, our work can achieve 10.6x (energy) and 41.6x (time) wider dynamic ranges by combining with task mapping and DVFS. Filter pruning based compression [6] has limited RRCR since most filters are needed to guarantee the DNN is still able to classify all image classes. Aggressive pruning can result in significant

TABLE III
COMPARISON WITH EXISTING WORKS

DNN	RRCR[1]	Dynamic range	Model size (KB)
Xu *et al.* [6]	20%	0.25x (time) 0.25x (energy)	773.9*N[3]
Tann *et al.* [9]	75%	1.29x (time) 1.49x (energy)	773.9
Proposed w/o D&T[2]	75%	3.53x (time) 3.53x (energy)	318.4
Proposed with DVFS	75%	30.8x (time) 6.76x (energy)	318.4
Proposed with D&T	75%	53.7x (time) 15.73x (energy)	318.4

[1]RRCR is runtime recoverable compression rate.
[2]D&T is DVFS and task mapping.
[3]N is set to 29 to cover all hardware settings on Odroid XU3.

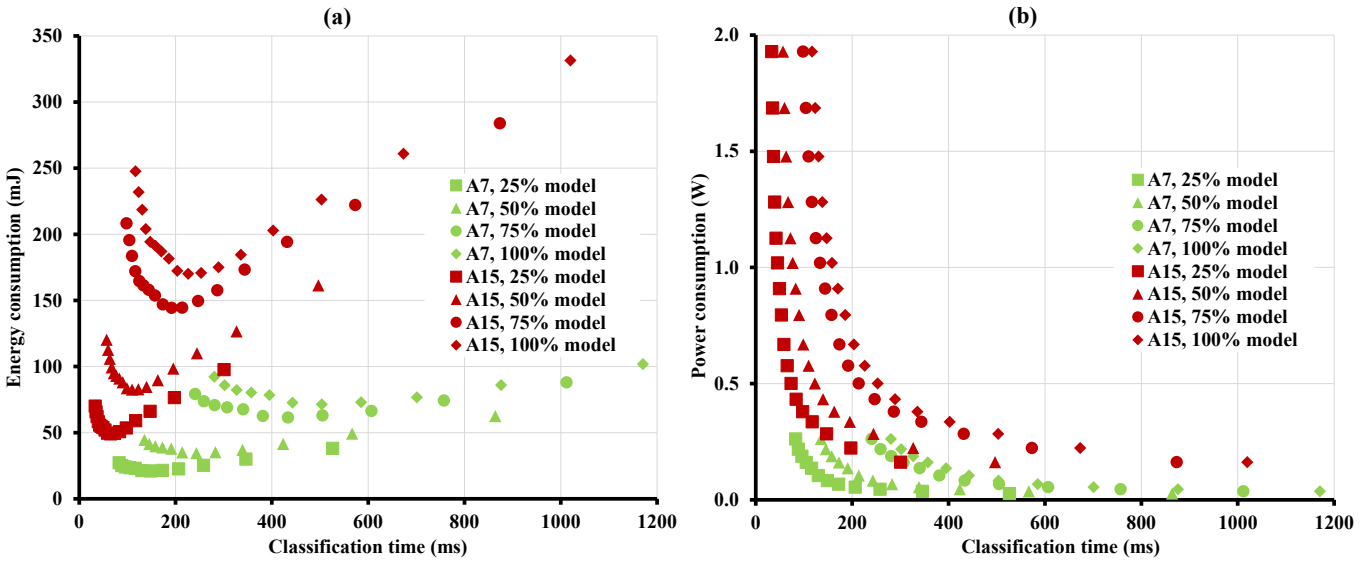

Fig. 5. Dynamic DNN (different symbols) is combined with task mapping (different colours) and DVFS (different points). Different configurations are used for different runtime energy (a), power (b) and time budget. A standalone dynamic DNN with four increments can only provide four trade-off points over a limited dynamic range. With the combination of task mapping and DVFS, the trade-offs are finer and the dynamic ranges are wider.

accuracy loss in some image classes where the features cannot be detected due to the missing filters. In addition, this approach optimises DNN for a pre-defined hardware setting (e.g. core, frequency). Therefore, it cannot be applied with task mapping and DVFS, and has significant memory storage overhead since multiple models are needed to cover all hardware settings.

V. Conclusion

In this paper, we proposed a dynamic DNN using incremental training and group convolution pruning. The channels of the DNN convolution layer are divided into groups, which are then trained incrementally. At runtime, following groups can be pruned for inference time/energy reduction or added back for accuracy recovery without model retraining. Compared to state-of-the-art, our approach can provide up to 2.36x (energy) and 2.73x (time) wider dynamic ranges with a 2.4x smaller memory footprint at the same compression rate. Moreover, our work can achieve 10.6x (energy) and 41.6x (time) wider dynamic ranges by combining with task mapping and DVFS.

VI. Acknowledgement

This work was supported in part by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/S030069/1. Experimental data can be found at DOI: 10.5258/SOTON/D1245.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[4] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014, pp. 1701–1708.

[5] T.-J. Yang *et al.*, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[6] Z. Xu, F. Yu, C. Liu, and X. Chen, "Reform: Static and dynamic resource-aware dnn reconfiguration framework for mobile device," in *Design Automation Conference (DAC)*, 2019, p. 183.

[7] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

[8] E. Park, D. Kim, S. Kim, Y.-D. Kim, G. Kim, S. Yoon, and S. Yoo, "Big/little deep neural network for ultra low power inference," in *International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 2015, pp. 124–132.

[9] H. Tann, S. Hashemi, R. Bahar, and S. Reda, "Runtime configurable deep neural networks for energy-accuracy trade-off," in *International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2016, p. 34.

[10] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6848–6856.

[11] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1492–1500.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[14] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 and CIFAR-100 datasets," [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[16] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.