Dynamic Channel Execution: on-device Learning Method for Finding Compact Networks

Simeon E. Spasov **Pietro Liò**
Department of Computer Science and Technology, University of Cambridge
`ses88@cam.ac.uk, pl219@cam.ac.uk`

Abstract

Existing methods for reducing the computational burden of neural networks at run-time, such as parameter pruning or dynamic computational path selection, focus solely on improving computational efficiency during inference. On the other hand, in this work, we propose a novel method which reduces the memory footprint and number of computing operations required for training *and* inference. We propose a framework integrated with the general training process which can identify a subnetwork within a baseline model which suffers from little or no performance degradation. Given a hard parameter constraint, our method constructs a computational path comprising only highly salient convolutional filters at each training iteration. Our methodology is designed such that a resource-constrained device would run forward and backward passes only on this compact network. As training evolves our algorithms learns to correctly identify the set of most salient channels. We validate our approach empirically on state-of-the-art CNNs - VGGNet, ResNet and DenseNet, and on several image classification datasets. Results demonstrate our framework for dynamic channel execution identifies compact subnetworks which reduce the computational cost up to $4\times$ and parameter count up to $9\times$, thus enabling efficient on-device learning.

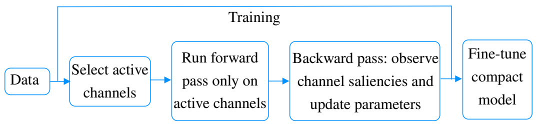
1 Introduction

In recent years, network architectures have become deeper and more complex, yielding highly overparametrized models with a higher memory footprint and many floating-point operations. These requirements have restricted the application of CNNs on resource-constrained devices. Therefore, many techniques, such as pruning [1], which eliminate unnecesary parameters at run-time have been studied. These current methods have solely focused on making inference more cost-efficient. In this work, we investigate how to lower the memory and computational demands of CNNs during *training* as well as inference. Our proposed framework considers the following scenario: given a hard parameter count constraint, how can we identify a computational path of convolutional filters in a CNN, i.e. a subnetwork contained within a baseline model, which results in minimal or no loss in performance? Our methodology is integrated in the training process by selectively activating and executing a subset of all convolutional channels at each training step (see Fig. 1). This is equivalent to introducing a pattern of structural sparsity at the filter level in the baseline network model at each training step. Hence, our methodology can be viewed as an attempt to train a structurally sparse network from scratch without firstly identifying "winning tickets" via pruning as in the Lottery Ticket Hypothesis [2] but rather by finding highly salient filters on the fly. Alternatively, our method can also be used as an efficient on-device learning technique since it reduces the computational and memory requirements for training. In this use case we assume that at each iteration only the compact subnetwork is copied onto the training device performing the forward and backward passes.

In our proposed methodology training is performed as follows: firstly, we formulate the channel selection problem at each training iteration as a combinatorial multi-armed bandit problem, where each arm is a convolutional filter, and propose to use the combinatorial upper confidence bound algorithm (CUCB) [3] to solve it. We use a saliency criterion introduced by [4] to track the relative contribution of executed channels to the network performance. In our work we assume that a salient channel ought to possess discriminative power regardless of its position in the network. Before each training step we employ the CUCB algorithm to select a subset of channels to activate based on the saliency information we have gathered from the beginning of the training process. We then run forward and backward passes only on these activated channels, hence executing them and learning their parameters. After identifying the most salient convolutional channels, we fine-tune their weights after freezing the network topology. As an example, we were able to identify a subnetwork within ResNet-50 trained on the SVHN dataset from scratch which outperforms the baseline model but comprises $4\times$ fewer parameters and utilizes $3\times$ fewer floating-point operations.

2 Methodology

We consider a supervised learning problem with a set of training examples $\mathcal{D} = \{\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\mathtt{N}\}, \mathbf{Y} = \{y_1, y_2, \ldots, y_\mathtt{N}\}\}$, where $\mathbf{x}$ and y represent an input and a label, respectively. Given a CNN model with L convolutional layers, let each layer $l \in 1 \ldots L$ comprise K_l channels, C_l^k, where $k \in 1 \ldots K_l$ is the channel index. In each training step t, we 1) sample a batch of B data samples ($\mathbf{x}^{1:B}$, $\mathbf{y}^{1:B}$); 2) select and activate a subset S of convolutional channels; 3) run a forward and a backward pass on the compact subnetwork, that is only on the active channels; and 4) observe the revealed saliency estimates ($\mathrm{SAL}_{l,t}^k$) of the activated channels. The saliency metric we employ was proposed by Molchanov *et al.* [4] and approximates the change in loss incurred from removing a particular channel. We maintain an empirical channel saliency mean $\hat{\mu}_l^k$ as well as the number of times T_l^k each channel has been activated in all training steps so far. More precisely, if channel C_l^k has been activated T_l^k times by the end of training step t, then the value of $\hat{\mu}_l^k$ at the end of training step t is $(\sum_1^t SAL_{l,t}^k)/T_l^k$. In addition, it is assumed that the number of channels to be activated in each training step, is predefined and set beforehand. After training the network for a given number of iterations, we fix the subset of active channels by selecting the top percentile according to their global rank of mean saliency estimates across all layers, and fine-tune the model. This final fine-tuning stage can be viewed as operating solely in exploitation mode.


Training
Data
Select active channels
Run forward pass only on active channels
Backward pass: observe channel saliencies and update parameters
Fine-tune compact model
Figure 1: Flowchart of the dynamic channel execution framework for efficient training.

2.1 Dynamic channel selection mechanism

In the combinatorial multi-armed bandit problem setting we are given a system of arms, or convolutional channels in our work, each having unknown utility, that is an unknown distribution of reward with an unknown mean. The task is to select a combination of arms so as to minimize the difference in total expected reward between always playing the optimal super-arm (the combination of arms), and playing super-arms according to the CUCB algorithm. We introduce two assumptions in our framework: firstly, we disregard channel position and connectivity, and assess each channel's importance independently of the others. Secondly, we assume that the channels' reward distributions, which we derive from their estimated saliencies, are stationary and do not drift as we train. Our modified version of the CUCB framework applied to dynamic channel selection is summarized in Algorithm 1. The algorithm can be loosely divided in two stages: firstly, an initialization round of exploring and creating an initial estimate of the channel saliencies, and a second stage where we start exploiting and refining the initial saliency estimates to guide the channel selection procedure. Note that during the second stage in step 7 we adjust the estimated channel saliencies to balance between exploration and exploitation.

Algorithm 1 The CUCB algorithm applied to selective neural network channel execution

```
 1: For each channel $k$ in each convolutional layer $l$ maintain: 1) $T_l^k$ as the total number of times the
    particular channel has been activated so far; 2) $\hat{\mu}_l^k$ as the mean of all saliency estimates observed
    so far.
 2: For each channel $k$ in each convolutional layer $l$, activate an arbitrary set of channels $S$, such that
    $C_l^k \in S$, run forward and backward passes through compact network, and update $T_l^k$ and $\hat{\mu}_l^k$.
 3: $t \leftarrow$ number of convolutional channels in network
 4: for j=1...J do
 5:     for batch ($\mathbf{x}^{1:B}$, $y^{1:B}$) in $\mathcal{D}$ do
 6:         $t = t + 1$
 7:         For each channel $C_l^k$, set $\overline{\mu}_l^k = \hat{\mu}_l^k + \sqrt{\frac{3 \ln t}{2T_l^k}}$
 8:         $S \leftarrow$ top percentile of channels according to $\overline{\mu}_l^k$
 9:         Activate channels in S
10:         Run forward and backward passes through compact network
11:         Update all $T_l^k$ and $\hat{\mu}_l^k$
12: $S \leftarrow$ top percentile of channels according to $\hat{\mu}_l^k$
13: Activate channels in S
14: Fine-tune final compact network
```

2.2 Estimating channel saliency

Our channel ranking approach is based on Molchanov *et al.* [4]. The intuition behind the approach is approximating the change in the loss function from removing a particular channel, which was active at training step t, via a first-order Taylor expansion. Let h_l^k be the feature map produced by the channel C_l^k and L be the training loss. The saliency estimate $S\hat{A}L_{l,t}^k$ can be shown to be equal to:

$$S\hat{A}L_{l,t}^k = \left| \frac{1}{M} \sum_{m=1}^{M} \frac{\delta L}{\delta h_{l,m}^k} h_{l,m}^k \right|, \tag{1}$$

where M is the length of the vectorized feature map. Eq. 1 assumes independence between channel parameters whereas in reality they are inter-connected. The computation of Eq. 1 is easy in practice as it only requires the first derivative of the loss w.r.t each feature map element, i.e. $\frac{\delta L}{\delta h_{l,m}^k}$, which is computed during backpropagation.

3 Experiments

In this section we empirically demonstrate the effectiveness of our framework for dynamic channel selection during training. We conduct all experiments in PyTorch [5]. We evaluate the performance of our method on three datasets: CIFAR-10, CIFAR-100 [6] and Street View House Number (SVHN) [7]. We evaluate our method for dynamic channel selection on three network architectures: VGGNet [8], ResNet [9] and DenseNet [10]. We use a VGG-19 architecture of type "VGG-E", however we use only a single fully connected layer (16conv + 1FC) [11]. For ResNet we employ a 50-layer architecture with a bottleneck structure (ResNet-50). For DenseNet we use a 121-layer architecture with a growth rate of 32 (DenseNet-121). We use batch normalization on all network models and remove all dropout layers. We do not apply the dynamic channel selection procedure on fully connected layers as they comprise only the final classification layer in our models. All network models are trained using SGD with Nesterov momentum of 0.9 without dampening. The minibatch size we use is 64 for all datasets. On the two CIFAR datasets we train for 160 epochs, and on the SVHN dataset for 20 epochs. The initial learning rate is set at 0.1 and is divided by 10 at 50% and 75% of the training epochs. We also use a weight decay of 10^{-4}. We adopt the weight initialization introduced by [12]. After training following the proposed framework, we fix the subset of active channels by selecting the most salient ones across all layers. This compact model is fine-tuned by repeating the training procedure. We evaluate the floating-point operations (FLOPs) and the number of parameters of our compact networks using [13].

3.1 Results

We evaluate the dynamic channel selection framework on all combinations of datasets and network architectures. We experiment with a ratio of active channels between 20% to 100% in 10% increments. We perform all experiments three times and report averaged results.

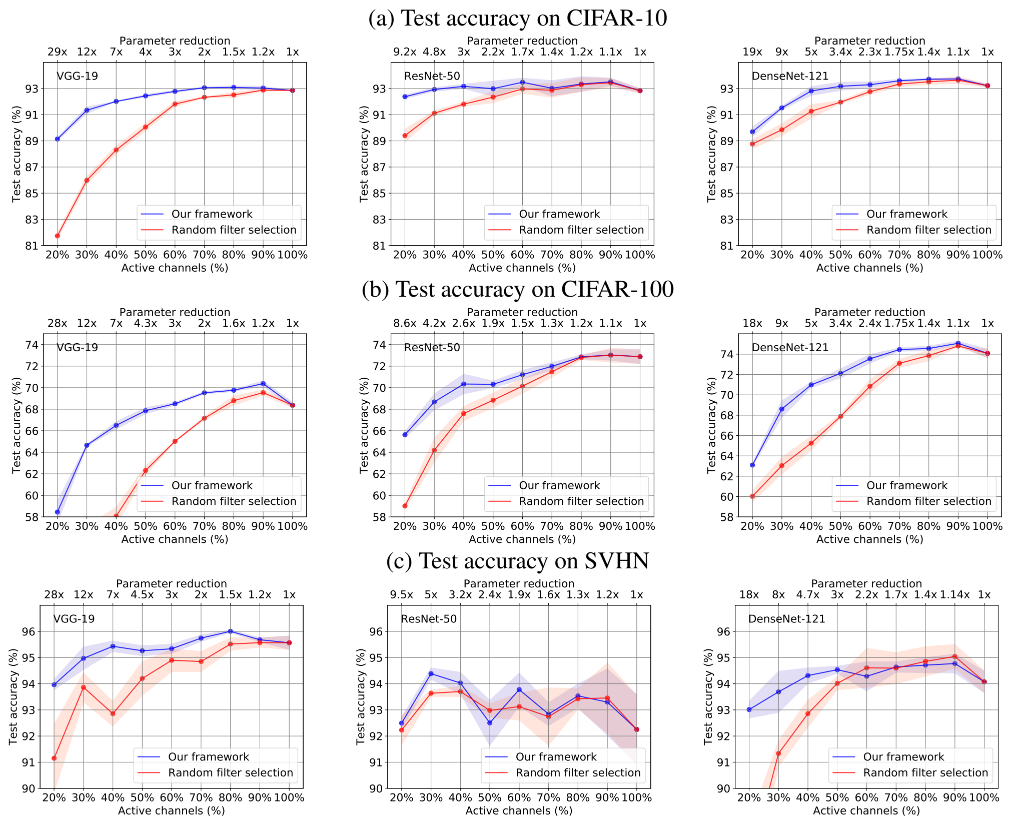


Figure 2: Comparing our proposed framework to random channel selection. The coloured areas denote standard deviation. All experiments were run three times. Our CUCB-based framework clearly outperforms random channel selection in 8 of the 9 experiments.

Regularization effect. For all datasets and network models the highest test accuracy is achieved when the percentage of active channels is typically between 70%-90%. The only exception is ResNet-50 evaluated on the SVHN dataset which achieves peak classification when 30% of the channels are active. We hypothesize that the increase in accuracy is due to the regularization effect of the dynamic channel selection procedure which could be viewed as feature selection applied on the hidden layers.

Parameter and FLOP reduction. Since the best performing compact models use active channels between 70%-90%, their parameter count and FLOPs are also lower compared to baseline models. For example, on CIFAR-10 the best performing VGG model achieves $2\times$ parameter reduction. DenseNet-121 and especially ResNet-50 are unable to achieve such parameter-efficiency owing to their bottleneck architecture. More specifically, the skip connections in ResNet-50 based on the addition operation require that certain convolutional layers have the same number of output channels. We observe comparable accuracy to baseline even for very compact networks ($30\% - 40\%$ channels) on the CIFAR-10 and SVHN datasets with parameter reduction between $3\times$-$7\times$ and FLOPs reduction $2\times$-$5\times$. Moreover, ResNet-50 can achieve over $9\times$ parameter reduction on CIFAR-10 and SVHN while maintaining baseline-level performance. Nevertheless, the very compact networks perform worse than baseline on the CIFAR-100 dataset (2%-3% accuracy drop for models with 30%-40% active channels). We conjecture this is because CIFAR-100 contains 100 classes and extra model capacity is required.

Results on CUCB vs random channel selection. We performed experiments where instead of activating the channels with the highest mean estimated saliency prior to fine-tuning, we randomly select a subset of channels to activate. It can be observed that randomly selecting channels to fine-tune performs worse in general than our proposed methodology. The difference in performance becomes more significant as the active channels become fewer. The sequential VGG-19 architecture adheres to this behaviour on all datasets. For ResNet-50 and DenseNet-121 the performance of models fine-tuned on randomly selected channels is on occasion on par with our framework. More specifically, on CIFAR-10 when the active channels are $\geq 70\%$, and on SVHN when random channel selection even outperforms our framework on a few instances. We hypothesize our methodology, which is based on the assumption of independence between channels, might be adversely affected by architectures with skip connections across layers, such as ResNet and DenseNet.

4 Conclusion

In this paper we have proposed an efficient on-device learning methodology to dynamically identify and utilize only the most salient convolutional channels at each training step given a hard parameter constraint. As a result, we can limit the memory and computational burden both during training *and* inference. Our method tracks the relative importance of each channel, and at each training step a subset of the most salient channels are activated and executed according to the combinatorial upper confidence bound algorithm. Experimental results on several datasets and network architectures reveal our framework is able to discover on-the-fly compact networks with lower computational cost compared to baseline (up to $4\times$) and parameter count (up to $9\times$) with little or no loss in accuracy.

References

[1] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143. 2015.

[2] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[3] Wei Chen, Yajun Wang, and Yang Yuan. Combinatorial multi-armed bandit: General framework, results and applications. In *International Conference on International Conference on Machine Learning - Volume 28*, pages I–151–I–159. JMLR.org, 2013.

[4] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. In *International Conference for Learning Representations*, 2017.

[5] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[6] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[7] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations, Conference Track Proceedings*, 2015.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 770–778, 2016.

[10] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 2261–2269, 2017.

[11] Fu Cheng-Yang. pytorch-vgg-cifar10. https://github.com/chengyangfu/pytorch-vgg-cifar10, 2019.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[13] Ligeng Zhu. Thop: Pytorch-opcounter. https://github.com/Lyken17/pytorch-OpCounter, 2019.