CAP'NN: A Class-aware Framework for Personalized Neural Network Inference

MAEDEH HEMMAT, JOSHUA SAN MIGUEL, and AZADEH DAVOODI,
University of Wisconsin–Madison, USA

We propose a framework for Class-aware Personalized Neural Network Inference (CAP'NN), which prunes an already-trained neural network model based on the preferences of individual users. Specifically, by adapting to the subset of output classes that each user is expected to encounter, CAP'NN is able to prune not only ineffectual neurons but also *miseffectual* neurons that confuse classification, without the need to retrain the network. CAP'NN also exploits the similarities among pruning requests from different users to minimize the timing overheads of pruning the network. To achieve this, we propose a clustering algorithm that groups similar classes in the network based on the firing rates of neurons for each class and then implement a lightweight cache architecture to store and reuse information from previously pruned networks. In our experiments with VGG-16, AlexNet, and ResNet-152 networks, CAP'NN achieves, on average, up to 47% model size reduction while actually *improving* the top-1(5) classification accuracy by up to 3.9%(3.4%) when the user only encounters a subset of the trained classes in these networks.

CCS Concepts: • **Computer systems organization → Neural networks;**

Additional Key Words and Phrases: Class-aware pruning, personalized inference, energy-efficient inference

ACM Reference format:
Maedeh Hemmat, Joshua San Miguel, and Azadeh Davoodi. 2022. CAP'NN: A Class-aware Framework for Personalized Neural Network Inference. *ACM Trans. Embedd. Comput. Syst.* 21, 5, Article 59 (December 2022), 24 pages.
https://doi.org/10.1145/3520126

1 INTRODUCTION

Deep learning algorithms have gained significant success in many domains, ranging from image and video classification to natural language processing. **Convolutional Neural Networks (CNNs)** are among the most widely used family of deep learning methods, providing unprecedented accuracy in applications such as object localization and object detection [25]. The high accuracy though comes with significant increase in required memory and computation resources, which mandates efficient hardware implementation of these networks on local devices. To address this, many approaches have been proposed to prune and compress the network. However, despite their success in reducing model size, prior approaches are fundamentally limited in their obliviousness to the *users* of the model.

This research was supported by Award No. 2006394 from National Science Foundation.
Authors' address: M. Hemmat, J. S. Miguel, and A. Davoodi, University of Wisconsin–Madison, 1415 Engineering Drive, Madison, Wisconsin, USA, 53706; emails: {hemmat2, jsanmiguel, adavoodi}@wisc.edu.

1539-9087/2022/12-ART59 $15.00
https://doi.org/10.1145/3520126

Our Goal: Personalized Inference. Our key insight is that an individual user only encounters a tiny fraction of the trained classes on a regular basis. Storing trained models (pruned or not) for all possible classes on local devices is costly and overprovisioned for the user's needs. For example, as prior studies show [23], 22% of the top 100 Android applications only use a single output class (e.g., faces, books, etc.) of the ImageNet dataset, which contains 1,000 output classes in total and is the widely used for image classification. Our study with a social media user also finds that among their 600 photos, seven classes make up 93% of the total observed classes. However, simply retraining bespoke neural network models for every user is not cost-effective, since there can be many users, all of whom are unique.

Our Solution: CAP'NN. We propose a **Class-aware Framework for Personalized Neural Network Inference (CAP'NN)**, which provides a *personalized* inference framework, taking a commodity trained model and pruning it based on the preferences of the user.[1] This minimizes the memory and computation overheads on the local device, processing neurons on a *need basis* (i.e., only when the user expects to encounter a specific output class).

This form of *class-aware pruning* is novel in its ability to consider which classes the user expects to encounter and weighs the pruning based on how frequently the user encounters them. It exploits the fact that not all neurons contribute equally towards correctly classifying a given output class. While prior class-unaware works are limited to pruning only ineffectual neurons (i.e., neurons that have low contribution to the final classification), CAP'NN uncovers the concept of **miseffectual neurons**, which are neurons that actually work against the correct identification of a specific class. Prior class-unaware pruning techniques are oblivious to miseffectual neurons, since, by nature of the training algorithm, all neurons contribute positively towards at least one class in the dataset. However, by personalizing the trained model and removing some classes, miseffectual neurons are left behind that are no longer useful and in fact confuse the classification of the remaining classes. By pruning miseffectual neurons, CAP'NN is able to achieve even higher accuracy than the original unpruned model, despite its much smaller model size. CAP'NN is also different from recent works [8, 23], which prune a network for a predefined subset of classes by identifying and removing unnecessary convolutional filters from the convolutional layers. Compared to CAP'NN, these works have limited opportunity for model size reduction and accuracy improvement, because (1) they do not consider per-class usage of each user and (2) they are oblivious to the concept of miseffectual neurons.

CAP'NN also exploits similarities among pruning requests from different users to reduce the timing overheads of pruning. Specifically, we first cluster the output classes of the network into groups based on the similarities between the firing rate of neurons. Next, we implement a light-weight cache architecture to store information about how the original network gets pruned for a given subset of classes. This information can be reused if a new user requests a pruning request for a different subset of classes with overlap and similarities with previous ones.

Contributions. Our work makes the following contributions:

- We introduce *CAP'NN*, a personalized inference framework that supplies neurons on a need basis, given the user's preferences.
- We propose new *class-aware pruning* schemes that take into account the distribution of classes that the user expects to encounter, achieving greater reductions in model size (up to 50% of the original model size as we pruned VGG-16 for 10 classes).

[1]The user is the *captain* now.

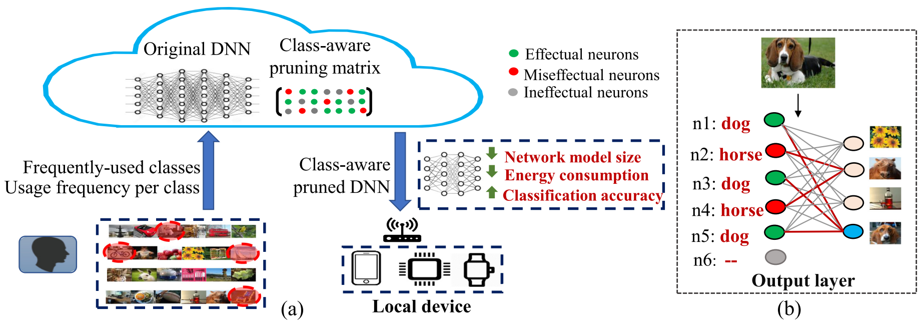

Fig. 1. (a) Overview of CAP'NN. (b) Example showing different types of neurons.

- We uncover the concept of *miseffectual neurons* and derive an algorithm for pruning them, achieving 2.3%(3.2%) improvement in top-1(5) accuracy of VGG-16 compared to the original unpruned model when pruning for 10 user-specified classes.
- We propose a lightweight cache architecture to minimize timing overhead of CAP'NN. Our technique exploits the similarities in the pruning requests from many users. Here, each user requests to prune the network for a different subset of classes but there may be overlap and similarities between the requested classes from different users.

2 OVERVIEW OF CAP'NN

In this section, we present our framework, CAP'NN, which enables personalized inference by pruning the network for a specific subset of classes based on the user's preferences. Figure 1(a) shows a high-level overview of our proposed framework. CAP'NN takes in a trained network as input and generates a class-aware pruned network, greatly reducing model size and improving accuracy for the user's subset of classes. Class-aware pruning exploits the correlation between neurons and output classes and without need to retrain the network [12].

Preprocessing Class-specific Firing Rates. First, as a preprocessing step, we need to obtain the correlation between neurons and output classes. To achieve this, we propose to measure the class-specific firing rate of neurons in the network. The class-specific firing rate represents how often a neuron gets fired when classifying the inputs that belong to a given output class. The key observation here is that in neural networks, the presence (or absence) of a particular feature in the received input is encoded as a positive (or negative) value for the feature's neurons. The negative neurons are clamped to zero as they pass through a ReLU function and are thus withheld from firing. As a result, the class-specific firing rate can serve as a proxy for how useful a neuron is in recognizing a given class. This preprocessing step is performed once offline, and class-specific firing rates are stored in the cloud server along with the network's architecture parameters.

Class-aware Pruning Techniques. Next, class-specific firing rates are utilized to prune the network for a subset of user-specified classes. We propose three variations of class-aware pruning:

- *CAP'NN-Basic pruning* (`CAP'NN-B`) uses the class-specific firing rates to first find a subset of neurons to be pruned for *each* output class while maintaining the per-class accuracy degradation below a threshold. The neurons that get pruned are thus ineffectual for a specific class. Then, for a user-specified subset of classes, `CAP'NN-B` prunes the neurons that are ineffectual for all classes.
- *CAP'NN-Weighted pruning* (`CAP'NN-W`) extends this to account for the expected per-class usage for each individual user, leading to more aggressive pruning. Specifically, `CAP'NN-W`

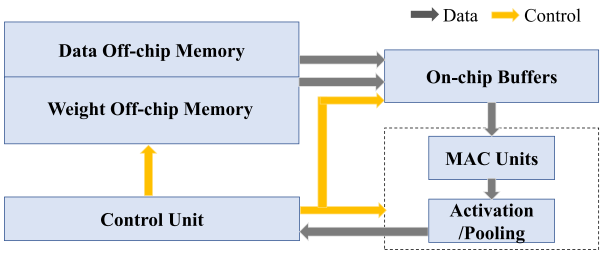

Fig. 2. Architecture view of the local device.

considers the likelihood of encountering each class to estimate an *effective* firing rate for each neuron. The effective firing rate is then used to find pruning candidates.

- *CAP'NN-Miseffectual pruning* (`CAP'NN-M`) enables pruning of miseffectual neurons in addition to the ineffectual ones already pruned by `CAP'NN-W`. It identifies and prunes neurons that work against the correct classification of the user classes.

Figure 1(b) shows an example of different types of neurons for a specific class (i.e., dogs). Here neuron n_6 is ineffectual, because it is not contributing to any of the output classes due to its low firing rate. Neurons n_2 and n_4 are miseffectual, because they work toward identifying a wrong output class (i.e., horses).

Pruning Process. In CAP'NN, the original network model is in the cloud. The cloud is responsible for pruning the network appropriately and sending the final model to the local device. Network pruning starts upon the user's request. To prune the network, the cloud receives the user's preferences (i.e., subset of classes as well as their expected usage). This information can either be directly provided by the user or obtained from a dedicated monitoring period. During the monitoring period, the cloud is responsible to track the network's prediction and determine the most frequently used classes and their usages, since original network resides in the cloud. Once the information is provided, the original network is pruned in the cloud (without the need for retraining) and is sent back to the device to be used for local inference. Note that the network can be pruned again if the user's preferences change.

Online Inference. A high-level overview of our hardware on the local device is shown in Figure 2. It is modeled after Google's **Tensor Processing Unit (TPU)** [16]. The weights and input data are stored in off-chip memory. To perform the inference computations, the weights and data are fetched from the off-chip buffer to fill up the on-chip weight and input buffers. The data values are then fed into various **multiply-accumulate (MAC)** units. The generated results are then propagated to the activations and pooling units. The outputs are stored in output buffers and are used as inputs to the next layer. A control unit is required to provide all necessary control signals. On-chip buffers are large enough to hold intermediate results generated by network. To execute the convolutional layers, we assume that the input feature maps are processed in parallel and the output feature maps are processed sequentially [30].

Summary. The advantage of CAP'NN is its novel approach of class-aware pruning, removing both miseffectual *and* ineffectual neurons based on user preferences. We show in our experiments that class-aware pruning of ineffectual neurons significantly reduces the network model size and energy consumption. On top of this, pruning miseffectual neurons can significantly boost per-class classification accuracy compared to the original network model. The next section describes our class-aware pruning algorithms in detail.

3 CLASS-AWARE PRUNING TECHNIQUES

Our class-aware pruning techniques require to first calculate class-specific firing rates for each neuron (or channel in case of convolutional layers[2]). This process is done once offline and the results are stored in the cloud. To obtain the neuron firing rates, we run the network using the training dataset with equal number of samples for each class. The class-specific firing rate of a neuron is then calculated as the fraction of effectual neurons (i.e., non-zero neurons that actively fire for the input samples representing a given class). In case of a convolutional layer, we calculate the class-specific firing rate of a channel as the average percentage of non-fired neurons in its corresponding feature map as described in Reference [14]. In this section, we focus our explanation on pruning neurons; it is straightforward to adapt the discussions to pruning channels. We propose three variations of `CAP'NN` for class-aware pruning [12].

3.1 CAP'NN-B: Basic Class-aware Pruning

Our basic class-aware pruning technique receives as input a CNN along with the user's preferences: a set of $\mathcal{K}$ output classes that the user expects to encounter as well as an acceptable degradation in classification accuracy denoted by ϵ. It also receives the class-specific firing rates for each neuron that we calculate in the cloud a priori. It outputs a set of neurons to be pruned while guaranteeing that the post-pruning degradation in accuracy is below ϵ for *each* output class. More specifically, the inputs are denoted as below:

- CNN specified by a set of output classes C and set of layers[3] $\mathcal{L}$ with neurons $\mathcal{N}_\ell$ $\forall \ell \in \mathcal{L}$
- Firing rate matrix F_ℓ of dimension $|\mathcal{N}_\ell| \times |C|$ $\forall \ell \in \mathcal{L}$
- Accuracy degradation ϵ
- Set $\mathcal{K}$ of user-specific classes

Offline Algorithm: Identifying Pruning Candidates. `CAP'NN-B` first applies Algorithm 1 to identify for each class, a set of neurons that can be pruned while guaranteeing the degradation in accuracy remains below ϵ in *all* the classes. Algorithm 1 is independent of $\mathcal{K}$. It receives the CNN as input along with ϵ and layer-specific matrix of neuron firing rates F_ℓ, as explained before. It returns a layer-specific pruning matrix P_ℓ of dimension $|\mathcal{N}_\ell| \times |C|$ for all $\ell \in \mathcal{L}$. An element (n,c) in P_ℓ is a binary value indicating if neuron n in layer ℓ may be pruned for class c. Given a class c, the non-zero elements in vectors $(:, c)$ in P_ℓs ($\forall \ell \in \mathcal{L}$) specify the set of neurons that may be pruned simultaneously across all layers for class c while guaranteeing the accuracy degradation is below ϵ, for *all* the output classes.

We first explain how Algorithm 1 operates and then explain how its output is utilized at runtime to find the neurons to prune for the user-specified classes. Starting from line 1, it visits each of the last layers in the network and computes pruning matrix P_ℓ for that layer, considering the accuracy degradation of all pruned layers up to that point. Specifically, in lines 7–15, it flags a set of candidate neurons to be temporarily pruned for each class c in the current layer. These are the neurons with firing rate below a threshold T (set to T^1_{start} in line 5). Next in lines 14–16, it adds the set of neurons that have been pruned so far from the previously visited layers. In lines 17–19, it measures accuracy degradation in each class for this combined pruning set. If the accuracy degradation is below ϵ in all classes, then the candidate neurons identified in the layer are permanently marked as pruned in line 21. Otherwise, the firing threshold T is decreased by a

[2]Channel pruning for convolutional layers reduces memory overhead of storing the firing rates and alleviates sparsity of the weight matrices and irregular memory accesses, leading to a more efficient hardware implementation.
[3]We consider $\mathcal{L}$ to be the last few layers of the CNN and prune only from the last layers, because earlier layers are typically not class-specific and extract more general features from the input.

ALGORITHM 1: CAP'NN-B(CNN, ϵ, $F_\ell\ \forall \ell \in \mathcal{L}$, &$P_\ell\ \forall \ell \in \mathcal{L}$)

Inputs: CNN given by set of output classes C and set of layers $\mathcal{L}$ with neurons $\mathcal{N}_\ell\ \forall \ell \in \mathcal{L}$, accuracy degradation ϵ, firing rates matrices F_ℓ of dimension $|\mathcal{N}_\ell| \times |C|\ \forall \ell \in \mathcal{L}$.
Outputs: Pruning matrix P_ℓ of dimension $|\mathcal{N}_\ell| \times |C|\ \forall \ell \mathcal{L}$.

```
1: for each layer $\ell = l_{start}$ to $|\mathcal{L}|$ do
2:     CAP-PerLayer($\ell$, $\mathcal{N}_\ell$, $F_\ell$ , $C$, $\epsilon$, &$P_\ell$)
3: end for
4: procedure CAP-PerLayer($\ell$, $\mathcal{N}_\ell$, $F_\ell$ , $C$, $\epsilon$, &$P_\ell$)
5:     $T = T^1_{start}$
6:     $H(n,c) = 0\ \forall n \in \mathcal{N}_\ell\ \forall c \in C$
7:     for each class $c \in C$ do
8:         for each neuron $n \in \mathcal{N}_\ell$ do
9:             if $F_\ell(n,c) \leq T$ then                  ▷ Firing rate for class c is below T
10:                $H(n,c) = 1$                             ▷ Flag to temporarily prune
11:            end if
12:        end for
13:        Temporarily prune neurons $n$ from $\ell$ if $H(n,c) = 1\ \forall n \in \mathcal{N}_\ell$
14:        for each layer $l = l_{start}$ to $\ell - 1$ do
15:            Temporarily prune neuron $n$ from layer $l$ if $P_l(n,c) = 1$
16:        end for
17:        for each $i \in C$ do
18:            Measure accuracy degradation $d_i$ at class $i$ of pruned CNN
19:        end for
20:        if $d_i \leq \epsilon\quad \forall i \in C$ then
21:            Set $P_\ell(n,c) = H(n,c)\ \forall n \in \mathcal{N}_\ell$       ▷ Prune permanently
22:        else
23:            $T = T - step$ and go to line 2
24:        end if
25:    end for
26: end procedure
```

step and the process is repeated to identify fewer neurons until accuracy degradation is below ϵ for all classes.

The outputs of this per-layer procedure are the neurons that are permanently marked for pruning for each class (i.e., $P(n,c) = 1$) in that layer (i.e., P_ℓ pruning matrix). The algorithm terminates once P_ℓ is computed for all layers in lines 1–3.

Online Pruning. At run-time, given the set of user-specified classes $\mathcal{K}$, the actual pruned neurons for a layer ℓ are given by $\cap_{c \in \mathcal{K}}(:,c)$ in P_ℓ, which computes the intersection of per-class pruning vectors over the classes in $\mathcal{K}$. This computation is done for each layer to find the final set of pruned neurons across all layers.

We note that Algorithm 1 is able to maintain the accuracy of the model after pruning for two reasons. First, the algorithm guarantees a bound of ϵ in accuracy degradation for each class; Given that the intersection of the neurons (which is a smaller set) are removed for a subset of classes, CAP'NN-B will not be an aggressive pruning scheme. Second, CAP'NN-B prunes the network layer by layer and sequentially, providing an for the model to compensate for accuracy loss from previous layers if an effectual neuron gets pruned.

Overall, CAP'NN-B relies on storing binary pruning vectors generated by Algorithm 1 and at run-time would only perform the intersection operation among related pruning vectors; thus, it is a fast online procedure. It also guarantees that the pruned vectors always result in an accuracy degradation of at most ϵ, regardless of $|\mathcal{K}|$.

Timing Complexity. Algorithm 1 involves finding per-layer pruning matrix (lines 4–26) for all layers in $\mathcal{L}$. For each layer, the main following operations are executed: (1) the algorithm iterates through all classes in the original network (line 7), (2) for each class, it iterates through all neurons in the layer (lines 8–11), and (3) the algorithm temporary prunes the neurons from previous layers to find pruning matrix for the current layer (line 14).

Assume the network has $|C|$ output classes and each layer has at most $|\mathcal{N}|$ neurons. Also, assume we are interested to find pruning matrices for the last $|\mathcal{L}|$ layers. Then, finding pruning matrix for each layer has timing complexity of $O(max(|\mathcal{N}|, |\mathcal{L}|).|C|)$. Given that number of neurons per layer in a network is generally greater than the number of the layers in the network, the complexity of CAP-PerLayer algorithm will be $O(|\mathcal{N}|.|C|)$ for each iteration. The algorithm though may run for multiple iterations to find the appropriate pruning threshold T for the layer. This is done by reducing the threshold at each iteration and repeating the same procedure (line 23). While the number of pruning iteration is not pre-determined, it is limited to $it^{max} = \lceil \frac{T^1_{start}}{step} \rceil$. (In our experiments, we show that only a small number of pruning iteration is required to prune the model.) With that said, CAP-PerLayer has time complexity of $O(it^{max}.|\mathcal{N}|.|C|)$. Since CAP-PerLayer is executed for last $|\mathcal{L}|$ layers, the complexity of CAP'NN-B will be $O(|\mathcal{L}|.it^{max}.|\mathcal{N}|.|C|)$.

3.2 CAP'NN-W: Weighted Class-aware Pruning

CAP'NN-W is a generalization of CAP'NN-B that takes into account the *likelihood of encountering each class*. The input user preferences include a weight $0 \leq w_k \leq 1$ for each user-specified class $k \in \mathcal{K}$. For a single user, these weights add to 1. Recall that this information is either obtained from user directly or is collected by cloud from a dedicated monitoring period, as discussed in Section 2.

Online Algorithm: Identifying Pruning Candidates. Similar to CAP'NN-B, pruning is conducted layer by layer and at each layer the neurons to be pruned are identified. This time, however, the per-layer pruning matrix P_ℓ is of dimension $|\mathcal{N}_\ell| \times 1$. Algorithm 2 shows the per-layer pruning procedure for CAP'NN-W.

First, in lines 2–6, for each neuron in the layer, the condition $\sum_{k \in \mathcal{K}} w_k \times F_\ell(n,k) \leq T$ is evaluated. This condition computes an *effective firing rate* given by $w_k \times F_\ell(n,k)$, which represents how often a neuron n fires for class k, accounting for how likely k will be encountered by the user. It then temporarily flags the neuron to be pruned if the sum of its effective firing rates across the classes in $\mathcal{K}$ is below a threshold.

The next steps in lines 8–17 are similar to Algorithm 1: it permanently prunes the flagged neurons if the per-class degradation in accuracy is below ϵ for all used classes, while accounting for the neurons pruned so far in earlier layers.

Note that Algorithm 2 cannot be performed offline, because it depends on the distribution of user-specified classes, which is only known (and may even change) at run-time. Therefore, CAP'NN-W takes longer to execute at run-time compared to CAP'NN-B. However, the procedure is still fast, especially when limited to only a few classes, i.e., $|\mathcal{K}|$ is small. This is because the per-class loop in Algorithm 1 (line 7) is completely removed. CAP'NN-W also incurs a higher memory overhead compared to CAP'NN-B, since it needs to store the floating-point firing rates to be used at run-time. This overhead can be alleviated by encoding the floating-point firing rates with fewer number of bits. Finally, CAP'NN-W guarantees that the pruned neurons cannot yield accuracy

Firing rate matrix

	n1	n2	n3
c1	0.08	0.04	0.26
c2	0.13	0.03	0.30
c3	0.03	0.07	0.14

Effective pruning rates in CAP'NN-W
(T = 0.1 and W_1 = 0.6 , W_2 = 0.1 , W_3 = 0.3)

n1	n2	n3
0.07	0.05	0.23

Pruning result

	Neuron 1	Neuron 2	Neuron 3
CAP'NN – B	×	✓	×
CAP'NN – W	✓	✓	×

✓ Pruned
× Not pruned

Fig. 3. Comparison of CAP'NN-B and CAP'NN-W.

ALGORITHM 2: CAP'NN-W-PerLayer(ℓ, $\mathcal{N}_\ell$, F_ℓ , $\mathcal{K}$, $\mathbf{w}$, ϵ, &P_ℓ)

Inputs: layer ℓ with set of neurons $\mathcal{N}_\ell$, firing rate matrix F_ℓ, set of used classes C with vector of class-usage weights $\mathbf{w}$, accuracy degradation ϵ.
Outputs: Pruning matrix P_ℓ of dimension $\mathcal{N}_\ell \times 1$.

```
1:  $T = T^1_{start}$; $H(n) = 0\ \forall n \in \mathcal{N}_\ell$
2:  for each neuron $n \in \mathcal{N}_\ell$ do
3:      if $\sum_{k \in \mathcal{K}} w_k \times F_\ell(n,k) \le T$ then
4:          $H(n) = 1$                                  ▷ Flag to temporarily prune
5:      end if
6:  end for
7:  Temporarily prune neurons $n$ from $\ell$ if $H(n) = 1\ \forall n \in \mathcal{N}_\ell$
8:  for each layer $l = l_{start}$ to $\ell - 1$ do
9:      Temporarily prune neuron $n$ from layer $l$ if $P_l(n) = 1$
10: end for
11: for each $i \in \mathcal{K}$ do
12:     Measure accuracy degradation $d_i$ at class $i$ of pruned CNN
13: end for
14: if $d_i \le \epsilon \quad \forall i \in \mathcal{K}$ then
15:     Set $P_\ell(n) = H(n)\ \forall n \in \mathcal{N}_\ell$    ▷ Prune permanently
16: else
17:     $T = T - step$ and go to line 2
18: end if
```

degradation beyond ϵ. This is because the algorithm estimates accuracy degradation for each class in $\mathcal{K}$ (line 11) before pruning neurons from a layer permanently. Also, similar to Algorithm 1, CAP'NN-W prunes the network layer by layer, providing more control to maintain network performance.

Online Pruning. Accounting for the likelihood of each class allows for more aggressive pruning of neurons in CAP'NN-W compared to CAP'NN-B at runtime. Consider the example shown in Figure 3 with three neurons and three classes. Assume the pruning threshold determined by both algorithms is $T = 0.1$. The per-class weights are listed for CAP'NN-W along with the effective firing rate for each neuron. Here, neuron n_1 is not pruned by CAP'NN-B, because its firing rate is above T for one of the three classes (i.e., c_2). In contrast, CAP'NN-W prunes n_1, because its effective firing rate is below T, considering that class c_2 is only encountered 10% of the time.

Timing Complexity. Timing complexity of Algorithm 2 is similar to Algorithm 1. The only difference is that in Algorithm 2 the per-class loop is removed and it only iterate through all neurons in each layer to find effective firing rates and temporarily prune the neurons from previous layers. With that said, the timing complexity of `CAP-PerLayer` algorithm will be $O(max(|\mathcal{N}|, |\mathcal{L}|, |\mathcal{K}|))$ for each iteration where $|\mathcal{K}|$ is the subset of classes the user is interested in. Again, assuming that number of neurons in each layer exceeds the number of layers and the number of user-specified classes, the timing complexity to find per-layer pruning matrix is $O(it^{max}.|\mathcal{N}|)$ and timing complexity for `CAP'NN-W` will be $O(it^{max}.|\mathcal{L}|.|\mathcal{N}|)$.

3.3 CAP'NN-M: Class-aware Pruning of Miseffectual Neurons

So far, our proposed schemes have only pruned ineffectual neurons. `CAP'NN-M` is an extension to `CAP'NN-W` that identifies and prunes *miseffectual neurons*. We say that a neuron is miseffectual if it fires in the direction of an incorrect class. That is, for a user-specified class $k \in \mathcal{K}$, a miseffectual neuron has a higher contribution to one or more wrong classes (i.e., class $c \neq k \; \forall c \in C$).

`CAP'NN-M` first identifies a set of miseffectual neurons $\mathcal{M}_c$ for each class $c \in C$. This is done as a one-time process offline. Next, the firing rate of each miseffectual neuron n for class c is set to 0 in the firing rate matrix ($F_{last}(n, c) = 0$). Next, our `CAP'NN-W` algorithm is invoked with this updated firing rate matrix to find the pruned set of neurons for `CAP'NN-M`.

To identify miseffectual neurons, `CAP'NN-M` follows a two-step procedure. In the first step, it identifies the top *confusing* classes using a confusion matrix. In the second step, it characterizes neurons as miseffectual if they contribute more to these top confusing classes. We describe these two steps in more detail.

Step 1: Finding Top Confusing Classes. Given a class $k \in \mathcal{K}$, we identify the top-5 classes[4] $c \neq k$ that have the highest probability of being triggered during inference if the inputs were only from class k. These classes are thus most likely to be confused with k and can be found using a confusion matrix of dimension $|\mathcal{K}| \times |C|$. Specifically, for class k, we run the network for N input samples from class k. We fill each entry (k, c) in the confusion matrix with the fraction of times that c is triggered. We then select the 5 classes with the largest entry values.

Step 2: Finding Miseffectual Neurons for Top Confusing Classes. At this step, we focus on identifying miseffectual neuron candidates among the neurons in the last layer, denoted as $\mathcal{N}_{last}$. These neurons connect to the $|C|$ output neurons as a fully connected layer. For each output neuron j:

$$c_j = \sum_{n \in \mathcal{N}_{last}} w_{ij} \times n_i + b_j. \tag{1}$$

The contribution of neuron $i \in \mathcal{N}_{last}$ to output neuron j is measured by $\frac{d(c_j)}{d(n_i)} = w_{ij}$. We measure the contribution of each neuron in $\mathcal{N}_{last}$ for each top-5 confusing class using its corresponding weight from the equation above.

Using the above two-step process, `CAP'NN-M` identifies and prunes miseffectual neurons in addition to the ineffectual ones found in `CAP'NN-W`. These two schemes are similar in terms of memory overhead and execution runtime. As we show in our experiments, removing miseffectual neurons actually improves classification accuracy in addition to pruning more neurons overall compared to `CAP'NN-W`.

4 CACHING PER-LAYER PRUNING THRESHOLDS

In this section, we propose a lightweight cache architecture to reduce the timing overhead of the `CAP'NN-W` algorithm. The proposed cache architecture is particularly helpful when several users

[4]We identify the top-5 classes, because it relates to the top-5 accuracy, which we report in our experiments.

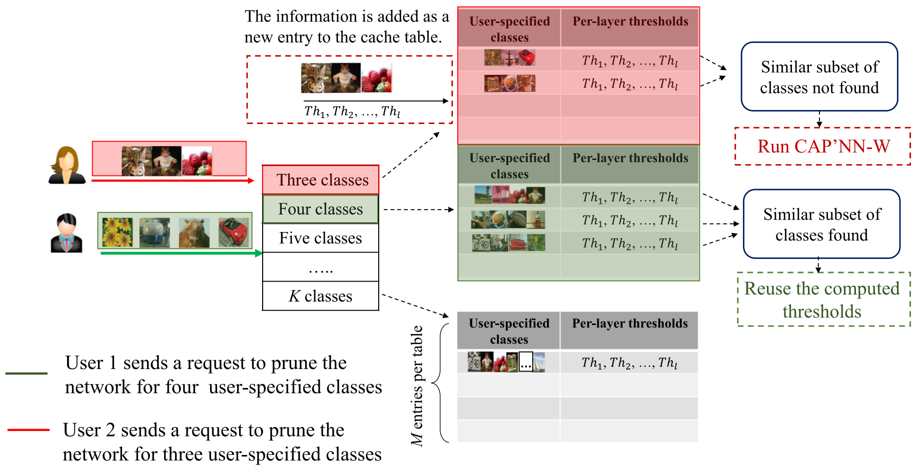
The information is added as a new entry to the cache table.
$Th_1, Th_2, ..., Th_l$
User-specified classes
Per-layer thresholds
Similar subset of classes not found
Run CAP'NN-W
Three classes
Four classes
Five classes
…..
K classes
Similar subset of classes found
Reuse the computed thresholds
M entries per table
User 1 sends a request to prune the network for four user-specified classes
User 2 sends a request to prune the network for three user-specified classes

Fig. 4. Overview of the proposed cache architecture.

are sending their pruning requests to CAP'NN, each requesting to prune the network for a different subset of user-specified classes but there is an overlap among the requested classes for some users.

As discussed earlier, CAP'NN-W requires finding per-layer pruning thresholds at run-time, which is determined based on user-specified classes and per-class usages. This can be a time-consuming process, since it involves finding a set of candidate neurons for pruning based on the effective firing rates of neurons and running the pruned network for each of the user-specified classes (lines 8–13 in Algorithm 2).

To alleviate the overhead, we propose to exploit similarities between the class-specific firing rate of neurons and reuse previously computed pruning thresholds if the two subsets of user-specified classes are found to have sufficient similarity. Our observation is that classes with similar layer-wise firing rates will have almost the same pruning threshold. This relationship is originated from how CAP'NN-W algorithm works. Recall in Algorithm 2, the firing rate of a neuron is compared against a threshold and the neuron is flagged for temporary pruning if its firing rate is below the threshold. The flagged neurons are then withheld from firing when measuring accuracy degradation. Let's assume neuron N_j has similar firing rates for classes c_1 and c_2. This is a proxy showing that neuron N_j has almost similar contribution toward the correct classification of inputs belonging to classes c_1 and c_2. With that said, if the neuron can be pruned for class c_1, it can also be removed for class c_2 without incurring much loss in accuracy. Given that decision on pruning a neuron depends on the pruning threshold $T2$, two classes have almost the same pruning threshold as well. This can be extended to the effective firing rate of neurons when a user is interested in several classes; The contribution of a neuron to a subset of classes K_1, denoted by the effective firing rate of the neuron, is almost similar to another subset of classes K_2 if classes in K_1 and K_2 have similar class-specific firing rates. To appropriately store and reuse the pruning thresholds, we propose to integrate a lightweight cache architecture into our design, and we follow a two-step procedure as below:

Step 1: Finding Similar Classes. For each layer in the network, we first run a clustering algorithm that divides the class-specific firing rates into P clusters by merely looking at the class-specific firing rate of individual neurons in that layer. Once clustering is done, output classes with the most similar firing rates are grouped together. The clustering is done once offline, and it works at granularity of layers. The clustering results are then stored in the cloud along with the network parameters to help evaluate the similarities among the pruning requests at run-time.

Step 2: Storing and Reusing the Computed Thresholds. Next, we use a lightweight cache architecture to store the computed pruning thresholds at run-time. More specifically, for each value of K (i.e., number of user-specified classes), we propose to store up to M subset of user-specified classes along with their corresponding per-layer pruning thresholds, obtained from running Algorithm 2, as shown in Figure 4.

At run-time and upon receiving a pruning request from the user, CAP'NN looks up the cache table with an equal number of user-specified classes instead of running `CAP'NN-W`. For each entry in the cache table, CAP'NN first measures the similarity between the current subset of user-specified classes and the existing one in the entry by utilizing the information from the clustering phase. Here, two classes are considered similar if they belong to the same cluster. The cache look-up results in a cache hit if an entry can be found such that all the classes in that entry are similar to the current subset of classes. In the case of a cache hit, CAP'NN picks the already-computed thresholds to prune the network. If such an entry is not found, then a cache miss occurs. In this case, `CAP'NN-W` is executed to find the pruning thresholds. The thresholds are then stored as a new entry in the cache table. If the cache is full, then CAP'NN evicts the least-recently used entry.

Figure 4 shows an example of our proposed architecture. In this example, two different pruning requests from two users, color-coded in green and red, are sent to CAP'NN. For each of the pruning requests, CAP'NN first looks up the appropriate cache table. In the case of User 1 with four user-specified classes, also color-coded in green, a similar cache entry is found. So, CAP'NN only takes the already-computed thresholds and prunes the network. For User 2 with three user-specified classes, the cache look-up results in a cache miss. Therefore, `CAP'NN-W` will be executed and the computed thresholds along with the user-specified classes are then stored in a new entry of the cache.

4.1 Details of Clustering

Here, we discuss our procedure to cluster per-layer class-specific firing rates. In this work, we opt to employ a K-means clustering algorithm, where the objective is to minimize **Within-cluster Sum of Squares (WCSS)** of the clustered elements [24], which in turn maximizes the similarity between elements in the same cluster. To compute WCSS for a cluster, the sum of squares is measured using the distances of the elements in a cluster from the centroid of that cluster. To determine the number of clusters for each layer, we use the Elbow algorithm [2]. In the Elbow algorithm, WCSS is reported as a function of the number of clusters, P. By increasing P, WCSS is initially decreased. Based on our observations, there exists a point where the rate of reduction in WCSS as a function of P will be the most, and after that point, increasing the number of clusters does not significantly reduce WCSS. We choose that point to select the number of clusters, which achieves a reasonably small value of WCSS with a reasonably small number of clusters.

To show the effectiveness of our clustering algorithm in finding similar classes, we measured intra-cluster and inter-cluster distances for each cluster. Intra-cluster distance measures the normalized distance between class-specific firing rate vectors of two classes in the same cluster while inter-cluster distance corresponds to the distance between firing rate vectors of two classes in different clusters.

Here, we define the average intra-cluster and inter-cluster distances of class c_i belonging to cluster p as follows:

$$Intra-cluster_p^{c_i} = \frac{1}{|p|-1} \times \sum_{\forall c_j \in p, c_i \neq c_j} \frac{\sum_{\forall n \in \mathcal{N}_\ell} |F_\ell(n, c_i) - F_\ell(n, c_j)|}{|\mathcal{N}_\ell|}, \tag{2}$$

$$Inter-cluster_p^{c_i} = \frac{1}{|C|-|p|} \times \sum_{\forall c_j \notin p} \frac{\sum_{\forall n \in \mathcal{N}_\ell} |F_\ell(n, c_i) - F_\ell(n, c_j)|}{|\mathcal{N}_\ell|}, \tag{3}$$

where $|p|$ refers to the number of classes in cluster p, F_ℓ is the firing rate matrix of layer ℓ, $|\mathcal{N}_\ell|$ is the number of neurons at layer ℓ, and $|C|$ is the total number of classes in the network.

More specifically, the intra-cluster distance of class c_i in cluster p measures the average distance between firing rate vector of class c_i and firing rate vector of all other classes in the same cluster, when normalized to the number of classes in p (excluding the class c_i). Similarly, the inter-cluster distance of class c_i in cluster p is the distance between the firing rate vector of class c_i and all other classes outside p, when normalized to the total number of classes outside the cluster. In our experiments, we show that intra-cluster distance is significantly smaller than the inter-cluster one, indicating that classes in the same cluster have the most similar firing rate vectors.

5 RELATED WORKS

5.1 Class-aware Pruning

The majority of existing pruning and structure simplification techniques seek to create an equivalent DNN model that operates on the same classes but requires less computation and smaller models. These techniques may be categorized into three groups: low-rank approximation, unstructured pruning, and structured pruning.

Low-rank approximation techniques achieve a reduction in both model parameters and computation using techniques such as Singular Value Decomposition [6] and Tucker decomposition [17]. Unstructured pruning techniques are based on eliminating unimportant weights and connections. They date back to the 1990s [5, 10], though recent works have emerged based on weight pruning [9]. The key observation in these works is that many weights in a trained network end up having relatively small magnitudes and hence may be reset to zero without affecting the network's performance.

Structured pruning applies pruning of entire layers or groups of weights. In the early 1990s, the authors of References [15, 18] sought to directly find and remove redundant neurons by analyzing the output neurons of the same layer as a function of the set of inputs. A neuron can then be pruned if its output can be well approximated via a weighted linear combination of the outputs of remaining neurons. More recently, channel pruning is studied [11, 14, 19–21, 27] as an effective way to develop compact and efficient models for CNNs, since the majority of inference energy is consumed by convolutional layers [3].

The above innovations can all be regarded as *class-unaware* pruning. They are fundamentally different from the class-aware approach in this work. These two sets of techniques are orthogonal; class-aware and class-unaware techniques may be applied simultaneously.

Recently, the works [8, 23] propose pruning techniques for a predefined subset of classes. Specifically both works focus on channel pruning for convolutional layers. CAP'NN is different from these prior works in several major ways. First, it takes per-class usage for each individual user into account and shows that it can fundamentally improve pruning opportunities while still guaranteeing that the classification accuracy never falls below a user-specified bound. Second, CAP'NN is applied to both fully connected layers (pruning neurons) as well as convolutional layers (pruning channels). Third, by introducing the notion of miseffectual neurons, CAP'NN is able to prune further and not only achieve a smaller model size but also *improve* classification accuracy, as we show in our experiments.

Class-aware pruning should not be confused with context-aware pruning such as Reference [4], where the pruning decision relies on the context of the received input, not based on a subset of classes. It is also different from input-dependent and early-termination techniques such as

Reference [26], where the correlation between neurons and output classes are utilized to prune the network at run-time and on an input-by-input basis.

CAP'NN is also different from knowledge distillation techniques. Knowledge distillation transfers the knowledge from a larger network (teacher) to the smaller one (student) by using the predicted probabilities generated by larger model as the labels for smaller one and minimizing the loss function for the distilled network. Therefore, it is oblivious to user's preference at run-time and it does not take into account the likelihood of encountering each class. Also, transferring the knowledge requires one to train the smaller model, which is not a cost-efficient approach for network personalization problem, since there can be many users with different preferences.

5.2 Content Caching

Content caching is a promising approach to reduce overheads of accessing repetitive data in different applications. Content caching exploits the benefits of caching commonly used data/content for future use by other tasks/users [1, 28]. For instance, to cope with explosive growth of mobile data on 5G networks, one solution is to find the most popular files for users and store them on local caches. This, in turn, results in faster access to the files for future users. In CAP'NN, we utilize the same approach to reduce the timing overheads of our class-aware pruning schemes by caching the information from previously pruned models and reusing them for future pruning requests.

6 RESULTS AND DISCUSSION

In our experiments, we use the VGG-16, AlexNet, and ResNet-152 networks to evaluate the efficiency of our CAP'NN implementations. All networks are recognized as representative networks in applications involving object classification and localization tasks. VGG-16 and AlexNet are considered heavy models with a huge number of parameters while ResNet-152 has a smaller model size and is particularly designed for mobile applications. All networks are implemented in Tensorflow and are trained and tested on the ImageNet (2012) dataset with 1,000 output classes.

To compute class-specific firing rates for each network, we ran the network with 200 images for each output class. In our implementation of different variations of CAP'NN, we set the following parameters for Algorithms 1 and 2: maximum allowed accuracy degradation $\epsilon = 3\%$, the start threshold to bound the firing rates $T^1_{start} = 0.4$, reduction step of $step = 0.025$. We also pruned from the last 6 layers of VGG-16 and AlexNet so $l_{start} = |\mathcal{L}| - 6$.

6.1 Comparison of the CAP'NN Variations

In this experiment, we prune the networks for different configurations when varying the number of user-specified classes $K = |\mathcal{K}|$. Specifically, for each value of K, we randomly selected 200 combinations of classes (i.e., each combination has K randomly selected classes). For each combination, we pruned the network using the three CAP'NN variations and measured the classification accuracy of the pruned network. Note, all variations of CAP'NN are applied to an already-trained network, and retraining is not required. We then report the average top-1/top-5 classification accuracies across the 200 combinations per K, along with the average model size.

Model Size. Figures 5, 6, and 7 show the average post-pruning model size for the three networks, when the networks are pruned for $|\mathcal{K}| = 2, 3, 4, 5$ user-specified classes. The model size is normalized to the number of parameters in their corresponding original network. The model size is measured by the number of (unique) parameters in each network including the number of weights and biases. Recall that `CAP'NN-W` and `CAP'NN-M` take into account the class usage distribution (i.e., likelihood of encountering each class). We consider different usage distributions, which are shown on the x-axis for each plot (i.e., for each value of K). For example for $K = 2$, we consider 10%–90% as

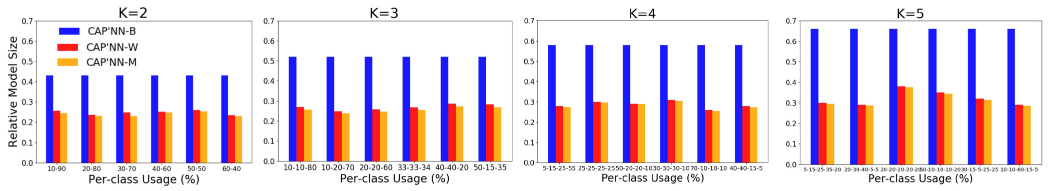

Fig. 5. Model size of VGG-16 of various pruning schemes with different number of user-specified classes (K) and usage weights.

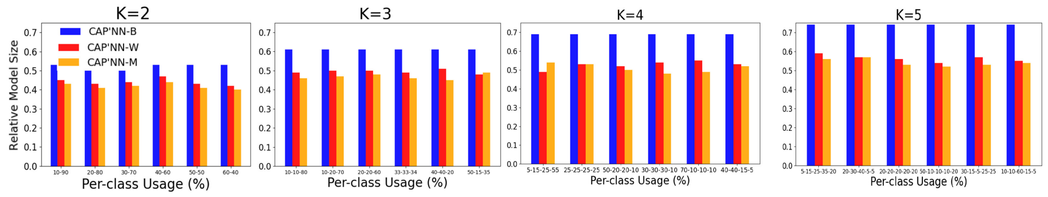

Fig. 6. Model size of AlexNet of various pruning schemes with different number of user-specified classes (K) and usage weights.

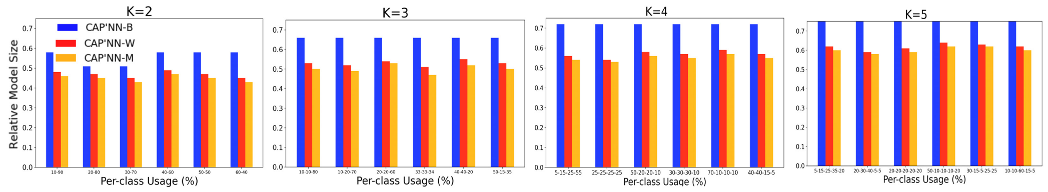

Fig. 7. Model size of ResNet-152 of various pruning schemes with different number of user-specified classes (K) and usage weights.

one usage scenario of the two classes. Overall, we consider 24 different variations by changing K and the usage distributions. (Note, the bars corresponding to CAP'NN-B do not vary within each plot, because it is independent of per-class usage.)

As the results show, all variations achieve significantly smaller model sizes compared to the original model. For example, in case of VGG-16 (AlexNet), when $K = 5$, CAP'NN-B, CAP'NN-W, and CAP'NN-M yield relative model sizes of on average 66% (73%), 30% (55%), and 29% (53%), respectively. For ResNet-152 architecture, the relative model size of CAP'NN-B, CAP'NN-W, and CAP'NN-M are around 75%, 62%, and 60%, respectively, when $K = 5$.

As can be seen, both CAP'NN-M and CAP'NN-W find more opportunities for pruning compared to CAP'NN-B, since they operate on effective firing rates defined by the usage distribution of classes. CAP'NN-M is able to achieve a slightly smaller model size compared to CAP'NN-W by also pruning miseffectual neurons; though as we show next, the main advantage of CAP'NN-M is its accuracy gain. We also observe that model size reduction is less in the case of ResNet-152 compared to VGG-16 and AlexNet. This originates from two facts. First, ResNet-152 is a more compact model with a fewer number of redundant parameters. Second, the network architecture of ResNet-152 has skip connections, which can limit the opportunity of CAP'NN for pruning neurons. More specifically, in ResNet architectures, a layer can be connected to more than one subsequent layer. In such cases, CAP'NN should consider the layer with a smaller number of pruned neurons while removing the input neurons in the next subsequent layer. AlexNet also achieves less model size reduction

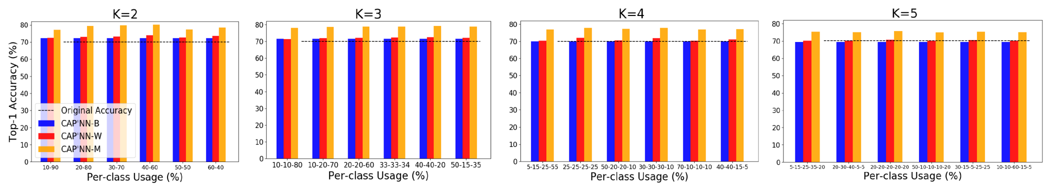

Fig. 8. Top-1 accuracy using various pruning schemes with different number of user-specified classes (K) and usage weights for VGG-16.

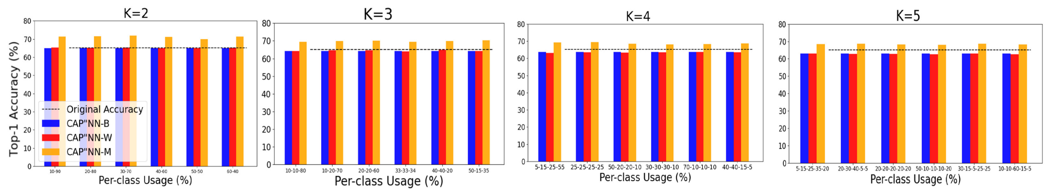

Fig. 9. Top-1 accuracy using various pruning schemes with different number of user-specified classes (K) and usage weights for AlexNet.

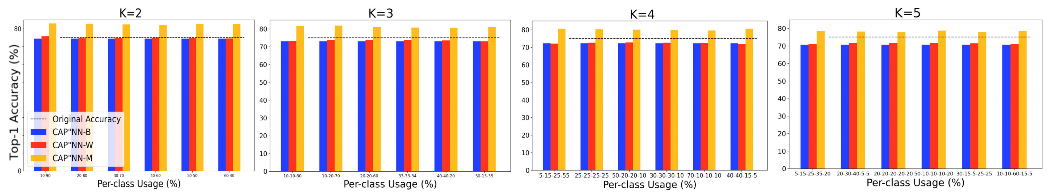

Fig. 10. Top-1 accuracy using various pruning schemes with different number of user-specified classes (K) and usage weights for ResNet-152.

compared to VGG-16. This is because AlexNet is a shallower network with fewer convolutional layers; therefore the last convolutional layers (which are pruned by CAP'NN) tend to extract less class-specific features from the inputs, limiting the potentials for model size reduction.

Accuracy. Figures 8, 9, and 10 show the comparison of top-1 accuracies for VGG-16, AlexNet, and ResNet-152, respectively. As expected, post-pruning accuracy degradation is within $\epsilon = 3\%$ for all of the 24 configurations of each network. More importantly, pruning miseffectual neurons in CAP'NN-M results in accuracy gains in all networks. For example, in case of VGG-16 (AlexNet), CAP'NN-M achieves up to 10% (5.9%) for $K = 2$ and up to 5.6% (3.2%) for $K = 5$.

The plots comparing the top-5 accuracies are similar (not shown for brevity). For all networks, top-5 accuracies are improved. Specifically, for $K = 5$, the top-5 accuracies of VGG-16, AlexNet, and ResNet-152 are improved on average by 4.8%, 2.9%, and 2.5%, respectively.

Next, we evaluate our framework on a larger number of user-specified classes. In this experiment, we applied CAP'NN-M to prune VGG-16 when varying K up to 100 user-specified classes. The results are shown in Figure 11. A higher K increases the relative model size, because it makes the pruning more conservative. We prune the network for only up to 100 user-specified classes, since with 100 classes, the relative model size is 90% of the original model; we can no longer achieve significant reductions beyond $K = 100$. However, the key takeaway of Figure 11 is that regardless

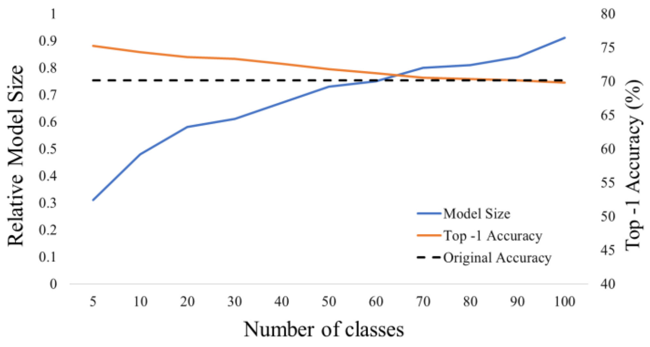

Fig. 11. Model size vs. accuracy tradeoff of VGG-16 for CAP'NN-M as a function of the number of user-specific classes K.

Table 1. Energy Consumption of Different Components in the Network and Relative Energy Consumption of Various Networks for Different Number of Classes

Component	Energy (pJ)	Number of classes	Relative energy consumption of VGG-16	Relative energy consumption of AlexNet	Relative energy consumption of ResNet-152
16-bit adder	0.4	**2**	0.33	0.46	0.51
16-bit multiplier	1.0	**3**	0.36	0.50	0.53
Max Pool/ReLU	1.2/0.9	**4**	0.38	0.53	0.59
SRAM	5	**5**	0.42	0.59	0.65
DRAM	640	**10**	0.56	0.64	0.73

of K, classification accuracy is bounded by $\epsilon = 3\%$ as ensured by all three variations of CAP'NN. We expect that in real use cases, the expected number of classes K is very small.

6.2 Energy Savings

We also evaluate the energy savings of CAP'NN after pruning the networks with CAP'NN-M. We first employ the analytical energy model proposed in Reference [29] and adapt it for our architecture, shown in Figure 2. The energy model is expressed in terms of the number of MAC operations, SRAM accesses, and DRAM accesses per inference. We use the energy numbers of various components from References [9, 22].

Table 1 reports the energy consumption of the components in the network as well as the average energy consumption of VGG-16, AlexNet, and ResNet-152 when normalized to the energy consumption of their corresponding original network model for different values of K. For each K, the relative energy consumption is averaged over different usage distributions and the 200 combinations of K randomly selected classes. As can be seen, CAP'NN can achieve up to 44%, 36%, and 27% energy saving for VGG-16, AlexNet, and ResNet-152 networks, respectively.

6.3 Inference Latency Measurement

In this experiment, we measure the inference latency of the networks when pruned for various numbers of user-specified classes. To measure latency, we first adapted the latency numbers of each individual computation unit including Pooling and ReLU from References [22, 25]. We have also used NVSIM [7] to measure latency of memory accesses. Then, according to the architecture of the underlying layers in each network such as number of input channels, number of output channels, and size of the convolutional kernels, we build our analytical model to map the network into the hardware. The underlying hardware is modeled according to Google's TPU design [16], which is also shown in Figure 2. Since the mapping is done based on the network architecture,

Table 2. Latency of Different Components in the Network and Relative Inference Latency of Various Networks for Different Number of Classes

Component	Latency (clock cycle)	Number of classes	Relative inference latency of VGG-16	Relative inference latency of AlexNet	Relative inference latency of ResNet-152
16-bit adder	1	**2**	0.37	0.49	0.59
16-bit multiplier	1	**3**	0.40	0.55	0.61
Max Pool/ReLU	2/1	**4**	0.42	0.58	0.65
SRAM	12	**5**	0.46	0.62	0.71
DRAM	120	**10**	0.59	0.67	0.79

our analytical model accounts for the available hardware resources. Last, we measure the network latency as the total number of clock cycles that is required to finish the inference for a given input. We then normalized the latency of the pruned network to the latency of the original unpruned network.

Table 2 summarizes the latency of different components in the network as well as the inference latency of the networks when pruned for different number of classes. As can be seen, CAP'NN is successfully able to reduce the inference time of the networks on local devices. For example, in VGG-16, the inference latency is reduced by more than 40% when the network is pruned for 10 user-specified classes. For ResNet-152, the latency improvement is about 20% for 10 user-specified classes.

6.4 Memory Overhead

As discussed earlier, `CAP'NN-W` incurs a larger memory overhead compared to `CAP'NN-B`, as it requires storing per-class firing rates. The incurred memory overhead in `CAP'NN-W` depends on two factors: (1) the number of bits to represent the firing rates, and (2) the size of the firing rate matrix per layer, which depends on the total number of classes and the number of output neurons/channels in the layers selected for pruning.

Note that we need to store the firing rates of the last 5 layers of VGG-16 and AlexNet but do not need to store the firing rates of the output layer. This is because we do not prune the output neurons of the last layer, as each neuron corresponds to one output class in the network. Therefore, CAP'NN requires storage for 5 firing rate matrices. These last 5 layers consist of 3 convolutional layers, each with 512 output channels, and 2 fully connected layers, each with 4,096 output neurons. Each matrix has 1,000 rows (one row per class) and 512 (4,096) columns for convolutional (fully connected) layers. In the case of AlexNet, CAP'NN requires storing 3 firing rate matrices for 3 convolutional layers with 384, 384, and 256 channels as well as 2 fully connected layers each with 4,096 output neurons.

We then linearly quantize the class-specific firing rates into 3-bit values. With this, the total memory overhead incurred by `CAP'NN-W` for VGG-16 is 3.6 MB, a mere 1.3% of the original, unpruned network (with 16-bit weights), which requires 276 MB of storage. For AlexNet, the memory overhead is 3.5% of the original network model.

6.5 Caching Per-layer Pruning Thresholds

Next, we assess the effectiveness of our proposed cache architecture to reduce the timing overheads of CAP'NN.

Clustering Results. First, we apply K-means clustering algorithm to cluster the firing rates of the last 5 layers (excluding output layer) in VGG-16 and AlexNet. The clustering led to 6 clusters for the last 3 convolutional layers and 8 clusters for the last 2 fully connected layers in both networks. Next, we measured intra-cluster and inter-cluster distances for each cluster to evaluate the effectiveness of the algorithm in finding similar classes.

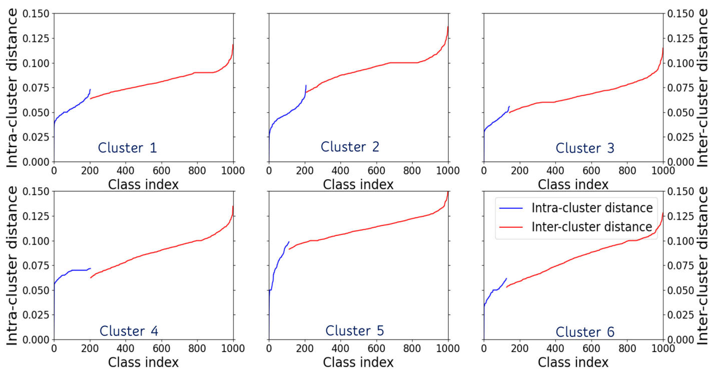

Fig. 12. Average intra-cluster and inter-cluster distances of different clusters in the first convolutional layer of VGG-16.

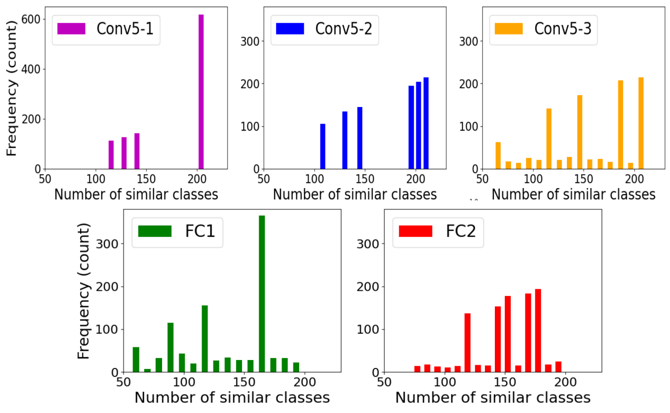

Fig. 13. Distribution of number of similar classes for last 5 layers of VGG-16.

Figure 12 shows average intra-cluster and inter-cluster distances of various clusters in the first convolutional layer of VGG-16. Here, the x-axis shows the class index, and left and right y-axes show average intra-cluster and inter-cluster distances, respectively. As the results show, for all clusters, intra-cluster distance is significantly smaller than the inter-cluster one, indicating that classes in the same cluster have the most similar firing rate vectors.

Next, we show the distribution of the number of similar classes for the last 5 layers in VGG-16. We note that for each class, the number of similar classes is equal to the size of the cluster to which it belongs. As the results in Figure 13 show, the number of per-class similar classes varies from around 50 classes to more than 200 classes, indicating that the clustering algorithm is successful at finding a sufficient number of similar classes for each class.

Cache Performance. Here, we evaluate the performance of our lightweight cache architecture. In this experiment, for each value of K, we randomly selected 600 combinations of classes (i.e., each combination has K randomly selected classes). Then, for each combination of classes, we followed the procedure discussed in Section 4 to find pruning thresholds. More specifically, rather than running Algorithm 2 for each combination, we first look up the cache table with a similar value of K. If a similar subset of classes was found, then we use already-computed thresholds. Otherwise,

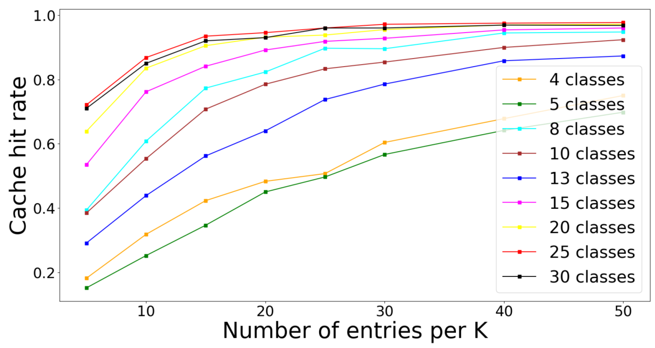

Fig. 14. Cache hit rate as a function of number of entries per cache for various numbers of user-specified classes (K) for VGG-16.

we run Algorithm 2. Finally, we evaluate the cache hit rate by measuring the fraction of pruning thresholds that were directly obtained from the stored thresholds in the cache table.

Figure 14 shows the cache hit rate as a function of the number of cache entries for various numbers of user-specified classes. As the figure shows, increasing the number of entries can improve the cache hit rate for any value of K. This is because, with a larger cache, a greater number of pruning requests can be stored, which in turn increases the chance of finding a similar subset of classes. More importantly, the cache hit rate saturates as we reach around 30–40 entries per cache table, indicating that a relatively small number of entries would be sufficient.

Accuracy. Next, we report post-pruning top-1 accuracy of VGG-16 and AlexNet networks with and without cache architecture for various numbers of user-specified classes, as summarized in Figure 15. Our results show that incorporating the cache architecture may slightly degrade the post-pruning top-1 accuracies given that the pruning thresholds that are obtained from the cache may differ from those that are actually computed by Algorithm 2. However, the post-pruning accuracies are still improved compared to the base unpruned models. For example, the cache architecture degrades accuracy for about 0.2% and 0.5% in AlexNet and VGG-16, respectively, for $K = 30$. Top-5 accuracies are also maintained after incorporating the cache architecture. Specifically, the cache architecture degrades top-5 accuracy for less than 1.1% for both AlexNet and VGG-16 when $K = 30$.

Time savings. Last, we report the time savings, obtained from the proposed cache architecture. For this, we calculate the average number of required pruning iterations to find per-layer pruning thresholds with and without the proposed cache architecture, averaged over 600 combinations of classes. The number of entries per cache is also set to 40 entries, as it maximizes the cache hit rate. Number of required iterations with and without cache as well as the time savings are summarized in Table 3 for the various numbers of user-specified classes in VGG-16 and AlexNet networks. As the results show, cache architecture can significantly reduce the number of required pruning iterations for both networks and under all variations of the number of classes. In VGG-16, the average number of pruning iterations is reduced by around 60% and 90% for 4 classes and 30 classes, respectively. For AlexNet, the time saving is around 37% and 84% for 4 classes and 30 classes, respectively.

6.6 Pruning Overhead

In this experiment, we estimate the time it takes for CAP'NN to prune the original model for a subset of user-specified classes. We also compare the pruning time before and after integrating the cache architecture.

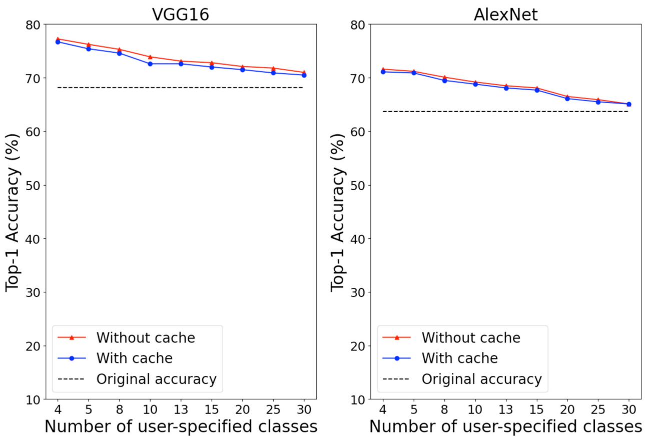

Fig. 15. The top-1 accuracy of the networks, pruned by CAP'NN-M, with and without the cache architecture.

Table 3. Average Number of Required Pruning Iterations with and without Cache Architecture

Number of classes	# Pruning iterations (without cache)	# Pruning iterations (with cache)	Time savings
	VGG-16		
4 classes	10	4	60%
5 classes	15	8	46%
8 classes	17	3	82%
10 classes	18	3	83%
13 classes	20	4	80%
15 classes	23	3	87%
20 classes	28	3	89%
25 classes	33	3	90%
30 classes	40	4	90%
	AlexNet		
4 classes	8	5	37.5%
5 classes	12	8	33%
8 classes	15	7	53%
10 classes	17	8	53%
13 classes	20	6	70%
15 classes	22	6	72%
20 classes	26	6	77%
25 classes	31	7	77%
30 classes	38	6	84%

Pruning time, incurred by CAP'NN, depends on three factors: (1) number of required iterations to find per-layer pruning thresholds, (2) number of required input samples to measure per-class accuracy degradation, and (3) inference latency on the cloud infrastructure for a batch of inputs (recall the cloud is responsible to prune the model). We have used the average number of pruning iterations for various numbers of user-specified classes from Table 3. Per-class accuracy is also measured by running the inference on 50 inputs from each class by CAP'NN. Finally, we have

Table 4. Average Pruning Time of CAP'NN (in Minutes) with and without Cache Architecture

	VGG-16		**AlexNet**	
Number of classes	**Without cache**	**With cache**	**Without cache**	**With cache**
4 classes	0.27	0.11	0.11	0.1
5 classes	0.52	0.28	0.21	0.14
8 classes	0.94	0.16	0.43	0.20
10 classes	1.25	0.20	0.61	0.29
13 classes	1.8	0.36	0.94	0.29
15 classes	2.4	0.31	1.2	0.32
20 classes	3.9	0.41	1.8	0.43
25 classes	5.7	0.52	2.8	0.63
30 classes	8.3	0.83	4.1	0.65

adopted the inference time of VGG-16 on cloud platforms from Reference [13]. As reported in Reference [13], up to 120 images can be processed in a second on Google cloud with 4x Tesla K80 GPU. The throughput can be increased by up to 230 images per second with 8× Tesla K80 GPU. For AlexNet, we scale the throughput by 1.9× according to latency numbers provided in Reference [23] for AlexNet and VGG-16.

Table 4 reports the pruning time of CAP'NN with and without cache architecture for VGG-16 and AlexNet on Google cloud with 4× Tesla K80 GPU. As the results show, without the cache architecture, the pruning time for VGGG-16 is around a minute for up to 10 user-specified classes and it is increased to 8 min for 30 user-specified classes. After incorporating the cache architecture, the pruning time is reduced to less than a minute for all variations of the number of classes. The same trend is observed for AlexNet.

6.7 Cache Overhead

The incurred overhead in our proposed caching scheme depends on two factors: (1) the number of user-specified classes (K) and (2) number of entries per cache (M). More specifically, at each cache entry, we need to store the class indices for the classes in $\mathcal{K}$ along with the obtained pruning threshold for this set. Thus, the cache overhead can be calculated as follows:

$$Cache\ size = M \times (K \times \log_2 |C| + Th_{bits}), \tag{4}$$

where $|C|$ is total number of classes and Th_{bits} is the required precision to store the pruning threshold. Note that one cache table is needed for each value of K.

In our estimation, we limit the number of user-specified classes to 30 classes, given that we expect users to encounter only a relatively small subset of classes. Also, to maximize the cache hit rate, the number of entries per cache is set to 40 entries, as shown in Figure 14. Last, we use 16-bit floating-point representation to store each threshold. We also note that we need five cache tables for each value of K, since the last 5 layers of VGG-16 and AlexNet are pruned.

Given this configuration, the cache size is around 1.4 and 8 KB for $K = 4$ and $K = 30$, respectively. Also, the total incurred overhead of cache for up to 30 classes is around 126 KB. The incurred memory overheads is significantly smaller compared to the unpruned network model size (which is around 276 MB for VGG-16 and 124 MB in case of AlexNet), thus negligible.

6.8 Comparison with Prior Works

As mentioned before, our proposed class-aware pruning techniques can be combined with existing class-unaware approaches to further reduce the model size given a set of user-specified classes.

Table 5. Results of Applying CAP'NN-M to Variants of VGG-16

#Classes	Thinet-Conv [20]		Channel pruning [11]	
Relative model size				
	Without CAP'NN	With CAP'NN	Without CAP'NN	With CAP'NN
2	0.94	0.21	0.90	0.24
3	0.94	0.22	0.90	0.26
4	0.94	0.24	0.90	0.28
5	0.94	0.26	0.90	0.30
Top-1 Accuracy (%)				
	Without CAP'NN	With CAP'NN	Without CAP'NN	With CAP'NN
2	68.0	71.4	69.0	69.8
3	69.3	70.8	69.0	69.1
4	70.1	70.5	67.0	68.8
5	69.8	70.1	67.8	68.5
Top-5 Accuracy (%)				
	Without CAP'NN	With CAP'NN	Without CAP'NN	With CAP'NN
2	88.5	90.8	88.6	89.5
3	89.1	91.3	88.9	89.7
4	89.3	91.6	89.1	90.1
5	88.6	90.9	88.8	89.5

(The variants are pruned with class-unaware pruning techniques).

Table 6. Comparison of Normalized Energy Consumption, Normalized FLOP, and Top-1 Accuracy with Reference [23]

	Normalized Energy		Normalized FLOPs		Top-1 Accuracy	
% of classes	**CAP'NN**	**[23]**	**CAP'NN**	**[23]**	**CAP'NN**	**[23]**
10%	0.36	0.72	0.42	0.67	93.1	90.1
20%	0.48	0.63	0.5	0.61	92.5	89.5
30%	0.56	0.69	0.57	0.62	91.3	89.1
40%	0.68	0.72	0.63	0.67	90.8	88.5
50%	0.76	0.78	0.69	0.75	90.1	88.5
60%	0.81	0.81	0.73	0.77	89.5	88.2
70%	0.87	0.82	0.77	0.78	89.1	88.1
80%	0.91	0.89	0.85	0.85	88.5	87.5
90%	0.92	0.94	0.90	0.92	88.1	87.3

To show this, we first prune VGG-16 using two recently proposed works [11, 20]. This is done by directly using their already-pruned, retrained models. We then apply CAP'NN-M to prune the already-pruned networks for up to $K = 5$ user-specified classes. The results are summarized in Table 5. As can be seen, CAP'NN-M is able to further reduce the relative model size of a class-unaware pruned model by up to 60% as the network is pruned for up to five classes (e.g., relative model size is reduced from 0.9 to 0.3 for $K = 5$ after applying CAP'NN-M on top of Reference [11]). In addition, top-1/top-5 accuracies are improved.

Next, we compare `CAP'NN-M` against a recent work that prunes based on classes [23]. In this work, the authors trained VGG-16 on the CIFAR-10 dataset with 10 output classes. The network achieves around 90.2% top-1 accuracy in this dataset. Then, they pruned the network for up to 10 user-specified classes and reported post-pruning energy consumption.

To compare with Reference [23], we follow the same procedure to train the network. Then, we apply `CAP'NN-M` to prune the network for varying number of classes and report the normalized energy consumption and number of FLOPs in Table 6. Energy consumption and the number of FLOPs are both normalized to those in the original network. As the table shows, for a small percentage of user-specified classes, `CAP'NN-M` yields lower post-pruning energy compared to Reference [23]. As the percentage of classes increases, the energy consumption of `CAP'NN-M` approaches the energy reported in Reference [23]. A similar pattern can be observed for normalized FLOPs. Also, as expected, the advantage of `CAP'NN-M` is most pronounced with a relatively small number of classes.

We also compare post-pruning top-1 accuracies for both `CAP'NN-M` and Reference [23], summarized in Table 6. As can be seen, CAP'NN achieves a higher accuracy for all number of user-specified classes. Also, for a smaller number of classes, `CAP'NN-M` is even able to improve the classification accuracy compared to that of the original unpruned model (i.e., 90.2%).

7 CONCLUSIONS

In this work, we propose CAP'NN as a framework that enables personalized inference by pruning the network for a specific subset of classes based on the user's preferences, thus reducing the network model size. Uncovering the concept of miseffectual neurons, CAP'NN is also able to improve post-pruning classification accuracy. Timing overheads of CAP'NN is significantly reduced by incorporating a lightweight cache architecture that exploits the similarities among different pruning requests from different users. As our experiments show, CAP'NN yields, on average, a relative model size of 65% of the original network and 2.5% improvement in top-1 accuracy as we prune different networks including VGG-16, AlexNet, and ResNet-152 for 10 classes. CAP'NN also reduces the timing overheads for pruning the network up to 90%, when pruning VGG-16 and AlexNet for various numbers of user-specified classes.

REFERENCES

[1] Islam Atta, Pinar Tozun, Anastasia Ailamaki, and Andreas Moshovos. 2012. SLICC: Self-Assembly of instruction cache collectives for OLTP workloads. In *Proceedings of the International Symposium on Microarchitecture*. 188–198.

[2] P. Bholowalia and A. Kumar. 2014. A clustering technique based on Elbow method and k-means in WSN. *Int. J. Comput. Appl.* 105, 9 (2014), 17–24.

[3] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2018. Understanding the limitations of existing energy-efficient design approaches for deep neural networks. *Energy* 2, L1 (2018), L3.

[4] J. Chio, Z. Hakimi, P. Shin, W. Sampson, and V. Narayanan. 2019. Context-aware convolutional neural network over distributed system in collaborative computing. In *Proceedings of the Design Automation Conference*. 1–6.

[5] Y. Le Cun, J. S. Denker, and S. A. Solla. 1990. Optimal brain damage. In *Proceedings of the Conference on Neural Information Processing Systems*. 598–605.

[6] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the Conference on Neural Information Processing Systems*. 1269–1277.

[7] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. 2012. NVSIM: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 31, 7 (2012), 994–1007.

[8] J. Guo and M. Potkonjak. 2017. Pruning ConvNets online for efficient specialist model pruning. In *Proceedings of the Computer Vision and Pattern Recognition Conference*. 113–120.

[9] S. Han, J. Pool, J. Tran, and W. Dally. 2015. Learning both weights and connections for efficient neural network. In *Proceedings of the Conference on Neural Information Processing Systems*. 1135–1143.

[10] Babak Hassibi, David G. Stork, and Gregory J. Wolff. 1993. Optimal brain surgeon and general network pruning. In *Proceedings of the International Conference on Neural Networks*. 293–299.

[11] Y. He, X. Zhang, and J. Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the International Conference on Computer Vision.* 1389–1397.

[12] Maedeh Hemmat, Joshua San Miguel, and Azadeh Davoodi. 2020. CAP'NN: Class-aware personalized neural network inference. In *Proceedings of the Design Automation Conference.* 1–6.

[13] LeaderGPU. [n.d.]. Retrieved from https://www.leadergpu.com/catalog/tensorflow.

[14] Hengyuan Hu, Rui Peng, Y. W. Tai, and Ch. Tang. 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. Retrieved from https://arXiv:1607.03250.

[15] Y. H. Hu, Qiuzhen Xue, and W. J. Tompkins. 1991. Structural simplification of a feed-forward, multi-layer perceptron artificial neural network. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing.* 1061–1064.

[16] N. P. Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture.* 1–12.

[17] Y. D. Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. Retrieved from https://arXiv:1511.06530.

[18] S. Y. Kung and Y. H. Hu. 1991. A Frobenius approximation reduction method (FARM) for determining optimal number of hidden units. In *Proceedings of the IEEE International Joint Conference on Neural Networks.* 163–168.

[19] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. Retrieved from https://arXiv:1608.08710.

[20] J. Luo, J. Wu, and W. Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the International Conference on Computer Vision.* 5058–5066.

[21] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning convolutional neural networks for resource efficient inference. Retrieved from https://arXiv:1611.06440.

[22] M. Nazemi, G. Pasandi, and M. Pedram. 2019. Energy-efficient, low-latency realization of neural networks through boolean logic minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference.* 274–279.

[23] Z. Qin, F. Yu, Ch. Liu, and X. Chen. 2019. CAPTOR: A class adaptive filter pruning framework for convolutional neural networks in Mobile applications. In *Proceedings of the Asia and South Pacific Design Automation Conference.* 444–449.

[24] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing. 2017. LookNN: Neural network with no multiplication. In *Proceedings of the Design Automation and Test in Europe.* 1779–1784.

[25] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the International Symposium on Computer Architecture.* 14–26.

[26] F. S. Snigdha, I. Ahmed, S. D. Manasi, M. G. Mankalale, J. Hu, and S. S. Sapatnekar. 2019. SeFAct: Selective feature activation and early classification for CNNs. In *Proceedings of the Asia and South Pacific Design Automation Conference.* 487–492.

[27] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Proceedings of the Conference on Neural Information Processing Systems.* 2074–2082.

[28] Zhengxin Yu, Jia Hu, Geyong Min, Haochuan Lu, Zhiwei Zhao, Haozhe Wang, and Nektarios Georgalas. 2018. Federated learning based proactive content caching in edge computing. In *Proceedings of the Global Communications Conference.* 1–6.

[29] B. Zhang, A. Davoodi, and Y. Hu. 2018. Exploring energy and accuracy tradeoff in structure simplification of trained deep neural networks. *IEEE J. Emerg. Select. Top. Circ. Syst.* 8, 4 (2018), 836–848.

[30] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays.* 161–170.

Received 29 June 2021; revised 11 January 2022; accepted 19 February 2022