CAP'NN: Class-Aware Personalized Neural Network Inference

Maedeh Hemmat, Joshua San Miguel and Azadeh Davoodi
Department of Electrical and Computer Engineering
University of Wisconsin–Madison, Madison, WI, USA
{*hemmat2,jsanmiguel,adavoodi*}*@wisc.edu*

***Abstract*—We propose *CAP'NN*, a framework for *Class-Aware Personalized Neural Network Inference*. CAP'NN 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 achieves up to 50% model size reduction while actually *improving* the top-1(5) classification accuracy by up to 2.3%(3.2%) when the user only encounters a subset of VGG-16 classes.**

I. 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 [12]. 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.

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 [11], 22% of the top 100 Android applications only use a single output class (e.g., faces, books, etc.) of the ImageNet dataset, which contains 1000 output classes in total and is the widely used for image classification. 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 *Class-Aware Personalized Neural Network Inference (CAP'NN)*. CAP'NN 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 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.

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

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).
- 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 compared to the original unpruned model when pruning for 10 user-specified classes.

II. 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.

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.

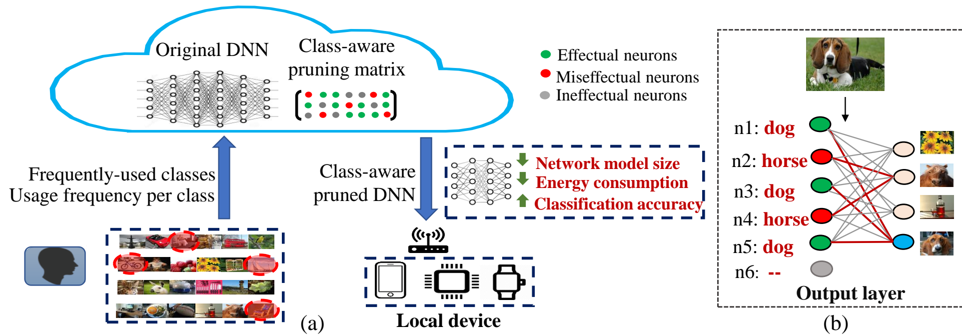

Fig. 1: (a) Overview of CAP'NN; (b) Example showing different types of neurons.

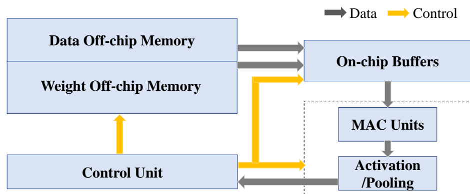

Fig. 2: Architecture view of the local device.

- *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` considers the likelihood of encountering each class in order 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 towards 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. In the monitoring period, the network's predictions are tracked and the most frequently used classes and their usages are determined. 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) [7]. 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.

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.

III. 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 [6]. 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.

A. 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 $\mathcal{C}$ and set of layers[3] $\mathcal{L}$ with neurons $\mathcal{N}_\ell$ $\forall \ell \in \mathcal{L}$
- Firing rate matrix F_ℓ of dimension $|\mathcal{N}_\ell| \times |\mathcal{C}|$ $\forall \ell \in \mathcal{L}$
- 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

[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.

pruning matrix P_ℓ of dimension $|\mathcal{N}_\ell| \times |\mathcal{C}|$ 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 which 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 run-time 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_{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, the candidate neurons identified in the layer are permanently marked as pruned in line 21. Otherwise, the firing threshold T is decreased by a 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.

Given that Algorithm 1 guarantees a bound of ϵ in accuracy degradation for the pruning set identified for each class, the intersection (which is a smaller subset of neurons) also has the same property.

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 $|\mathcal{K}|$.

B. 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.

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.

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 $\mathcal{C}$ 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 $|\mathcal{N}_\ell| \times |\mathcal{C}|\ \forall \ell \in \mathcal{L}$.
Outputs: Pruning matrix P_ℓ of dimension $|\mathcal{N}_\ell| \times |\mathcal{C}|\ \forall \ell \mathcal{L}$.

1: **for each** layer $\ell = l_{start}$ to $|\mathcal{L}|$ **do**
2: CAP-PerLayer(ℓ, $\mathcal{N}_\ell$, F_ℓ , $\mathcal{C}$, ϵ, $\&P_\ell$)
3: **end for**
4: **procedure** CAP-PERLAYER(ℓ, $\mathcal{N}_\ell$, F_ℓ , $\mathcal{C}$, ϵ, $\&P_\ell$)
5: $T = Tstart$
6: $H(n, c) = 0\ \forall n \in \mathcal{N}_\ell\ \forall c \in \mathcal{C}$
7: **for each** class $c \in \mathcal{C}$ **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 ℓ 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** $i \in \mathcal{C}$ **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**

Algorithm 2 `CAP'NN-W-PerLayer`(ℓ, $\mathcal{N}_\ell$, F_ℓ , $\mathcal{K}$, **w**, ϵ, $\&P_\ell$)

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

1: $T = Tstart$; $H(n) = 0\ \forall n \in \mathcal{N}_\ell$
2: **for each** neuron $n \in \mathcal{N}_\ell$ **do**
3: **if** $\sum_{k \in \mathcal{K}} w_k \times F_\ell(n, k) \leq T$ **then**
4: $H(n) = 1$ ▷ *Flag to temporarily prune*
5: **end if**
6: **end for**
7: Temporarily prune neurons n from ℓ 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** $d_i \leq \epsilon \quad \forall i \in \mathcal{K}$ **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**

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

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`.

it needs to store the floating-point firing rates to be used at runtime. This overhead can be alleviated by encoding the floating-point firing rates with fewer number of bits. Finally, similar to `CAP'NN-B`, `CAP'NN-W` guarantees that the pruned neurons cannot yield accuracy degradation beyond ϵ.

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.

C. 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 $c \neq k \ \forall c \in \mathcal{C}$).

`CAP'NN-M` first identifies a set of miseffectual neurons $\mathcal{M}_c$ for each class $c \in \mathcal{C}$. 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 $|\mathcal{K}| \times |\mathcal{C}|$. 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 $|\mathcal{C}|$ output neurons as a fully-connected layer. For each output neuron j:

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

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

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`.

IV. Related Works

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 and Tucker decomposition [8]. Unstructured pruning techniques are based on eliminating unimportant weights and connections. They date back to the 90s [2], though recent works have emerged based on weight pruning [4]. Structured pruning applies pruning of entire layers or groups of weights. In this space, channel pruning is recently studied [5], [13], [9] as an effective way to develop compact and efficient models for CNNs, since the majority of inference energy is consumed by convolutional layers.

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 [11], [3] 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 [1] where the pruning decision relies on the context of the received input, not based on a subset of classes.

V. Results and Discussion

In our experiments, we use the VGG-16 network to evaluate the efficiency of our CAP'NN implementations. VGG-16 is recognized as one of the representative networks in applications involving object classification and localization tasks. The network consists of 13 convolutional layers and 3 fully connected layers, with each layer being followed by a ReLU function. The network is implemented in Tensorflow and is trained and tested on the ImageNet (2012) dataset with 1000 output classes. To compute class-specific firing rates, 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 $T_{start} = 0.4$, reduction step of $step = 0.025$. We also pruned from the last 6 layers of VGG-16 so $l_{start} = |\mathcal{L}| - 6$.

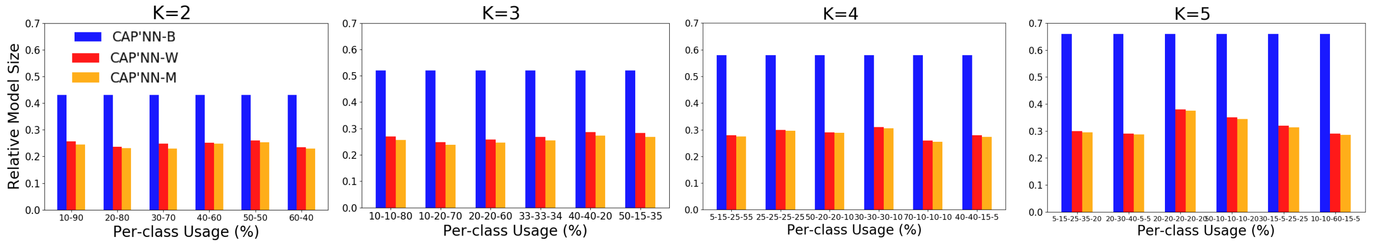

Fig. 4: Model size of VGG-16 of various pruning schemes with different number of user-specific classes (K) and usage weights.

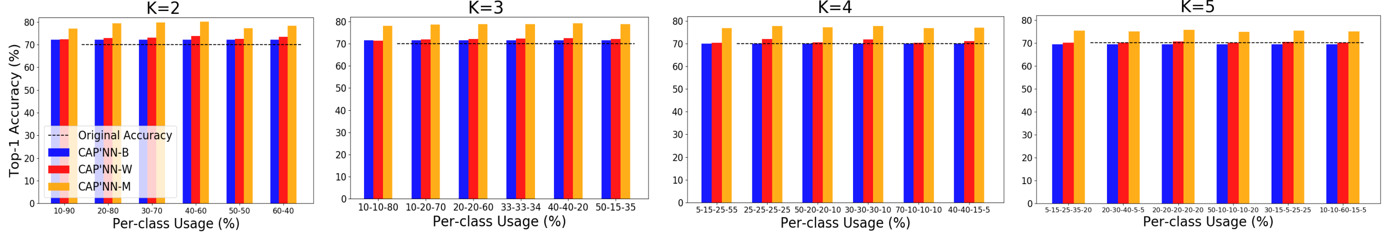

Fig. 5: The top-1 accuracy in VGG-16 of various pruning schemes with different number of user-specific classes (K) and usage weights.

A. Comparison of the CAP'NN Variations

In this experiment, we prune the network 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. Figure 4 shows the average post-pruning model size when the network is pruned for $|\mathcal{K}| = 2, 3, 4, 5$ user-specified classes, normalized to the number of parameters in the original network. Model size is measured by the number of (unique) parameters in the 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 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, when $K = 5$, CAP'NN-B, CAP'NN-W and CAP'NN-M yield relative model sizes of on average 66%, 30% and 29%, respectively. 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 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.

Accuracy. Figure 5 shows the comparison of top-1 accuracies. As expected, post-pruning accuracy degradation is within $\epsilon = 3\%$ for all of the 24 configurations. More importantly, pruning miseffectual neurons in CAP'NN-M results in accuracy gains of up to 10% for $K = 2$ and up to 5.6% for $K = 5$.

The plots comparing the top-5 accuracies are similar and are not shown due to lack of space. The top-5 accuracies are also improved on average, by 7.3% for $K = 2$ and 4.8% for $K = 5$. Extending the experiments to even more classes ($K = 10$), we find that the results are similar (not shown for brevity). The top-1 and top-5 accuracies are improved on average, by 2.3% and 3.2%, respectively. The relative model size is also 0.48 of the original network model size.

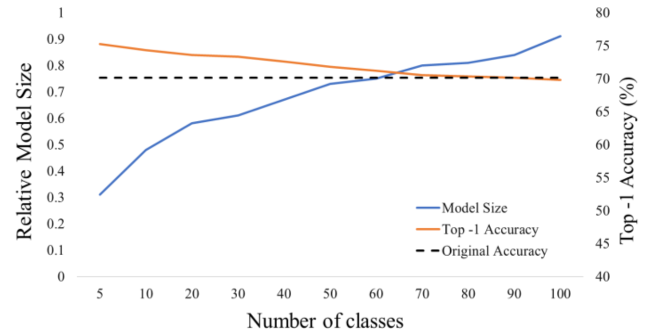

Fig. 6: Model size vs accuracy tradeoff for CAP'NN-M as a function of the number of user-specific classes K.

Next, we applied CAP'NN-M to prune the network when varying K up to 100 user-specified classes. The results are shown in Figure 6. 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 6 is that regardless 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.

B. Energy Savings

We also evaluate the energy savings of pruning VGG-16 with CAP'NN-M. We employ the analytical energy model proposed in [14] 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 [4], [10].

Table I reports the average energy consumption of VGG-16 when normalized to the energy consumption of the original network for different values of K. For each K, the relative energy consumption is averaged over different usage distributions and the 200 combinations

TABLE I: Energy of different components in the network and relative energy consumption of VGG-16 for different number of classes.

Component	Energy (pJ)	Number of classes	Relative energy
16-bit adder	0.4	**2**	0.33
16-bit multiplier	1.0	**3**	0.36
Max Pool / ReLU	1.2 / 0.9	**4**	0.38
SRAM	5	**5**	0.42
DRAM	640	**10**	0.56

of K randomly-selected classes. As can be seen, CAP'NN can achieve energy savings of up to 44% relative to the original model.

C. Memory Overhead

As discussed earlier, CAP'NN-W incurs 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) 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 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 4096 output neurons. Each matrix has 1000 rows (one row per class) and 512 (4096) columns for convolutional (fully connected) layers. We then linearly quantize the class-specific firing rates into 3-bit values. With this, the total memory overhead incurred by CAP'NN-W is 3.6MB, a mere 1.3% of the original, unpruned network (with 16-bit weights), which requires 276MB of storage.

D. 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. To show this, we first prune VGG-16 using two recently proposed works [5] and [9]. 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 II. 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 5 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 [5]). In addition, top-1/top-5 accuracies are improved.

Next, we compare CAP'NN-M against a recent work that prunes based on classes [11]. In this work, the authors trained VGG-16 on the CIFAR-10 dataset with 10 output classes. Then, they pruned the network for up to 10 user-specified classes and reported post-pruning energy consumption.

To compare with [11], 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 reported the normalized energy consumption in Table III. Energy consumption is normalized to the original energy of the network. As the table shows, for a small percentage of user-specified classes, CAP'NN-M yields lower post-pruning energy compared to [11]. As the percentage of classes increases, the energy consumption of CAP'NN-M approaches the energy reported in [11]. As expected, the advantage of CAP'NN-M is most pronounced with a relatively small number of classes.

TABLE II: Results of applying CAP'NN-M to variants of VGG-16. (The variants are pruned with class-unaware pruning techniques.)

#Classes	Thinet - Conv [9]		Channel pruning [5]	
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 / Top-5 Accuracy (%)				
	Without CAP'NN	**With CAP'NN**	**Without CAP'NN**	**With CAP'NN**
2	68.0 / 88.5	71.4 / 90.8	69.0 / 88.6	69.8 / 89.5
3	69.3 / 89.1	70.8 / 91.3	69.0 / 88.9	69.1 / 89.7
4	70.1 / 89.3	70.5 / 91.6	67.0 / 89.1	68.8 / 90.1
5	69.8 / 88.6	70.1 / 90.9	67.8 / 88.8	68.5 / 89.5

TABLE III: Comparison of normalized energy consumption with [11]

#Classes	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
CAP'NN	0.36	0.48	0.56	0.68	0.76	0.81	0.87	0.91	0.92	0.96
[11]	0.72	0.63	0.69	0.72	0.78	0.81	0.82	0.89	0.94	1.0

VI. 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. As our experiments show, CAP'NN yields a relative model size of 50% of the original network and 2.3% improvement in top-1 accuracy as we prune VGG-16 for 10 classes.

References

[1] J. Chio, Z. Hakimi, P. Shin, W. Sampson, and V. Narayanan, "Context-aware convolutional neural network over distributed system in collaborative computing," in *DAC*, 2019, p. 211.

[2] Y. L. Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS*, 1990, pp. 598 – 605.

[3] J. Guo and M. Potkonjak, "Pruning ConvNets online for efficient specialist model pruning." in *CVPR*, 2017, pp. 113 – 120.

[4] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015, pp. 1135 – 1143.

[5] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks." in *ICCV*, 2017, pp. 1389 – 1397.

[6] H. Hu, R. Peng, Y. Tai, and C. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv*, vol. 1607.03250, 2016.

[7] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit." in *ISCA*, 2017, pp. 1 – 12.

[8] Y. D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.

[9] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *ICCV*, 2017, pp. 5058 – 5066.

[10] M. Nazemi, G. Pasandi, and M. Pedram, "Energy-efficient, low-latency realization of neural networks through Boolean logic minimization," in *ASPDAC*, 2019, pp. 274 – 279.

[11] Z. Qin, F. Yu, C. Liu, and X. Chen, "CAPTOR: A class adaptive filter pruning framework for convolutional neural networks in mobile applications," in *ASPDAC*, 2019, pp. 444 – 449.

[12] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016, pp. 14–26.

[13] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *NIPS*, 2016, pp. 2074–2082.

[14] B. Zhang, A. Davoodi, and Y. Hu, "Exploring energy and accuracy tradeoff in structure simplification of trained deep neural networks," *IEEE JETCAS*, vol. 8, no. 4, pp. 836 – 848, 2018.