SMART: Runtime Support for Personalized AI

Boyuan Feng
boyuan@cs.ucsb.edu

Kun Wan
kun@cs.ucsb.edu

Yufei Ding
yufeiding@cs.ucsb.edu

***Abstract*—Runtime class distribution is an essential part in energy-efficient deployment of CNN models on mobile device. Unfortunately, existing papers only focus on the number of classes and overlook the importance of *class correlation*. Unlike such qualitative description as reducing number of classes can increase accuracy, our principles are quantitative: a metric has been proposed to measure *class correlation*. Based on the *quantitative metric*, *guided search* is brought up to replace the existing method *exhaustive search* and thus reduce tons of overhead. Further, *probability layer* is proposed to avoid runtime retraining. We formulate the problem of adjust models according to runtime distribution and energy budget into a scheduling problem and bring up a runtime supporting framework, called *SMART*, to choose model automatically.**

I. Introduction

Image classification is an important area due to their abundant applications in daily life. In recent years, unprecedented advance has been made and the state-of-the-art CNN models can even beat human's accuracy [1] on image classification. These CNN models are expected to be employed on mobile devices and give personalized suggestions. For example, AI-powered smart glasses are desired to detect different types of objects, identify current environment, and give user real-time suggestion. The obstacle to this exciting perspective is energy efficiency problem. The mobile device cannot serve CNN models due to the huge consumption of energy and memory. On solving this energy efficiency problem, runtime class skew has been found to be an effective method. [2] and [3] report that reducing number of classes can both increase accuracy and reduce resource consumption. Actually, only a small portion of classes can be seen in a specific scenario even if the CNNs tested in lab environment target thousands of classes. This method is called *runtime class skew*, since class skew could only be observed when a specific is using the device and cannot be predicted in lab environment.

We extend previous papers from only considering number of classes to incorporating similarity between classes, called *class correlation*. Figure 2 shows the intuition that classifying house, cat, dog, and tree is much easier than distinguishing four different cat species. Through a series of experiments, we found that classes with low *class correlation* have much higher testing accuracy than classes with high *class correlation* even if the model architecture and the number of classes remain unchaned. While number of classes consider how many classes have appeared in a scenario, *class correlation* considers how similar the group of classes are to each other, which is orthogonal to number of classes. To model the class correlation, a semantic tree representing closeness of labels in semantic is a straightforward idea. However, the way that algorithms found similarity is largely different from human's feeling. Thus a much more advanced quantitative method, *QuantClosenesses*, has been proposed to mimic the thinking process of CNN models.

The improvement on accuracy brought in by class skew provides the space for model simplification. There are two difference between our work and existing papers [2, 4].

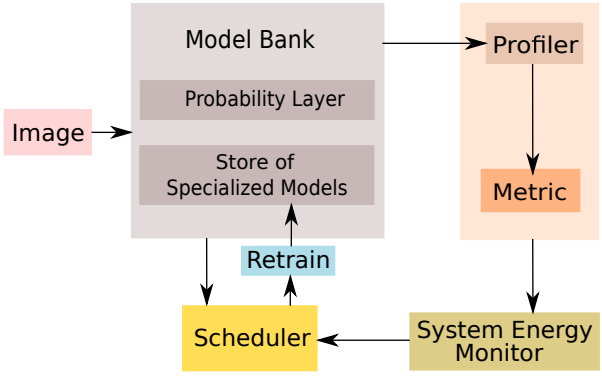
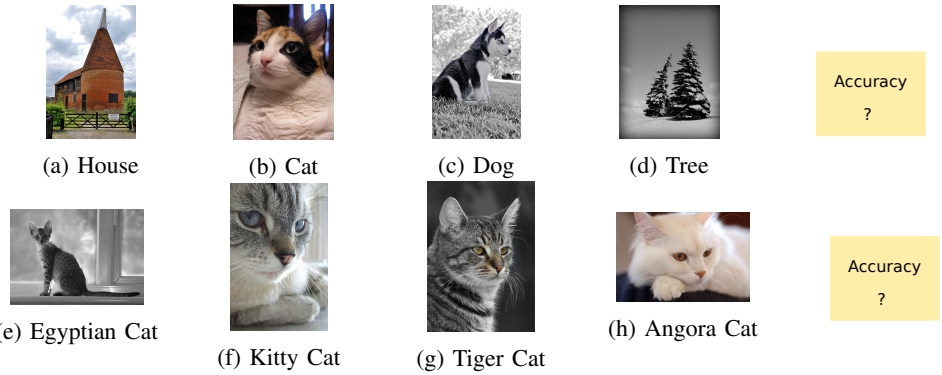


Fig. 1: Overall architecture: This graph shows the overall architecture of SMART framework. The input are images. Every image will be forwarded into the scheduler. The scheduler first refers to the profiler and check whether there is a class skew. Then the scheduler will estimate the energy constraint for this image and choose the model with best accuracy given this energy constraint. The executor will follow the scheduler's decision and grasp the corresponding model from model store, which is a on-disk storage of previous train-ed models. Finally, the predicted value will be recorded by profiler for future usage.

The first difference is how to search candidate models: we use guided search while existing papers use exhaustive search. Previous papers would search all possible combination of optimization methods exhaustively through trying every optimized model one by one without any theory guidance. This exhaustive method is unacceptable in practice since there are mind-bogglingly huge numbers of optimized models. Instead we bring up guided search to give estimation of most suitable models and reduce work in choosing-model process. We will have a mapping from groups of classes to suggested models, including every detail in the model such as input size, number of layers, and even number of nodes in each layer. Secondly, the interaction between optimization methods and class effect is measured in quantitative ways while existing papers have not discussed this interaction. Also, we introduce varying input size and distillation into this area, while all of existing papers only discussed reducing number of classes and number of layers.

To use class skew and optimization methods, retraining the model is unavoidable. Retraining is time-consuming and needs lots of resource. The only method that has been discussed in previous papers is to retrain the last few layers. This method is unsatisfiable due to the huge resource consumption on runtime retraining. We bring up two innovative methods for tackling this problem. Retraining can be totally avoided by using an extra layer, *probability label*, especially when energy budget is limited or cold start. If an environment has appeared for several times and the class skew shows a pattern, cold-training



(a) House (b) Cat (c) Dog (d) Tree

(e) Egyptian Cat (f) Kitty Cat (g) Tiger Cat (h) Angora Cat

Fig. 2: This set of images shows the class effect. It is much easier to distinguish house, cat, dog, and tree than classifying four cat species using the same architecture. This observation meets our intuition.

the model and saving it on disk for future usage is also a feasible method.

Finally, we will bring up a runtime supporting framework, called SMART. Figure 1 provides an overview of the architecture. An image will be sent to scheduler as input. Scheduler is an optimization algorithm to choose the model automatically according to current available resource. To support the decision, the scheduler will refer to model store for available pre-trained models and refer to class catalog for better understanding of difficulty in distinguish target classes. With these information, the scheduler will calculate the available energy budget and choose the best model to maximize the accuracy. Then the chosen model will be forwarded to executor and executor will load and execute that model. The predicted value will be recorded by profiler for future decision. The architecture will be covered in section 5.

In summary, our contribution remains in three parts.

1) Incorporated class correlation into runtime support and gave QuantClosenesses to measure class correlation quantitatively.
2) Upgraded search method from exhaustive search to guided search and generated a mapping from groups of classes to suggested models.
3) Brought up dynamic method *probability layer* for using runtime information and avoiding retraining at all. Added a method to use previous experience and avoid runtime retraining.
4) Constructed *SMART*, a runtime supporting architecture, for making use of run time class skew automatically according to available energy and existing models.

II. Class correlation

A. Impact on accuracy

Runtime class distribution has been widely reported as having power in increasing accuracy and reducing energy ([2, 4, 5]). The factor considered in existing papers about specialization is number of classes. It is believed that as we decrease number of classes, the accuracy would increase. Another belief is that the accuracy would keep similar once the number of classes is chosen, no matter which classes has been chosen. Based on these two belief, frameworks [2, 3, 4] are built and benefits are gained. In these papers, the scheduling framework will select CNN models according to runtime distribution based on model metadata like accuracy and energy cost. Whether the metadata is precise enough is the key to success of the framework, especially the accuracy. In existing papers,

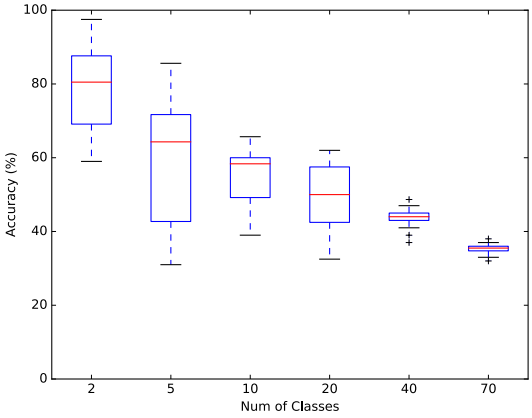


Fig. 3: Box plot shows specialization effect. Accuracy are grouped by number of classes. Red line is the average accuracy for each given number of classes. The blue boxes stand for the 25% and 75% percentile accuracy. The black lines show the minimum and maximum accuracy.

the only factor considered is the number of classes. The only method of collecting accuracy for an architecture is to sample a subset of classes according to the desired number of classes. From our experiments on CIFAR100 [6], we found that class effect is the important factor and the results without considering class effect is error-prone and hard to repeat.

Fig 3 shows the effect from *class correlation* on accuracy, which cannot be ignored. The model has four convolutional layers and three fully connected layers, which is similar to the simplified model commonly used in existing frameworks. The experiment is done on CIFAR100 with 100 fine labels and 20 coarse labels. Each coarse label will have five fine labels, which compose a natural measurement of class correlation. For each specific number of classes, the choice of classes is randomly selected, the model architecture is unchanged and trained with the data corresponding to the chosen classes. Then the accuracy is generated by using trained model to predict the corresponding dataset generated from testing dataset. For each specific number of classes, this experiment is done for 100 times and the accuracy is collected and presented by boxplot. In fig 3, we shows the result when number of classes is $2, 5, 10, 20, 40, 70$, which should be representative. Red line is the average accuracy for each

given number of classes. The blue boxes stand for the 25% and 75% percentile accuracy. Previously, we have the broad idea that reducing number of classes can always lead to an increase in accuracy. While our result also supports this broad idea, two fatal shortage of this broad idea is found and an important direction to use specialization better is found.

The first important observation is that, contrary to our general feeling, decreasing number of classes does not necessarily indicate the increase in accuracy. What make problems worse is that even a negative result may appear when reducing number of classes. As we decrease the number of classes from 10 to 5, instead of increasing, the minimum accuracy decrease from 39.6% to 31%. Further, the minimum accuracy when the number of classes equals five is even much lower than the minimum accuracy when that number equals 70. The accuracy for a given number of classes varies, because the chosen classes varies in each sampling. Since we cannot control what classes will appear in the runtime, the claim that decreasing number of classes can increase accuracy does not hold, or only hold sometimes.

The second pivotal observation is that the variance of accuracy for a specific number of classes is huge, which makes it infeasible to use a single value as the representative for the performance of a model architecture on a specific number of class. When the number of classes is 2, the accuracy varies from 59% to 98%. In this case, the different of the maximum accuracy and minimum accuracy is almost 40%. While 98% is a good enough accuracy for arguing that use simple model to replace original full model, 59% is such a bad result that no one wants to use the simple model, especially considering that the accuracy of random guess is 50% in this case. However, in existing papers, a single value is used to represent the accuracy for a pair of model and number of classes. No one knows whether 59% or 98% is the value being used. And the dramatic variance of accuracy makes neither of them acceptable. From fig 3, we can also find that the variance increases significantly as we decrease the number of classes. Considering that the specialization should be used when number of classes is much less than original number of classes, this observation makes the problem worse.

The reason behind the scene is the *class correlation*. Some classes have strong correlation between each other and it would be extremely hard to classify them. When the number of classes is small, it is more probable that two or five similar classes are chosen simultaneously. When the number of classes is large, it is hard to only select the classes which are hard to classify and the classes that could be classified easily will make the accuracy higher. Thus small number of classes would be more prone to produce worst and best accuracy and give higher accuracy and large number of classes tends to produce similar accuracy and have low variance.

To better support the importance of *class correlation*, we look into detail about which groups of classes give low accuracy and give an intuitive explanation of *class correlation*. Table I shows the choice of classes and the corresponding accuracy. Setting the number of classes as two, the accuracy of classifying boy and girl is 59.2%. If we try to classify fox and pine, the accuracy increase to 92.5%, even if we have changed nothing in the model architecture. The only difference that could lead to this 33% increase in accuracy is the chosen classes. Boy and girl shares more similarity than fox and pine. If we set the number of classes as 5, the accuracy of baby, boy, girl, man, woman is 31.49% while the accuracy corresponding to skyscraper, oranges, girl, bus, oak is 84.6%. This time, there are 53.11% difference in accuracy. Intuitively, the difference between second groups of classes is much larger than the first group. Through experiments, we have proven that this similarity between classes plays an crucial role in testing accuracy. We call this similarity between classes as *class correlation*.

Classes	Accuracy
(boy, girl)	59.2%
(orchid, tulip)	79.0%
(fox, pine)	92.5%
(baby, boy, girl, man, woman)	31.49%
(clock, keyboard, mushroom, orange, pear)	68.60%
(skyscraper, oranges, girl, bus, oak)	84.6%

TABLE I: This table shows the interaction between class correlation and accuracy.

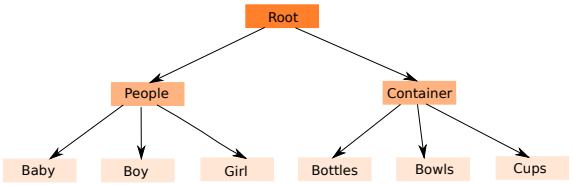


Fig. 4: An example of semantic tree generated from CIFAR100

B. Metric: Semantic Tree

Once have shown the importance of *class correlation*, a metric to represent the similarity is desired and this metric will be used as the base for directed search, a mapping from classes input to suitable model, which will be discussed in section 3. Table I shows that the similarity between natural language could be a good indicator of similarity between classes. From table I, we can find that boy and girl have low accuracy while these two classes belong to the same coarse label: people. This observation also holds for orchid and tulip both belonging to flowers. In contrast, fox and pine belong to different coarse label and thus have much higher accuracy 92.5%, while the other two groups only have 59.2% and 79.0% respectively. This observation also holds when there are five classes. Baby, boy, girl, man and woman have extremely low accuracy as 31.49%, only slightly better than the accuracy of randomly gussing, which is 20%. Again, these five classes have the same coarse label. If we choose classes from two coarse labels, the accuracy would increase dramatically. Clock, keyboard, mushroom, orange and pear are five labels from two coarse labels: electrical devices and fruit. The accuracy for these classes would increase to 68.6%. The accuracy would increase further, if we choose classes from five different coarse labels. Skyscraper, oranges, girl, bus and oak comes from five different labels and the accuracy is 84.6%, which is much higher than previous two groups. Based these observations, we find *semantic tree* is a good metric for class correlation, which measures the similarity between concepts from the perspective of human language.

Semantic tree is a tree-structured storage of images, widely used to organize images. In CIFAR100, there are 20

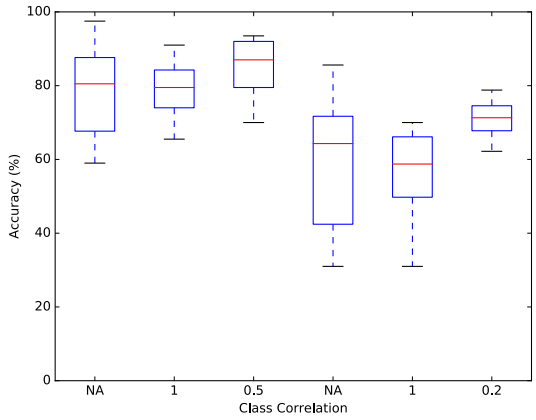


Fig. 5: Showing the boxplot of accuracy grouped by class correlation. The first three is the boxplot for class number equals 2. The second three are the boxplots when class number equlas 5. NA means the accuracy when we do not consider class correlation. Numbers are the class correlations calculated following the definition of metric tree.

coarse labels and five fine labels under each coarse label. 2500 images are labelled by each coarse label and each fine label contains 500 images. Thus there is a hierarchy structure organizing images naturally. ImageNet [7, 8] also organizes images in a tree-structured way. Thus semantic tree is widely existed in various large dataset and could be easily accessed. In a tree-structured storage, images will be grouped by labels and labels will be indexed like a tree according to their similarity in semantics. For example, the root might be mammal and has two children: dog and cat. Under the cat node, there are some children and each children node represents a cat species. Fig 4 shows a part of semantic tree in CIFAR100. The second layer contains coarse labels and the third layer contains fine labels. For example, "people" is a coarse label, and the 5 corresponding fine lables are "baby, boy, girl, man and woman" from "people" label. Thus "people" is the child of root and "baby, boy, girl" are the children of "people" node. If two images belongs to different coarse labels, we say that there is low class correlation between these two images. For images with the same coarse label but different fine labels, they are said to have high class correlation. If two images have the same fine label, they are indistinguishable from the point view of prediction. In CIFAR100, we can define the similarity of a groups of classes as

$$\frac{1}{|\{CL_1\} \cup \{CL_2\} \cup \cdots \cup \{CL_n\}|} \tag{1}$$

, where CL means coarse label and CL_i means the coarse label corresponding to the ith entry. The intuition of this value is that classes from same coarse label are treated as similar and as more classes come from the same coarse label, the class correlation will increase and the metric will also increase. Thus the larger the metric is, the more correlation selected classes have.

Semantic tree is powerful in predicting the testing accuracy. Using the same model, classifying classes close to each other in semantic tree tends to have low accuracy while high accuracy often comes with classes far from each other in the semantic tree. The accuracy is 75% when we use Model1 to predict from 5 coarse labels: quatic mammals, flowers, household furniture, insects, and people. In contrast, if we use finer labels inside each of these coarse labels, like "baby, boy, girl, man and woman" from "people" label, or "beaver, dolphin, otter, seal, whale" from "aquatic mammals" label, the accuracy would decrease dramatically to 55%, even if we have changed nothing with number of classes in the softmax layer and the model architecture. This result shows that semantic tree has certain power in predicting accuracy: classes with high similarity gets low accuracy and classes with low similarity gets high accuracy.

Fig 5 shows the benefit gained by semantic tree from a statistical perspective. The first three box plots are for two classes and the second three box plots are the accuracy when number of classes is 5. NA stands for the case when we do not use semantic tree. The numbers on x axis are the class correlation calculated by the formula mentioned before. When number of classes is 2, without considering class correlation, the accuracy ranges from 59% to 97%, where the difference is 38%. If the class correlation is 1, which means strong class correlation, the accuracy would range from 66% to 90%, where the difference decreases to 24%. When the class correlation is 0.5, the accuracy would range from 70% to 90%, where the difference decrease to 20%. When number of classes is 5, without considering class correlation, the accuracy ranges from 31% to 85%. In this case, the difference between minimum accuracy and maximum accuracy is 54%, which is extremely huge. After considering class correlation, this difference decrease to 39% and 13%, for class correlation equals to 1 and 0.2 respectively. Thus considering both number of classes and class correlation would decrease the variance of accuracy and make the metadata used in scheduling model more trustable.

III. Guided search: Mapping from Classes to Model

Through using runtime distribution and a series optimization methods, we will have a trade-off between accuracy and energy. These two parameters will be measured in advance and compose the base for runtime decision by the scheduling model *SMART*. *SMART* will take into consideration the remaining energy, desired accuracy, and pre-collected model accuracy. The intuition of *SMART* is to use a complex enough model such that the pre-collected accuracy for that model can match the target while the selected model are not too complex to cost too much energy. In this case, the precision of metadata becomes critical for the efficient runtime decision. If the variance of accuracy for a model on different groups of classes is too large, we cannot use a single number to represent the model accurac, which results the instability of the whole framework performance.

In existing papers [2, 3, 4], the framework for storing metadata and supporting runtime decision can be summarized in two equations. In storage, the metadata is organized in the following format

$$(model, numberClasses) \Longrightarrow accuracy, \tag{2}$$

while in runtime, the information is retrieved by the following function

$$model = f(target, numberClasses) \tag{3}$$

, where target is the desired accuracy. As we have demonstrated in section 3, previous framework does not consider class correlation and, for a given model and number of

classes, the difference between maximum and minimum accuracy could be as large as 50%. Thus the representative ability of this framework is very limited.

To give a more precise metadata, we must take the class correlation into consideration. The main obstacle is that, for a given number of classes, the combination of classes is mind-bogglingly huge. For example, if we have 100 classes in total, the number of possible combinations is 100!. It is entirely impossible to measure and record their accuracy one by one. Our solution is to use class correlation. We will first use metric tree to calculate a digit to represent the class correlation between selected classes. Then groups of classes with that class correlation will be sampled for 100 times and the accuracy for these groups are tested. After the experiments, we will collect the average accuracy and the difference between the maximum accuracy and minimum accuracy for each class correlation digits. In summary, the metadata will be organized in the format

$$(model, numberClasses, metric) \Longrightarrow accuracy, \quad (4)$$

in the preparation period and in the runtime we will query the databse for metadata by

$$model = f(target, numberClasses, metric). \quad (5)$$

This method will perform better than previous work since variance will decrease dramatically after we divide groups of classes by class correlation. One shortage is that sometimes the difference between maximum and minimum accuracy is still larger than 20% which makes the metadata not so trustable. To ease this problem, we will use 25-percentile accuracy, instead of accuracy, to represent the testing accuracy when the difference is larger than 20%. Through this way, we can have strong confidence that in more than 75% runs the selected model will perform at least not worse than the target accuracy. This is a system design choice favoring accuracy over energy. This design choice will force the framework accuracy better than user's expectation on accuracy. The overhead introduced by the design choice will be made up by scheduling model through considering many other models when possible, since different model will have various difference between maximum and minimum accuracy for a given number of classes and class correlation. To match this design choice, the storage and query format should also be modified appropriately as

$$(model, numberClasses, metric) \Longrightarrow adjusted\_accuracy, \quad (6)$$

for storage and

$$model = f(target, numberClasses, metric) \quad (7)$$

for query.

IV. Optimization Methods

The improvement in accuracy brought in by class skew provides the space for model simplification. In other words, even if we simplify a model dramatically and use much less resource, we can still achieve a similar accuracy with the original full model. Existing model optimization methods (Reference TO BE ADDED) does not make use of the class skew and cannot exploit thoroughly the space introduced by class skew. Thus we explore a series of class-skew related optimization methods, including number of nodes in softmax layer, number of layers, distillation, and reducing input size.

A. Reducing number of nodes in the softmax layer

Number of nodes in the softmax layer represents number of classes that we want to predict from. When less number of classes appears in an environment, a natural change in the CNN architecture is to reduce number of nodes in the softmax layer. The benefits are two fold. First, reducing number of classes can increase testing accuracy. If we randomly guess from 1000 classes, the expected accuracy would be 0.1%. The expected accuracy for random guess would increase to 50% if there are only 2 classes. This observation also holds for CNN models. Second, reducing number of classes can decrease memory usage. In VGG16 [9], 89.36% parameters exist in the fully connected layer. Reducing nodes in the softmax layer can reduce parameters proportionally and use less memory. A point worth noting is that most of computation exists in the convolutional layers. Thus reducing nodes in the softmax layer cannot save computation and power. This task is generally achieved by other optimizations.

B. Reduce number of layers

Reducing number of layers is a prevalent methods in specialization, which has benefit in reducing computation and shortage in decreasing accuracy. Reducing number of layers, especially the convolutional layers, is a good method in reducing computation. Taking VGG16 for example again, 99.20% computations exist in convolutional layers and spreads evenly across all the convolutional layers. Thus reducing number of layers can reduce computation proportionally. The negative side is the decreas of accuracy. Less layers means a shallower network which has been reported [2, 5] to have worse performance than deep models. However, this decrease in accuracy is made up by the increase introduced by reducing number of nodes in softmax layer. Thus these two methods together can decrease energy and increase accuracy, which is the key for the benefit claimed in previous papers.

C. Distillation

Distillation [10] is a method that can contribute to the model specialization and has not been introduced into specialization. The idea of distillation is using a deep model to train a shallow model. Deep models with higher accuracy would contain more information than shallower models. When we train the shallow model, we can treat the deep model as teacher and the shallow model as student. Instead of train the student model directly on the original datasets with a vector of only 0-1, we can train the student model to mimic the logits of large model, which indicates how strongly the teacher model believes that this image belongs to this specific class. In this way, the student model can learn more information from the teacher and achieve a higher accuracy.

Original paper [10] assumes that the number of classes in the shallow model is same as the number of classes in the deep model. This does not hold in specialization. In distillation for specialization, we will simply truncate the output of softmax layer and only use the output corresponding to the observed classes.

D. Input Size

Reducing input size is another method that exists in other areas and has not been introduced into specialization [6, 11]. Input size has influence on resource computation and model accuracy. With the same model on CIFAR100, if we reduce the input size from 32*32 to 16*16 and further to 8*8, the accuracy will only decrease from 25% to

23% to 21%, in the mean time the computation and memory cost has decreased by 4 and 16 times, respectively. Thus we can treat input size as a hyper-parameter and fine-tune this parameter according to available resource and targeted accuracy. In the training period, we need to downsample the training dataset into a smaller resolution and train the model on this down-sampled dataset. In the reference period, the input image should also be down-sampled to the same resolution. In this way, we can get significant benefit in energy efficiency while only lose little on accuracy.

V. Important factor in real life usage: Retrain

To use runtime class distribution, retraining is required since originally we only have a full model targeting thousands of classes. We cannot pre-train models for all possible combination of classes due to two reasons. First, we cannot predict the runtime class distribution before we actually see it. Second, the total combination of classes is a mind-boggling number such that it is unfeasible to pre-train models for each of the combination. Since the resource consumption by retraining is huge, methods for avoid retraining the whole network is desirable. We will discuss the shortage of method used by existing papers first and then bring up two innovative methods to avoid runtime retraining.

A. Retrain last few layers

The only existing methods for solving retraining problem is to retrain the last few layers, widely used in specialization [2, 4, 5] and transfer learning [12, 13, 14]. In this method, only the last few layers will be retrained. Parameters of preceding layers will be kept and architecture will remain unchanged. There is no strict standards to determine how many layers should be retrained or predict the accuracy for the choice. Literature reports that more layers are retrained, more benefit on accuracy is gained.

There are both good and bad sides for this method. Good side is that retrained model can still make use of features learned by leading layers and retraining from scratch can be avoided. The bad sides are the energy constrain and time latency. In the retraining process, hundreds of images need to go through the network for several rounds. In the resource constrained environment, this is a unbearable waste of energy. Without this waste, the device can process tens of thousand images and give user better endurance. Further, the time latency is also annoying even if the device has sufficient energy. The shortest retraining time reported by [3] is four seconds. Compared to network latency measured in milliseconds, four seconds latency is too long to be accepted and will deteriorate user experience dramatically. To solve both the energy wasting problem and long latency problem, we will bring up *cold retraining* and *probability layer* to supplement or even replace retraining last few layers.

B. Repeated patterns in daily life and cold retrain

In daily life, same environment and class distribution will appear again and again. Our family members are fixed for tens of years. In office, we will stay in the same company for at least several months and the collaborators are generally fixed. In films like *The Big Bang Theory*, characters are also fixed. Using this repeated pattern, we do not need to retrain models every time. We can retrain the model once and use that model for all similar environment afterwards. Actually we even do not need to retrain the model in the runtime. In the first time we confront a new scenario, we can record the distribution on disk and use *probability layer* described in the next subsection to be the runtime cold-start. When the users go back home and connect the mobile device to power and WiFi, we can cold retrain the model to whatever extent without worrying about the energy and memory limit, which would give us more benefit on accuracy.

C. Probability Layer

Probability layer is an extra layer after the softmax layer in the CNN models. With *probability layer*, we can use runtime class distribution without retraining. As we have discussed previously, there are gains in both reducing energy consumption and time latency. Existing methods, i.e. retraining last few layers, requires thousands of images to go through the network for tens of rounds, which waste lots of energy. The retraining also takes time and introduce several seconds latency during which new images cannot be processed. With *probability layer*, all of these inconvenience will be avoided and better performance will be gained. Thus *probability layer* is a brilliant supplement or even replacement of retraining last few layers.

To implement probability layer, the only thing we need to do is to add an extra layer after the softmax layer. Originally, the last layer of CNN model is softmax layer. The output of softmax layer is a vector $(p_1, p_2, ..., p_n)$, where n is the number of nodes in the softmax layer, i.e. the number of classes that the CNN model wants to predict from. Also, p_1, ..., p_n is a series of numbers between 0nd 1. Thus we would treat p_i as the probability that indicates how likely an image may belong to a specific class i. The main idea of probability layer is to add a constant to classes that is more likely to be the true label according to the runtime distribution. For example, if the runtime distribution shows that only the first 10 out of 100 classes could appear in a context, the probability layer will add a constant to p_1, ..., p_{10}, the predicted probability of these 10 classes. In other word, in this case, the probability layer is $c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, 0, ..., 0)$, where c_i is a constant between 0 and 1 and the output of the probability layer is $(p_1 + c_1, p_2 + c_2, ..., p_{10} + c_{10}, p_{11}, ..., p_{100})$. Here we treat c_i as hyper-parameters. The larger the c_i is, the stronger confidence we have in runtime distribution.

The success of probability layer depends on the accuracy of original model. Intuitively, adding a constant to a set of domain classes according to the run-time distribution would encourage the model to select among these domain classes. If the original model is precise, i.e. have high top-5 accuracy, this constrain could rule out some classes and help the model to get a correct result. For example, if the output of softmax layer is $(0.3, 0, ..., 0, 0.7)$ and the probability layer according to runtime distribution is $(0.5, 0.5, 0.5, 0.5, 0.5, 0, ..., 0)$, the output of probability layer would be $(0.8, 0.5, 0.5, 0.5, 0.5, 0, ..., 0.7)$. Assume the correct label is 0, the prediction without probability layer is 99, which is wrong, and the prediction with probability layer is 0, which is correct. In this case, probability layer helps us to rule out the unlikely class 99.

A possible case is that the probability corresponding to the true label are not highest even in domain classes. In this case, the probability will not be helpful. We found that top-5 accuracy is a good meric to measure how well the probability layer could help the original model. In other word, the benefit of probability layer is to improve the top-1 accuracy to be as large as the top-5 accuracy. If the top-5 accuracy is very high, the true label would have the

top-5 high probability. The probability layer will constrain the predicted label to be the intersection between these top-5 labels and the domain labels according to runtime distribution. Thus we can increase top-1 accuracy toward top-5 accuracy.

However, if the number of domain classes are 5 and the top-5 accuracy of the model is low, it is possible that the probability of true label is not the highest among the 5 domain classes. In this case, the benefit of probability layer would be slight. Thus, the higher top-5 accuracy the original model have, the better effect the probability layer would bring. In other word, if we use probability layer on a model with low top-5 accuracy, the benefit would be small.

D. Comparison with naive mask

A naive replacement of *probability layer* is to mask all non-domain classes, called *naive mask*. Probability layer is a generalization of naive mask. In fact, naive mask is equivalent to set the parameter c in probability layer to be 1.0. The intuitive explanation of naive mask is to only have nodes according to the main classes since the predicted label could only be one of these domain classes. If we use naive mask, all the prediction of images with classes besides the main classes would be wrong. This would limit the usage of naive mask dramatically. For example, if the domain classes occupy 70% in the runtime distribution, the accuracy with naive mask would be less than 70%, which is smaller than the accuracy without naive mask and makes naive mask useless, even if 70% skew is not a low skew in daily life.

Probability layer can avoid this problem if we set the parameter c to be strictly less than 1.0. This benefit depends on high accuracy of original model again. Generally, in denseNet [15], if an image could be classified correctly, the probability corresponding to the correct label after softmax layer is 1, and the probability corresponding to other labels are almost 0. For example, if the true label of an image is 11, the output of softmax would generally be $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, ...0)$. Even if we add a probability layer to first 10 classes and use the parameter as 0.9, the output would be $(0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 1, 0, 0, 0, 0, ...0)$. In this case, the model would still mark 11 as the correct label and the probability layer has no bad effect. However, if we use naive mask to rule out all classes other than first ten classes, the prediction would be constrained in the first ten classes and will be wrong.

In other words, the probability layer only has bad effect when the original model is not quite sure about how to classify an image. For example, if the true label of an image is 11, the output might be $(0.3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.7, 0, 0, 0, 0, ...0)$. After we add the probability layer to first 10 classes and use the parameter as 0.9, the output would be $(1.2, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.7, 0, 0, 0, 0, ...0)$. In this case, the model would mark 1 as the correct label and the probability layer have a bad effect. However, the happen of this bad effect needs two conditions. First, the labels with predicted probability 0.3 and 0.7 also appear in the first ten classes. Second, the predicted probability of true label is lower than the other one. Actually this is a rare case.

VI. Architecture

In this section, we will describe the architecture in detail. Figure 1 gives an overview of the architecture. Basically, we have 5 parts including two storage parts, class catalog and model store, one scheduler, one executor, and a profiler. The storage components, class catalog and model store, are used for saving metadata and providing supporting information for decision. The scheduler is a scheduling model to decide the choice of models in the runtime according to available resource and demanded accuracy. The target of the scheduler is to maximize the accuracy while adjust the energy consumption automatically according to remaining energy. The scheduler will also have a have high-accuracy mode just in case that users confront important scenario and want to get the best accuracy no matter how much energy the model will cost, or the user knows that a re-charge will be available soon. Once the scheduler made a decision, the executor will execute the model and the profiler will record the predicted value and update its record for future reference from the scheduler. Now we will discuss each component in detail.

A. Profiler

For every image, there would be a CNN model to classify it, either a full model or a specialized model. Thus every incoming image will have a predicted label. The profiler will record the distribution of classes and report the class skew to the scheduling model, which will use this information to choose the model. There would be two behaviors of profiler. First, the profiler will report the class skew every minute. Generally, the class skew would keep similar and change gradually as time goes and the class skew in current minute is a good approximation of the class skew in the following minute. And one minute is not a too short time period and hopefully a one-minute time window could see all the type of figures in current environment. This choice is a parameter instead of an unchangeable assumption. Second, after the profiler report the class skew to scheduler, it will still record the class skew. Through this way, the profiler can know, for this specific user, what kind of class skew will appear, how long the class skew will last, and how often this class skew will appear. This could be an important supporting information for the scheduler. If a class skew appears frequently, we can cold-retrain a complex model and save the model on disk. Once we see the class skew again, we can use this on-disk specialized model directly without retraining. Actually, it is quite common for a specific user that the same class skew appears everyday. For example, the people in family always remain same; the researchers will go to the lab everyday and the people in lab generally is changed once per year; the type and location of fitness equipments in a gym will also be same day by day. Once the profiler can detect this kind of class skew, the cold-retrain complex model could become a feasible choice.

B. Class catalog

When the profiler detect a class skew, we will consider not only the number of classes, but also the relationship between these classes. Class catalog is a tree-structured dictionary to group similar classes together. As we have discussed in section 2.2, if the detected class skew is close to each other in class catalog, it would be hard to classify these classes and the scheduler will use a more complex model to achieve the demanded accuracy.

C. Model Store

In the beginning, there would be a collection of models without consideration on class skew. These models are for different purpose and with different degree of optimization. In the running time, as the profiler detect the

class skew and usage pattern for the specific user, more specialized model, either hot retrained or cold retrained model, will be incorporated into the model store. When the device is connected to power and computation resource is enough, each model will be run on a benchmark, a collection of common datasets, like CIFAR100, and the average accuracy, energy and memory consumption will be recorded. In the run time, the scheduler can read this metadata from model store and use these information for decision.

Another task that model store will accomplish automatically is to get the most efficient model for every energy budget. Given a energy budget, only the model with highest accuracy will be kept. In the same time, only the model with lowest resource consumption will be kept given a accuracy. This can be done by model store automatically since it is straightforward to calculate the number of parameter and computations just by going through a model without actually running. In this way, only the model that achieves Pareto optimality will be kept in model store and the scheduler can easily choose the most suitable model according to current energy budget just like a hash function.

D. Scheduling Model

The task of scheduling model is to select a series of models to process the image stream under a resource constrain. The target is to maximize the accuracy while try to use less energy for each image. The choice is between a cascades of models with different accuracy and different energy consumption. To solve this optimization problem, we build a heuristic algorithm, in Algorithm 1 and Algorithm 2 to allocate energy for each input image, choose the most suitable model given the per frame energy, and wrap all the decisions automatically.

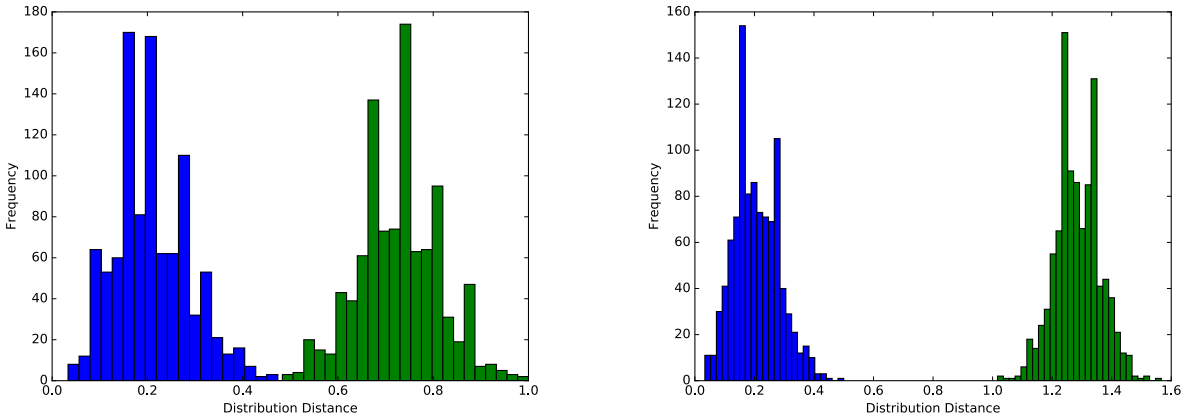
Through a series of experiments (TO BE ADDED) on real-life videos, we found that the contents of a video would almost remain same in a single second, a process rate of one frame per second is enough for detect objects in the video. For energy efficiency consideration, we will use this configuration and estimate the per frame energy based on this observation. In concrete, $AllocateEnergy$ (line 11) will retrieve the expected running time and estimate per frame energy by averaging the available energy over all remaining time. In this case, saving energy in each frame still makes sense, since the saved energy will be allocated to future inputs and increase the accuracy of these inputs.

For giving more freedom to users, a $mode$ option is given through $scheduler$ (line 1) function. This option is useful if the user feel that current scenario is very important such that we want to sacrifice energy-efficiency for exchanging higher accuracy. In this case, the scheduler will expect to run for at least 10 minutes, no matter how long the original settled remaining time is, and allocate more energy to each frame.

Given per frame energy budget and recorded class skew, $ChooseModel$ will return the best model. If no class skew exists, $ChooseModel$ do not need to consider retraining and will choose the model with highest accuracy given current energy budget. When there is class skew, the $ExistedRetrainedModel$ will check whether there are retrained model for current class skew and return a suitable model. When there is no retrained model on disk, the architecture will choose between retraining last few layers and using probability layer. Since retraining is costly, we expect to have enough energy to run that retrained model for at least 10 minutes, otherwise it is more feasible to avoid retraining and use probability layer directly.

Algorithm 1 Scheduling Model

1: **function** SCHEDULER($i, r, mode$) ▷ i is the input image, r is the report from profiler about current class skew, mode is the user preference, target indicates the desired accuracy if mode is set to be EFFICIENCY
2: $AE = RemainEnergy()$ ▷ AE stands for available energy
3: **if** mode == HIGHACCURACY **then**
4: $PFE = AE/600$ ▷ PFE stands for perFrameEnergy, which is the energy allocated for every frame.
5: **else if** mode == EFFICIENCY **then**
6: $PFE = allocateEnergy(AE)$
7: **end if**
8: $a, m = chooseModel(PFE, r)$ ▷ chooseModel(PFE,r) is a function to access the metadata from class catalog and model store to decide the best accuracy we can achieve under current energy constrain and which model we should use
9: $execute(i, m)$ ▷ execute(i,m) will retrieve the model m from model store and process the input image i with model m
10: **end function**
11: **function** ALLOCATEENERGY(AE) ▷ allocateEnergy(AE) allocates per frame energy according to current avalable energy and class skew
12: $n = RemainingTime()$ ▷ remainingTime() will give the expected time to keep running. We will process one image per seconds and n is the remaining time in second as well as the expected number of images to classify.
13: $PFE = AE/n$
14: **return** PFE
15: **end function**
16: **function** CHOOSEMODEL(PFE, r)
17: **if** $Parse(r) == NoSkew$ **then**
18: $a, m = getModel(PFE)$ ▷ If there is no class skew, getModel(PFE) will return the model with highest accuracy given the per frame energy budget PFE
19: **else if** $Parse(r) == Skew$ **then**
20: $a, m = ExistedRetrainedModel(r, PFE)$
21: **if** $m == NULL$ **then**
22: **if** $PFE > \Delta E/600 + FrameCost(m, r)$ **then**
23: $a, m = retrain(m, r)$
24: **else**
25: $a, m = probabilityLayer(m, r)$
26: **end if**
27: **end if**
28: **end if**
29: **return** a, m
30: **end function**



(a) Four classes overlapping

(b) Two classes overlapping

Fig. 6: Graphs showing the distance between data distribution of two classes groups

The scheduler will wrap up all the algorithms in scheduling model and give the most suitable model. For every input image i, The scheduler needs to get the class skew report r from profiler. In the high accuracy mode, enough energy is given to each image such that more freedom in choice of models is allowed. In the efficiency mode, $allocateEnergy$ will estimate per frame energy according to expected remaining running time. Given this per frame energy and class skew report, the chooseModel will give the best model m and the corresponding accuracy a. Finnaly, the $scheduler$ will call $execute(i, m)$ to run the model m on input image i.

VII. Experiments

A. Detection overhead

The fundamental of using runtime distribution is detecting runtime distribution, mainly focused on the number of classes and what classes are appearing. Profiler is the component in *SMART* taking care of this detection. As images come, full models will classify these images and profiler will store the results. For every 120 processed images, the profiler will calculate the distribution of these classified images and compare with the distribution for last 120 images. We choose 120 because generally we will process 1 image per seconds and 120 images will represent the distribution of classes in two minutes. Assuming there are n classes in the full model, the frequency of each class will be recorded as f_i, which is the proportion that this class appears. The profiler will calculate the difference between two distributions by

$$distance = \sum_{i=1}^{n} (f_i^1 - f_i^2) \tag{8}$$

, where f_i^1 is the frequency of class i in the first distribution and f_i^2 is the frequency of class i in the second distribution. If the distance is less than 0.5, we will claim that distribution has been stabilized and a runtime distribution has been detected. Fig 6 shows profiler performance. In fig 6a, the blue histogram is the distribution of distance when all images are selected from class 0 to 4 and the green histogram is the distribution of distance when images comes from classes 0 to 4 and 1 to 5 alternately. If the distance between two distribution is less than 0.4, we can assert that classes in these two consecutive time period are same. In fig 6b, the meaning of blue and green histogram are same as in fig 6a. The only difference is that the groups of classes in fig 6b is 0 to 4 and 3 to 7, where class groups have less overlap than fig 6a. In this case, the distance between two histograms larger. As we decrease the overlap further, the distance between two histograms will become larger. Thus the choice of 0.4 as detection criterion works. Based on these experiments, four minutes or 240 images would be enough for detecting runtime class distribution.

B. Probability Layer

A series of experiments on various kind of specialized datasets have shown the effectiveness of probability layer. CIFAR100 is a prevalent benchmark for various kind of CNN models, which has 100 classes, a training dataset with 500 images per class, and a testing dataset with 100 images per class. To mimic the runtime distribution, we generate a specialized dataset manually. For example, in generation of a specialized dataset with 80% data skew and 10 domain classes, we would use 1000 images from the testing dataset according to these 10 domain classes, and randomly select 250 images from other classes. We trained the model on original CIFAR100 training dataset and test the model on various kind of specialized datasets with different number of domain classes and selection of domain classes. We also take into account that the selection of domain classes because the choice of labels have strong effect on the test accuracy, even if we keep the number of domain classes unchanged. Actually some classes are easier to classify than others. To eliminate this random effect, we will choose different groups of labels for each number of domain classes.

Figure 7 summarizes the results. As a benchmark, the test accuracy on the CIFAR100 testing dataset without probability layer is 73.74%. If the number of domain classes is 2 and we choose the label 0 and 1, the accuracy without probability layer is 88.12%. The difference between specialized testing accuracy 88.12% and the overall accuracy 73.74% is due to selection of classes. If we change the label pair from (0,1) to (3,5), the accuracy would also change from 88.12% to 61.39%. However, in both case, the accuracy with probability layer are 98.51% and 99.01%, which performs very well consistently.

Then we increase the number of domain classes from 2 to 5 and choose domain classes as 0 to 4 and 5 to 9 respectively. This time, the effectiveness of selection of domain classes appears again. If we set domain classes as

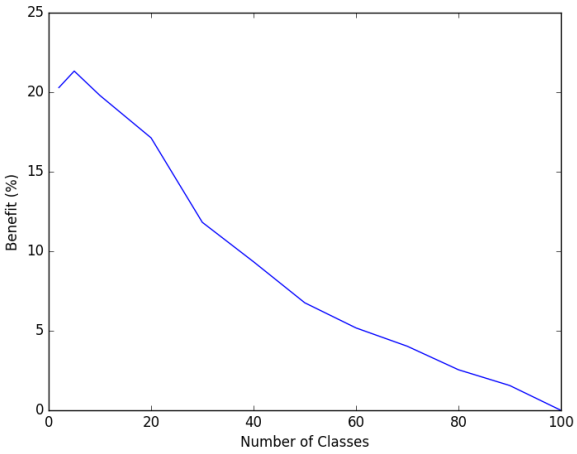


Fig. 7: Probability Layer Result

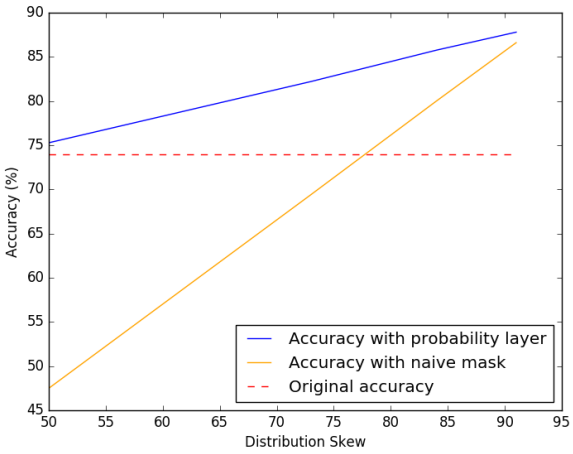


Fig. 8: Comparison Between Probability Layer and Naive Mask

0 to 4, the accuracy without probability layer would be 66.33%. If we set domain classes as 5 to 9, the accuracy would be 81.06%. The accuracy with probability layer in these two cases would be 93.03% and 97.01% respectively. The benefit of probability layer are 26.70% and 15.95% respectively.

When we increase the number of domain classes further to 10, the effectiveness of probability layer would still be dramatic. We choose 0 to 9, 10 to 19, 20 to 29 as domain classes. The accuracy without probability layer would be 75.35%, 67.16%, and 75.84%. The accuracy with probability layer would be 94.21%, 91.41%, and 92.22%. The benefits are 18.86%, 24.26%, and 16.38%.

To explore the interaction between probability layer and number of domain classes, we increase the number of domain classes gradually from 2 to 100. The benefit disappears gracefully. When number of domain classes is less than 20, the benefit are above 18%. Even if we have 40 domain classes, the benefit is still around 10%. When there are 60 classes, there are 5% benefit. Since we do not need retrain, these benefit are almost free. We do not need extra memory or energy and we have not introduced any extra latency. The only thing that we need is the run time distribution, which could be collected easily through an in-memory record.

Experiments show the significant of *probability layer* over *naive mask*. Figure 8 shows the comparison between probability layer and naive mask under different configurations. The red dash line represents the original accuracy without probability layer and naive mask, which is 74.57%. When the number of domain classes is 10 and the distribution skew is 50%, the accuracy with probability layer is 75.28%, while the accuracy with naive mask is only 47.49%. Thus probability layer has a positive effect even if the distribution skew is only 50%, while the naive mask has a strong negative effect in this context. As we increase distribution skew gradually, the accuracy with probability layer and naive mask keeps increasing. Only after the distribution skew becomes larger than 77.67%, naive mask starts to show positive effect on accuracy while probability layer keeps showing positive effect in all settings. As distribution skew approaches 100%, the effect of naive mask catches probability layer up, since the true labels start to only exist in domain classes. In short, probability layer is a generalization of naive mask and shows benefit over original model and naive mask consistently.

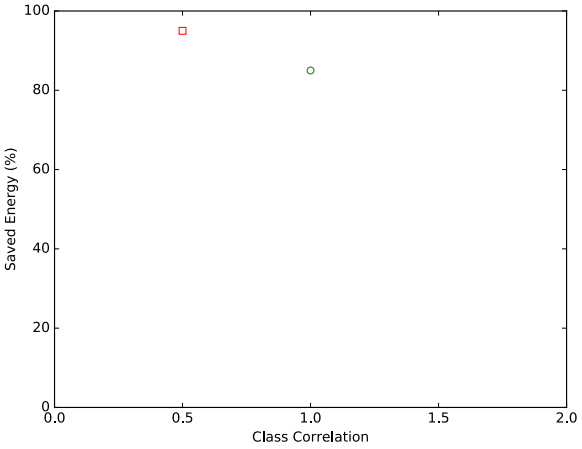


Fig. 9: Saved Energy

C. Energy efficiency

Considering class correlation can save more energy than only considering number of classes. Fig 9 shows the energy saved by class correlation. We generate a subset from CIFAR100 with 2 classes and different class correlation. The accuracy of VGG16 on CIFAR100 is 70.48%. We simplify VGG16 and reduce layers until the testing accuracy on specialized dataset is lower than 70%. Considering the innegligible impact of class correlation on testing accuracy, the experiments are done by different class correlation and, for each class correlation, the classes are selected randomly and the experiments are repeated for twenty times to get an average accuracy. When the class number is two and class correlation is 1, we need a model with one convolutional layer and two fully connected layer to get the accuracy as 72.9%. Thus we only need to consume 85% computation to get the same accuracy as the VGG16 with 19 layers. If we decrease the class correlation to 0.5, we only need a single fully connected layer to achieve 74.6% accuracy. In this case, only 5% computation of VGG16 is consumed. Thus, as we decrease class correlation from 1 to 0.5, the energy consumption is decreased by 66%

Test Data	Parameter	Accuracy	Benefit
Original CIFAR100	0	73.74%	
2 classes (0, 1)	0	88.12%	
2 classes (0, 1)	1.0	98.51%	+10.39%
2 classes (3, 5)	0	61.39%	
2 classes (3, 5)	1.0	99.01%	+37.62%
2 classes (0, 99)	0	84.65%	
2 classes (0, 99)	1.0	97.52%	+12.87%
5 classes $(0, ..., 4)$	0	66.33%	
5 classes $(0, ..., 4)$	1.0	93.03%	+26.70%
5 classes $(5, ..., 9)$	0	81.06%	
5 classes $(5, ..., 9)$	1.0	97.01%	+15.95%
10 classes $(0, ..., 9)$	0	75.35%	
10 classes $(0, ..., 0)$	1.0	94.21%	+18.86%
10 classes $(10, ..., 19)$	0	67.16%	
10 classes $(10, ..., 19)$	1.0	91.42%	+24.26%
10 classes $(20, ..., 29)$	0	75.84%	
10 classes $(20, ..., 29)$	1.0	92.22%	+16.38%

TABLE II: Test Probability Layer on Different Locality

VIII. Comparison with previous work

A. Distillation

Distillation has been discussed extensively in machine learning area [10, 16, 17, 18, 19, 20, 21]. With distillation, we can use a smaller model to learn a complex model, get a better performance than the small model, and consume less energy than the complex model. Thus distillation naturally fits great with energy-efficiency on mobile device. However, neither the distillation people nor the mobile energy efficiency people considered combining these two techniques together. We are the first to introduce distillation into mobile energy efficiency. We solved obstacles in both side. First, existing distillation methods could only learn a large model when there are equal number of nodes in the softmax layer while our method could proceed no matter the difference between these two numbers. Second, existing papers could either only retrain last few layers [22], which lose the ability to change model architecture, or retrain the model from scratch [2, 4, 5], which lose the information contained in large model. Introducing distillation into runtime optimization can solve these problems perfectly.

B. Specialization

Use specialization to generate a series of models composing trade-off between accuracy and energy is an emerging method for solving energy efficiency problem on mobile devices [2, 4, 5]. Generally the reduction in energy is achieved by reducing number of layers and the resulting decrease in accuracy is made up by reducing number of nodes in the softmax layer. Although existing papers introduced a new way to solve the energy efficiency problem, the implementation detail is very coarse-grained and there are lots of open questions to be answered. Instead of only considering number of classes in the runtime, our paper took class correlation into consideration and introduced semantic tree as a quantitative method describing similarity between classes. Based on the numeric metric, we brought up guided search to replace exhaustive search used in existing papers. The second contribution is to introduce distillation and varying input size into runtime optimization. Previously, the only method employed in this area is to reducing number of layers. We found that input size is also a simple but effective in specialization. Finally, *probability layer* was proposed to act as cold-start, solved the time latency introduced by retraining, and gives more choice for the scheduling model.

C. Model Compression

Model compression is to compress CNN architecture through matrix factorization and matrix pruning. Matrix factorization [20, 23, 24, 25] means using the multiplication of two low rank matrices to replace a single high rank matrix. Matrix pruning [26, 27] is to transform a matrix into sparse matrix by pruning small digits to be zero. Both directions of model compression do not change the number of nodes in the softmax layer or the model architecture. Thus techniques in model compression is orthogonal to using runtime distribution and could provide further optimization after reducing number of classes.

D. Early Stop

Early stop [28, 29] is an architecture with branches and will stop calculating once a branch is enough confident that an image has been classified correctly. Early stop contains various architectures to achieve energy efficiency through reducing unnecessary computation. This is orthogonal to runtime specialization and could be included into the model bank.

IX. Conclusion and Future Work

By using runtime class skew, including reduction in number of classes and similarity between a cluster of classes, we can increase accuracy without any change in model architecture. This phenonmenon gives us space to dramatically simplify our CNN models by changing input size, reducing number of nodes in softmax layer, reducing number of layers, and distillation. Further, we can use probability layer to make use of this class skew without retraining if the energy budget is limited, or retrain the last few layers if energy constraint is loose. By recording the repeated pattern of classes, cold retrain the model is a feasible way to make use of runtime distribution without runtime retrain. Finnaly, the end-to-end runtime support framework, SMART, can reduce energy consumption by

xx times while increase the accuracy by xx times. (EXPERIMENTS TO BE ADDED). In the future, we will provide a more detailed mapping from class cluster to suggested models.

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, 2015.

[2] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2016.

[3] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy, "Fast video classification via adaptive cascading of deep models," *arXiv preprint arXiv:1611.06453*, 2016.

[4] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: optimizing neural network queries over video at scale," *Proceedings of the VLDB Endowment*, vol. 10, 2017.

[5] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy, "Fast video classification via adaptive cascading of deep models," *arXiv preprint*, 2017.

[6] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Technical Report*, 2009.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[10] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[11] J. Fu, H. Zheng, and T. Mei, "Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition," in *Conf. on Computer Vision and Pattern Recognition*, 2017.

[12] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*. IEEE, 2014.

[13] F. Zhuang, X. Cheng, P. Luo, S. J. Pan, and Q. He, "Supervised representation learning: Transfer learning with deep autoencoders." in *IJCAI*, 2015.

[14] B. Sun, J. Feng, and K. Saenko, "Return of frustratingly easy domain adaptation." in *AAAI*, vol. 6, no. 7, 2016.

[15] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, vol. 1, no. 2, 2017.

[16] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in neural information processing systems*, 2014.

[17] Y. N. Dauphin and Y. Bengio, "Big neural networks waste capacity," *arXiv preprint arXiv:1301.3583*, 2013.

[18] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, "Learning efficient object detection models with knowledge distillation," in *Advances in Neural Information Processing Systems*, 2017.

[19] D. Lopez-Paz, L. Bottou, B. Schölkopf, and V. Vapnik, "Unifying distillation and privileged information," *arXiv preprint arXiv:1511.03643*, 2015.

[20] 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.

[21] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006.

[22] C. Li, Y. Hu, L. Liu, J. Gu, M. Song, X. Liang, J. Yuan, and T. Li, "Towards sustainable in-situ server systems in the big data era," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015.

[23] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[24] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv preprint arXiv:1412.6550*, 2014.

[25] J. Xue, J. Li, D. Yu, M. Seltzer, and Y. Gong, "Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014.

[26] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *International Conference on Machine Learning*, 2015.

[27] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015.

[28] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *Pattern Recognition (ICPR), 2016 23rd International Conference on*. IEEE, 2016.

[29] P. Panda, A. Sengupta, and K. Roy, "Conditional deep learning for energy-efficient and enhanced pattern recognition," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016.