Towards Deployment of Deep Neural Networks on Resource-Constrained Embedded Systems

By

Boyu Zhang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 08/26/2019

The dissertation is approved by the following members of the Final Oral Committee:

Azadeh Davoodi, Associate Professor, Electrical and Computer Engineering
Yu Hen Hu, Professor, Electrical and Computer Engineering
Mikko Lipasti, Philip Dunham Reed Professor, Electrical and Computer Engineering
Dan Negrut, Mead Witter Foundation Professor, Mechanical Engineering

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Deep Neural Network (DNNs) have emerged as an important computational structure that facilitate important tasks such as speech and image recognition, autonomous vehicles, etc. In order to achieve better performance, such as higher classification accuracy, modern DNN models are designed to be more complex in terms of network structure and larger in terms of number of weights in the model. This imposes a great challenge for realizing DNN models on computation devices, especially those resource-constrained devices such as embedded and mobile systems. The challenge arises from three aspects: computation, memory, and energy consumption. First, the number of computations per inference required by modern large and complex DNN models is huge, whereas the computation capability available in the given systems may not be as powerful as a modern GPU or a dedicated processing unit. So, accomplishing the required computation within certain latency is an open challenge. Second, the conflict between the limited on-board memory resource and the static/run-time memory requirement of large DNN models also need to be resolved. Third, the very energy-consuming inference process places a heavy burden on edge devices' battery life. Since the majority of the total energy is consumed by data movement, the goal is not only to fit the DNN model into the system but also to optimize off-chip memory access in order to minimize energy consumption during inference.

This dissertation aims to make contributions towards efficient realizations of DNN models on resource-constrained systems. Our contributions can be categorized into three aspects. First, we propose a structure simplification procedure that can identify and eliminate redundant neurons in any layer of a trained DNN model. Once the redundant neurons are identified and removed, the corresponding edges connected to those neurons will be eliminated as well. Then the new weight matrix is calculated directly by our procedure, while retraining may be applied to further recover the lost accuracy if necessary. We also propose a high-level energy model to better explore the tradeoffs in the design space during neuron elimination. Since both the neurons and their edges are eliminated, the memory and energy requirements are also get alleviated. Furthermore, the procedure also allows exploring the tradeoff between model performance and implementation cost.

Second, since the convolutional layer is the most energy-consuming and computa-

tion heavy layer in Convolutional Neural Networks (CNNs), we propose a structural pruning technique to prune the input channels in `conv` layers. Once the redundant channels are identified and removed, the corresponding convolutional filters will be pruned as well. There significant reduction in static/run-time memory, computation, and energy consumption can be achieved. Moreover, the resulting pruned model is more efficient in terms of network architecture rather than specific weight values, which makes the theoretical reductions of implementation cost much easier to be harvested by existing hardware and software.

Third, instead of blindly sending data to cloud and relying on cloud to perform inference, we propose to utilize the computation power of IoT devices to accomplish deep learning tasks while achieving higher degree of customization and privacy level. Specifically, we propose to incorporate a small-sized local customized DNN model to work with a large-sized general DNN model by using a "Mixture of Experts" architecture. Therefore, with minimal implementation overhead, the customized data can be handled by the small-sized DNN to achieve better performance without compromising the performance on general data. Our experiments show that the MoE architecture outperforms popular alternatives such as fine-tuning, bagging, independent ensemble, and multiple choice learning.

1 INTRODUCTION & MOTIVATION

In recent years, there has been tremendous development in using Deep Neural Networks (DNNs) to address challenging pattern recognition problems in cutting edge information technology applications such as object recognition, speech recognition, smart homes, health care, autonomous vehicle navigation, etc. [4, 17, 32, 64, 67, 78]. Empirically, it is found that deeper (more layers) and more complicated (more weights) DNN architectures often yield superior performance [77, 79]. This implies intensive numerical computation requirements for both training and deployment of DNN applications that can only be afforded using cloud computing or high-performance cluster computing with GPU (graphics processing unit) acceleration.

A DNN model designed (trained) on a cloud-based computing platform often exceeds the budget on computation resources which is available on edge devices such as mobile phones or internet of things (IoT). On the other hand, DNN-based Artificial Intelligence (AI) applications such as speech recognition, and smart homes would require edge devices such as sensors to gather application data. Current solution is to transfer such raw data via network to the cloud infrastructure for DNN processing and then transfer the results back to the edge devices. Examples such as home agent Amazon Alexa, Google Home all use this kind of remote processing architecture. Such an approach may not be easily scaled up to the future IoT deployment with billions of devices. It also cannot work when the network infrastructure is unavailable, and may not easily be customized to learn from individual needs or adapt to the changing environment.

The above concerns stem from a significant gap in current DNN design flow, namely the decoupling of design from deployment of the DNN architecture. When the implementation platform changes, the algorithm used to realize specific function often needs to be adapted as well. But for DNNs, a performance-oriented designing process may result in a complicated structure that may only be suitable for implementation on a cloud infrastructure. When such models are to be deployed to edge devices, the required computation often far exceeds that can be provided by these low power devices, not to mention modifying network structure or training weight values on devices to learn from users.

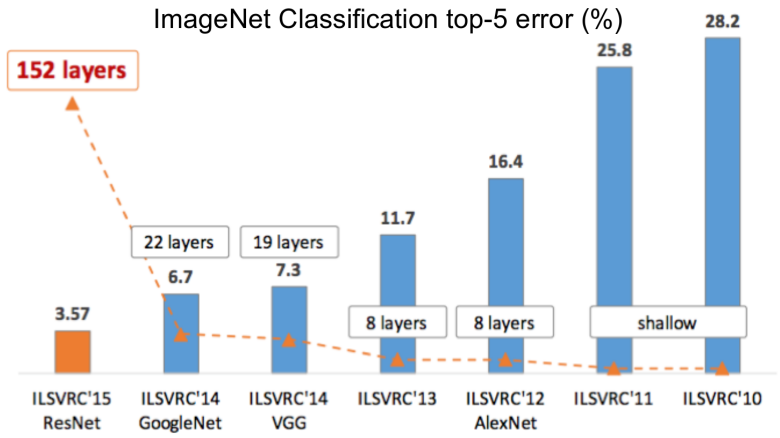

Figure 1.1: Top-5 error rate and number of layers of the winning model of ImageNet competition from 2010 to 2015 [29].

Therefore, two issues need to be addressed in order to deploy DNN models on edge devices and enable on-device user customization/adaptation:

The first issue is to fit DNN models into target edge devices while preserving their performance. Figure 1.1 shows the top-5 error rate and number of layers of the winning model of ImageNet competition from 2010 to 2015 [29]. Besides the availability of large-scale datasets and advances of modern GPUs, a major reason to enhance the performance of CNNs to higher level is the increasingly more complicated model architecture.

In general, the models with more layers (deeper) tend to achieve higher accuracy compared to the models with just a few layers (shallower). However, complicated models often contain millions of parameters and require billions of operations per inference, which hinders its deployment on resource-constrained platforms such as mobile devices and embedded systems. For example, Figure 1.2 shows the comparison of model size (as indicated by the size of the circles), top-1 accuracy (y axis), and number of computations (x axis) required per inference of various DNN models. As can be seen, from `AlexNet` to `VGG` to `Resnet` and `Inception`, although researchers have made progress in designing smaller DNN models to achieve higher accuracy, the average model size and number of operations required per inference are still very big for edge devices such as embedded and mobile systems. Specifically, the

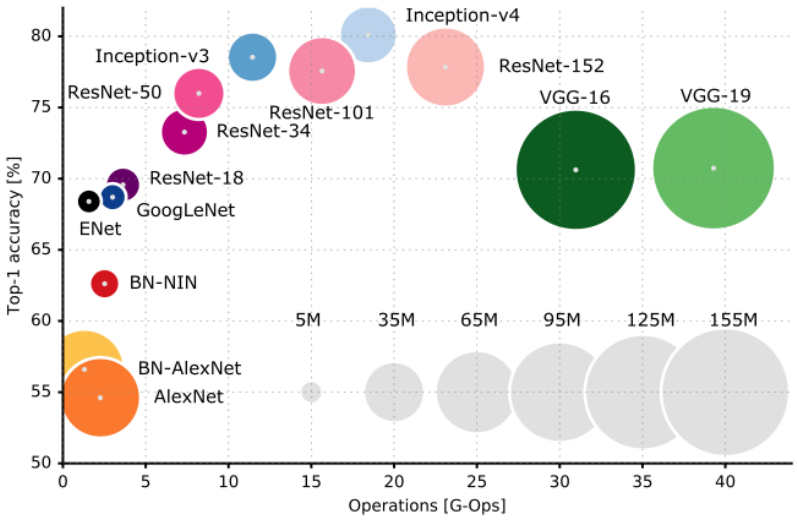

Figure 1.2: Comparison of model size, accuracy, and number of operation required per inference of various DNN models.

difficulties of deploying CNNs on resource-constrained edge devices arise from four aspects:

1. **Static memory:** Modern CNNs for practical applications often contain millions of parameters, which need to be loaded into memory during inference. Although the memory size of embedded devices is increasing in recent years, large-sized CNNs still impose a heavy burden on memory, especially when considering other applications running simultaneously;

2. **Run-time memory:** Besides model parameters, intermediate activation values also need to be stored in memory during inference. This type of memory requirement highly depends on batch size and whether gradient back propagation will be performed. Although it can be alleviated by reducing batch size at the cost of sacrificing throughput, runtime memory is not negligible even if batch size equals to 1;

3. **Computation capability:** Billions of computations are often required per inference, and they need to be accomplished in short time to deliver satisfactory user experience. This is a very challenging task for embedded processors due to

limited computation capability and other related factors, such as cache effects, memory accessing, and operating system;

4. **Energy consumption:** Conducting billions of computations and accessing (load and store) millions of model parameters/intermediate activations are highly energy-consuming. Thus, for devices that are powered by battery, battery life may become an issue.

The second issue to enable deployment on resource-constrained devices is to achieve on-device user customization for trained DNN models. The DNN models trained with large-scale training dataset may give satisfactory performance on the testing dataset, which means the DNN models may perform very well in general. However, when presented with user-specific data, such as accent in speech or specific handwriting style, the performance of the general DNN models may degrade and become unsatisfactory. This could impose a serious issue for each individual user. Furthermore, to make DNN enabled edge devices smart and not repeating their mistakes again and again, they need to be able to learn from and adapt themselves to each individual user. However, this is not an easy task due to two reasons:

1. **Availability of users' or vendors' data:** On one hand, users may not be able to access and modify the structures and weights of the trained DNNs because we envision that such trained (and potentially compressed) DNNs may be provided by vendors in the form of intellectual property. On the other hand, vendors may not in practice have access to the users' personal data, because it may be highly sensitive and private and users do not want to share it with vendors. In fact, even if users agree to share their personal data with vendors, centralizing all users' data, as well as training and deploying a customized model for each user may be unaffordable when the number of users scales up. Not to mention that connection to cloud may be unstable or even unavailable for devices in certain environments.

2. **Computation capability of edge devices:** While each inference pass is already very expensive to be carried out by edge devices, the amount of computation required by each training pass is about three times higher than the inference pass,

and all intermediate neuron values need to be stored during the forward pass and loaded during the back propagation pass. Moreover, effective training requires a large amount of training passes, besides the storage to hold the training dataset and the energy to load them. Basically, every obstacle that makes it difficult to perform inference on edge devices becomes even worse when performing training.

To address these issues and mitigate the design-to-deployment gap, two approaches may be considered: (a) down-size (simplify) the network structure to meet the energy and storage constraints without significantly sacrificing the performance; and (b) augment the trained DNN with other lightweight machine learning models to enable on-device local learning capability for user-specific data. Numerous efforts have been reported towards these approaches and we present our contributions in this dissertation.

In the remainder of this dissertation, we first give an overview of the related works in Chapter 2. Second, Chapter 3 summarizes the contributions of this dissertation. Third, the background, notations, and our high-level energy model is presented in Chapter 4. Then, in Chapter 5 and Chapter 6, we introduce our proposed techniques for structural simplification of DNNs and experiment results in detail. In Chapter 7, we present an architecture for our proposed on-device user customization model along with detailed experimental results which significantly improves system performance and only incurs minimal implementation overhead. Finally, we discuss future directions in Chapter 8.

2 RELATED WORKS

2.1 Techniques for Efficient Realization of DNNs

Efficient realization of DNNs has been pursued in the literature from two aspects: design of efficient DNN algorithms, and design of dedicated DNN processors. The second aspect is out of the scope of this proposal, so we will only discuss the details of the first aspect in the remaining part of this section. The research efforts in the first aspect can be categorized into three orthogonal directions: 1) reducing the hardware implementation cost per computation and memory access; 2) simplifying the structure of a trained DNN; and 3) designing new network modules and structures. The details of these three directions are discussed below.

Reducing the Hardware Implementation Cost

Since a huge amount of computation and memory access is involved in every inference process of DNN, the associated energy consumption and storage cost can be dramatically reduced if the cost per computation and memory access can be reduced. Several efforts have been spent on reducing these costs in an already-trained DNN that needs to be deployed, for example by means of quantizing the weights and activations, stochastic computing and so on [22].

Hashemi et. al investigates the impact of quantization on both network accuracy and hardware metrics including memory footprint, power consumption, and area of the design. It demonstrates comprehensive tradeoff curves between accuracy and implementation cost under various quantization schemes [27, 81, 82]. The works [12, 96] push the boundary to the extreme in which weights and/or activations are quantized into binary/ternary values, and thus achieve at least 16X model compression ratio. However, this impressive compression ratio cannot be easily translated into speedup in inference time without the support of dedicated hardware or bitwise arithmetic libraries. Furthermore, specialized gradients computation and accumulation algorithms need to be used during fine-tuning, otherwise the accuracy degradation may become unacceptable.

The use of stochastic computing has also been proposed [3], for example in [58],

to perform low-power multiplication in DNNs in exchange with reduced accuracy. In addition, Lin et. al proposed a rank decomposed statistical error compensation (RD-SEC) technique to allow the the entire system to operate in near threshold voltage regime [62]. As a result, they report significant energy saving while maintaining good accuracy.

In [6], the Eyeriss architecture is proposed to minimize energy consumption associated with data movement. It tries to identify the optimal dataflow so that all types of data reuses are maximized. This issue has also been highlighted in the Low-Power Image Recognition Challenge [16]. However, in the above class of techniques, the structure of the DNN remains intact and there is no guarantee that the existing DNN models are optimal for their corresponding task.

Simplifying the Structure of Trained DNNs

The second orthogonal direction of research is to develop techniques that can simplify the structures of trained DNNs as much as possible. Structure simplification techniques seeks to create alternative DNNs that require less computation and storage. They can be further divided into three categories: *unstructured pruning*, *structured pruning*, and *low-rank approximation*.

Unstructured pruning focuses on eliminating unimportant weights/connections in trained DNN models. These techniques can be dated back to the 90s with the goal to maximize the degree of structure simplification with negligible loss in accuracy. An early observation was that many weights in a trained network have relatively small magnitudes and hence may be reset to zero without affecting the network's (inference) performance. Techniques such as *brain damage* [13] and *brain surgeon* [28] were proposed to achieve this by using the second order derivatives of the loss function as a saliency measurement to determine if a weight should be pruned. In spite of its theoretical justification, high computational complexity is inevitable when applying to deep networks. There are also weight reduction methods during the training phase, such as *weight decay* [52]. Nowlan and Hinton [70] proposed a different approach called *weight sharing* where weights of similar values in the same weight matrix are grouped together so that those inputs that are to be multiplied to these grouped weights may be added first and then perform a single multiplication.

Many of these earlier methods have focused on the potential benefit of better generalization property due to a simpler structure with fewer weights, the so-called *Occam's razor* principle. Recently, Han et. al [23] proposed a three-step procedure to (a) train the network to learn which connections are important; (b) prune the weights with low magnitude; and (c) retrain the network to fine tune the weights of the remaining connections. They claimed 90% reduction of weights without sacrificing the performance. Later on, they enhanced this procedure by applying quantization and Huffman encoding on top of the pruning, and achieved up to 49X compression ratio [25]. Their procedure however requires time-consuming iterative pruning and fine-tuning to determine proper threshold parameters for each layer of the considered DNN. The work [21] improves over [25] by introducing mask variables and alternatively updating masks and model parameters, thus, recovering the incorrect pruned weights and reducing training iterations. However, although unstructured weight pruning can greatly compress the sizes of the models with the help of sparse representation, the structure of the resulting DNNs after applying these magnitude-based techniques is irregular, which requires specialized hardware or software libraries, such as Efficient Inference Engine (EIE) [24], to fully exploit the potential benefits (energy and computation reduction) brought by these techniques. This is because, with general-purpose hardware, every weight value still needs to be loaded at least once (even if it is 0), and the energy associated with data movement dominates the entire energy consumption [88].

Structured pruning tries to prune a DNN model in a structured manner, such as pruning set of neurons, groups of weights, or entire layers. In early 90s, [39, 53] seek to directly reduce redundant neurons by analyzing the output of a vector of neurons of the same layer with respect to a set of inputs. If the outputs of a subset of neurons can be well approximated via a weighted linear combination of the outputs of remaining neurons, this subset of neurons may then be removed from the network.

For Convolutional Neural Networks (CNNs), since the majority of inference energy is consumed by convolutional layers [7], pruning channels of convolutional layers is a very effective technique to develop compact and efficient models. In earlier works, [57] prunes filters and the corresponding channels based on the filters' ℓ_1 norm. [38] defines Average Percentage of Zeros (APoZ) to measure the percentage of zero activations of channels and neurons, and then prunes the channels/neurons with high

APoZ values. The work [68] first uses Taylor expansion to approximate the change of loss function with respect to pruning each channel, and then channels are pruned according to their impact on the loss function. The work [85] proposes Structured Sparsity Learning (SSL), which imposes structured sparsity on model weights in terms of filters, channels, kernel shape, and depth during training, thus, it effectively prunes channels at the cost of lengthy training process.

More recently, *ThiNet* [65] prunes the target layer by greedily selecting the input channel that has the least contribution to the output tensor in each iteration, and then this procedure is repeated until the specified number of channels have been pruned. The contribution of an input channel is defined as the squared summation of the corresponding partial output tensor values. The entire model is pruned layer by layer and fine-tuning is applied after pruning each layer. However, since all partial output tensor values are summed together to form the contribution value, this method may mistakenly treat an input channel as redundant even though its contributions to all output channels are relatively large but with near-to-zero summation.

For each layer to be pruned, *Channel Pruning* [31] introduces a coefficient for each input channel, which will be multiplied with its coefficient to form the scaled input tensor. The problem of selecting redundant channels is formulated as a LASSO regression optimization problem with the goal of minimizing 1) the approximation error between the original and the new output tensor which is obtained by using the scaled input tensor; and 2) the ℓ_1 norm of the coefficient vector. The ratio between these two terms affects the number of zeros in the resulting coefficients, thus it effectively determines how many channels may be pruned. Finally, the LASSO regression problem is solved by alternatively updating the coefficients and new weights until convergence. Similarly, *Network Slimming* [63] utilizes the scaling factors in Batch Normalization layers and trains the model under channel-level sparsity-induced regularization. After training, the channels with near-zero scaling factors will be pruned. However, besides the computationally-expensive alternative updating/training process, it is also not clear how to directly set the ratio between the regularization term and the regular loss in these methods in order to prune any specified number of channels.

Global Dynamic Pruning [61] shares the same alternative optimization scheme between the choice of channels to be pruned and model parameters. But instead

of pruning models layer by layer, it globally and tentatively selects the channels to be pruned according to the Taylor expansion of the loss function with respect to the weights of each channel. After the channels are selected, the parameters corresponding to the kept channels are fine-tuned. This two-step procedure is repeated until convergence, and finally, the selected channels are permanently pruned. However, since the redundant channels are globally selected across all layers, designers do not have fine-grained control over how many channels are pruned for each layer, which may lead to sub-optimal solution due to the lack of emphasis on pruning more beneficial (in terms of computation reduction) layers. However, these techniques are limited in scope because 1) they are typically not compatible with each other so they cannot be easily integrated together; 2) these techniques are usually defined to apply to an individual layer at each step. Systematic ways to combine structural simplification across all the layers to minimize the network size have not been fully explored in the literature.

The above methods all focus on the weight values of a neural network. A different approach, based on low-dimensional realization of a system has also been developed to simplify a trained DNN by exploiting low-rank property of the weight matrices. Specifically, *Low-Rank Approximation* aims to approximate a weight matrix with the product of lower-rank weight matrices, thus achieving reduction in both model parameters and computation. This property has been observed as early as late 80s [87] where Singular Value Decomposition (SVD) was used to break down the weight matrix. The works [15, 45] applied SVD to higher order tensors by extending it to monochromatic approximation and biclustering approximation. But the compression ratios on convolutional layers are not as impressive as those on fully-connected layers and the authors did not report reduction of computation and actual inference time. The work [49] approximates weight matrices with Tucker decomposition and reports up to 4.93X computation reduction. There are many follow-up techniques for low-rank approximation such as [14, 43, 80]. However, low-rank approximation techniques effectively expand each layer to two or three layers, which adversely impacts the actual inference time depending on the platform.

Designing New Network Modules and Structures

The third direction include efforts to either manually design or automatically search new network modules and structures. Examples include *Network in Network* [60], *ResNet* [29], Inception module in *GoogLeNet* [79], *Squeezenet* [42], and *neural architecture search (NAS)* by reinforcement learning [1, 97] or genetic algorithm [74, 86]. Also, *Knowledge Distillation* technique [33, 75] is widely used to train compact and efficient models by mimicking the intermediate activations, attention maps, or output probability distributions of larger models [5, 59, 66]. With these new modules and structures, researchers are trying to design small-sized DNN models that have superior performance and require less computation per inference. Putting it into perspective, these efforts are trying to draw small circles at the upper-left region in Figure 1.2. However, such design efforts require either tremendous insight about the targeted task itself or huge amount of computations, which makes it difficult to apply the ideas of designing networks for one type of task in order to design networks for another type of task.

Finally, although all of the above efforts pursue the goal of efficient realization of DNNs from different angles, most of them lack consideration of the architecture of the underlying hardware platform during the design phase of their techniques. As a result, the resulting DNN models after applying these techniques may not be able to achieve optimal performance when actually deployed to various platforms. For example, optimal reduction in both the parameters and the edges of the network, while correlated, is not equivalent to reduction in energy [88], especially in scenarios when tradeoff with accuracy is considered. Thus, accurate consideration of metrics such as energy consumption and their tradeoffs with accuracy becomes inevitable for deployment of DNNs on resource-constrained platforms, and such issues are extremely important for large-sized deep neural networks and emerging applications of today. However, this area has not been well studied, which may be because only recently the need for finding solutions for emerging big data applications with requirement of low energy but high error tolerance has emerged. Besides, the majority of structure simplification techniques and network design methods are published in communities with insufficient emphasis on hardware.

2.2 Applicable Techniques for Low-cost and Local Customization of DNNs

Transfer Learning

In recent years, transfer learning has emerged as a technique to achieve customization and accelerate the convergence of DNN training. The general idea of transfer learning is to improve the learning of target task on target domain by utilizing the knowledge learned from the source domain regarding a source task [71, 72]. In the task of customization, the source task and the target task are the same; the source domain is composed of generic dataset, and the target domain consists of user-specific dataset. One specific technique of transfer learning is *fine-tuning* [17], in which the given DNN model is first trained with generic dataset to reach a relatively-high performance level, then only the last few layers are fine-tuned with user-specific data while the weights of all the other layers are frozen. The reason why only the last few layers are fine-tuned with user-specific data is because the earlier layers of DNNs are usually treated as feature extractors and the last few layers make decisions based on the extracted features [89]. By training the earlier layers with only generic data, more general features can be learned and the whole model may achieve better generalization property.

The drawback of this transfer learning approach is that for large-sized DNN models, modifying the weight values cannot be easily accomplished on edge devices due to limited computation capability and energy budget. Furthermore, user may not be able to modify the weight values if the trained (and potentially down-sized) DNNs are provided by vendors in the form of intellectual property.

A recent work [26] tries to alleviate this problem by augmenting the base DNN with a task-specific network and an aggregation layer. All components are trained with generic dataset before deployment, then only the task-specific network and the aggregation layer are further trained on device with user-specific dataset after deployment. Although the task-specific network and the aggregation layer are designed to be small, deploying and training them still impose considerable energy and memory overhead on edge devices. Moreover, after user-specific training, the performance on generic dataset degrades significantly, which is undesirable.

A Mixture of Expert Ensemble Technique

The idea of local customization of machine learning models with user-specific or application-specific data can be dated back to 90s, when the *Mixture of Experts (MoE)* architecture [44, 47] was proposed to achieve local customization. The *MoE* architecture consists of multiple expert models each can provide its result for the given input, and a Gating Network (GN), which provides input-dependent weight values to combine the results of all experts. The work [44] suggested to reduce the coupling effect among experts and GN by defining a new error function, which makes the parameters of one expert not affected by the parameters of other experts. But there still existed some indirect coupling due to the lack of information on which expert should be responsible for which training sample. Later on, the *MoE* architecture was also shown to be very effective for classification of the patient-adaptable electrocardiogram beats [40, 84]. However, the problems considered at that time were considerably less complex and resulted in significantly simpler models compared to the ones today [76]. Furthermore, with increasing emphasis on edge computing, power efficiency becomes a critical factor when designing the components of *MoE*.

We note, in this dissertation we also use the MoE architecture. However, unlike previous works that focused on different expert architectures [11] and configurations [73], our work focuses on how to integrate the general *MoE* concept with DNN to achieve higher performance on complex tasks with minimal implementation overhead for resource-constrained devices.

Other Ensemble Techniques

The *MoE* architecture is similar to many ensemble techniques such as *independent ensemble (IE)*, *bagging*, and *multiple choice learning (MCL)* [55, 69]. They usually train **M** member models and make predictions based on the results of all member models. In *IE* [9], each member model is initialized with different random parameters and trained independently on the entire training set. In *bagging* [56], all member models have the same initial parameters but are trained independently on different subsets of data that are sampled (with replacement) from the original training set. In *MCL* [55, 69], instead of training each member model independently, the losses of all member models are combined to form the oracle loss, which is used to train all

models together. The oracle loss can be expressed as:

$$\mathcal{L}_{\mathrm{MCL}}(\mathbf{D}) = \sum_{i=1}^{N} \min_{m} \ell(y_i, P_i^m)$$

where $\mathbf{D}$ is the entire training set with N instances, m represents the m^{th} member model among all $\mathbf{M}$ models, ℓ is the loss function of each model, y_i is the ground truth label of the i^{th} training instance, and P_i^m is the predicted probabilities of all classes generated by the m^{th} model with respect to the i^{th} instance. Basically, by optimizing the oracle loss, the training process optimizes the most accurate model for each training instance, and drives each model to become a specialist on a subset of the target task.

Now we discuss the similarities and differences between MoE and these ensemble techniques. The similarity among all of these approaches (including *MoE*) is that all of them train multiple member models during training, and multiple models are involved to generate the final result during inference. Especially, *MCL* dynamically assigns each training instance to one or more member model(s), thus it encourages each member model to become a specialist. This is similar to *MoE* in the sense that we explicitly designed different experts to become specialists on generic and customized data.

However, there are major differences between *MoE* and other approaches which are enumerated below. For the purpose of easy illustration, we name the expert model targeted at generic data as Global Expert (GE), and the expert model targeted at customized data as Local Expert (LE).

1. Although there is no theoretical limitation, the member models in *IE*, *bagging*, and *MCL* are usually identical. While in our proposed *MoE* architecture, we deliberately designed LE and GN to be dramatically smaller than GE in order to 1) leverage the fact that customized data has much smaller size and less variance; and 2) minimize the implementation overhead brought by LE and GN. In fact, in our case study, the memory and computation overhead induced by LE and GN are about 2.52% and 0.50% of those of GE, respectively, while they would be at least 100% for the other three approaches.

2. In *IE*, *bagging*, and *MCL*, the final result is typically obtained by averaging the results of all member models. Thus, to get the final result, the inference of all member models must be performed. However, with *MoE* architecture, the final result is an input-dependent weighted combination of all experts' results. In the case where the weights for experts are binary values, the system could perform the inference of GN first, and then perform the inference of either LE or GE according to the GN's result. Since the majority input to the system would be customized data after its deployment, such selective inference can significantly reduce the computation and energy cost compared to other ensemble techniques.

3. *MCL* dynamically assigns each training instance to one or more member model(s) depending on how well each model performs regarding to that instance, thus encourages each model to become specialized on a subset of the original task. However, designers do not have the ability to control how the original task is partitioned and which member model is specialized on which sub-task. On the contrary, in our proposed *MoE* architecture, GE and LE are explicitly trained with and targeted at generic and customized data.

4. Even though only a subset of all training instances is used during the training of each member model in *bagging* and *MCL*, the entire training set is known prior to training in all techniques except *MoE*. However, in the problem of user customization, neither the vendor, nor the users have access to the entire dataset. Thus, training has to be done in two steps, which makes the other ensemble techniques not a suitable fit for user customization.

3 OUR CONTRIBUTIONS

This dissertation introduces techniques to reduce the implementation costs such as memory, computation, and energy for deploying and customizing trained DNNs on resource constrained embedded systems. The summary of our contributions is listed as follows:

- **High Level Analytical Energy Model [90, 91]**
  We propose a high-level analytical energy model to estimate the energy consumption of DNNs in order explore the design space, and bring awareness of energy as a metric within the DNN design/simplification process. Our energy model is based on an underlying hardware similar to the Tensor Processing Unit in [48]. Besides the energy consumption associated with computation, our model explicitly considers the energy related to data movement and memory hierarchy. In fact, they are the two main sources that make the number of parameters and computations not a good approximation for energy estimation: First, the energy consumption of accessing memory is significantly higher than that of computation. Second, the memory accessing pattern largely depends on the memory hierarchy. To reflect realistic energy consumption of various operations, we adopted the energy numbers of a 45nm process from [23]. As shown in our experiments, this model is used to guide the structural simplification technique described in Chapter 5 directly towards energy efficiency. Our high-level analytical energy model is presented in Chapter 4.

- **Structural Simplification via Neuron Elimination [90, 91]**
  To achieve an energy-aware structural simplification process, we propose a procedure that facilitates exploration of energy and accuracy trade-offs of varying configurations of a structurally simplified, trained DNN. This procedure seeks to eliminate redundant neurons at a considered layer and simultaneously updates the corresponding weight matrix. Therefore retraining is not necessary after neuron elimination. When a neuron is removed, all of its connecting edges are also removed, resulting in significant reduction of memory and energy while minimally affecting the accuracy. The task of eliminating redundant

neurons is formulated as a subset selection problem, which is approximately and efficiently solved by a novel approach based on pivoted QR factorization. For each layer, multiple configurations are realized and the configurations of multiple layers together form the solution space and energy-accuracy curves, from which the Pareto-optimal curve can be easily identified. We also discuss and validate strategies to apply our procedure to multiple layers in a DNN in order to maximize the compression ratio of the network subject to an overall budget in degradation of accuracy. Thus, according to the constraints of the target platform, the appropriate pruned model can be selected and deployed. In our experiments, we show energy-accuracy tradeoff provides clear guidance to achieve efficient realization of trained DNNs. We also observe significant implementation cost reductions with up to 11.50X in energy and 12.30X in storage while the accuracy degradation is negligible.

Our neuron elimination procedure is discussed in Chapter 5.

- **Structural Simplification via Channel Pruning [93]**
  To reduce the implementation cost of convolutional layers, we propose a channel pruning technique to identify and prune redundant input channels of convolutional layers. After a channel is pruned, its corresponding convolutional filters of both the current layer and the previous layer will be pruned as well, thus achieving significant reduction in terms of static and run-time memory, computation, and energy consumption. The proposed procedure investigates the intermediate results of convolutional layers. It formulates the task of identifying redundant channels as a subset selection problem, in which a subset of input channels of specified size is selected to maximally preserve the original outputs of the target layer and the other channels are identified as redundant. This NP-hard problem is approximately solved using the pivoted QR factorization algorithm as well. We also propose two techniques to explore additional pruning opportunities in ResNet-like models. Moreover, the proposed technique is orthogonal to others such as quantization and low-rank expansion, which can be combined to achieve further reduction. The results show our proposed technique is able to reduce the computation by 4.29X (in VGG-16) and 2.84X (in ResNet-50) while only sacrificing about 1.40% top-5 and 2.50% top-1 accuracies. Compared to

many prior works, our results are better in terms of both computation reduction and accuracies.

Our channel pruning technique is explained in Chapter 6.

- **User Customization via MoE [92, 94]**
  To achieve on-device user customization, we propose to utilize a Mixture of Experts (MoE) architecture to augment the DNN that is trained on generic data with lightweight machine learning models to enable on-device local learning capability for user-specific data. Thus, it achieves performance improvements with low implementation overhead. In the MoE architecture, the DNN trained with generic dataset is viewed as a Global Expert (GE). We then use a small DNN as a Local Expert (LE), which is trained with small-sized customized (i.e., user-specific) data to learn from user's input so that the system doesn't repeat the same mistake. The third component of the MoE architecture is a Gating Network (GN), also implemented as a small DNN, which determines whether the incoming data should be handled by GE or LE. Thus, improving system performance on customized data while preserving its performance on generic data. To minimize the associated implementation overhead, both LE and GN are designed to be very small and a novel technique is proposed which is based on *structure sharing*. Finally, this architecture may be easily extended to multiple local experts (LEs) to accommodate different classes of users, if desired. We evaluated the proposed MoE architecture with the task of recognizing custom handwritten digits and characters, and show that with minimal storage and computation overhead, the MoE architecture is able to achieve significant performance improvement over the customized data while preserving its performance on generic data. We conduct comprehensive experiments to show the superiority of MoE compared to many alternatives in the literature.

  Our MoE architecture for on-device user customization is presented in Chapter 7.

Overall, in this dissertation we analyze the major obstacles which hinder the deployment and customization of trained DNNs on edge devices, and propose three

techniques to alleviate them. We show our work can effectively reduce the memory, computation, and energy requirements, thus facilitating the deployment of DNNs on resource-constrained systems.

4 DNN BACKGROUND AND ENERGY MODEL

With the goal of achieving energy-efficient realization of DNNs, numerous efforts have been focused on structural simplification techniques and designing more efficient DNN models. As mentioned in Chapter 2, the primary metrics used in these works are the reduction in the number of parameters and computations, as well as reduction of bit-width. However, as pointed out by [88], reducing model size and the number of required computations per inference do not necessarily translate into a reduction in energy consumption – at the least the reduction in energy may not be proportional to the reduction in model size or computation. In other words, previous efforts in design of DNNs and structural simplification may result in small and accurate model, but they may not be the optimal model in terms of energy consumption.

This discrepancy between the energy consumption and the number of parameters/-computations comes from two sources: 1) Data movement. The energy consumption of accessing memory is significantly more than that of computation, which makes it dominates the overall energy consumption; and 2) Memory hierarchy and dataflow, which have a huge impact on data movement. When performing computation, the energy consumption of loading the required data (parameters and activations) into processing unit largely depends on the memory hierarchy and how the dataflow is organized. For example, to minimize energy consumption, [6] tries to identify the optimal dataflow so that all types of data reuses are maximized. In summary, the number of weights and computations are not a good approximation for energy estimation.

In this chapter, we propose a high-level analytical energy model that can be easily integrated into DNN design/simplification process to facilitate the estimation and incorporation of energy during the design phase. Our energy model is based on an underlying hardware similar to the Tensor Processing Unit [48]. Besides the energy consumption associated with computation, our model explicitly considers the energy related to data movement and memory hierarchy. The energy values of various operations are from [23], which are corresponding to a 45nm process.

In this chapter, the background and notations are introduced in Section 4.1, which is followed by detailed discussion of our energy model in Section 4.2.

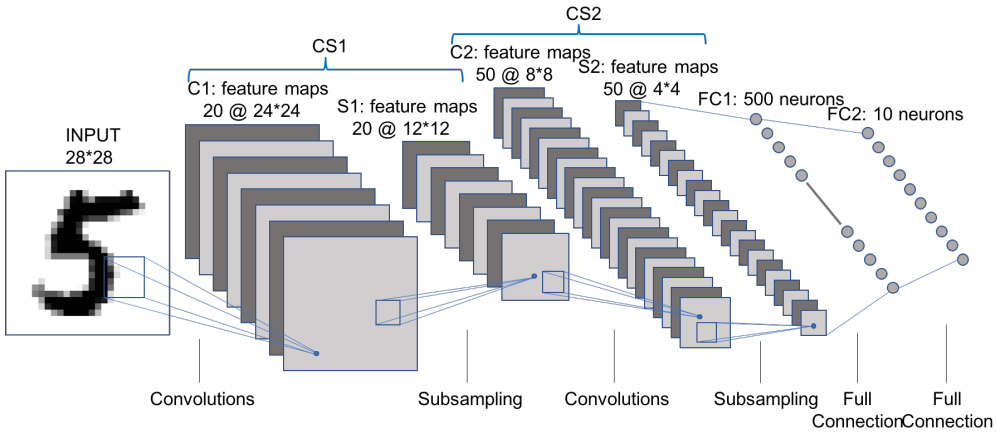

Figure 4.1: The structure of the `LeNet5` network from [54] for recognition of handwritten digits in MNIST dataset.

4.1 Background and Notations

A DNN often consists of many layers of neurons interconnected via a weight matrix between successive layers. At each layer, a common computation module has the following form:

$$\mathbf{y} = \mathrm{f}(\mathbf{W}\mathbf{x})$$

where $\mathbf{x}$ is an $\mathfrak{n} \times 1$ input vector to the current layer, $\mathbf{W} \in \mathbb{R}^{\mathfrak{m}\times\mathfrak{n}}$ is a weight matrix, and the matrix vector product $\mathbf{W}\mathbf{x}$ may also be used to realize convolution operations like those used in convolutional neural network (CNN). $\mathrm{f}(.)$ is an element-wise nonlinear function such as a Rectifier Linear Unit (Relu) [54]. $\mathbf{y}$ is a $\mathfrak{m} \times 1$ vector representing the output of the current layer and (in some cases, part of) the input to the next layer. The vector $\mathbf{y}$ may also incorporate subsampling similar to the pooling operator in CNN. In that case the dimension of $\mathbf{y}$ is $\mathfrak{q} \times 1$ with $\mathfrak{q} < \mathfrak{m}$. To train a DNN, a set of training vectors $\mathbf{X}$, which is a $\mathfrak{n} \times \mathrm{K}$ matrix is used to stochastically adjust the weights at each layer of the DNN until the final output achieves desired accuracy. The performance of the network (same as classification accuracy) then will be evaluated with a set of testing vectors.

Example: Figure 4.1 shows the structure of `LeNet5` [54] which is a CNN. It can be viewed as a a network with 4 layers. The first layer CS1 is made of a convolutional

layer C1 and a subsampling layer S1. The input to the first layer is a 28x28 greyscale image and it can be viewed as a $\mathfrak{n} \times 1$ vector with $\mathfrak{n} = 28 \times 28 = 784$. The input vector is then mapped to 20 feature maps via C1: each neuron in each output feature map is connected to a 5x5 neighborhood in the input image. It can be viewed as 20 5x5 *kernels* that are moved across the input image with a fixed *stride* (of 1 unit in this case). This stride makes consecutive neighborhoods in the input image to highly overlap. The size of each output feature map of C1 is 24x24. So $\mathfrak{m} = 24 \times 24 = 576$ neurons per feature map. Next the subsampling layer S1 maps each feature map to a smaller one of size 12x12. Therefore for CS1 layer, for each feature map $\mathfrak{q} = 12 \times 12 = 144$ for its $\mathfrak{q} \times 1$ output vector.

Similarly the second layer CS2 can be viewed as a convolutional layer C2 followed by subsampling layer S2. The final two layers in `LeNet5` are fully-connected layers (FC1 and FC2). Their dimensions are shown in the figure. `LeNet5` can be trained with the Caffe tool [46] and has been shown to achieve 99.1% accuracy on the MNIST dataset after training. MNIST [54] is a collection of images of hand-written digits and has 60K training data and 10K testing data.

Within the first layer, for each feature map, there is a weight matrix representing the weights of different edges. There are about 288,000 edges in C1, where each edge requires performing a multiplication between its weight and the value at its input-neuron. However the actual number of distinct parameters that are needed to be stored (and trained) for C1 is only about 500. This is because in convolutional layers there is significant weight sharing due to the same kernel is reused as it is moved in the input image when generating each output feature map. To reduce memory usage, within each feature map, edges are bundled in different groups and each group is stored with one weight. Thus the required memory to store a CNN depends on the number of distinct weights (same as parameters).

4.2 Energy Model

Figure 4.2 gives an overview of the hardware model used in this work for energy calculation which is similar to the recent Google's Tensor Processing Unit (TPU) in [48]. To perform the computations at each step, the image and parameters are fetched from the off-chip DRAM, to fill up the on-chip *activation buffer* and *weight buffer* and

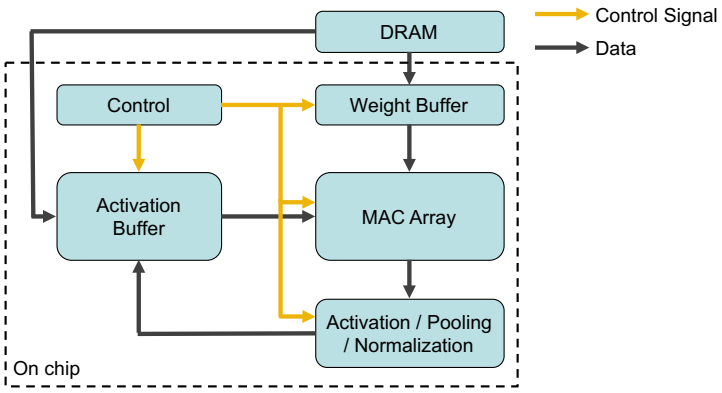

Figure 4.2: Overview of underlying hardware for energy modeling

are then fed into an array of Multiply Accumulate (MAC) unit. Next the generated results go through the unit implementing activation, pooling, and normalization and stored in on-chip *activation buffer* to be used as input to the next layer. The model in Figure 4.2 is similar to TPU except that we assume an on-chip control unit (implemented for example as a finite state machine) provides all necessary control signals, while in TPU these are provided through a PCIe Gen3 bus.

The sizes of the weight buffer and MAC array can be anything and are *not* required, for example, to be large enough to do all off-chip fetches of parameters and computations of each layer in one shot. But the size of the activation buffer is required to be large enough to store all intermediate activation values corresponding to each layer so off-chip write back won't be necessary during inference. This assumption is easy to realize and does not require a significant on-chip storage for the activation buffer as can be seen in the TPU implementation.

Using the above model we compute the inference energy by the following equation:

$$E = E_{MAC} + E_{DRAM} + E_{SRAM} + E_{REST} \tag{4.1}$$

where E_{MAC} represents the energy consumed by the MAC array and is calculated by

$$E_{MAC} = E(M) \times M \tag{4.2}$$

with $E(M)$ is energy consumed by one MAC operation and M is number of MAC

operations in the DNN.

Also E_{DRAM} represents the energy consumed by DRAM and given by

$$E_{DRAM} = E(D) \times (P + ImageSize) \tag{4.3}$$

where $E(D)$ is energy for fetching one word of a fixed quantization from DRAM and P represents the total number of parameters (same as distinct edge weights) in the considered DNN. $ImageSize$ is the number of word used to represent the image.

The term E_{SRAM} is energy consumed by the SRAM realizations of the activation and weight buffers and is expressed by

$$E_{SRAM} = E(S) \times P + 2E(S) \times A \tag{4.4}$$

Here $E(S)$ is energy for one SRAM access. (We assume read and write consume same amount of energy but the model can easily be extended to differentiate between the two). Also P is number of parameters and A is number of activations in the considered DNN.

The first term corresponds to the energy to fetch the parameters of each layer from the weight buffer into the MAC array. The energy represented by this term equals to the number of parameters multiplied by energy of a single read from the weight buffer.

The second term consists of two parts. Half of it corresponds to reading intermediate values generated by the previous layer from the activation buffer into the MAC array. Therefore the total energy represented by this term equals to total number of activations multiplied by the energy of a single access to the activation buffer. The second half of it corresponds to writing back the MAC results into the activation buffer (which again equals to the number of activations times SRAM access energy). Since there may exist very complicated memory hierarchy and dataflow pattern in reality, we note that Eq. 4.3 and Eq. 4.4 only provide a quick but rough estimate of the most dominant components of energy. For more accurate energy estimation, we refer the interested reader to [6].

Finally E_{REST} is the energy corresponding to the rest of the components such as control unit and activation, pooling, and normalization unit. These can be ignored

Table I: Operation energy based on 45nm process from [23]

Operation	**Energy**
32-bit ADD	0.9pJ
32-bit MULT	3.7pJ
32-bit SRAM	5pJ
32-bit DRAM	640pJ

compared to the other terms.

Note that based on this underlying hardware, this energy model depends on the specifications of a given DNN, specifically the values of M, P, and A. It also depends on the energy for a single MAC, SRAM, and DRAM access which are constant and are determined based on an assumed quantization and a process technology. In our experiments we used the numbers based on 45nm CMOS process reported in [23, 36] with relevant numbers are listed in Table I. We assumed 32-bit floating point MAC unit is used to perform both multiplication and addition. We also assumed each multiplication is followed immediately by an addition, thus, the energy consumption of MAC operation is the summation of the energy consumption of a multiplication and an addition.

5 STRUCTURE SIMPLIFICATION VIA NEURON ELIMINATION

Structural simplification techniques are based on simplifying the network model, for example by means of pruning the edges with weights below a threshold [23]. Structure simplification of artificial neural networks, including DNNs have been an active area of research for the past several decades [39, 53, 87]. These procedures typically aim to simplify the network as much as possible while ensuring negligible loss in accuracy. However, reduction in both the parameters and the edges of the network, while correlated, is not equivalent to reduction in energy [88]. In fact, considering energy consumption based on an underlying hardware model and its tradeoff with accuracy have not gained enough attention so far. This may be because only recently the need for finding solutions for big data applications which require low energy but can tolerate error has emerged. Besides, the majority of structure simplification techniques are published in communities with insufficient emphasis on hardware. This chapter aims to bridge this gap by proposing a simplification technique which directly relates to the underlying hardware model. We also show that considering the energy-accuracy tradeoff can impact how the network is simplified.

We propose a procedure that facilitates exploration of energy and accuracy tradeoffs of varying configurations of a structurally simplified, trained DNN. Our procedure seeks to eliminate redundant neurons at a considered hidden layer and simultaneously updates the weights connecting to the remaining neurons. When a neuron is removed, all edges (weights) connecting to that neuron are also removed, resulting in significant reduction of storage and energy while minimally affecting the accuracy. The task of eliminating redundant neurons in a hidden layer is formulated as a subset selection problem. We propose a novel approach using QR factorization with column pivoting to solve this problem efficiently. This procedure may be applied to different layers in a DNN. For each layer multiple configurations are realized and the configurations of multiple layers together form energy-accuracy tradeoff curves, from which Pareto-optimal can be easily identified. We also propose strategies for effective application of our algorithm in a sequential order to multiple layers in a DNN to decide the degree of simplification per layer such to maximize the compression ratio while an overall budget for accuracy degradation is not exceeded. Thus, according to the constraints of

the target platform, the appropriate pruned model can be selected and deployed.

A distinct feature of our structure simplification algorithm is that it does not require lengthy retraining because only the outgoing weights of remaining neurons of the given hidden layer need to be updated and this is done via matrix multiplication right after the redundant neurons are identified and removed. This proves to be advantageous in terms of time and energy, especially when our algorithm is applied to large scale DNNs such as the `AlexNet`. Yet, retraining may still be applied if desired to further fine tune the performance.

Our experiments show energy-accuracy tradeoff provides clear guidance to achieve efficient realization of trained DNNs. We also observe significant implementation cost reductions with up to 11.50X in energy and 12.30X in storage while the accuracy degradation is negligible. Moreover, our proposed procedure is also compatible with other types of techniques for structure simplification, and orthogonal with techniques such as weight quantization and stochastic computing. This is an advantage since the combination has potential for far more energy saving.

The remaining of this chapter is organized as follows. We discuss our procedure in Section 5.1 followed by methods for multiple layers in 5.1 and simulation results in 5.2.

5.1 Our Procedure

In this section we describe our procedure for structure simplification which exploits the redundancy among neurons organized in the same layer in a DNN. Neural networks are typically over-parameterized and have significant redundancy [14, 23], partially due to the fact that most models are manually designed. It is empirically observed that the output of the neurons within the same layer tend to be linearly dependent on each other. As such, the output of one neuron may be estimated using a linear combination of outputs of other neurons with little approximation error. If so, this neuron may be eliminated from the network without affecting the overall performance (after updating the corresponding weight matrices). Removing a neuron will also remove all the incoming and outgoing edges associated with that neuron and hence achieve the goals of reduction of computation, storage space of the weights as well as energy consumption.

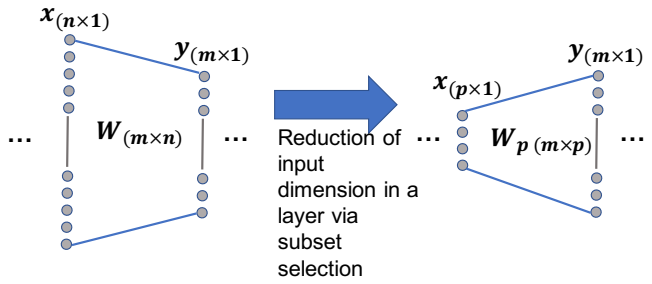

Figure 5.1: Overview of redundant neuron elimination technique

At a high level, our procedure consists of three steps: 1) identifying the most representative input neurons of the given layer through pivoted QR factorization; 2) removing all the other input neurons along with their incoming and outgoing edges; 3) manipulating the weight matrices of the immediately-affected layers to mostly preserve the values of the output neurons of the given layer. As a result, the subsequent layers will be unaware of the change made in the given layer and won't need any modification. In the remainder of this section, we provide detailed explanation of the above steps.

Overview of Formulation As A Subset Selection Problem

Figure 5.1 gives a high-level description of our neuron elimination technique. Let $\mathbf{x} \in \mathbb{R}^{n\times 1}$ be a $n \times 1$ input vector to the $m \times n$ weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ and $\mathbf{y} \in \mathbb{R}^{n\times 1}$ be the corresponding $m \times 1$ output vector. Our goal is to identify a subset of p elements in the $\mathbf{x}$ vector, denoted by $\mathbf{x}_p \in \mathbb{R}^{p\times 1}$, and a $n \times p$ matrix $\mathbf{G} \in \mathbb{R}^{n\times p}$ such that $\mathbf{x} = \mathbf{G}\mathbf{x}_p$ for any $\mathbf{x}$ in the training and testing dataset. If this subset of p neurons may be identified, the weight matrix feeding into these neurons will be reduced to p rows, and the new weight matrix ($\mathbf{W}_p \in \mathbb{R}^{m\times p}$) that these n neurons will be fed into will require only p columns. Moreover, once the indices of the selected p neurons are determined, the incoming weight matrix may be pruned to only keep the selected p rows without any change to the kept weights. On the other hand, the out-going weight matrix $\mathbf{W}_p$ of dimension $m \times p$ will be determined through subspace projection based on the set of provided input vectors.

To formulate this problem, we denote a matrix $\mathbf{X} \in \mathbb{R}^{n\times K}$ to be the output of

n neurons with respect to K training instances. Ideally, one would want these K instances to be same as the ones which trained this DNN. In practice, one may use cross-validation to obtain this $\mathbf{X}$ matrix. Given a positive integer p, our goal then is to find a matrix $\mathbf{X}_p \in \mathbb{R}^{p \times K}$ and a matrix $\mathbf{G} \in \mathbb{R}^{n \times p}$ such that:

1. Each row of $\mathbf{X}_p$ is a row of $\mathbf{X}$;

2. The Frobenius norm of the approximation error:

$$\mathbf{E}_p = ||\mathbf{X} - \mathbf{G}\mathbf{X}_p||_{\mathbf{F}} \tag{5.1}$$

   is minimized.

Condition 1 states that the p rows of $\mathbf{X}_p$ are a *subset* of p rows of the matrix $\mathbf{X}$. Condition 2 states that the approximation of $\mathbf{X}$ by $\hat{\mathbf{X}} = \mathbf{G}\mathbf{X}_p$ should minimize the sum of square of the approximation error over the K instances of the training data. In other words, the goal is to find a subset of exactly p rows of $\mathbf{X}$ that represent $\mathbf{X}$ as much as possible in a projection sense.

The approximated $\mathbf{X}$ can be obtained by projecting the p rows of $\mathbf{X}_p$ back to the space spanned by the rows of $\mathbf{X}$, which is:

$$\hat{\mathbf{X}} = \mathbf{X}\mathbf{X}_p^T(\mathbf{X}_p\mathbf{X}_p^T)^{-1}\mathbf{X}_p = \mathbf{X}\mathbf{X}_p^\dagger\mathbf{X}_p = \mathbf{G}\mathbf{X}_p \tag{5.2}$$

where $\mathbf{X}_p^\dagger$ is the pseudo-inverse of the $\mathbf{X}_p$ matrix such that $\mathbf{X}_p\mathbf{X}_p^\dagger = \mathbf{I}_p$ but $\mathbf{X}_p^\dagger\mathbf{X}_p \neq \mathbf{I}_n$, and $\hat{\mathbf{X}}$ is the approximated $\mathbf{X}$ from $\mathbf{X}_p$. In the above equation,

$$\mathbf{G} = \mathbf{X}\mathbf{X}_p^\dagger$$

is a $n \times p$ matrix with p rows consisting of 1s and 0s, and remaining $n - p$ rows that use the p rows of the $\mathbf{X}_p$ matrix to recover the remaining $n - p$ rows of the original $\mathbf{X}$ matrix. To compute the new weight matrix $\mathbf{W}_p$, note that

$$\mathbf{W}\mathbf{X} \approx \mathbf{W}\hat{\mathbf{X}} = \mathbf{W}\mathbf{G}\mathbf{X}_p \tag{5.3}$$

Thus the new weight matrix that will be connected to the reduced p neurons can be

found as:

$$\mathbf{W}_p = \mathbf{WG} = \mathbf{WX}\mathbf{X}_p^{\dagger} \tag{5.4}$$

As for the weight matrix of the preceding layer, as discussed earlier, with the removal of $n - p$ neurons, the corresponding rows will be removed, leaving a weight matrix with only p rows. The value of these p rows shall remain unchanged.

What is left to be decided is how to select the p neurons. This requires selection of p neurons from n candidates. Hence, this can be formulated as a *subset selection problem*: Selecting a subset of p neurons out of n possible candidates such that $\mathbf{E}_p$ is minimized. This is a very hard and well-known problem, although its NP-hardness is still an open problem [2]. It is obvious that we can find the optimal p rows in $\binom{n}{p} = \frac{n!}{(n-p)!p!}$ time. However, for practical DNNs, n is in the order of hundreds or thousands. For example, the FC1 layer in LeNet5 has 800 input neurons, and the FC1 layer in CaffeNet has 9216 input neurons. This makes exhaustive approach prohibitively slow even for one p value. Fortunately, there exists several approximation algorithms that can give approximate solution for this problem.

Next we discuss our algorithm based on QR factorization which sub-optimally but quickly solves this problem and also records energy and accuracy at each step to generate a tradeoff plot.

Algorithm

Algorithm 1 Reduce dimension of $\mathbf{X}_{n \times K}$

procedure REDUCEDIMENSION($\mathbf{X}_{n \times K}$, layer ℓ)
    p = rank($\mathbf{X}$); $E_{cur} = 0$
    do
        Select p rows of $\mathbf{X}$ using Algo. 2
        Compute $\mathbf{W}_p$ using Eq. 6
        Generate simplified network N as in Fig. 5.1
        Record *accuracy degradation* and energy E_{cur} of N
        $p = p - 1$
    while ($p \neq 0$ and $\mathtt{accuracy\ degradation} \leqslant \epsilon$)
    Generate accuracy vs energy tradeoff of stored configurations
end procedure

Algorithm 2 Find the most p representative rows in $\mathbf{X}_{n \times K}$

1: **procedure** FINDREPRESENTATIVEROWS($\mathbf{X}_{n \times K}$, p)
2: Perform SVD decomposition on $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$
3: Set $\mathbf{U}_p$ to be the first p columns of $\mathbf{U}$
4: Perform QRD on $\mathbf{U}_p^T$ and get $\mathbf{U}_p^T\mathbf{P} = \mathbf{QR}$
5: The first p columns in permutation matrix $\mathbf{P}$ identifies the p representative rows in $\mathbf{X}$
6: **end procedure**

The high-level algorithm is given in Algorithm 1. We start by setting $p = \text{rank}(\mathbf{X})$ rows and set the current energy E_{cur} to be 0.

Next in lines 4 to 6 we generate a simplified network *N* corresponding to this value of p. (We utilize Algorithm 2 to select the p representative rows, for a fixed value of p, using QR factorization which we discuss later.) Next we evaluate the accuracy degradation and update the energy corresponding to *N*. The accuracy degradation is found via performing simulation over the testing dataset and evaluating the rate of misprediction at the output of the simplified network. Energy is computed as described in Section 4.2. Next, we decrement p by one and evaluate the termination condition and continue simplifying the network until the condition is satisfied.

The termination condition is if p reaches 0, or if the degradation in accuracy at the output (measured as difference in percentage accuracy of the original network and the simplified one) exceeds a threshold percentage ϵ. Once the termination condition is satisfied, all simplified networks corresponding to the explored values of *p* will be generated as given in line 10. If ϵ is set to be maximum (=1), then all the configurations will be provided. Intermediate values of the threshold allows exploring the tradeoff in a smaller range. If the threshold is set to be a very small value (such as 2% in some of our experiments) only the configurations with negligible degradation in accuracy will be generated. We note that the energy consumption for each configuration is the output of our algorithm (along with the configuration itself) and the degree to which the given layer is pruned is only controlled by p. In other words, energy is only used as a metric to evaluate the pruned model but not to instruct how the original model should be pruned.

Next we explain the selection of p representative rows for a fixed value of p. As

mentioned before, optimal selection of p rows is infeasible. However, well-known approximation algorithms exist to approximately solve this problem efficiently [2]. For example, in [18], Golub proposed to incorporate a pivoted QR factorization to select the $\mathbf{X}_p$ matrix while obtaining the estimate of the G matrix. It has been shown in [2] that using pivoted QR factorization, the resulting matrix $\mathbf{X}_p$ satisfies

$$\mathbf{E}_p = \|\mathbf{X} - \mathbf{G}\mathbf{X}_p\|_{\mathbf{F}} \leqslant \sqrt{n-p} \cdot 2^p \cdot \sigma_{p+1}(\mathbf{X}) \tag{5.5}$$

where $\sigma_{p+1}(\mathbf{X})$ is the $(p+1)$th largest singular value of the $\mathbf{X}$ matrix.

Algorithm 2 describes the selection of p representative rows using pivoted QR factorization. We first perform singular value decomposition (SVD) on $\mathbf{X}$ to obtain matrix $\mathbf{U}$. We then select $\mathbf{U}_p$ to be the submatrix formed by the first p columns of $\mathbf{U}$. Then we apply QR factorization with column pivoting on $\mathbf{U}_p^T$. The QR factorization procedure generates the matrices $\mathbf{Q}$ and $\mathbf{R}$ as well as a *permutation* matrix $\mathbf{P}$. (A permutation matrix is one that is obtained by permuting the rows of an identity matrix.) The specific permutation implied by the first p columns in $\mathbf{P}$ identifies the p representative rows in $\mathbf{X}$.

Computational Complexity: In Algorithms 1 and 2 the main computational tasks are (a) performing QR factorization and (b) evaluation of the classification accuracy. The computational complexity of (a) is polynomial, for example $O(pn^2 - 2/3n^3)$ using the Householder algorithm [19]. The classification is performed for all K testing vectors. Assume there are L layers in a DNN, and each weight matrix has a similar size of order $m \times n$, then each run of classification with all testing vectors will incur $O(KLmn)$ operations. If all $\min(m,n)$ different p values are tested, the computation complexity will be $O(\min(m,n) \cdot KLmn) \approx O(KLn^3)$ where we assume $m \approx n$. So the computational complexity of (b) is $O(KLn^3)$ and the overall complexity is polynomial. For example, the runtime of our procedure on layer FC1 in `LeNet5` and K=10000 was under 35 seconds.

Dependency on the Input Dataset: Since the output of a hidden neuron is a function of the weight matrix and the output of the previous layer, which also depends on the input data. Thus, our method is indeed dependent on the (training) dataset. This is why we *uniformly* sample K instances from the training set to make sure they share the same underlying distribution as the testing instances. Otherwise we may experience a

much higher accuracy degradation. In fact, the weights of a DNN are trained using training dataset and hence are data dependent as well. Thus, if the same architecture is trained with a different dataset, our technique may select a different set of p neurons to be the most representative which may result in a different accuracy degradation.

Relationship with PCA: The proposed algorithm is similar to PCA in the sense that both of them aim to identify the most representative p features from the matrix $\mathbf{X}$. PCA seeks to map data points from high dimensional space to low dimensional space by projecting the data points on the directions specified by the top p eigen-vectors. Mathematically, the coordinates of the i^{th} data point in the new space are given by the i^{th} column of $\mathbf{VX}$, where $\mathbf{V} \in \mathbb{R}^{p \times n}$ is formed by the top p eigen vectors. In other words, the resulting point is a linear combination of the features of the original data point, and the resulting matrix $X_p \in \mathbb{R}^{p \times K}$ is obtained by linear combination of the n rows of X. It implies that the PCA procedure still requires preserving all the n neurons to generate the projected p neurons. This is the key difference compared to our technique in which only p out of the n neurons are kept and the rest are discarded.

Application to Convolutional Layers: At first glance, our algorithm seems mostly applicable to fully connected layers where the n input neurons over K training samples naturally map to the matrix $\mathbf{X}$. This makes the proposed technique perfectly suitable for architectures like MLP [35] and LSTM [34].

However, in theory it is possible to apply our technique to a convolutional layer. For example, we can represent the input neuron matrix in the form $X_{n \times K}$[1]. It results in the corresponding weight matrix to be one large sparse matrix with many elements repeated in a specific pattern. There are two reasons why the weight matrix in this approach is sparse and has repeated patterns. First, in a convolutional layer, each output neuron is only connected to a small portion of input neurons. Second, all the output neurons in the same feature map share the same set of kernel weights, which means the same set of weights is heavily reused for the output neurons in the same feature map. Even though this approach can be applied in theory, the specific pattern and sparsity of the weight matrix make our neuron elimination technique less effective in practice.

In the second approach (which can be considered a better one), we create a

[1] n is the number of input neurons of all input feature maps for the given layer and K is the number of training samples used in our algorithm.

separate weight matrix for each receptive window and apply our technique. Here the neurons in each receptive window are treated as input neurons. Each training sample corresponds to $\mathfrak{m}$ columns in $\mathbf{X}$ and $\mathbf{X} \in \mathbb{R}^{\mathfrak{n}' \times \mathsf{K}'}$, where $\mathfrak{n}'$ equals to the number of neurons in one receptive window, $\mathsf{K}' = \mathfrak{m} \times \mathsf{K}$, and $\mathfrak{m}$ equals to the number of output neurons in one feature map. In this approach, only the common neurons which are eliminated across all receptive windows and all feature maps can be ultimately eliminated which decreases the effectiveness of our technique.

Generally speaking there are better ways in practice to handle convolutional layers, for example by tensor approximation and low-rank expansion techniques as in [15] and [45]. Our proposed subset selection technique also relates to low rank approximation. Therefore, we believe that our method can also be extended to convolutional layers using similar extensions as in the above prior works. Besides the above, another effective way to handle a convolutional layer is to apply Fourier transformation first, and perform all calculations in that domain.

Extension to Multiple Layers

It is straightforward to apply the algorithm introduced in the previous subsection to just one layer in a Deep Neural Network. In order to achieve more energy savings and higher overall compression in the number of stored network parameters, it is natural to try to apply the algorithm to multiple layers of a DNN. However, this problem is not as straightforward as applying the algorithm to just one layer, because we need to decide to what extent the algorithm should be applied to each layer. This is because the the threshold ϵ defined in Algorithm 1 (for percentage degradation in accuracy at the outputs between the original and simplified networks) which controls the number of iterations and the termination condition should be defined a-priori when applying the algorithm to a layer. When a budget for this overall accuracy degradation across all the layers is specified, it is unclear how this threshold should be split per layer. In other words, it is unclear how individual ϵ should be specified when applying the algorithm to each layer.

In this subsection, we first study applying Algorithm 1 for two consecutive layers of a DNN, then we introduce three different methods which can be used to determine the individual ϵ per layer for an overall budget of accuracy degradation across all the

layers. In our experiments we set this overall budget to a small value (i.e., only 2%). This determination should be done in order to reach the maximum compression ratio of the DNN size, where compression ratio is defined as the ratio of number of weights in the original network to the one in the simplified network.

In general, this issue exists whenever we want to apply layer based structure simplification algorithms, such as low-rank approximation [15], to multiple layers of a DNN because these techniques also result in some degree of accuracy degradation after simplification. We note that even though we only validated our methods using our neuron elimination algorithm, they are generalizable to be used with other layer based structure simplification algorithms. Before discussing our three methods, we first make a note on the ordering of the layers when applying our algorithm to multiple layers. We use the layer ordering following the structure of the network as we move forward from the input layer to output layer. For example, in a DNN that layer l_1 is followed by layer l_2, if we want to eliminate n_1 neurons from l_1 and n_2 neurons from l_2, we will achieve higher accuracy if we apply the algorithm first to l_1 and then l_2 rather than apply it first to l_2 and then l_1. This is because our neuron elimination algorithm induces little approximation error at the output of the considered layer. Therefore it makes sense to simplify one layer prior to simplifying any of the subsequent layers after it. This ensures when applying the algorithm to a given layer, the structure of the network in all prior layers will remain unchanged; otherwise the algorithm won't be as effective if it simplifies a layer based on wrong assumption about the neurons that feed into it.

The above ordering will be used for the three methods that we discuss next.

In our first method, we first apply Algorithm 1 to each layer separately, each time keeping the other layer in its original form. This is done assuming the ϵ threshold applied to each layer is same as the overall budget in excessive error across all the layers. For each layer and each iteration of Algorithm 1, we record the degree of accuracy degradation (relative to the original network) as well as the degree of improvement obtained in compression ratio of the network corresponding to that iteration. Here the improvement in compression ratio is computed as compression ratio minus 1. We then plot these two parameters as shown for example in Figure 5.2 for two layers when the overall accuracy degradation budget is 2%. Each point corresponds to one iteration of the algorithm in the corresponding layer where 1%

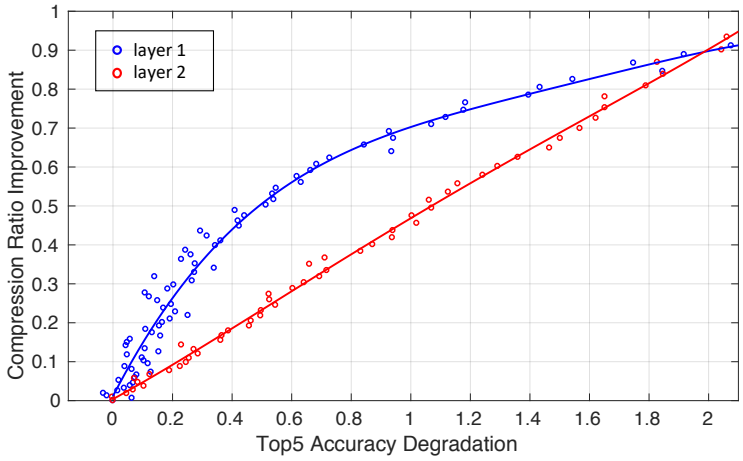

Figure 5.2: Improvement in compression ratio versus degradation in accuracy ϵ (i.e., difference in output accuracy of the original and simplified networks) for each iteration of Algorithm 1 when applied to two layers in the DNN independently

of the neurons are eliminated per iteration. (This is done by modifying line 8 of Algorithm 1 so that parameter p is decremented by 1% of neurons.) From these plots we use a mechanism to decide the individual thresholds in excessive error, denoted by ϵ_1 and ϵ_2 which should be specified for the first and second layers respectively. Specifically we first decide ϵ_1 and apply the algorithm to the first layer to follow the intuitive ordering that we discussed. Next, the simplified first layer is fixed and the second layer is simplified with $\epsilon_2 = \epsilon - \epsilon_1$ to ensure the overall excessive error budget is not exceeded. *The main challenge in specifying these thresholds is to maximize the overall compression of the network when Algorithm 1 is applied in a sequential order to the two layers, and when sum of* ϵ_1 *and* ϵ_2 *is the overall excessive error budget* ϵ.

Next we explain how ϵ_1 is determined in the first strategy to achieve the above goal. Let us consider the plot in Figure 5.2 again. We fit a curve to the points corresponding to each layer. The *slope of each curve* shows the rate in improvement in compression per change in degradation in accuracy. We pick ϵ_1 to be the point on the curve of the first layer in which the slope of the curve becomes lower than that of the second layer. This approximates the point before which it will be more beneficial to degrade accuracy by compressing layer 1, and after which it would be better to

degrade the remaining budget in accuracy by compressing layer 2. In the example in the figure, this occurs about 0.65% in the X-axis which corresponds to $\epsilon_1 = 0.65\%$ in the X-axis and $\epsilon_2 = 2 - 0.65 = 1.35\%$.

Our second method is based on the idea that if the first layer provides χ_1 compression ratio and the second layer gives χ_2 compression ratio, then the overall compression ratio can be approximated by $\chi_1 \times \chi_2$. By identifying the point where the highest overall compression ratio is achieved, we can estimate to what extent we should apply the algorithm to the first layer and then the second one. This method is similar to the first method in that we start with applying the algorithm to the two layers separately (keeping the other layer in the original form) and record accuracy degradation after each iteration. However, instead of recording the *improvement* in compression ratio, we record the actual compression ratio after each iteration. Similar to the first strategy, we first apply the algorithm to each of these layers until the accuracy degradation reaches ϵ in each case. Then, for each degradation value d from 0 to ϵ, we multiply the compression ratios corresponding to d of the first layer and the compression ratio corresponding to $\epsilon - d$ in the second layer to get the estimated overall compression ratio. The reason we use $\epsilon - d$ for the second layer is because with the first layer taking d degradation, the second layer will only be left with $\epsilon - d$ degradation budget. We then identify the degradation d_{max} where the estimated compression has the highest value, and we use it the same way as the first method.

We note that both methods mentioned so far are not precise because as we apply the algorithm to the first layer, the dynamics of the second layer will also change. So working with independently-generated curves is an approximation. These two methods just aim to provide a good estimation of the split point and hope to achieve a good, but may be sub-optimal, compression ratio.

In our third method, we consider various accuracy degradation points d between 0 and ϵ. For each d we first apply the algorithm to the first layer with $\epsilon_1 = d$. Then for the *simplified* layer 1, we apply the algorithm to the second layer with $\epsilon_2 = \epsilon - d$, and record the overall compression ratio obtained from both layers. This strategy effectively performs a thorough search in the solution space and it guarantees to produce the optimal solution subject to the step size of d. However, due to the thorough search of the solution space, the runtime of this method is significantly longer than that of the first two because it requires running Algorithm 1 for each layer

Table II: Information on experimented neural networks

	LeNet5	LeNet300100	CIFAR10	CaffeNet
CS1	5×5×1×20	-	5×5×3×32	11×11×3×96
CS2	5×5×20×50	-	5×5×32×32	5×5×48×256
CS3	-	-	5×5×32×64	3×3×256×384
CS4	-	-	-	3×3×192×384
CS5	-	-	-	3×3×192×256
FC1	4×4×50×500	784×300	4×4×64×10	6×6×256×4096
FC2	500×10	300×100	-	4096×4096
FC3	-	100×10	-	4096×1000
#Edgs	2293K	266K	12.3M	724M
#Parm	431K	266K	89.4K	61M

for each value of d, while the first two methods only require applying Algorithm 1 to each layer once.

5.2 Simulations Results

Table II shows the networks used in our experiments. `LeNet5` is a CNN that has two sets of convolutional and subsampling layers (CS1 and CS2), and two fully-connected (FC) layers (FC1 and FC2). `LeNet300-100` is composed of three FC layers. `CIFAR10` has three sets of convolutional and subsampling layers, followed by one FC layer. `CaffeNet` which is Caffe's replication of `AlexNet` [51] has five convolutional layers for which CS1, CS2, CS5 are followed by subsampling layers. The convolutional layers are followed by three FC layers. For each layer we list the number of parameters. We also report total number of edges and parameters per network.

Both LeNet networks were trained using Caffe [46] with 60K images designated for training from the MNIST dataset [54]. Similarly we trained and tested `CIFAR10` network using Caffe with the CIFAR-10 dataset [50]. The `CaffeNet` network and its parameters (when trained for the ImageNet 2012 dataset) was downloaded by using a script provided by Caffe. Table III column 8 reports the achieved classification accuracy for each network after training (for the original case when not applying any structure simplification). For `CaffeNet` we report top-1 / top-5 accuracies. These

Table III: Comparison of required memory

	Original		After				Accuracy	
	Params	**Total**	**Params**	**Total**	**%original**	**Ratio**	**Original**	**After**
LeNet5-FC1	13.8 Mb	13.9 Mb	1.1 Mb	1.2 Mb	8.13%	12.30X	99.10%	97.26%
LeNet5-FC2	13.8 Mb	14.0 Mb	1.7 Mb	1.9 Mb	12.67%	7.88X	99.10%	97.25%
LeNet300-100-FC1	8.5 Mb	8.5 Mb	1.6 Mb	1.6 Mb	18.85%	5.30X	98.21%	96.57%
LeNet300-100-FC2	8.5 Mb	8.5 Mb	1.5 Mb	1.5 Mb	17.64%	5.67X	98.21%	96.24%
LeNet300-100-FC3	8.5 Mb	8.5 Mb	7.6 Mb	7.6 Mb	89.40%	1.12X	98.21%	96.92%
CIFAR10-FC1	2.9 Mb	3.2 Mb	2.5 Mb	2.8 Mb	89.21%	1.12X	81.49%	79.57%
CaffeNet-FC1	243.8 MB	244.7 MB	129.1 MB	130.0 MB	52.94%	1.89X	56.67% / 79.59%	53.72% / 77.67%
CaffeNet-FC2	243.8 MB	244.7 MB	134.8 MB	135.3 MB	55.30%	1.81X	56.67% / 79.59%	54.19% / 77.80%
CaffeNet-FC3	243.8 MB	244.7 MB	186.2 MB	187.1 MB	76.37%	1.31X	56.67% / 79.59%	53.57% / 77.61%

classification accuracies are similar to what are reported in the literature [50, 51, 54] for these networks. In all experiments, to measure the (classification) accuracy, we used the subset of MNIST, CIFAR10, and ImageNet datasets which were designated for testing. We simulated each network for each test image and report the rate of correct predictions in the output.

Comparison of Energy and Memory for A Single Layer

In our first experiment we evaluate our procedure with the threshold ratio in Algorithm 1 is set to 2%. We then pick the solution from the last iteration of the algorithm. This setting outputs the configuration with the smallest energy and little loss in accuracy.

We apply our procedure to different fully connected (FC) layers of different networks. In Table III, each row corresponds to simplification of a specific layer in one network. We report results for original case and after simplification. For each case, we report the required memory to store network parameters as well as the total memory which additionally accounts for storing the activations. These are reported for the entire network even though the simplification is applied to one layer per row. For example in the first row, the memory saving after applying Algorithm 1 to FC1 in the `LeNet5` is 12.30X. For `CaffeNet` the largest savings is 1.89X which was found at layer FC1. In the last two columns we also report accuracy for the original and after cases. The degradation in accuracy remains negligible. The grey rows highlight the

Table IV: Comparison of number of parameters and post-retraining accuracy of pruned models with the SVD technique [15]

	Original	After		Accuracy		
	#Params	#Params	Ratio	Original	After	Retrain
LeNet5-FC1	430.5k	35.0k	12.30X	99.10%	97.26%	99.01%
LeNet5-FC2	430.5k	54.7k	7.88X	99.10%	97.25%	99.24%
LeNet5-FC1(SVD)	430.5k	44.8k	9.61X	99.10%	97.40%	99.23%
LeNet5-FC2(SVD)	430.5k	429.6k	1.00X	99.10%	97.81%	99.28%
LeNet300-100-FC1	266.2k	50.2k	5.30X	98.21%	96.57%	98.37%
LeNet300-100-FC2	266.2k	50.0k	5.67X	98.21%	96.24%	97.57%
LeNet300-100-FC3	266.2k	238.0k	1.12X	98.21%	96.92%	98.06%
LeNet300-100-FC1(SVD)	266.2k	57.0k	4.67X	98.21%	96.74%	97.32%
LeNet300-100-FC2(SVD)	266.2k	239.0k	1.11X	98.21%	97.30%	98.03%
LeNet300-100-FC3(SVD)	266.2k	266.0k	1.00X	98.21%	97.51%	97.97%
CIFAR10-FC1	89.4k	79.8k	1.12X	81.49%	79.57%	82.17%
CIFAR10-FC1(SVD)	89.4k	88.5k	1.01X	81.49%	81.49%	81.77%
CaffeNet-FC1	61.0M	32.3M	1.89X	56.67% / 79.59%	53.72% / 77.67%	-
CaffeNet-FC2	61.0M	33.7M	1.81X	56.67% / 79.59%	54.19% / 77.80%	-
CaffeNet-FC3	61.0M	44.2M	1.31X	56.67% / 79.59%	53.57% / 77.61%	-
CaffeNet-FC1(SVD)	61.0M	25.3M	2.41X	56.67% / 79.59%	53.74% / 77.82%	-
CaffeNet-FC2(SVD)	61.0M	45.0M	1.36X	56.67% / 79.59%	52.91% / 77.38%	-
CaffeNet-FC3(SVD)	61.0M	57.8M	1.06X	56.67% / 79.59%	51.64% / 77.56%	-

layer with the highest ratio column per network.

To put the proposed technique into perspective, we also applied low-rank approximation method (denoted by SVD) given in [15] to the same layers as we did with our proposed technique. Note this technique is not energy-aware. We show the results in Tables IV and V. In Table IV, columns 2 and 3, we also report the number of parameters of the entire network before and after pruning different layers of each network using our technique and SVD technique. The results suggest that with 2% accuracy degradation, neuron elimination technique achieves higher compression ratio and energy savings in all cases compared to SVD, except for CaffeNet layer FC1. In terms of accuracy, both our technique and SVD have negligible degradation compared to original as can be seen in columns 5 and 6.

We also applied retraining to the pruned model using our technique and SVD and report post-training accuracy in the last column in Table IV except those of CaffeNet due to limited resource. As can be seen, retraining can recover the accuracies to a

Table V: Comparison of energy to perform classification of 1 image in corresponding dataset with SVD technique [15]

	Computation Energy	**Communication Energy**			**Total**	**Accuracy**
	MAC	$\text{SRAM}_{weights}$	$\text{SRAM}_{activations}$	DRAM		
`LeNet5` (Original)	10.55 uJ	2.15 uJ	0.16 uJ	276.02 uJ	288.88 uJ	99.10%
`LeNet5` (After)	1.91 uJ	0.18 uJ	0.13 uJ	22.90 uJ	25.11 uJ	97.26%
Original/After	5.52X	11.94X	1.23X	12.05X	11.50X	
`LeNet5 (SVD)`	8.77 uJ	0.22 uJ	0.16 uJ	29.17 uJ	38.33 uJ	97.40%
Original/After	1.20X	9.77X	1.00X	9.46X	7.54X	
`LeNet300-100` (Original)	1.22 uJ	1.33 uJ	8.02 nJ	170.87 uJ	173.43 uJ	98.21%
`LeNet300-100` (After)	0.22 uJ	0.23 uJ	5.54 nJ	30.56 uJ	31.02 uJ	96.24%
Original/After	5.55X	5.78X	1.45X	5.59X	5.59X	
`LeNet300-100 (SVD)`	0.26 uJ	0.29 uJ	8.26 nJ	36.99 uJ	37.55 uJ	96.74%
Original/After	4.69X	4.59X	0.97X	4.62X	4.62X	
`CIFAR10` (Original)	56.57 uJ	0.45 uJ	0.47 uJ	59.21 uJ	116.69 uJ	81.49%
`CIFAR10` (After)	43.50 uJ	0.40 uJ	0.43 uJ	53.03 uJ	97.37 uJ	79.57%
Original/After	1.30X	1.13X	1.09X	1.12X	1.20X	
`CIFAR10 (SVD)`	56.57 uJ	0.44 uJ	0.47 uJ	58.61 uJ	116.09 uJ	81.49%
Original/After	1.00X	1.02X	1.00X	1.01X	1.01X	
`CaffeNet` (Original)	3.32 mJ	0.30 mJ	7.37uJ	39.11 mJ	42.75 mJ	56.67% / 79.59%
`CaffeNet` (After)	2.98 mJ	0.16 mJ	7.02uJ	20.75 mJ	23.90 mJ	53.72% / 77.67%
Original/After	1.12X	1.88X	1.05X	1.88X	1.79X	
`CaffeNet (SVD)`	3.17 mJ	0.13 mJ	7.37uJ	16.27 mJ	19.57 mJ	53.74% / 77.82%
Original/After	1.05X	2.31X	1.00X	2.40X	2.18X	

level that is comparable to that of the original model. After retraining, some pruned models even have higher accuracy than the original model, and we believe this is partially due to the fact that pruned models generalize better than the original one.

Next in Table V we report the energy corresponding to the highlighted (grey) rows of Table IV. We calculate and report the breakdown of energy between computation (MAC array), DRAM, two SRAM terms corresponding to weight and activation buffers which are calculated according to Eq. 4.2, Eq. 4.3, and Eq. 4.4, which are discussed in Section 4.2. We also report the breakdown of energy consumption with respect to layer type in Figure 5.3. As it can be seen the majority of energy consumed at each network is due to DRAM energy, and then due to MAC energy. In terms of layer type, the majority of energy is consumed by FC layers because the energy consumption per DRAM access is more than two orders of magnitude of that of performing a MAC operation, and FC layers have many more parameters compared

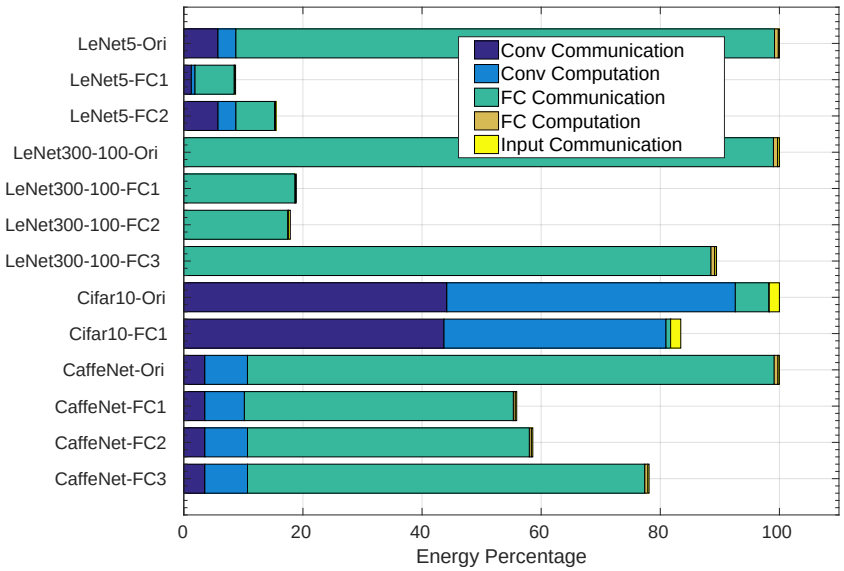

Figure 5.3: Communication versus computation energy distribution among different types of layers for all experimented models. For each model, the energy consumption of the original network is used as reference.

to convolutional layers. The only exception here is CIFAR10 because its FC layer is very small compared to its other layers. Overall, compared to the original, the energy saving ratios are 11.50X, 5.59X, 1.20X, 1.79X, in `LeNet5`, `LeNet300-100`, `CIFAR10`, and `CaffeNet`, respectively.

Since only one FC layer is pruned in each experiment, the ratio of energy saving is less related to the absolute size of the network, but rather highly related to the structure of the given network. In general, if the size (in terms of number of parameters) of the given layer is relatively large compared to the entire network, we could expect a relatively high energy saving ratio. For example, the relative sizes of the highlighted layers of LeNet5, CIFAR10, and CaffeNet in Table III are 92.92%, 11.45%, and 61.88%, respectively. This order is correctly corresponding to the order of their energy saving ratios, which are 11.50X, 1.20X, and 1.79X. Another factor that comes into play is the size of the previous layer in the network. If the number of input neurons of the previous layer is large, by eliminating input neurons of the given layer, we will

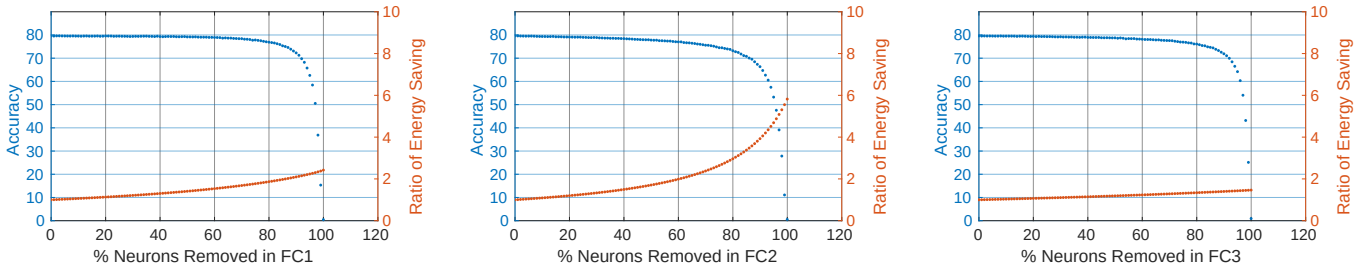

Figure 5.4: Tradeoff in accuracy and rate of energy saving when eliminating neurons in different layers (FC1, FC2, FC3) of `CaffeNet`

also eliminate a large number of edges in the previous layer, thus, achieving a high ratio of energy saving. This explains why the FC2 layer in LeNet300-100 gives a higher energy saving ratio than the FC1 layer. This also explains why the FC2 layer in CaffeNet achieves almost the same amount of compression ratio as that of FC1 layer with only half size of FC1 layer. Note, we define the size of a layer in terms of number of parameters rather than number of edges because energy consumption to access parameters from DRAM is more than two orders of magnitude of that of performing a MAC operation corresponding to an edge. In other words, the number of parameters, which is related to communication energy, is the dominant factor.

Tradeoff Between Accuracy and Energy for A Single Layer

In this experiment we set the threshold ratio in Algorithm 1 to its maximum value (=100%). So the algorithm stores all configurations obtained from the steps of structural simplification. We then plot the tradeoff curves between (top-5) accuracy and rate of energy saving for the `CaffeNet` network which is the largest network. In Figure 5.4 we show three scatter plots when our algorithm is (independently) applied to the FC1, FC2, and FC3 layers in `CaffeNet`. In each plot, the X-axis reports the percentage of neurons removed locally within the layer to which our algorithm is applied to (ranging from 0 to 100%). The Y-axis shows accuracy and rate of energy saving. These two metrics are reported for the entire network while only one layer is simplified in each plot.

We make the following observations:

(1) For each plot, the total degradation in accuracy remains small for the majority of the iterations, until a clear turning point when accuracy degrades exponentially and becomes 0 when all neurons are removed.

(2) The rate of energy saving however is different across the layers. For FC1 and FC3 the rate is closer to a linear approximation while in FC2 the rate of energy saving has an hyperbolical growth rate after the initial iterations.

(3) If the goal was to pick the minimal energy solution for negligible degradation in accuracy, among the 3 layers, FC1's solution would be found as the best (slightly better than FC2), as reported in the previous experiment in Tables IV and V.

(4) If however, a higher degradation in accuracy is allowed, FC2 allows significantly higher rate of energy saving than FC1. For example for a 10% degradation in accuracy (corresponding to 70% top-5 accuracy level), the rate of energy saving is 3.48X in FC2 while it is 2.15X in FC1, and 1.41X in FC3. Interestingly, this is despite the fact that FC1 is the largest fully-connected layer in `CaffeNet` as can be seen from Table II. Intuitively, the reason that FC2 gives higher rate of energy savings when allowing more degradation in accuracy may be because removal of neurons also results in removing both incoming and outgoing edges connecting to them. For FC2, this means removal of edges in both FC1 and FC2 which are both large-sized fully-connected layers. This unique benefit gets amplified hyperbolically as more neurons are removed from FC2.

(5) Generally speaking, under a given constraint such as energy or accuracy, the optimal result should be searched across the results from different layers across different structure simplification techniques. For example, Fig. 5.5 shows the energy saving ratio achieved by different layers versus classification accuracy. The figure indicates that the layer which gives the maximum energy saving for a given accuracy is initially FC2 and then switches to FC1 after the 77.5% accuracy point.

Overall, from this experiment we observe that not only the degree of energy saving strongly depends on degree of accuracy when considering any individual layer, but also, the choice of the layer which yields the maximum energy savings depends on the accuracy threshold as well. Furthermore, this behavior may indicate the opportunity to perform aggressive pruning on layers that can give us the most energy saving when sacrificing same amount of accuracy, while compensating the lost accuracy by adding neurons in those less energy sensitive layers. We plan to explore this opportunity in

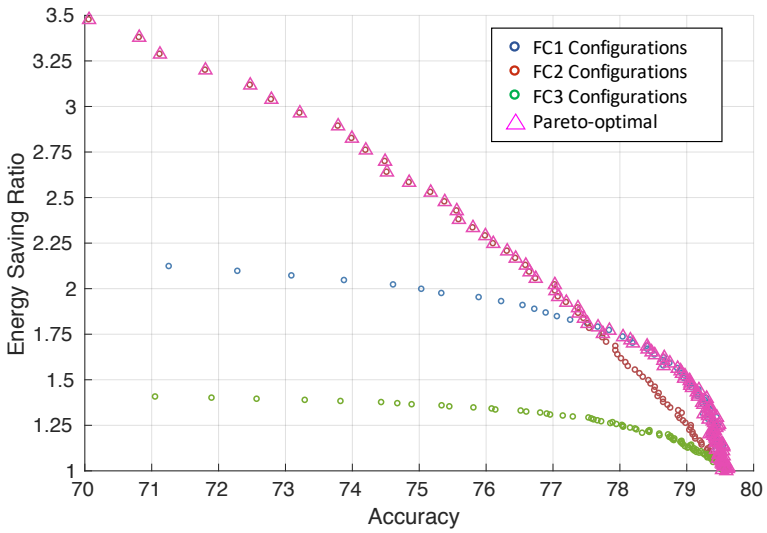

Figure 5.5: Achievable energy saving ratios from different layers versus final classification accuracy. The points with same color correspond to the same layer and the points marked with △ sign are Pareto-optimal of the proposed technique.

our future work.

Comparison when Simplifying Multiple Layers

In this subsection, we show the results of applying our algorithm to two consecutive fully connected (FC) layers in a DNN according to the guidelines given by the three methods introduced in Section 5.1. We show the results of `CaffeNet` since it is our largest DNN. Here Algorithm 1 is first applied to FC1, followed by FC2 and then FC3 for all three methods. Our methods help decide the threshold to apply to each layer so the overall degradation in accuracy between the simplified and original networks is bounded by 2% while aiming to maximize the overall compression ratio.

For our first method, consider Figure 5.6 which shows the improvement in compression ratio versus top-5 accuracy degradation curve for all three FC layers in `CaffeNet`. Recall, at each point in the X-axis, we want to simplify the layer that has the highest slope in its curve, assuming the layers are processed sequentially in the given order. For example, if we are simplifying FC1 and FC2, by inspecting

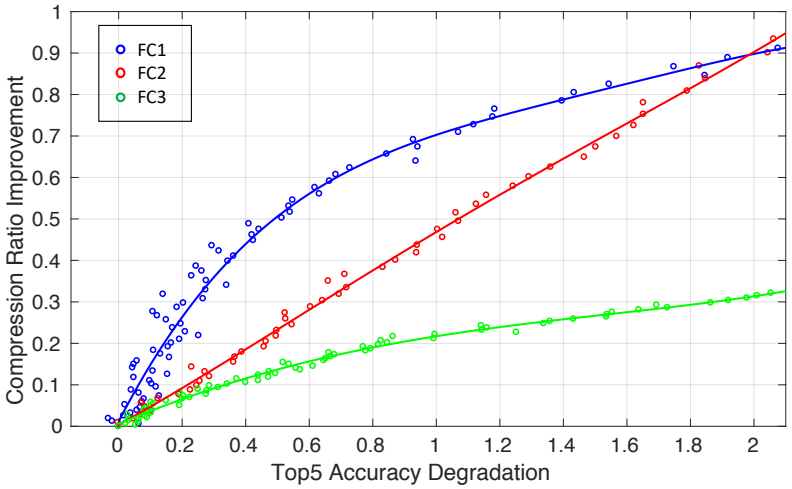

Figure 5.6: Improvement in compression ratio versus degradation in accuracy ϵ (i.e., difference in output accuracy of the original and simplified networks) for each iteration of Algorithm 1 when applied to all FC layers in CaffeNet independently.

their corresponding curves, we identify 0.65% as the turning point where the slope of FC2 becomes greater than the slope of FC1. So we apply the algorithm to FC1 for $\epsilon_1 = 0.65\%$ and fix the simplified FC1, and then we move to FC2 until we reach 2% degradation so $\epsilon_2 = 1.35\%$. By doing this, we are able to have 2.35X overall compression ratio.

When the two considered layers are FC2 and FC3, since the slope of the curve of FC3 is always less than that of FC2, we will only apply the algorithm to the FC2 layer. By doing this, we will get 1.81X overall compression at the end.

For our second method, consider the green, blue, and purple curves in Figure 5.7 which show the results of simplifying FC1 and FC2 layers. The X-axis represents the accuracy degradation threshold for FC1 (i.e., ϵ_1). The Y-axis (blue and green curves) shows the compression ratios when applying Algorithm 1 separately on each layer. The Y-axis also shows the purple curve of our estimate of the combined compression ratios (by multiplying the corresponding compression ratios on the two curves).

The green curve corresponds to FC1 when ϵ_1 varies 0% to 2%. For each value of d between 0% to 2% in the green curve for FC1, the blue curve represents how much compression can be achieved with $(2 - \mathrm{d})\%$ degradation budget in FC2. For

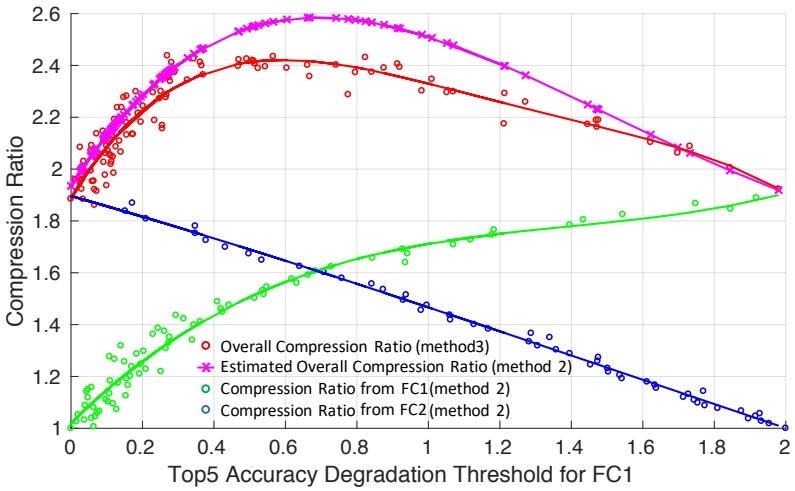

Figure 5.7: Compression ratio versus accuracy degradation used by the second and third strategies when applied to FC1 and FC2

example, when d $= 0$, FC2 has 2% budget, thus it can compress the model 1.81X. When d $= 2$, FC2 has no degradation budget, thus it cannot compress the model at all. The purple curve is generated by multiplying the corresponding compression values of the green and blue curves. The idea is that if we assume the overall degradation equals the summation of the degradation values from the two layers and the overall compression equals the multiplication of the compression ratios from the two layers, then the purple curve will represent the overall compression at various ϵ_1 values. Of course these two assumptions are not true because the dynamics of the second layer will change as the first layer changes. So, the purple curve is only an estimation. We then pick the ϵ_1 corresponding to the highest value in the purple curve, and set it as the degradation threshold for the first layer. By doing so, we achieved 2.38X overall compression.

The red curve in Figure 5.7 shows the **results of our third method**. This curve represents the actual compression ratio that can be achieved at various ϵ_1 values. As we can see, the highest compression (2.43X) is when $\epsilon_1 = 0.56$. Therefore as expected method 3 performs better than method 2 in this example.

Figure 5.8 shows the results of the second and third methods when applying to FC2 and FC3 layers. It can be interpreted in the same way as Figure 5.7.

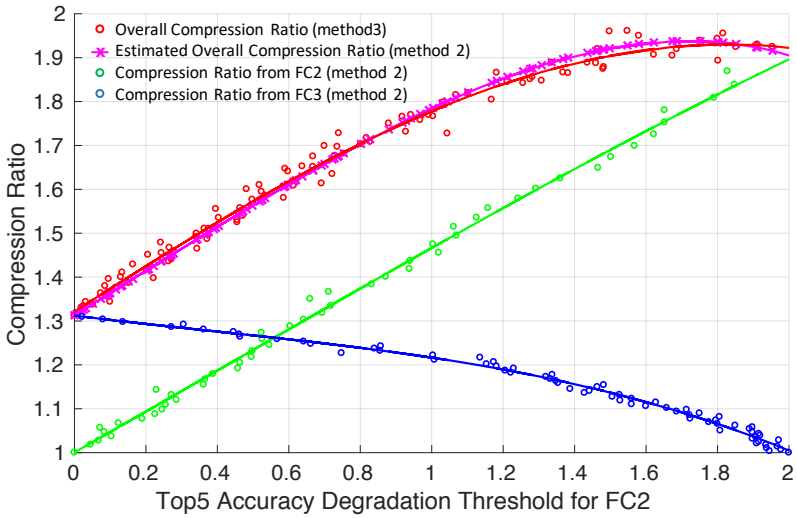

Figure 5.8: Compression ratio versus accuracy degradation used by the second and third strategies when applied to FC2 and FC3

Table VI: Comparison of Compression ratio and runtime of three methods

	FC1_FC2		FC2_FC3	
	Compression	Runtime	Compression	Runtime
Method 1	2.35X	6.82h	1.81X	1.47h
Method 2	2.38X	6.82h	1.91X	1.47h
Method 3	2.43X	241.74h	1.96X	80.53h

Table VI shows the compression ratio and runtime of the three methods for two cases when simplifying FC1 and FC2, and when simplifying FC2 and FC3. The overall accuracy degradation is 2% in all cases. We make the following observations from this table: (1) Both the first and second methods can achieve very good, although sub-optimal, compression ratios. This validates our reasonings behind them and verifies that they both provide very good estimations about how to achieve the highest compression ratio; (2) The runtime of the third method is significantly larger than that of the first two methods. This is due to the fact this method effectively performs a thorough search in the solution space. However we note that this method gives the highest compression; (3) Between the first and second methods, we recommend the

second one because it gives somewhat higher compression with the same runtimes as the first one.

6 STRUCTURE SIMPLIFICATION VIA CHANNEL PRUNING

To overcome the obstacles of deploying DNNs on edge devices, many existing techniques are applicable including model compression [15, 25, 27] and more efficient network architectures such as MobileNet [37] and ShuffleNet [95]. Model pruning techniques can be further divided into unstructured pruning and structured pruning. As a representative unstructured pruning technique, the work [25], pruned weights and neurons whose values were below a certain magnitude and demonstrated very good theoretical compression ratio and speedup results. However, like other unstructured pruning techniques, the effectiveness of pruning depends on the sparsity of weight matrices rather than the network architecture. Thus, even though the static memory issue is greatly alleviated by sparse representation, other issues like huge number of memory accesses and computation are still not resolved. In other words, the benefits of the resulting pruned model cannot be easily harvested without the support of specialized hardware and software.

In contrast, structured pruning aims to prune models in a structured manner, such as pruning sets of neurons, groups of weights, or entire layers. Thus, the resulting pruned models become more efficient in terms of network architecture, which allows the benefits brought by structured pruning techniques to be further utilized by existing hardware and deep learning libraries. In fact, the *neuron elimination* technique presented in Chapter 5 belongs to this category. Its effectiveness on fully-connected layers makes it well suited for models like multilayer perceptron and recurrent neural networks [34]. However, due to the heavy reuse of kernel weights, its performance is very limited on convolutional layers. On the other hand, as the key model for many vision related tasks, CNNs are designed to have fewer fully-connected layers and more convolutional layers [29, 64], which is the most energy consuming [7] and computation heavy layer type. Thus, being able to perform structured pruning on convolutional layers is very important in order to deploy CNNs on edge devices.

In this chapter, we propose a structured pruning technique that identifies and prunes redundant input channels of convolutional layers. After a channel is pruned, its corresponding kernel weights of both the current layer and the previous layer will be pruned as well, thus achieving significant reduction in terms of static and run-time

memory, computation, and energy consumption. The proposed procedure investigates the intermediate results of convolutional layers and formulate the task of identifying redundant channels as a subset selection problem, in which a subset of input channels of specified number is selected to maximally preserve the original outputs of the target layer and the other channels are identified as redundant. This NP-hard problem is approximately solved using pivoted QR factorization algorithm. We also propose two techniques to explore more pruning opportunities in ResNet-like models. Moreover, the proposed technique is orthogonal to others such as quantization and low-rank expansion, which can be combined to achieve further reduction.

To validate the effectiveness of our procedure, we survey many prior works and conduct experiments on the most two popular models (VGG-16 and ResNet-50) and dataset (ImageNet 2012). The experimental results show our proposed technique is able to reduce the computation requirement by 4.29X (VGG-16) and 2.84X (ResNet-50) while only sacrificing about 1.40% top-5 and 2.50% top-1 accuracies. Compared with many prior works, our results are much better in terms of both computation reduction and accuracies.

The remaining of this chapter is organized as follows: we introduce the notations that will be used throughout this chapter in Section 6.1, and then present our channel pruning algorithm along with the techniques for ResNet-like models in Section 6.1. We show experimental results and the corresponding analysis in Section 6.2.

6.1 Notations and Our Algorithm

In this section, we first introduce the basics and notations of channel pruning, then go through the details of our proposed technique.

Basics and Notations of Channel Pruning

Fig. 6.1 shows two convolutional layers and their corresponding input and output tensors. For the sake of simple illustration, we assume batch size equals to 1 and thus ignore the dimension of batch size in the following discussion. The input to convolutional layer l is a rank-3 tensor $\mathbf{I}_l \in \mathbf{R}^{H_l^i \times W_l^i \times C_l^i}$. The output rank-3 tensor $\mathbf{O}_l \in \mathbf{R}^{H_l^o \times W_l^o \times C_l^o}$ is obtained by performing convolution operations between the

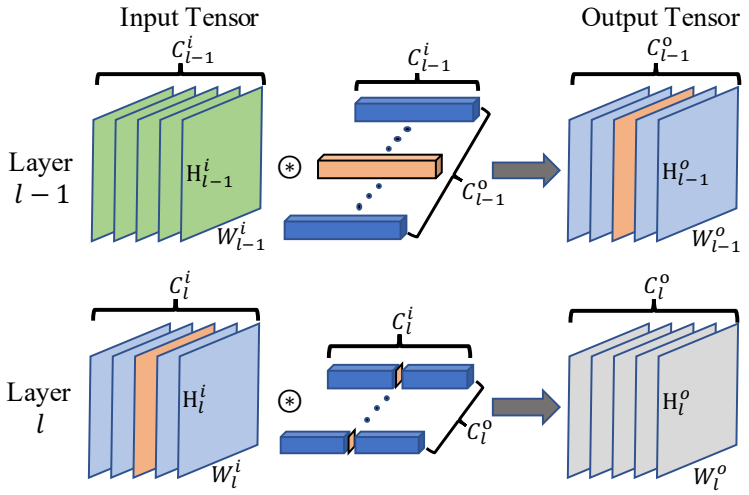

Figure 6.1: Diagram of convolutional layers in typical CNN models. The redundant channels and their corresponding weights are shown as orange slices and blocks, respectively.

input tensor and layer l's kernel $\mathbf{M}_l \in \mathbf{R}^{K_l \times K_l \times C^i_l \times C^o_l}$. Where, $H^{i/o}_l, W^{i/o}_l$, and $C^{i/o}_l$ are the height, width, and number of channels of the input/output tensors of layer l, respectively, and K_l is the spatial dimension of layer l's kernel. Then, $\mathbf{O}_l$ is subject to activation function and possibly max/average pooling to form the input tensor for the next layer. For each layer, the goal of channel pruning is to identify and prune redundant channels (among C^i_l channels) of the input tensor $\mathbf{I}_l$ such that the overall accuracy does not degrade too much after pruning. Once the redundant channels are pruned, their corresponding weights in both $\mathbf{M}_l$ and $\mathbf{M}_{l-1}$ can be pruned as well. For example, if we can identify and prune m channels in $\mathbf{I}_l$, then the resulting kernels of layers l and $l-1$ will become $\mathbf{M}_l \in \mathbf{R}^{K_l \times K_l \times (C^i_l - m) \times C^o_l}$ and $\mathbf{M}_{l-1} \in \mathbf{R}^{K_{l-1} \times K_{l-1} \times C^i_{l-1} \times (C^o_{l-1} - m)}$.[1]

Although the ultimate goal is to maximally preserve the final output tensor during pruning, we propose to address this problem with a layer-by-layer approach. Thus, for each layer, the core of channel pruning is to identify and prune redundant channels of $\mathbf{I}_l$ and modify the remaining kernel weights, such that the output tensor $\mathbf{O}^{pruned}_l$ obtained by using only the remaining input channels $\mathbf{I}^{pruned}_l$ and modified weights

[1] Usually $C^o_{l-1} = C^i_l$.

$\mathbf{M}_l^{pruned}$ can maximally approximate the original output tensor $\mathbf{O}_l$. Mathematically, how well a tensor is approximated by another tensor can be described by the Frobenius norm of their difference. Thus, channel pruning can be formulated as:

$$\begin{aligned} min \quad & \|\mathbf{O}_l - \mathbf{O}_l^{pruned}\|_{\mathbf{F}} \\ where \quad & \mathbf{O}_l^{pruned} = \mathbf{I}_l^{pruned} \circledast \mathbf{M}_l^{pruned} \\ s.t. \quad & \mathbf{I}_l^{pruned} \subseteq \mathbf{I}_l \\ & \mathbf{I}_l^{pruned} \in \mathbf{R}^{H_l^i \times W_l^i \times (C_l^i - m)} \end{aligned}$$

$m(< C_l^i)$ is the number of channels to be pruned, and $\circledast$ denotes convolution operation.

Our Proposed Technique

To solve the aforementioned problem, we investigate the linearity between the intermediate results of convolution operations and then identify redundant channels as those which can be approximated by linear combinations of the other channels. By considering the intermediate results instead of the input tensor $\mathbf{I}_l$, we actually exploit the redundancy in both input tensor $\mathbf{I}_l$ and kernel $\mathbf{M}_l$ together. Specifically, for an input image, an element o_j of the j^{th} channel of $\mathbf{O}_l$ is obtained by:

$$o_j = \sum_{c=1}^{C_l^i} \sum_{k_h=1}^{K_l} \sum_{k_w=1}^{K_l} \mathbf{I}_{l(k_h,k_w,c)} \times \mathbf{M}_{l(k_h,k_w,c,j)} \tag{6.1}$$

Note that the bias term is ignored for clarity and does not affect the correctness of the algorithm. Denoting the contribution of o_j by input channel c as o_j^c, as shown in Fig. 6.2, we have:

$$o_j = sum(\overrightarrow{o_j^c}) \tag{6.2}$$

$$\overrightarrow{o_j^c} = [o_j^1, o_j^2, ..., o_j^{C_l^i}]^T \tag{6.3}$$

$$o_j^c = \sum_{k_h=1}^{K_l} \sum_{k_w=1}^{K_l} \mathbf{I}_{l(k_h,k_w,c)} \times \mathbf{M}_{l(k_h,k_w,c,j)} \tag{6.4}$$

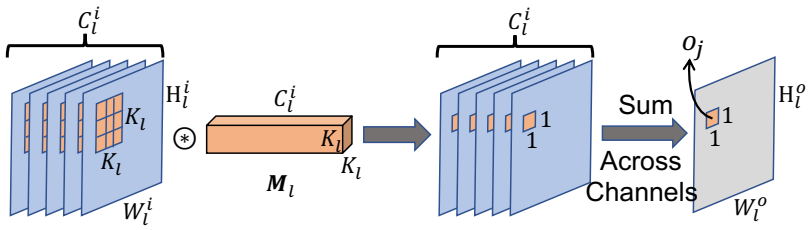

Figure 6.2: Diagram of obtaining an output channel by performing convolution operation between its kernel and the input tensor.

Considering the $\overrightarrow{o_j^c}$ vector, the idea is that if we can identify a subset of o_j^cs of size m from $\overrightarrow{o_j^c}$, such that each element in this subset can be approximated by linear combination of the o_j^cs outside of this subset, then these elements can be safely pruned and the original o_j can be well approximated by the summation of the scaled remaining o_j^cs. Thus, if we only care about approximating the o_j element, the aforementioned idea can be translated into pruning the input feature maps and kernel weights (of both current and previous layers) corresponding to those m elements, and then scaling the remaining kernel weights of current layer.

However, this is only for one element in one output channel with respect to one input image. In order to make this idea work for an entire convolutional layer, we need to consider $\overrightarrow{o_j^c}$ with three more dimensions:

1. Consider $\overrightarrow{o_j^c}$ across all elements of the j^{th} output channel, thus the redundancy is identified for the entire j^{th} channel of the given input image.

2. Consider $\overrightarrow{o_j^c}$ across all training samples, thus the redundancy is identified not only for one given image, but the entire training set.

3. Consider $\overrightarrow{o_j^c}$ across all output channels in $\mathbf{O}_l$, thus the redundancy is identified for all output channels.

With all of the three dimensions being considered, the column vector $\overrightarrow{o_j^c}$ is extended to a matrix $\mathbf{A} \in \mathbf{R}^{C_l^i \times N}$, which is formed by concatenating $\overrightarrow{o_j^c}$ over N samples along the column direction. Ideally, N is the number of $\overrightarrow{o_j^c}$s that can be collected from all $\mathbf{O}_l$s across all training images, which means $N = \#\ training\ images \times \#\ output\ feature\ maps \times size\ of\ each\ feature\ map$. But that is too big to be

stored and processed. So, in practice, we perform random sampling to collect enough $\overrightarrow{o_j^c}$s from different images, output channels, and spatial locations. The exact number of samples used in our experiments will be explained in Section 6.2. Since each row in $\mathbf{A}$ corresponds to an input channel, the problem becomes identifying m redundant rows from $\mathbf{A}$ with the goal of best approximating the matrix $\mathbf{B} \in \mathbf{R}^{1\times N}$ by linearly combining the remaining rows, where $\mathbf{B}$ is formed by extending o_j over the samples.

Algorithm 2 is adopted to approximately and efficiently solve the inverse of this NP-hard subset selection problem. Here, instead of identifying the most p representative rows in $\mathbf{X}$, our goal is to identify the most $C_i^i - m$ representative rows in $\mathbf{A}$, by pivoted QR factorization. Let's denote the remaining matrix after pruning the redundant rows as $\mathbf{A}_{pruned}$. To best approximate the matrix $\mathbf{B}$, the optimal scaling factors for the remaining rows, and thus their corresponding input channels, can be obtained by $\mathbf{B}\mathbf{A}^{\dagger}_{pruned}$, where $\mathbf{A}^{\dagger}_{pruned}$ is the pseudo-inverse of $\mathbf{A}_{pruned}$. For current layer, the input channels and kernel weights corresponding to those pruned rows can be eliminated, and the effect of scaling the remaining channels can be achieved by scaling the corresponding kernel weights, which effectively generates the new kernel weights for the current layer. For the previous layer, since its output channels corresponding to those pruned rows are disregarded in the current layer, it does not need to generate those channels any more regardless of its activation function. So, the corresponding kernel weights of the previous layer can be pruned and no further adjustments are needed.

Handling ResNet Architecture

Fig. 6.3 shows the first two bottleneck units in ResNet-50 model, which is designed for ImageNet image classification task. Besides the typical pipelined convolutional layers in classic CNN models, it also has a shortcut path in each unit connecting the input and output tensors of each unit. The shortcut path in $unit_1$ is a projection path, which consists of a convolutional layer that projects 64 input channels to 256 output channels with the same spatial size. The shortcut path in $unit_2$ is an identity path, which does nothing more than feed forward the input tensor to the output tensor. In prior works [31, 61, 65] that have discussed the technical details about how ResNet's bottleneck unit is handled, only the input tensors of layer `conv2` and `conv3` are pruned, while

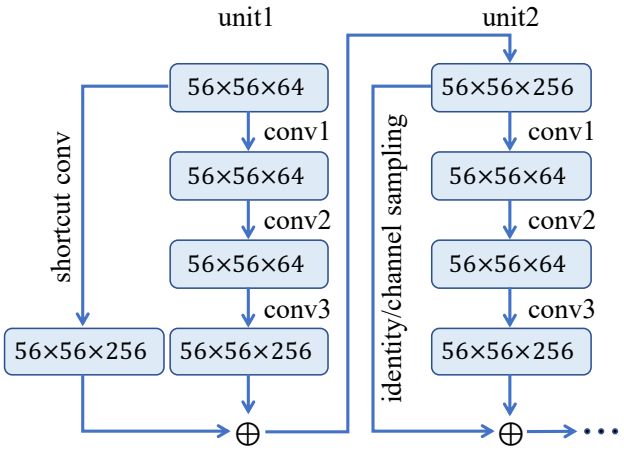

Figure 6.3: Diagram of the first two bottleneck units in ResNet-50 model for ImageNet classification task.

the input tensor of layer `conv1` is intact. This limitation is due to two reasons: 1) for bottleneck units with projection path, they were not able to analyze and prune layer `shortcut conv` and `conv1` simultaneously; and 2) for bottleneck units with identity path, pruning the input tensor of layer `conv1` may cause misalignment problem when merging it with the output tensor of layer `conv3`.

To overcome the above limitation and explore more pruning opportunities, we apply two tweaks to our technique to successfully prune all layers in ResNet architecture. First, for units have projection shortcut path, the redundant channels need to be identified for both layer `shortcut conv` and layer `conv1` at the same time. To handle this, when extending vector $\overrightarrow{o_j^c}$ to matrix $\mathbf{A}$ (refer to Sec. 6.1 for details), we sample o_j from the output tensors of both layer `shortcut conv` and layer `conv1`. Thus, matrix $\mathbf{A}$ has the information from both layers and the redundant input channels are identified for both layers simultaneously.

Secondly, for units which have identity shortcut path, the issue is that after pruning layer `conv1`, the resulting input tensor may not be able to merge with the output tensor of layer `conv3` due to misalignment between channels. Moreover, some channels that may *not* be identified as redundant by the next bottleneck unit can potentially be identified as redundant by the `conv1` layer of the current unit and

be pruned out, which is undesired. To handle this, we propose to prune layers in backward direction for the entire ResNet model, and propagate the set of indices of remaining input channels identified by the next unit back to the current unit. Later on, when pruning the `conv1` layer of the current unit, another set of indices of remaining input channels will be identified, and the union of these two sets will finally be used as the remaining channels for the `conv1` layer and propagated back to the previous unit. Since the final remaining input channels for layer `conv1` is a superset of the channels that should be passed through the identity shortcut path, we perform *channel sampling* to select the channels needed by the next unit and only pass them to the next unit.

6.2 Simulation Results

Experiment Setup

The computer used to run the experiments has one Intel i7-8700K CPU, 16GB RAM, and one Nvidia GTX 1080Ti GPU. We used 30K samples when generating the matrix $\mathbf{A}$ mentioned in Section 6.1 and this number was determined by experiments. We first tried 20K samples and 30K samples, and observed that the results of 30K samples were little bit better than those of 20K samples but not significant. Then we also tried 40K samples, which crashed the program due to memory error when performing SVD using *Numpy* package. At the same time, 30K samples was not the computation bottleneck of the experiments so we did not have to bother with finding the smallest number of samples for the problem. So, for each target layer, the experiment was conducted with 30K samples, which are randomly selected across various spatial locations, output channels, and training images. Note that, since the sampling process was performed randomly, roughly equal number of training images are drawn from each class.

As for the CNN models tested in the experiments, after examining many prior works, we found that the most popular experimented models are VGG-16 [77] and ResNet-50 [29], which are designed for and trained with the ImageNet ILSVRC 2012 dataset. So we performed our experiments with these models and dataset in order to have a fair comparison with others. VGG-16 is a classic pipelined CNN model with

16 layers, which are divided into 5 convolution groups with each group has 2 or 3 convolutional layers and 3 fully connected layers. ResNet-50 mainly consists of 4 bottleneck blocks with each of them having 3 to 6 bottleneck units. Each bottleneck unit has 3 convolutional layers and one *identity/projection* shortcut path depending on whether this unit is the first unit in its block. The original models are downloaded from TensorFlow's official website. ImageNet is a large-scale image dataset that contains 1.2M training images and 50K validation images at various resolutions of 1000 classes. Standard data augmentation schemes such as resizing, random cropping, and horizontal flipping are used when fine-tuning each model. All accuracies are measured by using the single-view test approach (central crop only) on the validation set.

Sensitivity Analysis of Layers

In order to verify the effectiveness of the proposed technique as well as to determine to what extent each layer should be pruned when pruning the entire model, we performed sensitivity analysis by applying our technique to each layer independently for all layers in both VGG-16 and ResNet-50 models and tested how accuracies are impacted.

Fig. 6.4 shows how the top-1/5 accuracies change when various percentages of input channels are pruned for several convolutional layers in VGG-16. Here, we only show the sensitivity curves for 5 layers, which are the second convolutional layer of all 5 convolution groups. Since these layers have different number of input channels, the x-axis is normalized to show the percentage of channels being pruned instead of the absolute numbers. We make two observations from this figure: 1) Some layers, especially layer `conv1_2` and `conv2_2`, experience abnormal drops in accuracy (circled in red) as their input channels are pruned. However, pruning fewer channels should always generate at least equally good results compared to pruning more channels because we can simply set the weights of those extra channels to zero and obtain the same accuracies. Thus, these glitches are caused by the built-in randomness in the sampling procedure (refer to Sec. 6.1 for details) of our algorithm, and they can be eliminated by either averaging the results from multiple runs or sampling more data points; 2) Comparing the curves between layers, it is obvious that early layers are more robust than later layers when the same percentage of input

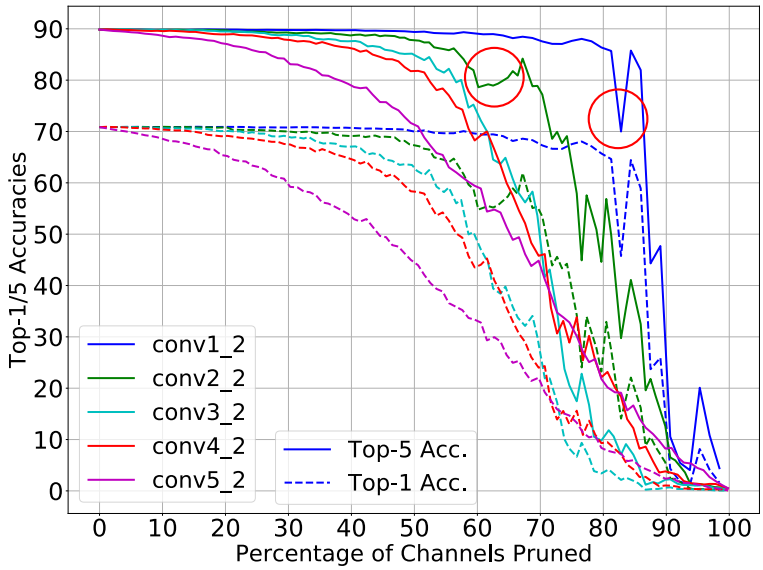

Figure 6.4: The sensitivity curves of the second convolutional layers of all 5 convolution groups in VGG-16 model under pruning.

channels is pruned. This behavior is expected for typical CNN models like VGG-16, because early layers are responsible for extracting low-level features such as edges, colors, and corners of various orientations, and these features can easily be approximated by linear combinations of others. In contrast, features extracted by later layers are more complex and closely related to specific classes, thus it is harder to approximate the pruned channels with the remaining channels in these layers.

Fig. 6.5 shows the sensitivity curves of 6 convolutional layers in ResNet-50 model. Here, we choose to show 6 representative curves from two bottleneck units, which are `unit_2` from `block1` and `unit_1` from `block4`. From the figure, we make the following observations:

First, it is obvious that the sensitivity curves of the `conv1` layers of `block1/unit_2` and `block4/unit_1` are dramatically different; the curve from `block4` is much more sensitive than that from `block1`. This difference is partially due to the reason we mentioned above, but more importantly, it is because we handle the identity and projection shortcut paths differently. Since `unit_1` is the first bottleneck unit in `block4`, it has a projection path, which is pruned simultaneously with layer `conv1`.

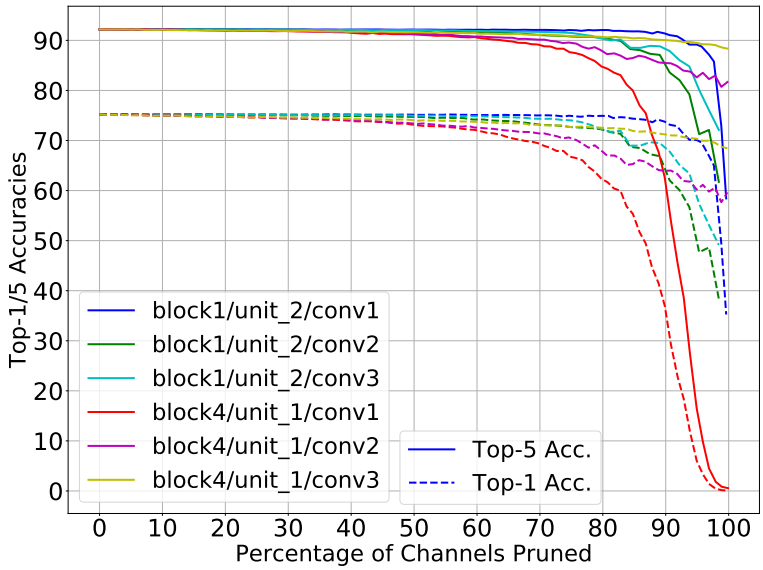

Figure 6.5: The sensitivity curves of the convolutional layers of two bottleneck units in ResNet-50 model under pruning.

While `unit_2` is the second bottleneck unit in `block1`, it has an identity path, which is intact to avoid the misalignment issue when pruning the `conv1` layer. So it can still pass information to later layers. Therefore, when almost all input channels are pruned, accuracies drop to 0 for layer `block4/unit_1/conv1` but remain relatively high for layer `block1/unit_2/conv1`.

Secondly, when comparing the curves of layers (except the `conv1` layer of the first bottleneck unit in each block) within the same bottleneck unit, we observe that layer `conv2` is almost always more sensitive than the others. This is because only `conv2`'s kernel size is 3×3 and all the others are 1×1. Thus, when scaling the original kernel weights to get the new weights during pruning, the adjustments applied to `conv2`'s weights are more coarse-grained than those to the other layers (refer to Sec. 6.1 for details). This makes it harder to accurately approximate the pruned channels for `conv2` layers, which in turn makes these layers more sensitive. This phenomenon is not observed from the curves of VGG-16 because all of its convolutional layers have 3×3 kernel.

Finally, comparing the curves between VGG-16 and ResNet-50 models, we can

see that the curves from ResNet-50 are much more robust than those from VGG-16 in general. This is because a ResNet architecture with `i` bottleneck units actually has 2^i different paths from the input to output, which can be seen clearly by unrolling the network [83]. On the other hand, typical CNN models like VGG-16 only have one effective path. Thus, pruning one single layer (except the `conv1` layer of the first unit in each block) of ResNet-50 only affects a subset of its paths, while the remaining unaffected paths can still contribute to high accuracies. However, pruning any layer in VGG-16 or any `conv1` layer of the first unit in any block in ResNet-50 affects all paths, thus explaining why accuracies can drop to 0 when pruning these layers.

Results of Pruning the Entire Model

Based on the sensitivity analysis of the layers, we set the pruning target for each layer accordingly and prune the entire model. Specifically, for layers that are robust to pruning, we prune 50% to 70% of their input channels, and not pruning more even if accuracies are still high. This is because the sensitivity analysis is performed independently for each layer, and how previous layers are pruned affects the behaviors of later layers. So, instead of calculating the comprehensive interaction of sensitivity between layers (which is impractical due to the amount of computation required), we simply chose to leave some margin for later layers. For layers that are sensitive to pruning, we prune up to 50% of their input channels so that the accuracies do not degrade much. Like many prior works, after pruning each layer, we fine tune the pruned model to recover the lost accuracies. Table VII shows the comparison of top-1/5 accuracies, number of FLOPs required per inference, average inference time of a minibatch of images, and *Overall Score* for the original and various pruned VGG-16 and ResNet-50 models. Because the pruned models have different computation requirements and top-1/5 accuracies, sometimes it may not be obvious to judge if one pruned model is better than the other when comparing two models. For example, one pruned model may require higher number of computation while achieving better accuracies. Thus, to directly compare these pruning techniques, we define a unified metric called *Overall Score* that incorporates both computation and accuracy metrics. Since more computation reduction with less accuracy degradation is desired, for each

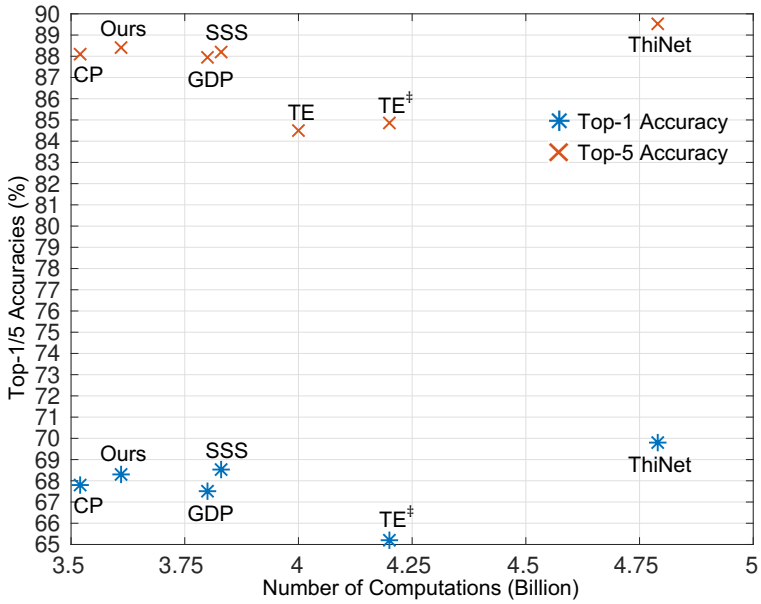

Figure 6.6: Accuracy versus computation requirement for various pruned VGG-16 models.

pruned model, the overall score is defined as:

$$\begin{aligned} \text{Overall Score} = & \frac{\#\text{ FLOPs}_{original} - \#\text{ FLOPs}_{pruned}}{\#\text{ FLOPs}_{original}} * 100 \\ & - \frac{\text{Top1 Acc}_{original} - \text{Top1 Acc}_{pruned}}{\text{Top1 Acc}_{original}} * 100 \\ & - \frac{\text{Top5 Acc}_{original} - \text{Top5 Acc}_{pruned}}{\text{Top5 Acc}_{original}} * 100 \end{aligned}$$

So, higher score is better. We also note that the *Overall Score* is only comparable between the pruned models of the same original model, but not across different original models.

For pruned VGG-16 models shown at the top part of Table VII, Channel Pruning (CP)[31] only reported speedup values instead of the absolute number of FLOPs required per inference, so we calculated it based on their released source code. Taylor Expansion-based pruning (TE)[68] only reported top-5 accuracy of the pruned model in their original paper, so we further included the numbers reported by[61]

Table VII: Comparison of top-1/5 accuracies, # FLOPs, average inference time, and overall score of the original and various pruned VGG-16 and ResNet-50 models on ImageNet ILSVRC 2012 dataset

	Model Name	Top-1 Accuracy	Top-5 Accuracy	# FLOPs	Overall Score	Inference Time
VGG-16	Original	70.85%	89.85%	15.47B	-	208.25ms
	ThiNet[65]	69.80%	89.53%	4.79B	67.20	-
	GDP[61]	67.51%	87.95%	3.80B	68.61	-
	CP[31]	67.80%	88.10%	3.52B	70.99	-
	TE[68]	-	84.50%	4.00B	-	-
	TE‡[68] ([61])	65.20%	84.86%	4.20B	59.32	-
	SSS[41]	68.53%	88.20%	3.83B	70.13	-
	Ours	68.30%	88.41%	3.61B	71.46	111.43ms
ResNet-50	Original	75.22%	92.20%	3.86B	-	105.90ms
	ThiNet[65]	71.01%	90.02%	1.71B	47.74	-
	GDP[61]	70.93%	90.14%	1.57B	51.39	-
	CP[31]	72.30%	90.80%	2.60B	27.24	-
	SFP[30]	74.61%	92.06%	2.25B	40.75	-
	PF‡[57] ([41])	72.88%	91.05%	1.54B	55.75	-
	PF‡[57] ([41])	72.98%	91.08%	1.70B	51.77	-
	Ours	72.74%	90.88%	1.36B	60.04	67.49ms

X‡[Y]([Z]) indicates the data point is from reference *Z*, which implements the technique *X* that was originally introduced by reference *Y*.

for comprehensive comparison (the data source is shown in parentheses). Figure 6.6 plots the data points of all pruned models for better illustration. If a model is at the lower-right(upper-left) region of another model, it is clearly worse(better) than the other because it requires more(less) computation while achieving lower(higher) accuracy. However, it is not obvious to judge which one is better if one model is at lower-left or upper-right region of the other, so the *Overall Score* can be used for comparison in this case. Compared to the pruned model generated by ThiNet, ours requires 24.63% less computation while sacrificing 1.50% (top-1) and 1.12% (top-5) accuracies. Compared to Sparse Structure Selection (SSS)[41], our pruned model requires 5.74% less computation while providing slightly higher top-5 accuracy and

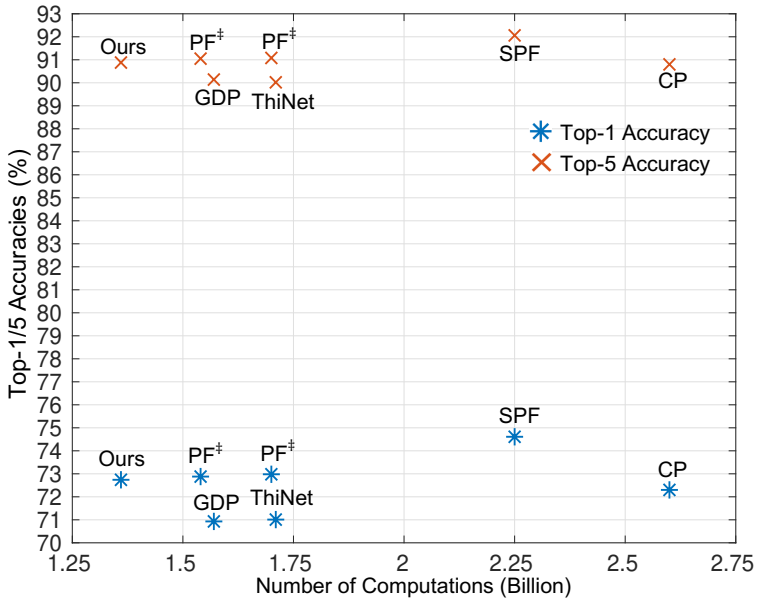

Figure 6.7: Accuracy versus computation requirement for various pruned ResNet-50 models.

slightly lower top-1 accuracy. In practice, among these models, one might be chosen over the other depending on the target platform, accuracies, and runtime specification. For all the other techniques, our pruned model always achieves significantly higher top-1/5 accuracies with less computation requirement (except for CP, which is only 2.49% lower than ours). Overall, our technique has the highest *Overall Score* among all compared techniques.

The statistics of pruned ResNet-50 models are shown at the bottom part of Table VII. The data of the pruned models generated by CP and Pruning Filters (PF)[57] are calculated based on the released source code or borrowed from[41] due to similar reasons mentioned above. These data points are plotted in Figure 6.7 for better illustration. Compared to ThiNet, GDP, and CP, our pruned model requires up to 47.69% less computation while still achieving significantly higher top-1/5 accuracies. This is mainly because our proposed technique for handling ResNet-like architectures can explore more pruning opportunities than those prior works. When compared with Soft Filter Pruning (SFP)[30] and PF, our pruned model still requires 11.69%

to 39.56% less computation but at the cost of sacrificing 0.14% to 1.87% top-1/5 accuracies. These four pruned models (and potentially with many others) together form the tradeoff between computation requirement and model performance, and the choice of the optimal model to be deployed depends on the specific requirements. In terms of the *Overall Score*, our technique still achieves the highest score among all compared techniques.

The last column of Table VII shows the average actual inference times of a minibatch of 64 input images. Because different GPUs and libraries were used in prior works, and TensorFlow version of their pruned models cannot be easily obtained, we only report the inference time of the original and our pruned models for a fair comparison. All inference times are measured with one Nvidia GTX 1080Ti GPU and averaged over the entire validation set. Compared with the theoretical 4.29X and 2.84X computation reductions, the actual speedup achieved on our GPU are only 1.87X and 1.57X, for pruned VGG-16 and ResNet-50 models, respectively. Since we observed that both CPU and PCIe interface are far from being fully utilized, we believe the gap between the theoretical and actual speedup values is mainly caused by computation parallelism, cache effect, and memory accessing pattern in GPU, which is in turn impacted by the hardware itself, network architecture, and TensorFlow library implementation. Furthermore, we anticipate that mobile and embedded systems may benefit more from the pruned models and such gap may be narrowed when performing inference on them due to their limited memory resources compared to powerful GPUs.

7 LOW-COST AND LOCAL CUSTOMIZATION OF DNNS

The use of Deep Neural Networks (DNNs) allows efficient embodiment of intelligence into emerging application domains. Often, an intelligent assistant such as Siri or Alexa may be equipped with sensory devices (cameras, microphones) to capture and transmit sensory data to the cloud to be processed by a backend DNN model for cognitive tasks. While a cloud-tethered model may provide acceptable performance for occasional use, it may be insufficient to provide satisfactory performance when presented with user-specific data (e.g., accent in speech). Moreover, the cloud-tethered architecture may raise privacy concerns when dealing with highly private user data, suffer from often unpredictable remote server status, network communication latency, and even fail when network connection is unavailable, while users may even refuse to upload personal data to cloud.

An alternative approach would be to leverage edge computing that utilizes local resources to perform DNN inference, and thus resolve the above-mentioned privacy and network connection issues. However, as mentioned in previous chapters, performing inference of modern DNNs often requires considerable amount of resources in terms of computation, memory, and energy. Such requirements make it difficult and expensive to deploy DNN models in resource constrained environment, such as embedded and mobile platforms. Even though many techniques [8] have been proposed to compress trained DNNs so that their inference may be accomplished on stand-alone edge devices, these techniques alone may not be sufficient to make edge devices be able to modify DNNs' structures or parameters to achieve user customization, because training requires much more resources than inference. Moreover, we envision that such trained (and compressed) DNN may be provided by vendors in the form of intellectual property and cannot be modified by the user. On the other hand, even if users agree to share their personal data with vendors, centralizing all users' data, training and deploying a customized model for each user may be unaffordable when the number of users scales up.

Given the above challenges, local learning with user-specific data on edge devices may be utilized as a viable option. First, since the size of user-specific data is remarkably smaller than that of generic data, the structure of a local learning model

may be significantly simpler than a general DNN model. So, local learning only imposes minimal implementation overhead to edge devices. Second, it can improve system performance and enhance user experience by customizing the behavior of intelligent assistants to correct the mistakes made by the general DNN model.

In this chapter, to achieve on-device user customization, we introduce a novel architecture for local learning, namely Mixture of Experts (MoE), to customize a trained DNN that is deployed on an edge device with performance improvement and low implementation overhead. In the MoE architecture, the DNN trained with generic dataset is viewed as a Global Expert (GE). We then use a small DNN as a Local Expert (LE), which is trained with small-sized customized (i.e., user-specific) training data to learn from user's input so that the system doesn't repeat the same mistake. The third component of the MoE architecture is a Gating Network (GN), also implemented as a small DNN, which determines whether the incoming data should be handled by GE or LE. Thus, improving system performance on customized data while preserving its performance on generic data. To minimize the associated implementation overhead, both LE and GN are designed to be very small and a novel techniques based on *structure sharing* is proposed. Finally, we demonstrate the effectiveness of the proposed MoE architecture with the task of recognizing custom handwritten digits and characters.

In the remainder of this chapter, we first discuss the details of the MoE architecture in Section 7.1. Then the datasets and training procedures used in our experiments are presented in Section 7.2. Finally, Section 7.3 shows our experimental results and the comparison with other related techniques in detail.

7.1 The MoE Architecture

A block diagram of the proposed MoE architecture is shown in Fig. 7.1. It consists of a Global Expert (GE), one or more Local Experts (LEs), and a Gating Network (GN), which provides data-dependent weight signals (w_i) to combine the outputs (o_i) of GE and LEs. In the case of multiple LEs, each one may represent one class of users. Ideally, if the GE provides a correct result with respect to a given input, the GN will just pass through the GE's output as the final output and suppress the LE's output. If an LE is more likely to provide a correct result, then the output of the GE

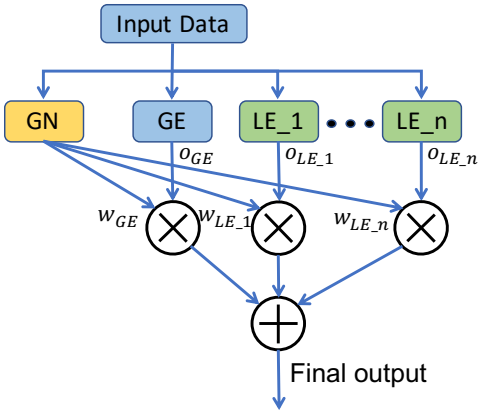

Figure 7.1: Diagram of a general MoE architecture

(and other LEs) will be suppressed. Therefore, the GN plays a central role in the MoE architecture. The details of these three components are discussed below. For simplicity, we only discuss the case with one LE but the discussions can be easily extended to multiple LEs.

Details of the Three Components in MoE

Global Expert: We envision there will be an IP market for trained DNNs on very large generic datasets. These carefully designed, trained, and potentially compressed DNNs may be licensed by vendors to be incorporated into edge devices to facilitate intelligent cognitive services, such as speech and face recognition. Many design and compression techniques such as knowledge distillation [33] and weight pruning [8] may be adopted by vendors to develop and fit DNN models into the target edge devices. But these techniques are orthogonal to the proposed MoE architecture, thus we do not include detailed discussion of those techniques in this chapter. These trained large DNNs will be designated as the Global Expert (GE). Their weight values and internal structures are proprietary information, which cannot be revealed to or modified by individual users. Therefore, given an input data, a GE will provide its output and, depending on the specific licence, intermediate neural activations (e.g., extracted features) to the user, but it cannot be retrained. On the other hand, the limited energy and computation budgets of edge devices may not be capable of training these large

and complicated DNNs.

Since the GE is trained with generic dataset, it is expected to make more mistakes on customized data. The users may then opt to provide these mistakenly classified customized data back to the vendor. As such, the vendor may further design and train the GE to improve its performance. Then the updated GE is released to users as a firmware update. However, such an update is expected to be a rare event and can be disabled if the communication with cloud is unavailable.

Local Expert: A Local Expert (LE) is a local trainable DNN residing within edge devices. It will be trained on a small-sized local customized dataset with the purpose of handling user-specific data. The customized training dataset is collected by the system during daily use. Since the size of the customized dataset is much smaller than that of the generic dataset, LE is designed to have much smaller and simpler structure compared to that of the GE. Techniques for designing compact DNN models can be used to achieve this goal, and we will present ours in the next subsection. This lightweight structure incurs minimal implementation overhead, which is critical for deploying the entire system on embedded and mobile platforms. Moreover, because both the customized dataset and the LE are small, the training of LE can be carried out by the edge device itself.

Gating Network: In its most general form, the Gating Network (GN) provides two outputs: w_{GE}, and w_{LE} such that $0 \leqslant w_{\mathrm{GE}}, w_{\mathrm{LE}} \leqslant 1$ and $w_{\mathrm{GE}} + w_{\mathrm{LE}} = 1$. The final output of the system is a linear combination of the outputs of all experts (i.e., o_{GE} and o_{LE}) weighted by their corresponding weights. The GN acts as a mediator between the GE and the LE: If it is likely that GE will provide a correct answer, then $w_{\mathrm{G}} >> w_{\mathrm{L}}$, meaning the GE's output will be selected as the final output. On the other hand, if it is more likely that the LE's output is correct, then $w_{\mathrm{G}} << w_{\mathrm{L}}$ meaning the LE's output will be the final output. However, in such configuration, all experts and GN are strongly coupled together because the system error is measured against the combined output.

In this work, we demonstrate a case study based on a specialization of the above GN description where GN outputs binary values, formally, $w_{\mathrm{GE}}, w_{\mathrm{LE}} \in \{0, 1\}$ and $w_{\mathrm{GE}} + w_{\mathrm{LE}} = 1$. Therefore, the GN's task is to strictly select the output of either GE or LE to be the final output and suppress the other. In this sense, the GN behaves like an intelligent multiplexer, multiplexing the outputs of GE and LE(s) according to the

predicted likelihoods of their output to be correct. Under such configuration, during training, GN is trained with a mixture of customized instances and equal number of generic instances (randomly sampled from the generic dataset). Since the customized dataset is much smaller than that of the generic dataset, only a tiny portion of the generic dataset is selected. The customized data instances share a common label of $[w_{\mathsf{GE}} = 0, w_{\mathsf{LE}} = 1]$, and the sampled generic data instances have a common label $[w_{\mathsf{GE}} = 1, w_{\mathsf{LE}} = 0]$. During inference, such configuration can be explored to reduce energy and computation cost by first performing the inference of GN and then selectively executes GE or LE(s) according to the GN's result.

Optimization of the LE and GN Architectures

We make the observation that when realizing the MoE architecture, it is not required that all its three components share the same input data. Depending on the license agreement, the GE, being a licensed IP of a trained DNN, can also provide some extracted features to both LE and GN as shown in Fig. 7.2(a). Such structural sharing can help provide a very good starting point to LE and GN because GE is trained with large size generic dataset, which shares the same fundamental properties with the customized data. In fact, the structural sharing technique can be viewed to contain elements from transfer learning, where the knowledge learned from generic data by the earlier layers of GE is transferred to LE and GN. More importantly, structural sharing can be also leveraged to simplify the structural design and minimize implementation overhead for both LE and GN.

More specifically, instead of feeding raw input data to LE and GN, we draw intermediate feature maps from the GE. Thus, the GE shares outputs of one or more of its convolutional layers (for higher level feature extraction) with both LE and GN, and the shared features are then processed by subsequent layer(s) in LE and GN to make their own specific decisions. Feature sharing is done by providing *only* the intermediate features extracted from the input without revealing GE's internal weight values. Therefore, the only information revealed (to a potential hacker) would be the dimension of these intermediate feature maps.

Sharing feature maps effectively helps reduce convolutional layers from LE and GN. In our case study, we found that feeding in the feature maps from the first

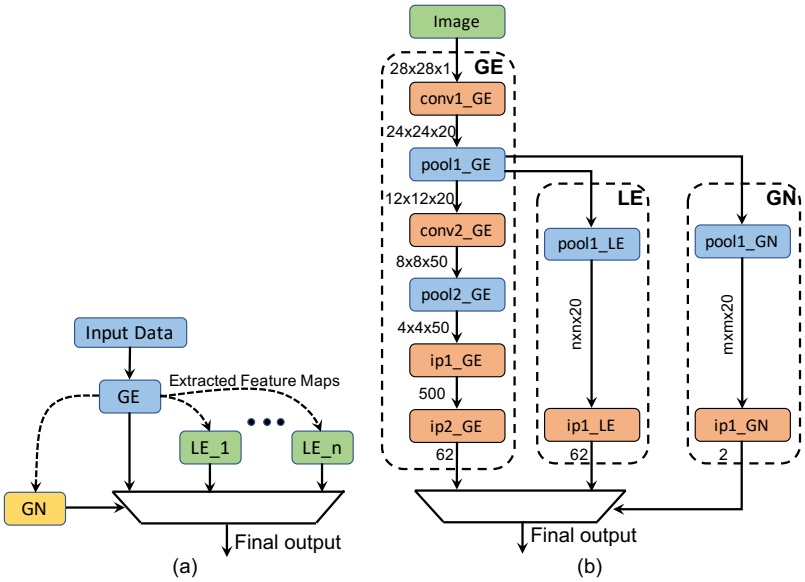

Figure 7.2: Structure sharing diagram and network structure of the prototype MoE model for the customized EMNIST recognition task

convolutional layer of GE and connecting them to one fully connected layer in LE and GN are enough to provide satisfactory performance. Another reason we choose to share earlier layers rather than later layers is because the features extracted by the earlier layers of a DNN are more general [89] and these features are shared by both generic and customized datasets. It is worth noting that, the MoE architecture is only a method to organize and coordinate multiple machine learning models to generate a final result. When being applied to deep neural networks, the proposed structural sharing technique is adopted to provide regularization/generalization power and to minimize the implementation overhead brought by LE and GN. Neither the MoE architecture nor the structural sharing technique limits LE or GN to only fully connected layers. Both LE and GN can definitely have any number of layers of any type if needed by other more complicated tasks, and there exists a tradeoff between the system's performance and the implementation overhead brought by LE and GN.

To efficiently share these feature maps, one approach is to use pooling layer to reduce the dimension of the shared feature maps before feeding them to the subsequent layer(s) in LE and GN, thus further reducing implementation overhead. This approach

treats each feature map equally and LE/GN may choose different pooling degrees. Another approach is to analyze the importance of different feature maps with respect to the goals of LE and GN, then only feed the important feature maps to LE and GN.

An Alternative Training Option

Besides the training scheme mentioned in Section 7.1, there exists another training option, which is explained below:

A pattern classifier such as the GE or the LE will partition the customized data's feature space R into two disjoint sub-regions: in one of them the classifier's output is deemed correct and in the other it is deemed incorrect, over the customized data. In the table below, let us consider four disjoint regions $\{R_k; 1 \leqslant k \leqslant 4\}$ where the outputs of the GE and the LE will be correct (Y) or incorrect (N). The row titled with GN gives the desired output of the GN where the lower-case letter d represents either 1 or 0, a don't care situation.

	R_1	R_2	R_3	R_4
GE	Y	N	Y	N
LE	Y	Y	N	N
GN	[d d]	[0 1]	[1 0]	[d d]
MoE	Y	Y	Y	N

First, just like the previous training scheme, the GE is trained on generic dataset. Then, given the training dataset R, the GE will partition it into $R_1 \cup R_3$ (correct classification) and $R_2 \cup R_4$ (mis-classification). The randomly initialized LE will partition it into $R_1 \cup R_2$ (correct classification) and $R3 \cup R4$ (mis-classification). From these results, the four regions R_1, R_2, R_3, and R_4 can be identified.

From this table, with MoE, the overall classification accuracy may be increased from $R_1 \cup R_3$ using only GE to up to $R_1 \cup R_2 \cup R_3$ using GE and LE. To achieve this performance enhancement, the LE should strive to correct mistakes made by the GE by maximizing region R_2 and shrinking region R_4.

This is also pictorially shown in Fig. 7.3. The outermost rectangle ($R_1 \cup R_2 \cup R_3 \cup R_4$) represents all customized samples. $R_1 \cup R_3$ represents the samples that can be correctly classified by GE. $R_1 \cup R_2$ represents the samples that can be correctly classified by LE. R_4 represents the samples that cannot be correctly identified by GE

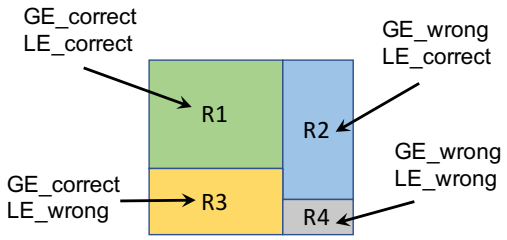

Figure 7.3: Diagram of the sample region covered by GE and LE. Region R1 + R3 represents the samples that are already correctly classified by GE. Region R2 represents the samples that can *only* be correctly classified by LE.

and LE. The ability of LE to maximally complement GE is shown by the relative size between R_2 and R_4. If R_2 is much larger than R_4, it indicates that LE is able to provide correct predictions on those samples that GE cannot handle, and LE complements GE very well.

Thus, this training option requires to train LE with samples in $R_2 \cup R_4$, and train GN to identify $R_2 \cup R_4$ from the rest of the region R. This is equivalent to changing the functionalities of LE and GN so that GN identifies when GE makes a mistake (rather than distinguishing customized versus generic data). However, our training experiences showed overfitting issue may occur due to insufficient training samples in $R_2 \cup R_4$; for example the size of $R_2 \cup R_4$ is only about 25% of R and the size of R is already very small in our experiments. Thus, in this work, we train LE on all customized data and train GN to predict if an input data is customized or generic, as explained in Section 7.1. Fortunately, we observe that LE still complements GE very well under this training setting.

7.2 Case Study of Recognizing Handwritten Digits & Letters

We evaluate the proposed MoE model on the classification problem of recognizing handwritten digits and letters. The Extended MNIST (EMNIST) dataset [10] is used as the generic dataset. The images of handwritten digits and letters released by [26] are used as the customized dataset for customized training and testing.

Generic and Customized Datasets

Generic Dataset: The EMNIST dataset is generated by applying Gaussian blurring, centering, padding, and down sampling to all the images in NIST Special dataset 19 [20]. The entire dataset is released by [10]. The structure of the EMNIST dataset is exactly the same as that of the NIST dataset. Specifically, the EMNIST dataset contains 814,255 28×28 grayscale images in total, and each of these images belongs to one of 62 classes ('0'-'9', 'a'-'z', and 'A'-'Z'). These images are further divided into training and testing sets with the same probability for each class, which results in 697,932 and 116,323 images in training and testing sets, respectively. Since the number of samples in each class is highly unbalanced in both training and testing sets, we first balanced the classes in these two sets before using them. Specifically, for both sets, we identified the class that has the least number of samples in each set and then randomly down sample other classes so that the number of samples in other classes roughly matches that of the class we identified at the beginning in the corresponding set. This results in about 2700 samples per class in training set and about 453 samples per class in testing set. Overall, there are 165,092 samples in training set and 27,537 samples in testing set, which correspond to 23.65% and 23.67% of the original training and testing sets.

User Customized Dataset: The user customized dataset is collected and released in [26]. It contains 28×28 grayscale images of 62 alphanumeric characters written by 10 Asian users. All images are collected and pre-processed using similar techniques as those in the EMNIST dataset. For each user, it has 30 images per class in training set and 10 images per class in testing set. This results in a total of 1860 training and 620 testing images per user. Compared with the size of the generic dataset, this corresponds to 1.13% and 2.25% in terms of training and testing sets, respectively.

The MoE Network Structure

Figure 7.2(b) shows the network structure of the prototype MoE model used in the experiment. The GE is a variation of LeNet5 [54] that is modified to produce 62 outputs. The LE shares its first convolutional layer with the GE by using the output feature maps of the first pooling layer of GE as its input feature maps. The LE then performs further max pooling operation on its input feature maps, which down samples

them to the size of $\mathfrak{n} \times \mathfrak{n} \times 20$. The exact value of $\mathfrak{n}$ depends on the receptive window size and stride of the kernel of the pooling layer. We will discuss how the value of $\mathfrak{n}$ is determined in our experiments to achieve the minimal implementation overhead with negligible loss in classification accuracy. The down-sampled feature maps is then fed into a fully connected layer that outputs the result of LE's classification. The size of this fully connected layer is $\mathfrak{n} \times \mathfrak{n} \times 20 \times 62$.

The GN has a similar structure to the LE with the exceptions that: 1) the size of the down-sampled feature maps is $\mathfrak{m} \times \mathfrak{m} \times 20$, which can be different from that of LE; 2) its fully connected layer only outputs two values indicating whether the final classification result should be selected from GE or LE. Similar to the LE case, the size of this layer is $\mathfrak{m} \times \mathfrak{m} \times 20 \times 2$ and we will discuss how the value of $\mathfrak{m}$ is determined in our experiments.

Training Procedure

Tensorflow is used for building and training the entire network. The training is performed in three steps as described below:

1. The GE is trained with the training set of the generic dataset. Note, in practice, this training is done prior to the deployment of edge devices. After training, all the parameters in GE are fixed in the subsequent steps.

2. For each user, LE is trained with the training set of that user's customized dataset.

3. For each user, GN is trained with the training set of that user's customized dataset plus equal number of instances from the generic dataset. The label of each training instance is either [1, 0] or [0, 1] indicating whether the instance is from generic or customized

In the following sections, the first step is referred as generic training and the last two are referred as customized training. During all aforementioned training steps, a small portion of the corresponding training set is used as validation set to tune the hyper-parameters of the training process, such as learning rate and number of training epoch.

Table VIII: Implementation overhead vs. overall classification accuracies for different degrees of subsampling from GE

Subsampled Input Feature Map Size	**Storage Overhead**	**Computation Overhead**	**Accuracies**	
			Customized	EMNIST
12×12×20	40.38%	8.05%	91.58	75.57
6×6×20	10.09%	2.01%	92.53	75.52
4×4×20	4.49%	0.89%	92.81	75.24
3×3×20	2.52%	0.50%	92.74	73.93
2×2×20	1.12%	0.22%	89.03	72.29
1×1×20	0.28%	0.05%	47.77	69.33

7.3 Experimental Results

Determining the Size of Pooling Layers in LE and GN

We experimented with different values of $\mathfrak{m}$ and $\mathfrak{n}$ to determine the sizes of the pooling layers in LE and GN which are denoted by *pool1_LE* and *pool1_GN* in Fig. 7.2(b). Specifically, for each set of values of $\mathfrak{m}$ and $\mathfrak{n}$, we trained the LE and GN networks and recorded classification accuracies as well as the storage (i.e., network size) and computation overhead due to LE and GN together, relative to GE. The storage overhead is measured with respect to number of distinct parameters in LE and GN.

Table VIII reports the percentage increase in storage and computation due to LE and GN relative to GE, and the classification accuracy of the overall network, which is reported as separate quantities over the customized and generic testing datasets. For each row, the same value is used for $\mathfrak{m}$ and $\mathfrak{n}$ so the sizes of the pooling layers in GN and LE are equal to each other. The first column reports the size of the down-sampled input feature maps in LE and GN, and the first row shows the default size if pooling is not performed.

As can be seen the overheads in both storage and computation drops significantly (from 40.38% to 0.28%, and from 8.05% to 0.05%) as the down-sampling becomes more aggressive. The accuracies on both customized data and generic data remain relatively stable before the down-sampled input feature maps size shrinks to $2 \times 2 \times 20$ (so $\mathfrak{n} = \mathfrak{m} = 2$). After that, both accuracies experienced a noticeable drop. The

Table IX: Overheads in storage and energy consumption of the LE and GN relative to GE

Storage (%)	**Overhead in Energy Consumption (%)**			
$\|\mathsf{LE}+\mathsf{GN}\|/\|\mathsf{GE}\|$	MAC	SRAM	DRAM	Total
2.52%	0.50%	2.58%	2.52%	2.45%

Table X: Post customized training accuracy of various components in the MoE Model

User ID	**Customized Data**					**EMNIST**
	GE	LE	GN	$Overall$	$\mathsf{LE} \mid \mathsf{GE}_{wrong}$	$Overall$
1	65.48	86.29	98.55	86.45	74.77	74.53
2	80.97	96.77	97.74	96.61	90.68	74.51
3	82.58	95.00	96.13	94.84	87.04	73.01
4	80.81	97.42	96.13	96.45	91.60	73.02
5	72.58	90.00	96.94	89.52	80.59	74.20
6	79.35	91.77	93.87	91.13	82.03	72.05
7	73.34	93.86	99.03	93.86	85.45	74.74
8	78.55	93.39	97.90	93.55	82.71	74.68
9	80.48	97.42	98.71	97.42	92.56	74.34
10	67.90	88.06	97.42	87.58	79.40	74.25
Average	76.21	93.00	97.24	92.74	84.68	73.93

highlighted row represents a good configuration of LE and GN in the sense that it only incurs minimal storage and computation overhead while still provides very high accuracies. Thus, it is used in the following experiments.

Table IX shows the overhead of storage and energy consumption of LE and GN together, reported as the percentage relative to GE. For energy estimation, we use the energy model mentioned in Section 4.2 and report energy break down of different components such as MAC, SRAM, and DRAM, and total. Table IX indicates that the overhead of storage and energy brought by LE and GN together are minimal, at 2.52% and 2.45% respectively, relative to GE.

Performance of Components in MoE

Table X reports the performance of various components of the MoE model after the training is completed.

- Column ‘GE’ shows the accuracy of GE on customized testing set. It represents the case when only GE is available and provides a reference point for evaluating the performance of the proposed MoE model.
- Column ‘LE’ shows the accuracy of LE on the customized testing set. It evaluates how well LE performs in classifying user customized data *if* GN behaves perfectly.
- Column ‘GN’ shows the accuracy of GN on customized testing set. It indicates how well GN performs in classifying whether the input image is from customized or generic datasets.
- The ‘Overall’ column under **Customized Data** shows the overall accuracy of the proposed MoE model on the customized testing set.
- Column ‘LE | GE_{wrong}’ measures the accuracy of LE on the customized testing samples that are wrongly classified by GE. This metric reflects how well LE complements GE.
- The ‘Overall’ column under **EMNIST** measures the accuracy of the entire model on the generic testing set.

As can be seen from Table X, on average, GE only has 76.21% accuracy on customized dataset, while LE is able to provide 93.00% accuracy after customized training (assuming GN behaves perfectly). Fortunately, after customized training, GN indeed performs very well with 97.24% accuracy in distinguishing whether an image is from customized or generic dataset. Thus, the overall accuracy of the entire model is significantly improved to 92.74%, on average.

The table also suggests that LE is indeed a good complement for GE on customized data, because 84.68% of the times, LE is able to provide correct classification on samples that cannot be correctly classified by GE. The overall accuracy on generic dataset after customized training is 73.93%, indicating a minimal drop compared to GE, which is 75.72% (not shown in the table).

Comparison with [26] and with Fine-tuning

Besides the proposed MoE model, there exists other alternative approaches for customizing large-size trained DNNs. Specifically, the recent work [26] proposed to augment the trained DNN with a task-specific network and an aggregation layer. In comparison, the *MoE* architecture has more flexibility due to the use of GN and has higher interpretability level because every component in *MoE* has its clear responsibility. Another approach that is widely used in transfer learning is to fix the parameters of early layers of the trained DNN and fine-tune the last few fully connected layers. In both approaches, the entire model (for [26], including the task-specific network and aggregation layer) is first trained with generic data by service providers before shipping to customer.

Later on, after shipping to customer, only the task-specific network and the aggregation layer (for [26]), or the last few fully connected layers (for fine-tuning) are trained with customized data by user. In order to compare with these alternative approaches, we implemented them in Tensorflow and carried out the training procedure as described in [26] with the same datasets mentioned in Section 7.2. We also note that the same LeNet5 model is used as the trained DNN as in the MoE model, and all of the two fully connected layers are retrained in the fine-tuning approach since it generates better results than only retraining the last fully connected layer.

The comparison of before/after customized training accuracies and implementation cost between these approaches and the proposed MoE model are shown in Table XI. For each user, it only shows the accuracies after customized training due to limited space. Compared to both [26] and fine-tuning, the MoE model has much lower accuracies on both datasets before customized training. This is because, at this point, only GE is trained with generic dataset and both LE and GN are initialized with random weights. Thus, LE and GN only introduce random noise to the entire system before customized training. In contrast, since the entire model in both [26] and fine-tuning are trained with generic data before deployment, their pre-customized training accuracies on both generic and customized datasets are similar and not noisy. So, by showing the noisy pre-customized training accuracies of MoE, our purpose was not to emphasize the absolute increase in accuracies as a result of customized training. But rather to show: 1) the accuracies of LE and GN in MoE are noisy before

Table XI: Comparison of pre & post customized training accuracies and implementation cost

User ID	[26]		Fine-tuning		MoE	
	Customized	EMNIST	Customized	EMNIST	Customized	EMNIST
1	87.58	67.65	86.29	71.87	86.45	74.53
2	97.90	70.39	96.61	72.41	96.61	74.51
3	94.68	69.68	95.48	71.91	94.84	73.01
4	96.29	70.15	96.77	72.32	96.45	73.02
5	90.97	67.51	91.13	71.51	89.52	74.20
6	91.13	71.33	92.10	73.59	91.13	72.05
7	92.29	67.49	93.91	70.82	93.86	74.74
8	94.52	69.70	94.68	71.74	93.55	74.68
9	98.87	69.61	97.90	72.14	97.42	74.34
10	88.23	67.97	88.71	71.31	87.58	74.25
Avg. Acc. (After training)	93.25	69.15	93.36	71.96	92.74	73.93
Avg. Acc (Before training)	75.05	75.94	75.09	75.72	29.76	28.90
(a) # Param.	19.59K		0		11.52K	
(b) # Inf. Comp.	44.34K		0		11.52K	
(c) # Cus. Train. Comp.	2452.03K		3181.00K		322.56K	
(d) # Avg. Total Comp.	2363.34K		2319.00K		355.27K	

customized training; and 2) compared to the accuracy of the base model (i.e., GE) on generic data (which was 75.72%), both [26] and fine-tuning have similar accuracies before customized training. *However*, after customized training, MoE has much higher accuracy (73.93%) compared to [26] (69.15%) and fine-tuning (71.96%). In fact, this noisy behavior of MoE can be addressed by two improvements, which will be introduced in Section 7.3

After customized training, the accuracies of the MoE model are dramatically increased. Specifically, its accuracy on customized dataset reaches 92.58%, which is similar to that of [26] and fine-tuning. Meanwhile, its accuracy on generic data is 73.93%, which is higher than [26]'s 69.15% and fine-tuning's 71.96%. Compared to GE itself (75.72%), this represents only 1.24% accuracy degradation versus 6.57% in [26] and 3.76% in fine-tuning, even though their pre-customized training accuracies are similar to GE's.

Regarding the implementation cost, we compared these approaches in terms of four metrics:

(a) *Total number of model parameters excluding the base model*, which directly reflects static memory overhead. In the proposed MoE architecture, this includes the parameters of both LE and GN. In [26], this includes the parameters of the task-specific network and the aggregation layer. Finally, there is no additional parameters in the fine-tuning approach since it only fine tunes the last two fully connected layers of the base model.

(b) *Number of computations needed for a single inference excluding the base model*. Like the first metric, this only counts the computations required by LE and GN in the MoE architecture, and by the task-specific network and the aggregation layer in [26]. There is no additional computation in the fine-tuning approach.

(c) *Number of computations required per customized training pass*. In general, each training pass includes one inference pass and one gradients back propagation pass of the *entire* model. But, in our case, for customized training, only a *subset* of layers are trained in all three approaches, so we define this metric to provide more clarifications about the required computations during customized training. First, during the inference pass, computation is needed from the input to the point where the probabilities of all classes are generated so that the value of the loss function can be calculated. Next, in the back propagation pass, computation is *only* needed from the loss to the earliest layer that needs to be trained so that the gradients of those layers' parameters can be calculated based on their current values, and then these parameters can be updated base on their gradients. We don't consider the generic training process since it is performed at the vendors' side where we assume limit in computation for this task is not a concern.

(d) *Average of total number of computations needed per inference when classifying user's inputs after customized training*. This metric evaluates the computation cost of the entire model after it becomes stable (i.e., after customized training) and is used to classify the user's inputs (customized data, not generic data), which is the majority use case.

As can be seen from Table XI, due to our proposed structural sharing technique, metrics (a) and (b) of MoE are 58.81% and 25.98% of those of [26], respectively, while

the fine-tuning approach does not incur any additional parameters and computations because no other network(s) is added to the base model.

Metric (c) of MoE is 13.15% and 10.14% of that of [26] and fine-tuning, respectively. This is because, besides the customization components, only the first layer of GE is involved in the inference pass when performing customized training on MoE architecture, while the entire base model is involved in [26] and fine-tuning. This is also the main reason for the huge gap between MoE and the other two approaches in terms of metric (d). Once customized training terminates, during inference, the MoE approach first computes the results of GN and then performs the inference of either GE or LE according to the GN's result. In the majority use case, the system will be presented with customized inputs, and the GN can correctly identify them as customized data most of the times (97.24% according to Table X). Thus, in the MoE approach, the inferences of GN, LE, and the first layer of GE are performed in 97.24% of the times, and the inferences of GN and entire GE are only performed in 2.76% of the times. However, in the other two approaches, the inference of the base model is always performed, which induces relatively-high computation cost. Overall, the proposed MoE approach is efficient in terms of both memory and computation compared to the other two approaches.

Comparison with Other Ensemble Techniques

Several other ensemble techniques such as *bagging*, *independent ensemble (IE)*, and *multiple choice learning (MCL)* are introduced in Section 2.2. Since models trained with *MCL* and *IE* were proved to have better performance than those trained with *bagging* [55, 56], here we ignore *bagging* and perform comparison between *MoE*, *IE*, and *MCL*. This is despite the fact that *IE* and *MCL* do not fit perfectly in our problem setting.

To ensure a fair comparison, the member models in *IE* and *MCL* are designed to have similar architectures compared to those in *MoE*. Specifically, both *IE* and *MCL* have only two member models. For *IE*, one of its member models is exactly the same as the GE in *MoE*, and another one consists of one convolutional layer and one fully connected layer. The `conv` layer in *IE* is the same as the first `conv` layer in GE and the fully connected layer is the same as that in LE. For *MCL*, we delete GN from the

MoE model and use GE and LE as the member models in *MCL*. Furthermore, in order to obey our problem setting, all member models in both *IE* and *MCL* are trained in two steps: first with generic data and then customized data. Finally, we test accuracies and calculate the implementation cost, which are shown in Table XII.

Table XII: The post-customized training overall/oracle accuracies on both customized and generic datasets of the three approaches, and their implementation cost.

User ID	**MoE** (Overall/Oracle)		**IE** (Overall/Oracle)		**MCL** (Overall/Oracle)	
	Customized	EMNIST	Customized	EMNIST	Customized	EMNIST
1	86.45 / 91.29	74.53 / 79.71	89.35 / 91.61	66.82 / 73.78	74.03 / 94.19	55.31 / 74.50
2	96.61 / 98.23	74.51 / 80.19	97.58 / 98.23	70.39 / 75.81	87.90 / 99.68	59.08 / 80.40
3	94.84 / 97.74	73.01 / 80.39	95.48 / 97.42	68.73 / 76.19	88.87 / 98.71	59.33 / 78.67
4	96.45 / 98.39	73.02 / 79.81	97.74 / 98.23	68.79 / 74.64	87.26 / 98.71	61.30 / 80.49
5	89.52 / 94.68	74.20 / 79.96	91.45 / 94.84	64.09 / 72.95	84.19 / 95.81	54.31 / 73.34
6	91.13 / 96.29	72.05 / 80.53	91.45 / 94.68	70.39 / 76.79	81.94 / 97.74	60.14 / 80.67
7	93.86 / 96.12	74.74 / 80.15	94.67 / 96.12	69.37 / 74.81	87.40 / 98.55	57.05 / 76.81
8	93.55 / 96.29	74.68 / 79.59	95.16 / 96.45	69.49 / 75.17	87.90 / 98.87	59.69 / 77.84
9	97.42 / 98.55	74.34 / 80.12	98.55 / 99.03	68.03 / 73.31	90.16 / 99.35	60.01 / 78.24
10	87.58 / 93.39	74.25 / 79.81	90.32 / 94.19	64.55 / 72.55	79.52 / 93.71	55.40 / 74.71
Average	92.74 / 96.10	73.93 / 80.03	94.18 / 96.08	68.07 / 74.60	84.92 / 97.53	58.16 / 77.57
#Param	468.02K		468.16K		467.66K	
#Comp-Train	322.56K		7854.48K		2962.32K / 6968.52K	
#Comp-Inf	355.27K	2221.60K	2618.16K		2330.52K	

Like many ensemble techniques, all three approaches are evaluated in terms of the overall accuracy and the oracle accuracy. The overall accuracy measures the accuracy of the system if the system is only allowed to make one prediction. For *MoE*, the final prediction is picked from either GE or LE depending on the GN's result. For *IE* and *MCL*, the final prediction is determined based on the average probabilities of their member models. The oracle accuracy measures if any of the member models makes the correct prediction. The last three rows in Table XII show the number of parameters, and number of computations required per training and inference pass.

Comparing the accuracies on the customized dataset, *MoE* has 1.44% lower overall accuracy than *IE* while both of them are significantly higher than that of *MCL*. There are two reasons that *IE* has higher overall accuracy than *MoE*: 1) the process of averaging probabilities in *IE* reduces the variance of member models, thus improves overall accuracy; 2) all member models are first trained with generic data,

which provides a good initialization for customized training. In fact, we also tested a variation of *MoE* with LE trained on generic data at vendors' side, and it indeed improved the overall accuracy on customized data by 0.34% (not shown in table). *MCL* has the lowest overall accuracy due to the overconfident issue, i.e., member models can make very confident predictions (probabilities are very skewed) even though the predictions are wrong. Thus, it adversely impacts the overall accuracy when averaged with other member models.

Regarding the oracle accuracy, *MoE*'s and *IE*'s are similar while *MCL*'s is much higher. This is because the loss function in *MCL* is designed to encourage the diversity between member models, thus covering more label space. As for the accuracies on generic dataset, *IE* and *MCL* have the same trend as on the customized data due to the aforementioned reasons. But they have much lower accuracies than *MoE* because all of their member models are trained with customized data in customized training, while the GE in *MoE* is intact during customized training. Therefore, models in *IE* and *MCL* forget the properties of generic data and have lower accuracies than those of *MoE*.

In terms of the implementation cost, since the member models are designed to have similar architectures, all three approaches have very close number of parameters. But *MoE* requires dramatically less computation than *IE* and *MCL* during both training and inference. During training, *MoE*'s computation requirement is about 4.1% to 10.9% of those of the others, and during inference, it is about 14.4% or 90.4% depending on the type of input data. This degree of saving is mainly due to the fact that all member models in *IE* and *MCL* need to be executed while only the selected components need to be executed in *MoE*. Please note that there are two inference computation cost numbers under *MoE* because different components in *MoE* are executed depending on whether the input is classified as customized or generic data by GN. Since *MCL* dynamically assigns training instances to one of its member models, the computation cost of training also has two values corresponding to the two member models.

Addressing the Noisy Behaviors of MoE

Finally, below we propose and evaluate two options that can overcome the noisy behavior of MoE model. The options are listed below and the results are presented in

Table XIII: The overall accuracies before/after customized training presented for customized and generic datasets, for the two options to overcome the noisy behavior.

User ID	**Option 1** (before/after)		**Option 2** (before/after)	
	Customized	EMNIST	Customized	EMNIST
1	58.06 / 87.58	72.56 / 74.51	65.48 / 86.45	75.75 / 74.53
2	78.39 / 97.10	72.56 / 74.74	80.97 / 96.61	75.75 / 74.51
3	78.06 / 95.32	72.56 / 74.78	82.58 / 94.84	75.75 / 73.01
4	78.55 / 97.58	72.56 / 73.25	80.81 / 96.45	75.75 / 73.02
5	68.55 / 88.87	72.56 / 74.66	72.58 / 89.52	75.75 / 74.20
6	76.29 / 90.48	72.56 / 73.51	79.35 / 91.13	75.75 / 72.05
7	72.70 / 94.18	72.56 / 75.15	73.34 / 93.86	75.75 / 74.74
8	72.10 / 94.19	72.56 / 74.57	78.55 / 93.55	75.75 / 74.68
9	75.81 / 97.90	72.56 / 75.39	80.48 / 97.42	75.75 / 74.34
10	62.58 / 87.58	72.56 / 74.87	67.90 / 87.58	75.75 / 74.25
Average	72.11 / 93.08	72.56 / 74.54	76.21 / 92.74	75.75 / 73.93

the following table.

1) Similar to [26], we can train both GE and LE with generic data at the vendors' side. Thus, LE can learn from generic data and make decent predictions. Even though GN is still noisy right after deployment, the pre-customized training accuracies of the whole system won't be noisy.

2) We use the result from GE as the final result right after deployment, and gradually enable LE and GN as they are trained with more and more customized data.

The overall accuracies of these two options are presented in Table XIII for before/after customized training, for both the customized and generic datasets. Based on the results, we make the following observations: (a) Compared to the noisy behaviors shown in the original paper, the proposed two options can significantly improve the pre-customized training accuracies. Thus, these options may be adopted in practice; (b) Option 2 has higher pre-customized training accuracies than option 1. This is because option 2 only uses GE to perform classification right after deployment. In comparison, although the LE in option 1 is trained with generic data, its performance is still not as good as the GE's performance and the accuracies of the entire system

are adversely impacted by the LE; (c) After the same amount of customized training, option 1 has slightly better performance than option 2. This is because training LE with generic data provides a better initialization for customized training, and the LE in option 1 has higher post-customized training accuracies than that in option 2.

8 FUTURE DIRECTIONS

In the previous chapters of this dissertation, we presented our contributions towards deployment of DNNs on resource-constrained embedded systems. Specifically, we proposed: 1) a high-level analytical energy model that is able to quickly estimate the energy consumption of DNNs, thus facilitate the inclusion of energy into DNN design/simplification process; 2) structural simplification techniques such as neuron elimination and channel pruning that can significantly reduce the static/run-time memory, computation, and energy requirements of DNN models, so that inference of modern DNNs may be accomplished locally on edge devices; and 3) an novel Mixture of Experts (MoE) architecture for local learning to customize a trained DNN that is deployed on an edge device, thus achieving performance improvement on user-specific data with minimal implementation overhead. Although our contributions take us one step closer to our ultimate goal, some issues remain unresolved. So, based on our work, we identify two directions for future research.

First, in the current MoE architecture, although the structural sharing technique is already used to minimize the implementation overhead of LE(s) and GN, but the fact that both LE(s) and GN use the same set of feature maps as their input leave us some room for further optimization. The responsibility of GN is to distinguish between generic and customized data, while the duty of LE(s) is to generate results with respect to the original task. Thus, it is highly likely that the most important features for LE(s) and GN are different. For example, the features cared by LE(s) and GN could be different channels of the same convolutional layer of GE, or they could be the features extracted by different layers of GE, or the mixture of both. Our goal of this direction is to identify and share the important features for LE(s) and GN separately. This improvement may not seem to be significant in our case study because the implementation overhead is already very small anyway. However, as we mentioned in Section 7.1, both LE(s) and GN can have any number of layers of any type if necessary. Thus, for more complicated tasks that require more complex LE(s) and GN models, this improvement may further reduce the implementation overhead.

Second, all the structural simplification techniques mentioned in Chapter 2 leave the label space intact, in other words, the simplified model is able to generate pre-

dictions for the same number of classes as the original model. Also, the inference of the entire DNN model (no matter simplified or not) is usually assumed to be fully executed by one edge device. However, the computation resources available on one edge device may be insufficient for complex DNNs, which are required for high performance. On the other hand, in the era of IoT, there are many edge devices around us, and they are able to work collaboratively. Thus, to fully utilize the resources of edge devices, we propose to decompose and simplify a complex DNN model into several simpler children DNNs with each only in charge of generating results of a subset of the entire label space. For example, we want to decompose a CNN that can classify 100 classes into 10 children CNNs with each able to classify 10 classes. Then, each of these children DNNs can be deployed to one edge device, and their results can be aggregated together to solve the original task. There are three related questions need to be addressed to achieve a good balance between the complexity of children DNNs and the overall performance: 1) how to partition the original label space; 2) what the output of each children model should be and how to aggregate them; and 3) how to aggressively simplify a model if only a subset of labels is considered. The goal of this direction is to make contributions towards solving these three questions. Actually, the third question itself is also useful for user customization in the case that the user only cares about a subset of the label space.

REFERENCES

[1] Baker, Bowen, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2016. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

[2] Boutsidis, C., M. W. Mahoney, and P. Drineas. 2009. An improved approximation algorithm for the column subset selection problem. In *Acm-siam symposium on discrete algorithms*, 968–977.

[3] Canals, Vincent, Antoni Morro, Antoni Oliver, Miquel L Alomar, and Josep L Rosselló. 2016. A new stochastic computing methodology for efficient neural network implementation. *IEEE Transactions on Neural Networks and Learning Systems* 27(3):551–564.

[4] Chen, Chenyi, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Ieee international conference on computer vision*, 2722–2730.

[5] Chen, Guobin, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. 2017. Learning efficient object detection models with knowledge distillation. In *Advances in neural information processing systems*, 742–751.

[6] Chen, Y-H, J. Emer, and V. Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Acm sigarch computer architecture news*, vol. 44, 367–379. IEEE Press.

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

[8] Cheng, Yu, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*.

[9] Cireşan, Dan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*.

[10] Cohen, Gregory, Saeed Afshar, Jonathan Tapson, and André van Schaik. 2017. EMNIST: an extension of MNIST to handwritten letters. *arXiv preprint arXiv:1702.05373*.

[11] Collobert, Ronan, Samy Bengio, and Yoshua Bengio. 2002. A parallel mixture of svms for very large scale problems. In *Advances in neural information processing systems*, 633–640.

[12] Courbariaux, Matthieu, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*.

[13] Cun, Y. Le, J. S. Denker, and S. A. Solla. 1990. Optimal brain damage. In *Advances in neural information processing system*, vol. II.

[14] Denil, Misha, Babak Shakibi, Laurent Dinh, Nando De Freitas, et al. 2013. Predicting parameters in deep learning. In *Advances in neural information processing systems*, 2148–2156.

[15] Denton, E. L., W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, 1269–1277.

[16] Gauen, K., R. Rangan, A. Mohan, Y.-H. Lu, Wei Liu, and A. C. Berg. 2017. Low-power image recognition challenge. In *Ieee asia and south pacific design automation conference*, 99–104.

[17] Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 580–587.

[18] Golub, G. 1965. Numerical methods for solving linear least squares problems. *Numerische Mathematik* 7(3):206–216.

[19] Golub, G., and C. Loan. 2013. *Matrix computations, 4th ed.* London: The Johns Hopkins University Press.

[20] Grother, Patrick J. 1995. NIST special database 19 handprinted forms and characters database. *National Institute of Standards and Technology*.

[21] Guo, Yiwen, Anbang Yao, and Yurong Chen. 2016. Dynamic network surgery for efficient dnns. In *Advances in neural information processing systems*, 1379–1387.

[22] Gysel, Philipp, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*.

[23] Han, S., J. Pool, J. Tran, and W. J. Dally. 2015. Learning both weights and connections for efficient neural network. In *Annual conference on neural information processing systems*, 1135–1143.

[24] Han, Song, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd annual international symposium on computer architecture*, 243–254. IEEE.

[25] Han, Song, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[26] Harris, Barend, Mansureh S Moghaddam, Duseok Kang, Inpyo Bae, Euiseok Kim, Hyemi Min, Hansu Cho, Sukjin Kim, et al. 2018. Architectures and algorithms for user customization of CNNs. In *Ieee asia and south pacific design automation conference*, 540–547.

[27] Hashemi, Soheil, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. 2017. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Design, automation & test in europe conference & exhibition*, 1474–1479.

[28] Hassibi, Babak, David G Stork, and Gregory J Wolff. 1993. Optimal brain surgeon and general network pruning. In *Ieee international conference on neural networks*, 293–299. IEEE.

[29] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 770–778.

[30] He, Yang, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. 2018. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*.

[31] He, Yihui, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the ieee international conference on computer vision*, 1389–1397.

[32] Hinton, Geoffrey, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29(6):82–97.

[33] Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[34] Hochreiter, S., and J. Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

[35] Hornik, K., M. Stinchcombel, and H. White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2(5):359–366.

[36] Horwitz, M. 2015. Energy table for 45nm process. In *Stanford vlsi wiki*.

[37] Howard, Andrew G, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[38] Hu, Hengyuan, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

[39] Hu, Y. H., Qiuzhen Xue, and W. J. Tompkins. 1991. Structural simplification of a feed-forward, multi-layer perceptron artificial neural network. In *International conference on acoustics, speech, and signal processing*, 1061–1064.

[40] Hu, Yu Hen, Surekha Palreddy, and Willis J Tompkins. 1997. A patient-adaptable ecg beat classifier using a mixture of experts approach. *IEEE Transactions on Biomedical Engineering* 44(9):891–900.

[41] Huang, Zehao, and Naiyan Wang. 2018. Data-driven sparse structure selection for deep neural networks. In *Proceedings of the european conference on computer vision*, 304–320.

[42] Iandola, Forrest N, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

[43] Ioannou, Yani, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. 2015. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*.

[44] Jacobs, Robert A, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural Computation* 3(1):79–87.

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

[46] Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *ACM international conference on multimedia*, 675–678.

[47] Jordan, Michael I, and Robert A Jacobs. 1994. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation* 6(2):181–214.

[48] Jouppi, Norman P, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 1–12. ACM.

[49] Kim, Yong-Deok, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*.

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

[51] Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

[52] Krogh, A., and J. A. Hertz. 1991. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, vol. 4, 950–957.

[53] Kung, S. Y., and Y. H. Hu. 1991. A Frobenius approximation reduction method (FARM) for determining optimal number of hidden units. In *Ieee international joint conference on neural networks*, 163–168.

[54] Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the ieee*, 2278–2324.

[55] Lee, Kimin, Changho Hwang, Kyoung Soo Park, and Jinwoo Shin. 2017. Confident multiple choice learning. In *Proceedings of the 34th international conference on machine learning-volume 70*, 2014–2023. JMLR. org.

[56] Lee, Stefan, Senthil Purushwalkam, Michael Cogswell, David Crandall, and Dhruv Batra. 2015. Why m heads are better than one: Training a diverse ensemble of deep networks. *arXiv preprint arXiv:1511.06314*.

[57] Li, Hao, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

[58] Li, J., A. Ren, Z. Li, C. Ding, B. Yuan, Q. Qiu, and Y. Wang. 2017. Towards acceleration of deep convolutional neural networks using stochastic computing. In *Ieee asia and south pacific design automation conference*, 115–120.

[59] Li, Quanquan, Shengying Jin, and Junjie Yan. 2017. Mimicking very efficient network for object detection. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 6356–6364.

[60] Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400*.

[61] Lin, Shaohui, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baochang Zhang. 2018. Accelerating convolutional networks via global & dynamic filter pruning. In *Proceedings of the international joint conferences on artificial intelligence*, 2425–2432.

[62] Lin, Y. Y., S. Zhang, and N. R. Shanbhag. 2016. Variation-tolerant architectures for convolutional neural networks in the near threshold voltage regime. In *Ieee international workshop on signal processing systems*, 17–22. IEEE.

[63] Liu, Zhuang, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the ieee international conference on computer vision*, 2736–2744.

[64] Long, Jonathan, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 3431–3440.

[65] Luo, Jian-Hao, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the ieee international conference on computer vision*, 5058–5066.

[66] Luo, Ping, Zhenyao Zhu, Ziwei Liu, Xiaogang Wang, and Xiaoou Tang. 2016. Face model compression by distilling knowledge from neurons. In *Thirtieth aaai conference on artificial intelligence*.

[67] Miotto, Riccardo, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. 2017. Deep learning for healthcare: review, opportunities and challenges. *Briefings in Bioinformatics*.

[68] Molchanov, Pavlo, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning convolutional neural networks for resource efficient inference. In *Proceedings of the international conference on learning representations*.

[69] Mun, Jonghwan, Kimin Lee, Jinwoo Shin, and Bohyung Han. 2018. Learning to specialize with knowledge distillation for visual question answering. In *Advances in neural information processing systems*, 8081–8091.

[70] Nowlan, J. S., and G. E. Hinton. 1992. Simplifying neural networks by soft weight-sharing. *Neural Computation* 4(4):473–493.

[71] Pan, Sinno Jialin, and Qiang Yang. 2010. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* 22(10):1345–1359.

[72] Raina, Rajat, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y Ng. 2007. Self-taught learning: transfer learning from unlabeled data. In *Proceedings of the 24th international conference on machine learning*, 759–766. ACM.

[73] Rasmussen, Carl E, and Zoubin Ghahramani. 2002. Infinite mixtures of gaussian process experts. In *Advances in neural information processing systems*, 881–888.

[74] Real, Esteban, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *Proceedings of the 34th international conference on machine learning-volume 70*, 2902–2911.

[75] Romero, Adriana, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*.

[76] Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

[77] Simonyan, Karen, and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[78] Sutskever, Ilya, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112.

[79] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 1–9.

[80] Tai, Cheng, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. 2015. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*.

[81] Tann, Hokchhay, Soheil Hashemi, R Bahar, and Sherief Reda. 2016. Runtime configurable deep neural networks for energy-accuracy trade-off. In *Proceedings of the eleventh ieee/acm/ifip international conference on hardware/software codesign and system synthesis*, 34. ACM.

[82] Tann, Hokchhay, Soheil Hashemi, R Iris Bahar, and Sherief Reda. 2017. Hardware-software codesign of accurate, multiplier-free deep neural networks. In *Design automation conference (dac), 2017 54th acm/edac/ieee*, 1–6. IEEE.

[83] Veit, Andreas, Michael J Wilber, and Serge Belongie. 2016. Residual networks behave like ensembles of relatively shallow networks. In *Advances in neural information processing systems*, 550–558.

[84] Watrous, R, and G Towell. 1995. A patient-adaptive neural network ecg patient monitoring algorithm. In *Computers in cardiology 1995*, 229–232. IEEE.

[85] Wen, Wei, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, 2074–2082.

[86] Xie, Lingxi, and Alan Yuille. 2017. Genetic cnn. In *Proceedings of the ieee international conference on computer vision*, 1379–1388.

[87] Xue, Q., Y.H. Hu, and W.J. Tompkins. 1989. A neural network weight pattern study with ECG pattern recognition. In *Annual international conference of ieee engr. in medicine & biology soc.*, 2023–2024.

[88] Yang, T.-J., Y-H. Chen, and V. Sze. 2016. Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv preprint arXiv:1611.05128*.

[89] Yosinski, Jason, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, 3320–3328.

[90] Zhang, Boyu, Azadeh Davoodi, and Yu Hen Hu. 2018. Exploring energy and accuracy tradeoff in structure simplification of trained deep neural networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8(4): 836–848.

[91] ———. 2018. Exploring energy and accuracy tradeoff in structure simplification of trained deep neural networks. In *Proceedings of the 23rd asia and south pacific design automation conference*, 331–336. IEEE Press.

[92] ———. 2018. A mixture of expert approach for low-cost customization of deep neural networks. *arXiv preprint arXiv:1811.00056*.

[93] ———. 2019. Efficient inference of cnns via channel pruning. *arXiv preprint arXiv:1908.03266*.

[94] ———. 2019. A mixture of expert approach for low-cost dnn customization. *IEEE Design & Test (under revision)*.

[95] Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 6848–6856.

[96] Zhu, Chenzhuo, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*.

[97] Zoph, Barret, and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.