Customized Systems of Deep Neural Networks for Energy Efficiency

by

Babak Zamirai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Babak Zamirai

zamirai@umich.edu

ORCID iD: 0000-0001-8379-0834

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. Scott Mahlke for guiding me through the challenging journey of PhD. He taught me how to conduct research by patiently giving me enough space and freedom to explore various ideas on my own, while being an excellent mentor who provided insightful and informative feedback when I was stuck on technical or even nontechnical aspects of problems. I also owe thanks to the remaining members of my dissertation committee, Prof. Dreslinski, Prof. Kim and Prof. Tang. They all devoted their precious time to help form this dissertation into what it has become today.

I am proud to be a member of the Compilers Creating Custom Processors (CCCP) Research Group. I enjoyed having numerous conversations, both serious and fun, about various topics with my current and former labmates. They always helped me a lot with my projects, and I never felt left alone the stressful nights before deadlines. Mehrzad and Daya helped me transform from a student to a researcher when I started my PhD. Jiecao, Salar and Pedram contributed significantly to different chapters of this work. In addition, I had the great chance of collaborating with Parker Hill, Michael Laurenzano and Animesh Jain. Ankit, Anoushe, Janghaeng, Gaurav, Shruti, Andrew, Jason and John were a welcoming group of people who I had the unique opportunity of joining and working alongside. I want to thank Sunghyun, Jonathan, Shikai, Hossein, Ze, Armand, Brandon, Sanjay and Yunjie for all the interesting discussions that we had and proofreading my papers.

Ann Arbor is a completely different city from my hometown, Tehran, in many aspects from the size of the city and population to weather. However, my large Iranian extended family of friends made this transition smooth and full of enjoyable memories. I would like to thank Samin, Amir, Nima, Amirhossein, Mehran, Ashkan, Ramin, Alireza, Farima, Milad, Nahal, Shayan, Pouyan, Parisa, Sadaf, Negar, Mehrdad, Donya, soroush, Javad, Javid, Navid, Mehdi, Omid, Hossein, Mohammad, Arman, Mohammadreza, Moin, Mina, Mahsa, Hedieh, Tara, Sajjad, Sina, Farshad, Bahram Mahmoud, Mostafa, Heidar, Ahmad, Morteza, Shahab, Saeed, Reza, Vahid, and many others. We made countless fantastic memories together from my birthday parties and celebrating cultural events to playing soccer every single week. I cannot imagine living in this city even for a single week without having these awesome friends. I would like to express my special thanks to all members of Alborz Sport Club. Playing soccer with these skillful people was a weekly stress relief and source of joy and happiness during long cold Winters of Ann Arbor.

Finally and most importantly, my family deserves major gratitude for everything I have achieved in my life. I have not seen my mother, Farahnaz, and my father, Siamak, for the past few years due to the single-entry visa and travel ban. However, they have always encouraged me to stay strong and pursue my dreams. I am very fortunate that my brother, Pedram, and his wife, Agreen, are both graduate students at the University of Michigan. Hence, we had the opportunity of living together for the past three years, which was an amazing and unique experience for me to stay hopeful and happy even in the most difficult situations of PhD life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The size and complexity growth of deep neural networks (DNNs), which is driven by the push for higher accuracies and wider ranges of functionality, is outpacing the growth of hardware for mobile systems. Thus, edge devices must collect and transmit sensory data for analysis on a remote server. However, offloading the data to be processed on the backend server can be problematic due to the high cost/latency of wireless communication of the raw data as well as potential privacy/security concerns.

Hence, there has been significant research interest to increase performance and energy efficiency of deep learning computation on both mobile devices and servers. These techniques can be divided into two major categories: input-invariant and input-variant. Input-invariant methods leverage hardware and software techniques, such as pruning, to accelerate a single DNN for the entire dataset. On the other hand, input-variant approaches focus on the fact that most of the inputs do not require the entire computational power of the model to produce an accurate final output. Consequently, they employ a combination of simple and complex models to dynamically adjust the complexity of the DNN to input difficulty.

Both DNN pruning and intelligent combination of DNNs require machine learning expertise and manual design, which makes them hard to implement and deploy. This thesis proposes techniques to improve performance and applicability of both input-invariant and input-variant methods. First, it introduces a new category of input-variant solutions to maximize energy efficiency and minimize latency of DNNs by customizing systems of DNNs based on input variations. Instead of conventional DNN ensembles, it proposes a data heterogeneous multi-NN system to divide the data space into subsets with one specialized learner for each subset. In addition, an intelligent hybrid server-edge deep learning system is introduced to dynamically distribute DNN computation between the cloud and edge device based on the input data and environmental conditions. Furthermore, it suggests an input-driven synergistic deep learning system, which dynamically distributes DNN computation between a more accurate big and a less accurate little DNN. At the end, it introduces a noniterative double-shot pruning method, which takes advantage of both architectural features and weight values to improve the simplicity and applicability of pruning.

Compared to the conventional approaches, on average, an energy consumption reduction up to 91% and a performance improvement up to $12.3\times$ are achieved, while maintaining the original accuracy.

CHAPTER 1

Introduction

In the last few years, due to the large and rapidly increasing quantities of available data, Deep Neural Network (DNN) models trained on a large dataset significantly outperform traditional machine learning techniques in various domains [1], such as object detection, e.g. Fast R-CNN [2, 3], image classification [4, 5], speech recognition, e.g. Microsoft Cortana [6], natural language processing [7], and handwriting recognition [8].

Current mobile devices are capable of generating huge amounts of high-quality data in the form of video and images to feed DNNs for various applications. Since DNNs are the state-of-the-art algorithms for many mobile vision applications, they are becoming an essential part of using and sharing photos and videos taken by smartphone cameras. However, the size and complexity of DNNs are increasing rapidly to improve their accuracy and functionality [9, 10].

These massively complex models are not necessarily suited to low-power edge devices [11]. Therefore, traditional mobile devices offload DNN computation to the cloud, which may require transferring large amounts of data to a remote server [12, 13]. However, this additional latency and energy consumption are not tolerable by many time-sensitive applications on power-limited devices. For instance, adding delay to image translation applications (Google image translate), which are expected to respond immediately, decreases the user satisfaction. High-speed network connectivity may also not always be available depending on location.

To overcome these problems, numerous approaches were proposed to improve performance and energy efficiency of feedforward inference. These methods can be divided into input-invariant and input-variant techniques. The input-invariant approaches are further classified to two main categories: hardware-based and software-based.

Hardware-Based: Because of the inherent parallelism, DNNs can be efficiently accelerated with specialized hardware. Many recent hardware platforms have special features for DNN processing, such as special vector instructions and reduced precision arithmetic support to perform faster deep learning computation. In addition, there are various systems that are built specifically for DNN processing, in the form of custom DNN servers and embedded System-on-Chips (SoCs).

These hardware-based approaches concentrate on utilizing intra- and inter-layer parallelism to accelerate computation and improve energy efficiency, which is limited to layer-level designs instead of considering the whole structure of DNNs.

Software-Based: Traditionally, DNN models were designed to maximize accuracy without considering the underlying hardware. However, this approach can lead to designs that are complex to implement and deploy. To address this problem, recently, there has been an increased interest in maximizing energy efficiency and performance of DNNs while maintaining their original accuracy or minimizing the accuracy loss.

DNNs are typically overparameterized, which results in a large amount of inherent redundant computation per inference. Software-based methods focus on reducing the storage and computational resources required for DNNs by eliminating unnecessary information [14]. This redundancy can be removed by reducing the precision of operations and operands (weights/activations), including using fixed-point instead of floating-point, reducing the bitwidth, nonuniform quantization, and weight sharing. In addition to precision reduction, there are various methods to reduce the number of operations and model size, such as compression, pruning, compact network architecture design, and knowledge distillation.

Many of these input-invariant approaches require rigorous hardware modifications or machine learning expertise to heavily tune hyperparameters and are non-trivial to be extended to new architectures and various tasks. For example, magnitude based network pruning approaches [11] rely on pretraining and expensive iterative prune – fine-tune cycles. In addition, both hardware- and software-based approaches try to improve the performance and efficiency of a single DNN for all input instances existing in a dataset without considering their variations and dissimilarities.

Input-Variant: DNNs are usually overprovisioned and most of the inputs do not require the entire computational power of the model to produce an accurate final output. Although very large and computationally intensive models are necessary to correctly classify the hard inputs, much smaller models would also work decently for most of the inputs [15]. Input-variant approaches combine simple and complex models to run easy inputs efficiently on simple models and utilize complex models only for hard inputs for maintaining the high accuracy.

DNNs with early exits, such as BranchyNet [16], combine multiple DNNs in a single DNN by adding multiple branches to the model and terminating the computation as soon as the classification confidence is high enough. Another group of approaches for taking advantage of input variations augment DNNs with gating units to decide whether to bypass the subsequent layer based on the previous layers [17].

All these input-variant approaches require machine learning expertise and manual design to create DNNs with the desired behavior. For example, DNNs with early exits require the user to determine the location of each exit and design the side branch classifiers. In addition, multiple exit points result in multiple loss functions which make the training process more complicated.

This thesis addresses the limitations of the traditional DNN computation complexity reduction techniques by customizing systems of DNNs based on input variations existing in datasets and simplifying the use of pruning techniques for nonexpert users. We demonstrate that additional energy efficiency and performance improvements could be achieved through input-variant acceleration techniques while eliminating the manual design by replacing a single DNN with multiple off-the-shelf or automatically created DNNs, and offloading most of the computation on the simple ones. In addition, we show that it is possible to obtain extremely sparse networks using noniterative pruning approaches with no additional hyperparameters or complex pruning schedules, while maintaining the same accuracy as the existing baselines. Chapter 3 proposes a data heterogeneous multi-NN system, which partitions the data space into subsets and employs one specialized learner for each subset. In Chapter 4 and 5, intelligent hybrid server-edge deep learning systems are introduced to evaluate the input data and environmental conditions and dynamically divide DNN computations between the cloud and edge device. Then, in Chapter 6, an input-driven synergistic deep learning system is proposed. This system dynamically decides between activating a more accurate big and a less accurate little DNN. Chapter 7 introduces a noniterative double-shot neural network pruning method, which combines architectural insights and information about the importance of weight values to improve the simplicity and applicability of pruning approaches.

1.1 A Data Heterogeneous Multi-NN System

To reverse the energy trend, we take a counterintuitive approach to investigate ensemble methods. Traditional ensemble methods use a group of weak learners operating collectively to create a strong learner that is capable of achieving higher accuracy and supporting a wider range of functionality. However, ensembles often suffer from high energy consumption due to the inherent inefficiencies of activating multiple, often redundant, learners. We adapt the traditional approach by creating a data heterogeneous multi-NN system. In contrast to traditional ensembles, data heterogeneous multi-NNs divide the data space into subsets with one specialized NN for each group of data. For inference, a selector identifies the input subset and activates the corresponding NN. To demonstrate the feasibility of data heterogeneity, a systematic methodology is developed to replace an NN with a set of automatically generated data specialized neural networks (dsNNs) and the corresponding selector. By activating only one dsNN for each input instance and reducing the interference between different input behavior patterns, our technique provides lower energy consumption while maintaining output accuracy. A multi-dsNN system comprised of two dsNNs is implemented and evaluated across a microcontroller, a mobile CPU, and a mobile GPU.

1.2 An Intelligent Hybrid Server-Edge Deep Learning System

In comparison to offloading CNN computations on the cloud, edge devices offer potential advantages of faster response time, lower energy consumption, continuous operation regardless of wireless network conditions, and better privacy/security. To achieve frequent edge device execution, we propose *SIEVE*, Speculative Inference on the Edge with Versatile Exportation, which dynamically distributes CNN computation between the cloud and edge device based on the input data and environmental conditions to maximize efficiency and minimize latency. A speculative version of the CNN is created through aggressive precision reduction techniques to run most of the inferences on the edge device, while the unmodified CNN is run on the cloud server. A runtime system directs each input to either the edge or cloud based on the input size and environmental conditions. The runtime system also decides whether to accept speculative inferences made on the edge or invoke recovery by replaying the inference on the cloud.

1.3 Extension of Cloud-Edge Hybrid Systems

To expand the applicability and design choices of intelligent hybrid server-edge deep learning systems, we propose deploying a pair of lightweight CNNs on the edge device to run most of the inferences speculatively, instead of aggressively compressed speculative CNNs, while the unmodified CNN is run either on cloud server or edge device depending on its complexity and network connection speed. Same as the original proposed intelligent hybrid server-edge deep learning system, A runtime system decides between running either on the edge or cloud based on the input size and environmental conditions. And, the runtime system is also responsible for invoking recovery when the output is not reliable.

1.4 An Input-Driven Synergistic Deep Learning System

To combat the substantial computation and memory demands inherent to deep learning queries and facilitate the efficient execution of complex models on cloud servers, we proposes *Dynamic Duo (DD)*, an input-driven synergistic deep learning system, which dynamically distributes CNN computation between a more accurate big and a less accurate little CNN to achieve maximum performance and efficiency. The most synergistic pair of CNNs is chosen through heuristic searches in the model pool to run most of the inferences on the little CNN, while the big CNN is invoked only when the little CNN has low confidence.

1.5 Double-Shot Neural Network Pruning

To improve the performance of single-shot network pruning approaches, which work on randomly initialized weights to identify essential connections and ignore the importance of weight values, we investigate the combination of architectural insights and information about the importance of weight values. We propose a double-shot noniterative pruning approach. Our technique converts the global sparsity rate to layer-wise local sparsity rates using single-shot approaches. Then, it prunes the trained dense network utilizing local sparsity rates and connection importance metrics. And at the end, it trains the sparse network in the same way as the dense one. Noniterative double-shot pruning does not require any complex pruning schedule or additional hyperparameters, which makes it easy-to-use for nonexpert users.

CHAPTER 2

Background and Related Work

2.1 Neural Networks

Neural networks (NNs) consist of layers of neurons that have trainable parameters including weights and biases. The output of each neuron (neuron activation), which is the dot product of its inputs and weights followed by an optional nonlinear activation function, could be the input of other neurons in other layers. Figure 2.1 shows the structure of a feedforward fully-connected NN [18]. It is composed of layers of computational units (neurons), with connections between every pair of neurons in adjacent layers. Each neuron combines its weighted inputs ($X_i \times W_i$) and bias value to determine the output (Y) via the activation function (f). These combination of fully-connected layers could perform as a stand-alone NN to process the input or as a part of a bigger NN to process the preprocessed data by other layers, such as convolutional layers.

NNs are capable of performing classification and regression tasks. For classification, the NN receives an input instance, which is a vector of input elements, and computes the output vector of K elements. By assigning a softmax ($\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$) activation function on the output layer of the NN, it estimates the probability of that the correct output is j (for $j = 1, ..., K$), which could be interpreted as its confidence in each class.

Backpropagation is a common method of training artificial NNs used alongside an optimization method such as gradient descent. In backpropagation, a loss function, typically cross-entropy loss for classification problems, is used to measure the prediction quality of the NN. It calculates the

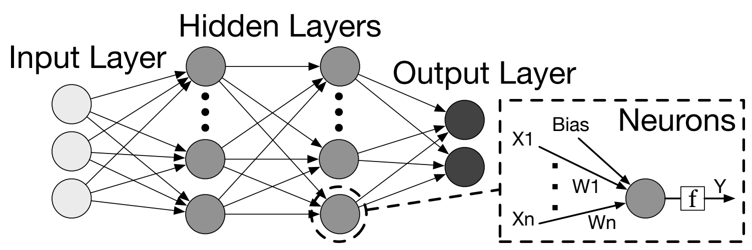


Figure 2.1: Feedforward fully-connected neural network.

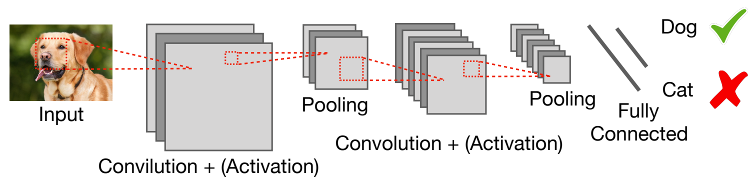


Figure 2.2: A typical convolutional neural network for image classification containing convolutional, activation, pooling and fully-connected layers.

gradient of the loss function with respect to all weights existing in the NN, and the gradient is fed back through all the layers to update connection weights for minimizing the loss function. There are two main problems in the training phase, model overfitting and model underfitting. Overfitting occurs when the NN captures the irrelevant information (noise) of the training data, and underfitting will occur if the NN cannot capture all behavior patterns of the training data.

2.2 Convolutional Neural Networks

There exist different kinds of neural networks (NNs) for various application domains, such as fully-connected, convolutional and recurrent neural networks. Figure 2.2 shows the typical layers including convolutional, activation such as ReLU, pooling for subsampling, and fully-connected, in the architecture of feed-forward deep convolutional neural networks (CNNs). CNNs for image recognition and classification are responsible for converting image pixels to class scores.

Figure 2.3 shows the top-5 accuracy and depth of NNs for top competitors of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [19], which is a software image classification

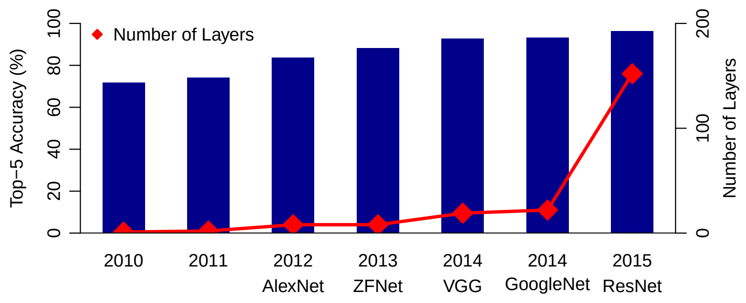


Figure 2.3: Accuracy vs. depth of NNs on ImageNet dataset. Deeper NNs require more computation and memory, which translates to lower performance and higher energy.

and object detection competition, from 2010 to 2015. As demonstrated, the depths of the DNNs are increasing rapidly to achieve higher accuracies. Deeper NNs require more computational power and memory, which results in lower performance and higher energy consumption.

2.3 Efficient Neural Networks

Various techniques for improving energy efficiency and performance of DNNs can be divided into input-invariant and input-variant methods. The input-invariant approaches are further classified to two main categories: hardware-based and software-based.

2.3.1 Input-Invariant Techniques

Hardware-based: Because of the inherent parallelism, DNNs can be efficiently accelerated with specialized hardware [20, 21, 22, 23, 24, 25, 26, 27]. Moreover, many DNN applications are implemented on mobile platforms to avoid the latency and power implications of connecting to the cloud server [28, 29]. A neural processing unit (NPU) [30] is a hardware implementation of DNNs, which is tightly integrated into the processor pipeline. Temam [31] exploits the error tolerance of DNNs to design a defect-tolerant and energy-efficient neural accelerator for heterogeneous multi-cores. Merolla et al. [32] propose a fully digital implementation of DNNs which consists of digital integrate-and-fire neurons and SRAM crossbar memory for synapses. FPGA-based hardware accelerators are another category of neural accelerators [33, 34, 35]. Lei et al. [36] use Single

Instruction Multiple Data (SIMD) architecture to speed up DNNs. Albericio et al. [37, 38] accelerate inference of the CNNs by eliminating all zero bits of the input activation and only processing essential bits. DaDianNao [39] introduces a custom multi-chip machine-learning architecture which enables high-degree parallelism of CNN/DNN based on their algorithmic characteristics. Venkataramani et al. [40], Ovtcharov et al. [41] and Hauswald et al. [42] give architecture designs on the servers and datacenters. RedEye [43] and ShiDianNao [44] place CNNs next to sensors. PRIME [45] applies Processing-in-memory to DNNs. Cambricon [46] is an ISA designed for DNNs. SCNN [47] proposes an accelerator for the compressed-sparse CNNs. Hill et al. [48] introduce a GPU framework to address irregular computations and on-chip memory bandwidth bottleneck by eliminating synapse vectors and using near-compute data fission.

Software-based: These methods focus on reducing the storage and computational resources required for DNNs by eliminating unnecessary information [14]. Han et al. [49] prune the unimportant connections and fine-tune the weights. Distillation approaches transfer the knowledge acquired by a large model to a single small model [50]. Nakkiran et al. [51] propose a rank-constrained topology to compress a trained DNN with a low-rank approximation of the weights associated with individual nodes in the first hidden layer. Yu et al. [52] propose SIMD-aware pruning to improve the performance of pruned DNNs. Ding et al. [53] use FFT-based multiplications to reduce computational and storage complexity with negligible accuracy loss. Vanhoucke et al. [54] utilize fixed-point representation of DNNs to reduce required computational resources. Batched lazy computation is also introduced to utilize the temporal locality and exploit the tradeoff between computational efficiency and execution latency. Frame skipping [55] computes the DNN once for a single input and uses the output for multiple consecutive inputs. Courbariaux et al. [56] propose Binarized NNs to improve power-efficiency. Lavin et al. [57] suggest computing small convolutions using Winograd's algorithm. Attention algorithms help DNNs to focus on critical regions of the input [58, 59]. There are also some techniques to approximate the function of layers [60].

Input-invariant approaches try to improve the efficiency and performance of a single DNN for the entire dataset. Consequently, most of the mentioned approaches are orthogonal to our techniques and could be utilized to further optimize each DNN of our proposed systems.

2.3.2 Input-Variant Techniques

Although very large and computationally intensive models are necessary to correctly classify the hard inputs, much smaller models would also work decently for most of the inputs [15]. Therefore, DNNs with early exits, such as BranchyNet [16], combine multiple DNNs in a single DNN by adding multiple branches to the model and terminating the computation as soon as the classification confidence is high enough. On the other hand, SkipNet [17] augments residual networks with gating units to decide whether to bypass the subsequent layer based on the previous layers. Another example is SlowFast networks [61], which use low frame rate for spatial semantics and high frame rate for temporal information because spatiotemporal orientations are not equally likely.

Adaptive DNNs [62, 63] train the partitioner and the networks at the same time from scratch. Hence, they require a more sophisticated training approach than the single network and use more complex loss functions. Such training complexity is time consuming and limits the applicability scope of these approaches to only small datasets. BL-DNN [64] and IDK [65] are not able to automatically select DNN configurations and use a single confidence threshold. Network cascades [66, 67, 68, 69] are limited to detection tasks.

All these methods require machine learning expertise and manual design to create DNNs with the desired behavior. For example, DNNs with early exits require the user to determine the location of each exit and design the side branch classifiers. In addition, multiple exit points result in multiple loss functions which make the training process more complicated. Conversely, our techniques simply combine off-the-shelf or automatically developed DNNs and do not require redesign or an expert to be involved. In fact, BranchyNets or MSDNets can be included as some of the DNNs that our systems consider.

CHAPTER 3

A Data Heterogeneous Multi-NN System

3.1 Introduction

To decrease energy consumption while maintaining high accuracy, we investigate ensemble methods [70, 71]. Conventional ensemble methods train multiple NNs and combine their outputs by voting or weighted averaging to obtain accuracy improvements [72, 73, 74, 75]. There are two major classes of methods to construct ensemble NNs. The first category, including bagging [76] and boosting [77], change the distribution of the original training data to create explicitly different training sets for different NNs. The second category, such as mixture of experts (MoE) [78], train different NN experts on different subsets of the training data to create implicitly different training sets [79]. The combination of multiple learners can reduce the impact of a single learner getting stuck in local optima or suffering from overfitting. It can also expand the space of the representable hypothesis. Therefore, it provides higher prediction accuracy.

Although traditional ensembles can enhance the accuracy of a learning algorithm, they are counterproductive for our goal of energy efficiency. Ensemble methods require the activation of more than one NN per input instance, which can multiplicatively increase energy consumption. Boosting/bagging generally require large numbers of learners to achieve accuracy gains [80]. Even with MoEs, subsets of training data should not be mutually exclusive to avoid limited local generalization of experts, which results in complex and power-hungry gating NNs for data partitioning

This work is a collaboration with Salar Latifi, Jiecao Yu, Daya S Khudia and Mehrzad Samadi.

and activation of multiple NNs during inference [81].

Instead of increasing accuracy by consuming extra energy, we take an opposing approach and explore multi-NN systems where only *one* learner is activated per input instance to decrease energy consumption and maintain accuracy. By carefully selecting the active learner, the prediction accuracy is not sacrificed, but at a fraction of the energy consumption of the monolithic NN. We term this approach *data heterogeneity* wherein datasets are partitioned into subsets and a specialized learner is created and optimized for each subset. The combination of specialized learners form a data heterogeneous multi-NN system. Data heterogeneity differs from traditional heterogeneous design that is computation focused, where processing elements are specialized for particular computation tasks (e.g., an audio processor or h.264 decoder). Data heterogeneous learners all perform the same task, but are optimized for different subsets of the data space.

To demonstrate the potential of data heterogeneity, we develop a systematic methodology to create a heterogeneous system of data specialized neural networks (dsNNs), or multi-dsNN, with different topologies and configurations to create an energy-efficient learner. Multi-dsNN systems consume less energy than monolithic NNs, while their accuracy remains almost the same because of three reasons. First, multi-dsNNs select and activate only a single dsNN per input instance, thereby avoiding the multiplicative increase in energy consumption of ensemble methods. Second, multi-dsNNs can employ stronger learners than conventional ensembles, thereby not leading to model underfitting. Finally, weak learners of conventional ensembles are trained on the entire training set, randomly sampled subsets or overlapped subsets. In contrast, multi-dsNN partitions the training set systematically and trains each dsNN on a specific subset of the training set. Since each subset contains instances with similar behavior patterns that are easier to learn, dsNNs tend to be smaller with similar accuracies as monolithic and ensemble NNs. Moreover, these mutually exclusive subsets of training data result in simple and power-efficient partitioning mechanisms.

This chapter makes the following contributions:

- We propose a multi-dsNN system that takes advantage of data heterogeneity to decompose the input dataset into subsets and design a set of learners customized to each specific subset.

For inference, the input is classified and the corresponding specialized learner is activated. Multi-dsNNs improve energy efficiency and performance of monolithic NNs by utilizing specialized designs.

- We demonstrate the feasibility of data heterogeneous designs by developing a systematic approach to automatically create an optimized multi-dsNN from a baseline monolithic NN.
- A selector unit is designed to learn the behavior of the training set and predict the categories of data instances in the test set. It is responsible for partitioning training and test sets into smaller subsets based on the output error distribution of a monolithic NN and assigning a dsNN to each subset. Surprisingly, multi-dsNNs are robust to moderate misclassification rates of noncomplex lightweight selectors.
- We implement a multi-dsNN consisting of two automatically generated dsNNs, $dsNN_{main}$ and $dsNN_{tail}$ on three hardware platforms (ARM Cortex-M4 Microcontroller, Kryo 280 octa-core Mobile CPU and Adreno 540 Mobile GPU) to measure energy consumption reductions and performance improvements across a set of three CNNs and six fully-connected DNNs. In comparison to a monolithic NN, on average, multi-dsNN decreases energy consumption by 61%, 53% and 55% and increase performance by $2.6\times$, $1.9\times$, and $1.8\times$ on microcontroller, mobile CPU and mobile GPU while maintaining 100% of the baseline prediction accuracy.

3.2 Background and Motivation

Different input instances of the NN show different behavior patterns. Due to the limited size of the NN, it is difficult for it to learn all behavior patterns across the input space, which is called model underfitting. To decrease the impact of model underfitting, a common solution is to increase the size of NNs to learn the behavior patterns of the entire input space. Figure 3.1a shows a typical plot of the error rate against NN size (implementation cost). Each point represents a specific NN

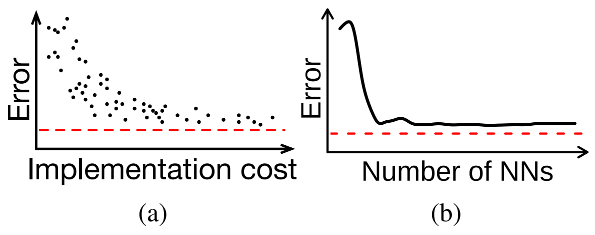


Figure 3.1: Error versus implementation cost for (a) a monolithic NN and (b) an NN ensemble.

topology. The size of the NN increases much faster than the reduction of the output error rate. Therefore, an NN with high accuracy will often have a high implementation cost which leads to low performance and poor energy efficiency.

Another well-known approach to increase the accuracy is using NN ensembles [77, 76, 70, 71], which train multiple NNs on different subsets of the training set and combine their result during inference. An ensemble NN combines a group of low-bias high-variance NNs to form a low-bias low-variance NN. Although NN ensembles result in higher accuracies, they worsen the problem of high energy consumption and low performance. Figure 3.1b shows that increasing the number of active NNs in the ensemble reaches a plateau in accuracy; however, increases the implementation cost multiplicatively.

3.2.1 Data Specialization

To develop an intuition for preventing unnecessary growth of NNs while maintaining their accuracy, we investigate various behavior patterns of input instances in MNIST dataset [82]. The fully-connected NN design containing three hidden layers of 256 neurons is obtained from Minerva [21]. We measure the output error for all test input instances. Although misclassification rate is used for measuring the accuracy of NNs, it is a binary value (correctly/wrongly classified) per input instance. Hence, it is not a proper metric to measure output quality of the NN. To solve this problem, we use 1.0 minus confidence for computing this error. As explained in Section 2.1, the confidence of NN for each input instance is equal to the maximum output of the softmax layer and is between 0.0 and 1.0. The high (low) error value means low (high) confidence and is interpreted

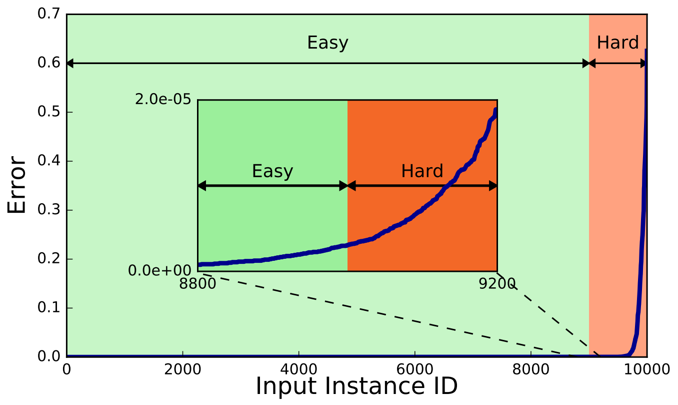
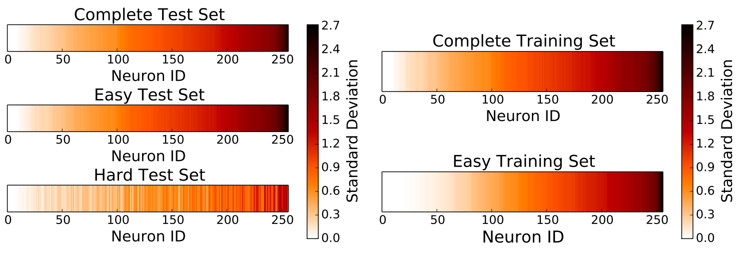


Figure 3.2: Error distribution of the test set of MNIST divided to two parts: easy-to-classify and hard-to-classify.



(a) Complete training set. (b) Easy test set.

Figure 3.3: Standard deviation of activations for third hidden layer of MNIST for two experiments. (a) Train on complete training set and test on complete/easy/hard test set. (b) Train on complete/easy training set and test on easy test set.

as hard (easy) to classify.

Figure 3.2 shows error distribution of all 10000 inputs for the test set of MNIST. The y-axis represents errors, and the x-axis is inputs that are sorted from low to high error. Therefore, the left part of the plot contains easy-to-classify inputs (e.g., high confidence), and as we move to the right part, inputs become harder to classify (e.g., low confidence). We divide the test set into two subsets: easy and hard, which contains 90% and 10% of the inputs, respectively. The same mechanism is used on the training set, and the easy and hard subsets contain 90% and 10% of inputs, respectively. The size of subsets is manually selected for this study, but we have a more systematic approach for partitioning that is described in Section 3.3.3.

Figure 3.3a presents the activations for all neurons in the third hidden layer of the NN trained on the complete training set for three different test sets to demonstrate the behaviors on different subsets of the test set. The x-axis of each plot shows neurons from 0 to 255. Colors represent standard deviation of neuron activations for each test set. If the standard deviation of a neuron activation is high, that neuron is more sensitive to different inputs and can be used to distinguish and classify various inputs more effectively. In other words, the dark neurons in the plots are more effective than the light ones.

The top graph represents the result of testing on the complete test set, the middle one on the easy test set, and the bottom one on the hard test set. Neurons in all three plots have the same order and are sorted based on the standard deviation of their activations in the top plot. As expected, the top graph and the middle one are very similar because the easy part is the major subset of the test set and contains low-error instances. On the other hand, the bottom graph has a smaller subset of dark neurons, which means it contains less effective neurons. In addition, the distribution of important neurons is very different from two other graphs. Hence, this NN is not properly designed and trained for hard-to-classify instances.

Figure 3.3b has the same format as Figure 3.3a and compares the result of using the complete and limited training sets to demonstrate the potential benefits of specializing an NN for a subset of data. The top graph is related to training on the complete training set and testing on the easy test set. And, the bottom graph represents training on the easy training set and testing on the easy test set. As depicted, the light part of the bottom graph is considerably larger than the top one while the dark part has almost the same size. In other words, the NN with easy training and test set has more unimportant neurons but almost the same number of important ones. Therefore, it is possible to reduce the NN size by omitting the unimportant neurons because their impact on the output is minimal.

Summary: The subset of neurons that are important for classifying easy inputs is different from the subset for hard ones. Consequently, to cover all the input space, a monolithic design must grow. And, the larger NN is still not well-designed for the harder-to-classify inputs. On the other hand, the monolithic model is an overdesigned NN for easy inputs, and we could classify

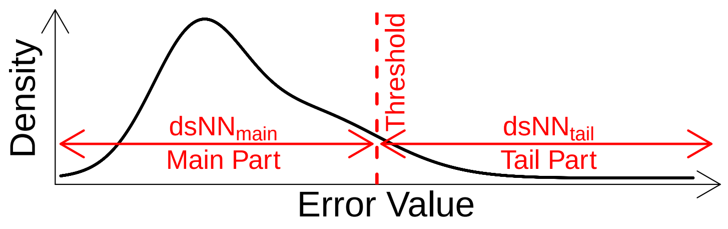


Figure 3.4: Dividing input space based on the output error distribution to two different parts. The main part is much dense and contains low-error instances. The tail part is sparse and contains instances with high errors.

them by utilizing a smaller NN without any loss of accuracy. Therefore, partitioning datasets and designing specialized NNs for each subset could help reduce the size of NNs while maintaining their accuracy.

3.3 Data Heterogeneous Multi-NN

In this chapter, we propose an energy-efficient system of data specialized neural networks (dsNNs), which leverages data heterogeneity to increase energy efficiency. After describing the concept of data heterogeneity and the inherent potential of NNs for this technique, the main components of a multi-dsNN system are described. Then, we go over various implementation decisions, design choices, and parameters to come up with an automatic method of constructing dsNNs.

3.3.1 Multi-dsNN System

Figure 3.4 shows the error distribution of a typical NN on a dataset. The majority of the dataset has a relatively small error. However, parts of the input set still suffer from high output errors. As explained in Section 3.2.1, training a monolithic NN to learn all these behavior patterns results in an unnecessary growth of the NN. This data heterogeneity is a great potential for replacing the monolithic NN with an energy-efficient data heterogeneous multi-NN. In contrast to conventional heterogeneous systems, which dedicate specialized processors to separate tasks, each NN of our system is designed to focus on a subset of input data, which contains input instances with sim-

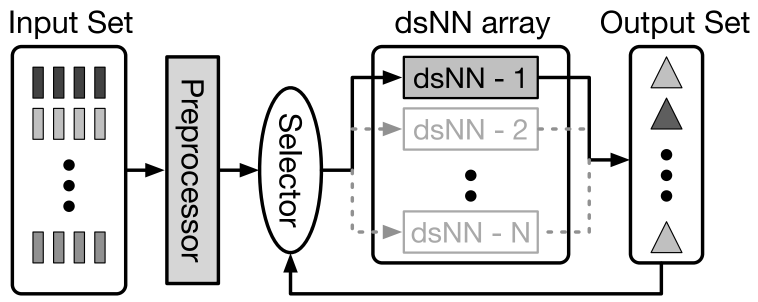


Figure 3.5: Overview of a general multi-dsNN. It contains an optional preprocessing unit for feature extraction. Then, the selector unit activates the most proper dsNN based on the features of the current input.

ilar behaviors. All NNs are then combined to create a strong learner which has a lower energy consumption.

Specializing NNs for different subsets of the input space provides a system with only one active dsNN per input instead of a monolithic NN or a conventional NN ensemble. This multi-dsNN system delivers the following potential benefits:

Lower cost: Each dsNN does not need to cover a broad range of data. Therefore, it can be constructed with fewer layers and neurons per layers than a monolithic NN. One active dsNN trained on mutually exclusive subsets of training data results in lower costs than conventional ensembles.

Better performance: In addition, a smaller NN needs a lower amount of computation, thus it will generally be higher performance depending on the specific hardware.

Higher accuracy: A single NN is no longer used for various inputs with different patterns. Thus, in some cases a multi-dsNN gains accuracy improvements over a monolithic NN.

Figure 3.5 shows an overview of a general multi-dsNN which consists of three major components: preprocessor, selector unit, and an array of dsNNs. The preprocessor extracts useful features of input to provide the selector unit and dsNNs with more meaningful data for the classification task. Instead of a monolithic NN, a set of dsNNs is available in the dsNN array. For each input, the selector unit chooses the best dsNN to become active and sends the current input to that. Then, the dsNN computes the final output of the system and sends the feedback to the selector unit for online modifications.

3.3.2 Preprocessor

The preprocessor is trained on the training set to become capable of extracting useful features of the input. In other words, it is responsible for converting the original input to the practical input for the selector and dsNNs. In our design, if the monolithic NN is a CNN, the convolutional part will remain as the preprocessor unit, and the fully-connected part will be replaced by a selector and an array of dsNNs. On the other hand, if the monolithic NN is a fully-connected DNN, the multi-dsNN system will not use any preprocessor, and the original input data will be sent to the selector and dsNNs.

3.3.3 Input Partitioning

The next step in implementation of a multi-dsNN is dividing the input dataset into subsets. First, we need to define a **mechanism for partitioning**. Three alternatives are considered:

- **Based on Input:** The training dataset is expected to be representative of the test dataset. Hence, we can extract the distinguishing features of the training inputs and use them to partition the test inputs.
- **Based on Output History:** If the distinguishing features are extracted by examining the exact training outputs, a record of limited number of previous test outputs will provide good clues for partitioning the current test input.
- **Based on Simultaneous Multiple Outputs:** Another partitioning mechanism is running multiple dsNNs per test input, comparing their outputs, and sending it back to the selector unit as feedback. This feedback could be used to modify the behavior of the selector. If all dsNNs converge to the same results, we can predict that current dsNNs are working properly, else we need to substitute some of them with other available dsNNs.

Although partitioning based on the output history produces acceptable results for applications where there is a correlation between different input instances (such as different tiles of an image), it does not necessarily work well for applications with isolated input instances. On the other hand, partitioning based on simultaneous multiple outputs makes the system complex and inefficient because activating multiple dsNNs per input instance is power hungry and makes the selector more complicated. Consequently, partitioning based on input is chosen for the multi-dsNN.

After specifying the partitioning mechanism, it is necessary to determine the **partitioning granularity**. Different input instances of a dataset or various input elements of an input instance might participate in isolated computations and have different distributions. Our goal is to partition them based on their features to use customized dsNNs for each subset. Input sets could be divided to subsets based on two approaches:

- **Element Partitioning:** If we have N input elements per input instance, it will be possible to form K groups in a way that each contains correlated elements. However, this approach needs K active dsNNs for inference.
- **Instance Partitioning:** Each input instance consists of input elements, so input instances could be classified based on their values. In contrast to element partitioning, this method needs only one active dsNN for inference.

As we mentioned before, a single active dsNN is energy-efficient and could be handled by a simple and low-cost selector unit. Consequently, we limit ourselves to instance partitioning which results in design options that consist of one active dsNN for inference.

Partitioning input instances based on the output error distribution is an input-based instance partitioning mechanism that is used by multi-dsNN for dividing training and test sets to multiple subsets. After examining the error distribution of different output elements of various applications, we find that in general the error is distributed similar to Figure 3.4. The distribution of the error is composed of dissimilar parts, the main and tail. The main part contains lots of low-error data points spread on a relatively limited interval. But the tail part is a wider range consisting of sparse high-error data points.

Using a monolithic NN is not a good solution because each part tries to modify the NN to perform more accurately for its own data. In other words, the data instances in the main part might change the weights of the NN in a way that decreases the quality of final answers for the tail part and vice versa. Even in the case of using ensembles, each NN suffers from the same problem because the training data is randomly sampled or subsets of training data have overlaps. Hence, an input-based mechanism could partition data based on the error distribution to design one dsNNs for each part. Dedicating specialized NNs to the tail and main part reduces the size of each dsNN and possibly increases its accuracy. It is possible to find proper thresholds for dividing the error distribution to different parts by examining the error distribution produced by different instances of the training set. Then, we are able to divide the training set into subsets, and utilize them to train different dsNNs. Finally, the same mechanism is used to dispatch test input instances to the proper dsNN during inference.

3.3.4 Selector Unit

The selector unit uses a filter to choose the appropriate dsNN during inference for each test input. The feature used for partitioning is the error distribution. However, computing the final error for each input instance needs exact outputs, and computing exact outputs by NNs is impossible. Therefore, we need to predict the amount of error based on input elements.

Since the selector unit should be active for all input instances, it is critical to utilize a lightweight predictor. However, making this unit too simple might cause a high rate of misclassification which means sending data to wrong dsNNs. High misclassification rates result in less accurate results and more expensive dsNNs. After comparing different methods of classification, we find that using a decision tree is the best fit for our system. A decision tree is cheap in terms of energy consumption and accurate enough to keep misclassification rate low. The decision tree is trained using the training data to predict the error for each test input instance and choose the most suitable dsNN in the array to activate.

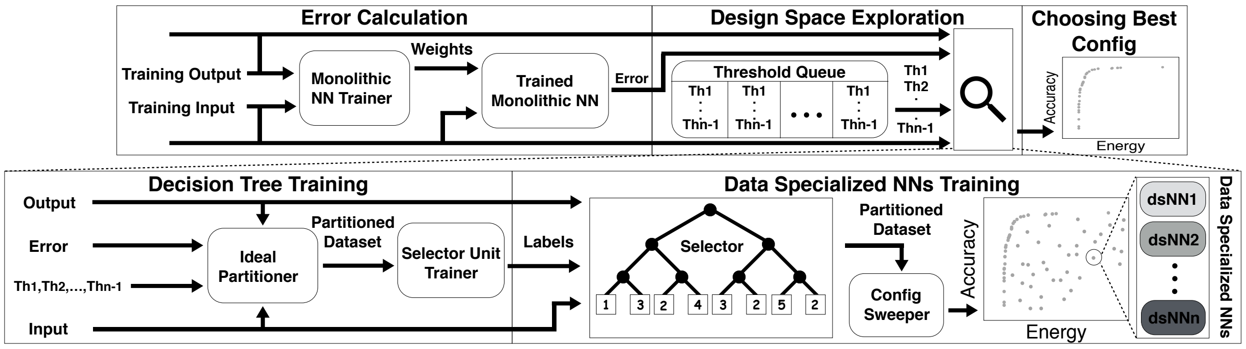


Figure 3.6: Finding the optimal configuration of the multi-dsNN. The training data is sent to the monolithic NN to obtain error values. Then, design space exploration uses error values and training data to evaluate different partitioning thresholds and dsNN topologies. Finally, the best configuration is selected considering available resources and limitations.

3.3.5 dsNN Array

In design space exploration, the most appropriate dsNN topology for each subset and its weights are saved in the dsNN array. During inference, the selector examines each input and picks the most suitable dsNN from the dsNN array. The activated dsNN processes inputs and computes the final output.

3.4 Training and Inference

After designing a multi-dsNN, we need to configure and train it. Figure 3.6 depicts different stages of configuring and training. First, we use the training dataset and monolithic NN to compute error values for each training input instance. Then, different combinations of partitioning thresholds and dsNN topologies are swept to identify all optimal design points. Each design point is a trade-off between accuracy and energy consumption. At the end, we choose the design point which meets the constraints of our problem such as maximum energy consumption and minimum output quality. Note that the size of the design space can quickly explode, so a heuristic is used to limit the search as described in Section 3.5.7.

3.4.1 Error Calculation

We use the training dataset to train the monolithic NN based on the backpropagation method. Then, we send back the training input to the monolithic NN to compute outputs for the training set. After that, error values are computed based on the confidence of the NN for each input instance and saved in an error file. Then, we send the training dataset augmented with error values to the next stage as shown in the upper left portion of Figure 3.6.

3.4.2 Design Space Exploration

Threshold Queue: To find the best configuration, we need to sweep the design space. In addition to the data provided from the previous stage, design space exploration needs partitioning thresholds to divide the training set. All possible combinations of partitioning thresholds to divide training set to N subsets for N dsNNs are predetermined in the threshold queue. We need to run the decision tree training and data specialized NNs training steps for each entry of the queue.

Decision Tree Training: The thresholds are used to partition the training set ideally based on the error file. Then, the training input and partitioning information are used to train a decision tree with a proper depth as the selector unit.

Data Specialized NNs Training: For each threshold set, we try all different topologies for each dsNN to find the optimal NN and train them. Although it seems more reasonable to train dsNNs by the ideally partitioned data, this approach might cause each dsNN to become too specialized. Considering the fact that the decision tree is not able to partition the test data perfectly, it is better to use the trained decision tree to partition the training data. Then, use this imperfect classified training data to train each dsNN to help the system in better tolerating the misclassification rate of the selector unit. Ideal classification in the training phase might cause the system to react undesirably to the misclassified test inputs.

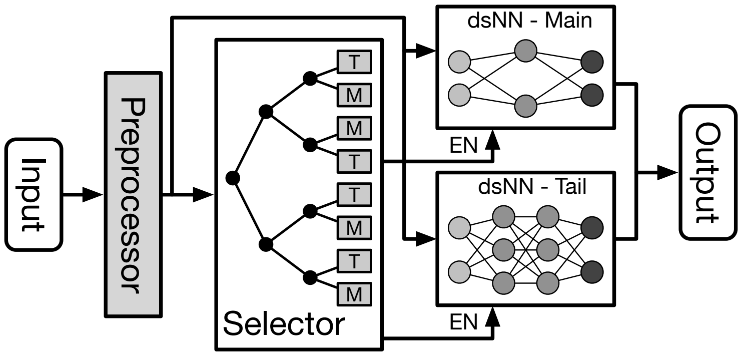


Figure 3.7: 2-way multi-dsNN with decision tree as the selector and two dsNNs trained on main and tail parts of the training set.

3.4.3 Choosing the Best Configuration

After training each dsNN with its specific dataset, that dataset is reused to compute the accuracy of the corresponding dsNN. At the next step, we compute the weighted sum of energy and accuracy of the dsNNs to find out the final energy and accuracy for each combination of selector configuration, dsNN topologies and partitioning thresholds. Finally, we plot the Pareto frontier of accuracy-energy and pick the thresholds, related dsNN topologies, and decision tree configurations to meet the energy and quality constraints of the application. In fact, the Pareto frontier provides users with a wide range of trade-offs between energy efficiency and accuracy.

3.4.4 Inference

Figure 3.7 shows the inference of a 2-way multi-dsNN. During inference, the preprocessor transforms input elements. Then, the selector predicts the category of the input instance and activates the proper dsNN. The active dsNN processes the input data and provides the final outputs.

3.5 Evaluation

To maintain a simple design space, our experiments are limited to two dsNNs (i.e., single partitioning threshold). A case study of exploring multi-dsNNs containing more than two learners is explained in Section 3.5.6. We call one of the dsNNs, which is designed for low-error input instances, $dsNN_{main}$ and the other one $dsNN_{tail}$. We limit each dsNN to at most three hidden layers and 4096 neurons per layer. In addition, the partitioning queue is occupied by nine different partitioning thresholds. To limit the number of trainings required for designing our dsNNs, a heuristic search is utilized instead of the exhaustive search for design space exploration. This heuristic is explained in detail and compared to brute-force search in Section 3.5.7.

3.5.1 Experimental Methodology

Hardware Platforms: We evaluate multi-dsNN on three different hardware platforms: microcontroller, mobile CPU, and mobile GPU. They represent different kinds of processors which are widely used in embedded and mobile systems.

- Microcontroller: An ARM Cortex-M4 microcontroller available on STM32 Nucleo board, which has 128KB of SRAM and 512KB of flash storage is used to run benchmarks. Matrix-vector/matrix multiplication libraries from ARM are used to implement fully-connected layers.

- Mobile CPU: A Qualcomm Snapdragon 835 chipset is used which consists of four high performance and four energy efficient Kryo 280 cores (semi-custom built cores based on ARM license) and 4GB of DDR4 RAM. To run the benchmarks, we use CNNdroid [83] library, which utilizes the Android Renderscript framework to accelerate matrix multiplications. Qualcomm's Trepn Profiler is used for energy/performance measurements.

- Mobile GPU: An Adreno 540 GPU included in Snapdragon 835 is used for evaluation on mobile GPUs. Similar to the mobile CPU, the CNNdroid library and Trepn Profiler are used for implementation and analysis.

Benchmarks: For evaluating multi-dsNN, seven classification datasets from various domains are used. A brief description of each benchmarks is presented in Table 3.1. The datasets consist of ImageNet [84], CIFAR-10 [85], MNIST [82], 20NG [86], Forest [87], Reuters-21578 [88, 89]

Table 3.1: Application, topology, and accuracy of baseline NNs.

	Dataset	Description	Input	Output	Topology	Accuracy(%)
CNN	**CIFAR-10**	Image classification	1024	10	C1->C2->C3->64	80.48
	ImageNet	Image classification	9216	1000	C1->C2->C3->C4->C5->4096->4096	56.90
	MNIST	Handwritten digits recognition	800	10	C1->C2->500	99.19
FC	**20NG**	Newsgroup posts for text classification	21979	20	64->64->256	82.00
	CIFAR-10	Image classification	3072	10	4000->1000->4000	55.95
	Forest	Classifying the forest cover type	54	8	128->512->128	71.42
	MNIST	Handwritten digits recognition	784	10	256->256->256	98.31
	R52	News articles for text categorization	2837	52	128->64->512	94.19
	WebKB	Web pages from different universities	3418	4	128->32->128	90.07

and WebKB [90]. For top three rows of Table 3.1, which are CNNs, input data is preprocessed by convolutional layers of the monolithic NN, and the fully-connected part is replaced by a multi-dsNN. For other six fully-connected DNNs, no preprocessing is required, and the input is directly fed to the dsNN. AlexNet [91], ConvNet [92] and LeNet-5 [93] are our monolithic baseline CNNs. And, monolithic NN topologies and accuracies for fully-connected DNNs are obtained from Minerva [21] and Lin et al. [94].

In Table 3.1, the input column represents the number of input elements in each input instance after preprocessing. The output column specifies the number of classes. The topology column shows the optimal topology of the baseline NN for each benchmark. For example, C1->C2->500 for MNIST states that the CNN has two convolutional layers (Convolution+Pooling+ReLU) and one fully-connected layer of 500 neurons. The final column presents accuracy (100% minus misclassification rate) of the baseline NN after testing on the test dataset. Accuracy values could be slightly different from prior work because they depend on many hyperparameters including random initialization of weights, learning rate, weight decay, etc.

Training: We use Caffe [95] framework to implement, train and test our NNs. In addition, the Scikit-learn [96] library is used for implementing, training and evaluating the decision tree as the selector unit.

Output Accuracy: We use 100% minus misclassification rate to measure the accuracy of NNs. To show the improvements over the baseline NN designs, we report accuracy for each benchmark relative to its baseline from Table 3.1.

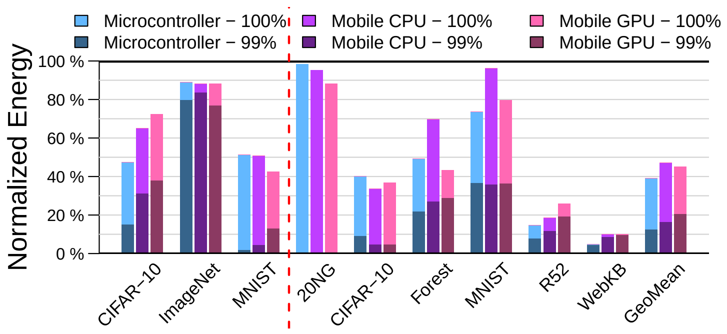


Figure 3.8: Normalized energy of multi-dsNN on different hardware platforms for 99% and 100% normalized accuracy.

3.5.2 Energy Consumption Reduction

To demonstrate energy savings of our approach, we evaluate two multi-dsNN designs per hardware platform for each benchmark. The first design reduces energy consumption while maintaining 100% accuracy of the monolithic NN. And, the second one relaxes the minimum acceptable normalized accuracy to 99% for situations in which the user is willing to sacrifice a tiny amount of accuracy for higher energy savings. The only exception is 20NG, for which there is no point on the Pareto frontier between 99% and 100% accuracy. Hence, we only report the savings for 100% accuracy.

Figure 3.8 demonstrates the energy consumption of the multi-dsNN designs for the nine benchmarks on the three hardware platforms in the case of 99% and 100% accuracy. In each group, the left bar represents microcontroller results, the middle bar reports the mobile CPU, and the right bar shows the mobile GPU. All energy values are normalized to the energy of the monolithic NNs on each platform. Except for 20NG, each bar consists of two overlapped bars: the bright bar, which represents the energy of multi-dsNN for 100% accuracy, and the dark bar, which shows the energy while accuracy is 99%. Hence, the visible bright part depicts additional energy savings by relaxing accuracy constraints.

Based on Figure 3.8, multi-dsNN obtains energy savings for all benchmarks on all three hardware platforms while maintaining 100% accuracy of the monolithic NN. These energy savings fall into the range of 2% to 95%. On average, multi-dsNN decreases energy consumption to 39%,

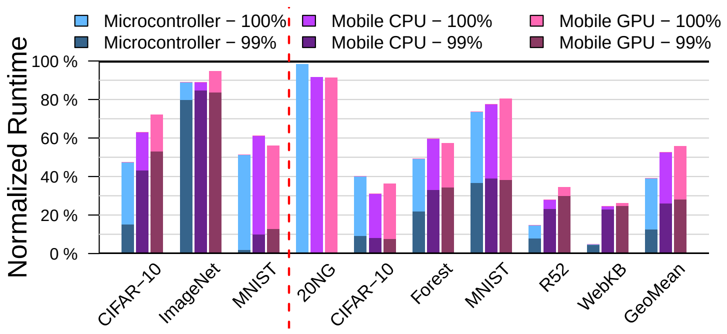


Figure 3.9: Normalized runtime of multi-dsNN on different hardware platforms for 99% and 100% normalized accuracy.

47% and 45% for 100% accuracy and to 13%, 17% and 21% for 99% accuracy on microcontroller, mobile CPU and mobile GPU, respectively. The main reason for these significant energy savings is that the classification task is divided into two parts. First, the input instances are classified to high-error and low-error. Then, the proper dsNN produces the final output. The first part is offloaded to a lightweight decision tree, which is much less energy consuming than the monolithic NN. In addition, the second part is less complex than the original task, therefore requires a smaller NN than the monolithic one for both main and tail for all benchmarks.

20NG achieves the lowest energy savings. Since the number of input elements in the 20NG dataset is at least two times bigger than other benchmarks, multi-dsNN training cannot find an accurate splitter and chooses a small decision tree to improve energy savings. The average height of the tree for 20NG is about seven, which is the lowest in comparison to other benchmarks. This decision tree sends a considerable number of $high\_error$ training inputs to $dsNN_{main}$ and vice versa. Therefore, subsets of training data become similar to the original training set, so both $dsNN_{main}$ and $dsNN_{tail}$ have almost the same size as the monolithic NN.

3.5.3 Performance Improvement

Figure 3.9 shows the normalized execution time of the multi-dsNN on the three hardware platforms across the nine benchmarks for 99% and 100% accuracy using the same format as Figure 3.8. On average, multi-dsNN reduces execution time to 39%, 53% and 56% of the monolithic NN for 100%

accuracy and to 13%, 26% and 28% for 99% accuracy on microcontroller, mobile CPU and mobile GPU, respectively. Replacing the monolithic NN by multi-dsNN improves performance for all benchmarks on all hardware platforms even for 100% accuracy. The lowest speedup is $1.02\times$ while the highest one goes up to $21\times$.

By comparing the difference of bright and dark bars for different benchmarks, we can observe that the dsNN related to CNN of MNIST achieves the highest performance improvement when the accuracy requirement is relaxed from 100% to 99%. The monolithic NN for this benchmark consists of a relatively small input layer connected to a relatively big hidden layer. The small input layer helps the decision tree to partition the input accurately and efficiently, and the big layer provides more opportunities for reducing the size of each dsNN, resulting in considerable performance gains. Another interesting observation is that WebKB's performance is not very sensitive to this constraint relaxation. As depicted, multi-dsNN with 100% accuracy results in the highest speedup, which means the highest size reduction, for WebKB. Consequently, there is not much room for reducing the size of dsNNs further when switching to 99% accuracy, and most of the performance improvement comes from changing the partitioning threshold to send more input instances to the smaller dsNN.

3.5.4 Comparison with NN Ensembles

In order to compare our proposed solution with traditional NN ensembles, we compare accuracy and energy efficiency of the multi-dsNN with both boosting and MoE.

Monolithic Boosting: We use SAMME method [97], which is an extension of AdaBoost [98] algorithm to multi-class cases, with NNs used as base learners [75, 99, 100]. The most common choice for base learners of an ensemble is the monolithic baseline NN. Then, these learners are trained sequentially on different subsets of the training set. Although this approach leads to accuracy improvements, adding each learner increases the overall energy of the system multiplicatively.

Energy-Conscious (EC) Boosting: Given our focus on energy efficiency, we compare multi-dsNNs to NN ensembles with a limited computation budget. This budget is determined by the number of FLOPs of the monolithic NN inference per input instance. As a result, it is crucial to choose weaker NNs as base learners so as not to exceed the predefined energy limits while adding more learners.

Energy-Conscious (EC) MoE: We utilize a modified version of MoE which focuses on energy efficiency rather than accuracy. There are two different approaches in order to come up with the EC MoE. First, the size of each expert could be reduced to contain less number of neurons and improve energy efficiency. Second, reducing the number of active experts in MoE helps decrease energy consumption. Similar to the EC boosting, the energy budget is determined by the FLOPs of the monolithic NN.

In our experiments for monolithic boosting, ensembles containing 2 to 100 base learners with the same configuration as the monolithic NN are trained and evaluated. In this approach, the highest accuracy is chosen as the optimum design. On the other hand, the EC boosting method can utilize any arbitrary NN architecture for the base learner as long as its overall number of FLOPs stays lower than or equal to the monolithic NN. Consequently, the maximum number of learners will be derived from the specified computation budget and the base learner's configuration, capped to 100. For the EC MoE experiment, the capacity of the model is increased by containing 100 experts. However, the number of active experts per input instance is determined by the computational budget, which is limited to the FLOPs of the monolithic network. To explore the design space, the topology of each expert and the number of active experts are swept in a range that the overall computational constraints are met.

Figure 3.10 shows comparison of multi-dsNN with monolithic boosting, EC boosting and EC MoE for CNN of CIFAR-10 benchmark. The y-axis represents the accuracy of the learner normalized by the accuracy of the monolithic NN. And, the x-axis is the average number of FLOPs normalized by the baseline NN to compare energy consumption of different designs in a hardware-agnostic manner. For multi-dsNNs, the amount of the dynamic energy consumption depends on the size of each dsNN and the depth of the decision tree. Moreover, the partitioning threshold and misclassification rate of the selector have an impact on the amount of energy because it determines

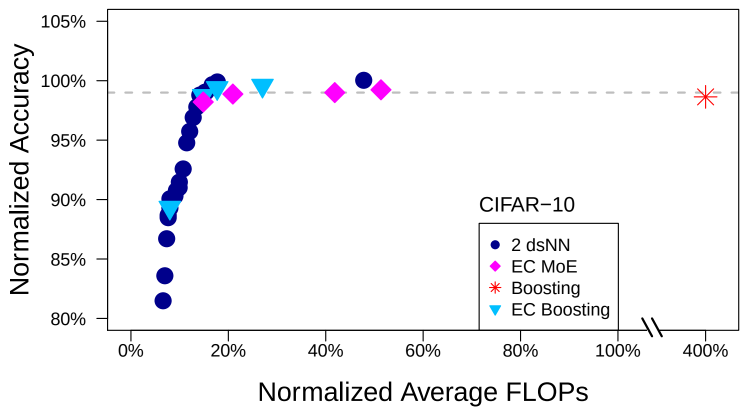


Figure 3.10: Accuracy-computation trade-off for multi-dsNN, EC boosting, EC MoE versus and monolithic boosting.

how many input instances run on each dsNN. The gray dashed line indicates accuracy of 99%. Figure 3.10 consists of a Pareto frontier of the multi-dsNN depicted by circles, a Pareto frontier of EC boosting plotted by triangles and a Pareto frontier of EC MoE plotted by diamonds. The final plotted data is a single star point for monolithic boosting, which requires a discontinuity of the x-axis to be displayed.

Based on the plot, multi-dsNN provides the most energy-efficient design. For 100% accuracy of the monolithic NN, multi-dsNN reduces computation by 52% while EC boosting and EC MoE does not generate any point with the same accuracy as baseline. Even if we decrease the minimum acceptable accuracy threshold, multi-dsNN provides more efficient designs. If we relax the minimum required accuracy of applications to 99%, multi-dsNN could reduce the average number of FLOPs by 2% more than EC boosting and 27% more than EC MoE while adding only 0.94% error in comparison to the monolithic NN.

Although the focus of multi-dsNN is improving energy efficiency for the same accuracy as the monolithic NN, it provides more accurate design points than EC boosting and EC MoE. An interesting observation is that monolithic boosting does not provide any accuracy improvement. The problem is that by moving forward in monolithic boosting, the training set for each iteration is a sample of the original training set, which does not even cover 10% of the training set in more than half of the iterations. Considering the large number of input neurons in this benchmark, these

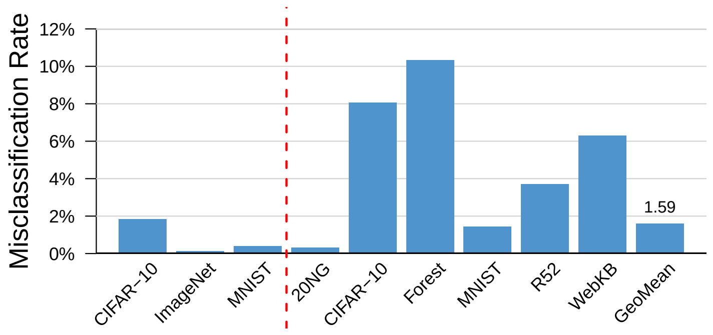


Figure 3.11: Misclassification rate of the selector.

limited subsets of training data result in overfitting, which causes higher error rates.

3.5.5 Partitioning Quality

We use an incomplete decision tree with a maximum average depth of 79, limited to 18K nodes and 18K-1 leaves, to partition the input dataset to two subsets. As with any machine learning technique, the output of this decision tree is not 100% accurate and causes some misclassifications. In other words, during inference, some points might migrate from main to tail, and vice versa. These migrations require more complex dsNNs to handle inputs from both parts, which result in less energy savings. Furthermore, each dsNN might produce high errors for misclassified inputs because that dsNN is specifically designed for dealing with a part of input instances. However, since the decision tree is active during training, dsNNs are trained to tolerate moderate misclassification rates. Hence, replacing the decision tree with an oracle classifier does not change energy savings by more than 10%.

A misclassification happens whenever, for an input instance, the decision tree sends the input to one dsNN which classifies the input wrongly while the other dsNN is able to classify this input instance correctly. We divide this number by the number of all input instances to calculate the misclassification rate for the decision tree. After running multi-dsNN with different misclassification rates of the selector unit, we find that keeping this rate less than 15% helps our system to achieve acceptable accuracy and improve energy efficiency. Figure 3.11 shows the misclassification rate of the decision tree for each benchmark. As you can see, the maximum misclassification rate of

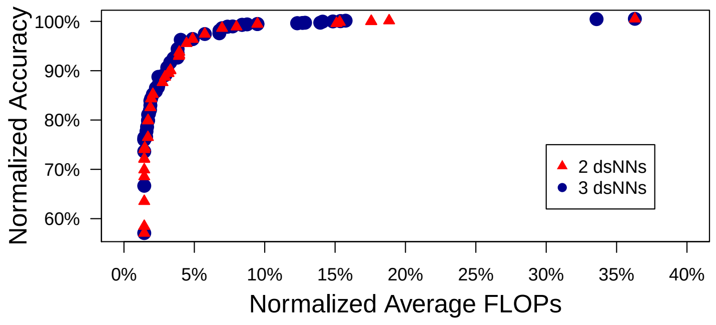


Figure 3.12: Comparing accuracy and average FLOPs of 2-way multi-dsNNs and 3-way multi-dsNNs for R52.

our system among all benchmarks is about 10%. And, the average misclassification rate is about 1.6%.

Although 20NG and ImageNet have the largest number of input elements among all benchmarks, their misclassification rate is the lowest. As explained in Section 3.5.2, both $dsNN_{main}$ and $dsNN_{tail}$ are very similar to the monolithic NN for these benchmarks. Therefore, it is unlikely for these similar NNs to produce different results for one input instance. Consequently, the misclassification rate goes down. The important conclusion is that even if the decision tree does not work precisely for a benchmark because of its huge input layer, multi-dsNN training automatically chooses complex NN topologies to decrease misclassification rate and maintain the prediction accuracy while not consuming more energy than the monolithic NN.

3.5.6 More than Two dsNNs

A multi-dsNN is not necessarily limited to two dsNNs. Figure 3.12 depicts accuracy-computation Pareto frontier of R52 for two different multi-dsNN systems. The first one contains two dsNNs, and the second one consists of three dsNNs. The y-axis shows accuracy of each configuration normalized to the baseline NN. The x-axis is the normalized average number of FLOPs for different configurations. The plot is limited to the normalized FLOPs less than 40%, where all the demonstrated points need less computation than the monolithic NN. Based on the number of points on each Pareto, for R52, switching from two dsNNs to three dsNNs makes the multi-dsNN more flexible, which enables the user to choose desirable designs at a finer granularity. Moreover, as depicted on Figure 3.12, for some points, adding the third dsNN makes the accuracy and efficiency of the system slightly better. But, overall gains are negligible when the number of dsNNs is increased beyond two. We confirmed that this trend held for a subset of the other benchmarks as well.

3.5.7 Heuristic Search

An exhaustive search for a 2-way multi-dsNN in our design space, which is limited to three hidden layers of 4096 neurons and nine partitioning thresholds, requires more than 10^{12} different dsNNs to be trained. Since it is not practical to train this many dsNNs, it is necessary to use heuristic search techniques to explore the design space. Our approach is composed of two alternative search strategies that are detailed below: energy and accuracy optimization. To select which heuristic to use, the system designs dsNNs identical to the baseline NN but with half of the number of neurons in the hidden layers. If the normalized accuracy of this multi-dsNN is more than 100%, energy optimization continues. Otherwise, accuracy optimization is called.

For energy optimization, we modify the random-restart hill climbing technique [101]. The initial point for the algorithm is a dsNN without any hidden layers. For each iteration, if the accuracy is not improving by increasing the size of the dsNN, there is no hidden layer, or the last hidden layer is full (has 4096 neurons), the algorithm will jump to another configuration. Otherwise, it will take a normal step and multiply the number of neurons in the last hidden layer by two. Instead of random-restarts, we use deterministic jumps. A jump is defined as multiplying the number of neurons of the last nonfull hidden layer by two or adding a new hidden layer containing one neuron after the input layer in the case that all hidden layers are full. When there are no more jump options, then the search concludes.

Accuracy optimization uses the same approach as energy optimization with the following changes. The initial point is the baseline dsNN. It will jump if the normalized accuracy goes below 100% or the first hidden layer is empty. The normal step is reducing the number of neurons

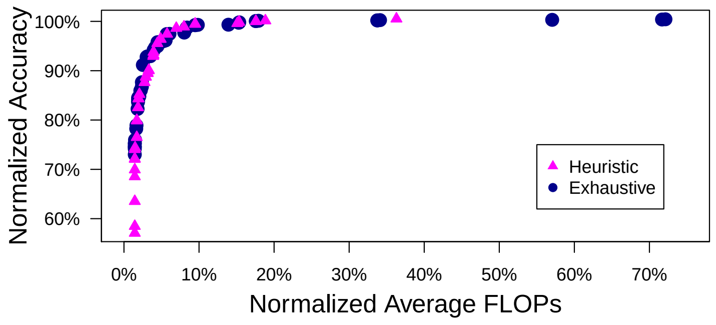


Figure 3.13: Comparison of exhaustive and heuristic search.

by 5% in the first hidden layer. A jump is reducing the number of neurons of the first nonempty hidden layer that does not result in reducing normalized accuracy by more than 5%.

This algorithm reduces the number of trainings to less than 3000 dsNNs per benchmark. Figure 3.13 compares the accuracy-computation Pareto frontier obtained from our heuristic and an exhaustive search for R52. As illustrated, the difference of these two plots is negligible for optimum points on the top-left part of the figure, which have about 100% accuracy while requiring the minimum FLOPs. Moreover, the heuristic does not generate the inefficient points on the top-right part of the graph, which have about the same accuracy as the optimum points while using three to four times more FLOPs.

3.6 Related Work

Meta learning and ensemble methods: These algorithms are proposed to combine different learning algorithms to shape a stronger learning system. Generally, they can be classified to three different groups [102]:

The first group targets the variations in data and includes bagging [76, 103] and boosting [99, 104, 100, 105, 66]. These methods use several learners, such as NNs, and train them on different subsets of the dataset. For inference, all of the base learners are activated, and final output is produced via a voting scheme. In bagging, each base learner is trained on a uniformly sampled version of the original dataset, while in boosting, the training set is sampled according to the weaknesses of the previous learners. Because of high computation cost of this group, Zhou et al. [99] try to eliminate unimportant NNs, but all remaining NNs are still active for inference.

The second group aims at variations of learners and consists of stacking [70], cascading [106, 107, 108, 109], delegating [110] and arbitrating techniques [111]. Similar to bagging and boosting, stacking uses a number of learners, but instead of voting, another learner is used to predict the final output based on the predictions of learners. Cascading is similar to boosting in the aspect of training set sampling, but it deploys different learners with more complex topologies in further steps. Delegating is similar to cascading with the difference that each learner is only trained on the weaknesses of the previous learner. Arbitration also uses different learner topologies as cascading, but all of them are trained on the original dataset.

The last group focuses on both data and learner variations and includes meta decision trees (MDT) [112] and mixture of experts (MoE) [78]. Unlike the other groups which require activation of all or a subset of the base learners, MDT deploys a decision tree to select the suitable classifier for each test data. In MDTs, the training set is randomly partitioned into different groups using stratified sampling, and each base learner is trained on different combinations of subsets to generate the metadata for training of the decision tree. MoEs [113, 80, 114, 115] are based on conditional computation methods [116, 117, 118] which assign an activation function per neuron or block of neurons to activate only a part of the NN based on the input features. Hence, each NN is specialized on a subset of input dataset. For inference, a classifier activates all or some of them to make the prediction.

In all mentioned methods, either the computation cost is high due to the activation of multiple learners, or the training is based on random sampling and does not fully exploit different inherent patterns of the data space. In contrast, dsNNs are specialized learners and systematically trained to exploit available data heterogeneity in the input space, which results in activating only a single specialized learner and higher energy efficiency while maintaining the original accuracy.

3.7 Conclusion

In this chapter, we propose a system of *data specialized neural networks (dsNNs)* to increase energy efficiency of NNs. A multi-dsNN consists of multiple simple and specialized NNs, which are customized for different subsets of the input space. In contrast to conventional NN ensembles, multi-dsNN selects and activates a single dsNN per input instance dynamically, and that single activated dsNN is not responsible for the entire range of input instances. Hence, each dsNN is less complex than the monolithic NN, which results in considerable energy savings. In some cases, this specialization can reduce the interference between different input behavior patterns to provide a higher accuracy while consuming less energy. Compared to conventional monolithic NNs, a 2-way multi-dsNN reduces the energy consumption by an average of 61%, 53% and 55% and increases the performance by an average of $2.6\times$, $1.9\times$ and $1.8\times$ on microcontroller, mobile CPU and mobile GPU with 100% accuracy of the baseline.

CHAPTER 4

An Intelligent Hybrid Server-Edge Deep Learning System

4.1 Introduction

Due to the prohibitive energy and latency of communicating data to cloud servers, there is a growing trend towards CNN execution on edge devices themselves. The processing power of modern edge devices has increased to help them handle more computationally-intensive applications [119]. For example, the Qualcomm Snapdragon 835 with machine learning capabilities enables running some trained models directly on the mobile device [120]. However, the size and complexity of CNNs are increasing more rapidly to improve their accuracy and functionality than the hardware capabilities, which results in computations with energy requirements beyond device's battery constraints even with hardware accelerators [49].

To address the shortcomings of cloud-only and mobile-only approaches, two categories of hybrid cloud-edge systems have evolved, which partition the computation between cloud servers and edge devices for maximizing performance and efficiency [121]. First, applications, such as Apple's Siri [122], Google Assistant [123], Amazon Alexa [124], and Microsoft Cortana [6], consist of specialized speech recognizers for keyword spotting on the device and automatic speech recognition, natural language interpretation, and various information services on the cloud. In these applications, the computation is partitioned manually during the design process, and servers and

This work is a collaboration with Salar Latifi, and Pedram Zamirai.

devices are responsible for different tasks. In the second category, since the energy efficiency and data transfer rates vary for different wireless technologies, edge devices dynamically decide whether to offload and which parts of the computation need to be offloaded to maximize the energy efficiency and performance [125]. For example, MCDNN [126] utilizes Approximate Model Scheduling to manage multiple DNN execution requests under resource constraints. It enables trading off classification accuracy for resource use by reasoning about on-device/cloud execution trade-offs. Another example is Neurosurgeon [127], which dynamically finds the best partitioning point at layer granularity by analyzing the CNN topology and network connection. Then, the computation of the first set of layers is kept on the device and offloads the remaining layers to the cloud.

Our analysis of prior CNN computation partitioning approaches reveals that they over utilize the cloud because they are input data invariant. The partitioning decision is either hardwired or dynamically changes only in response to the environment, e.g., network connectivity. In other words, as long as the topology of the CNN and the data connection speed are constant for an application running on the device, the partitioning decision remains unchanged. However, CNNs are usually over provisioned and most of the inputs do not require the entire computational power of the model to produce an accurate final output [16, 15, 128]. Consequently, we hypothesize that efficiency improvements can be achieved through input-variant partitioning.

In this chapter, we take inspiration from traditional speculation-recovery techniques and present *SIEVE*, Speculative Inference on the Edge with Versatile Exportation, which is input-variant and dynamically distributes CNN computation between the cloud and mobile device to achieve maximum efficiency and minimum latency in various environments. SIEVE aggressively compresses a CNN to form a small CNN that can speculatively perform inferences for a large fraction of the inputs on the device. A runtime system sends each input to either the edge or cloud based on the user preferences, input size and environmental conditions. In addition, the runtime system can selectively invoke an unmodified original CNN on the cloud server to recover from misspeculation, wherein the speculative CNN can only provide a low-confidence answer.

SIEVE has four main advantages. First, the runtime system automatically adapts how and where inferences are performed to the environment (network conditions and battery lifetime) in order to reduce energy consumption on inferences. Second, relying on the original CNN for recovery enables aggressive precision reduction of the speculative CNN, which increases the opportunities of processing more complex CNNs efficiently on the edge device. Third, the average latency of the system is improved in comparison to both cloud-only and mobile-only approaches because of the elimination of the extra round trips to the server and the excessive complexity of CNNs for marginal accuracy improvements. Last, not sending requests to the server for all inputs reduces the load on the server and releases some resources to support more users on the cloud.

The contributions of this chapter are as following:

- We propose an intelligent hybrid cloud-edge CNN computation partitioning technique to achieve efficiency and performance gains. A lightweight software runtime is designed to dynamically select between speculative inference on the edge device or inference on the server using the original CNN. It models the latency and energy of the speculative and original CNNs, available hardware on the edge device, data transfer latency, and speculative CNN accuracy and its associated misspeculation recovery costs.

- We provide a misspeculation detector that dynamically determines a confidence threshold on the speculative CNN output based on the distribution of correctable errors learned during the training phase and the target output quality to detect potential faulty outputs, which are sent to the original CNN for replay.

- We develop an automatic procedure to deterministically prune the design space, then search for the most compressed floating-point format to represent weights and activations of individual layers separately. A set of heuristics are used to combine those layers and form the final speculative CNN design with maximum compression rate and minimum accuracy loss, while not requiring any fine-tuning or retraining.

- We also introduce a lightweight (0.05% area overhead) hardware compressor and decom-

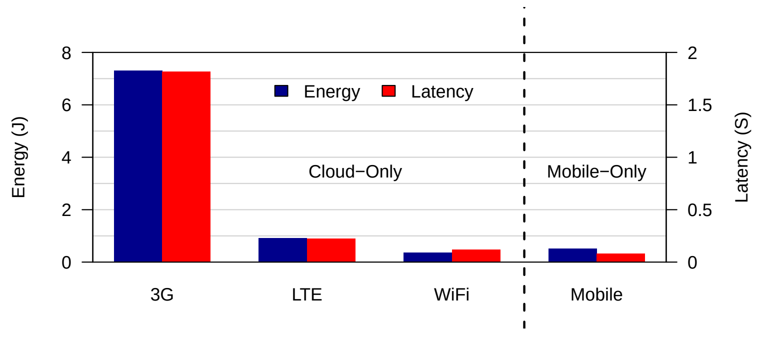


Figure 4.1: Energy and latency of AlexNet for mobile-only and cloud-only. The optimum computation platform varies for different communication speeds and optimization goals. Data transmission is considerably energy- and time-consuming.

pressor unit responsible for efficient floating-point reformatting for the speculative CNN.

Our evaluation on a benchmark suite of nine CNNs shows that SIEVE on average reduces mobile energy consumption by 91%, 57%, and 26% and improves latency by $12.3\times$, $2.8\times$, and $2.0\times$ for 3G, LTE and WiFi connections in comparison to the cloud-only approach, without any accuracy loss.

4.2 Background and Motivation

4.2.1 Mobile Deep Learning

As the energy-efficiency of CNN computation and computational power of mobile devices are increasing, the popularity of running CNN applications on mobile devices (mobile-only) is growing. However, the size and complexity of CNNs are increasing more rapidly, so it is difficult for mobile devices to keep up. Hence, offloading CNN computation onto cloud servers (cloud-only) is another widely-accepted approach.

Figure 4.1 compares the energy and latency for AlexNet inference on a Jetson TK1 mobile platform [129] for mobile-only and cloud-only approaches, considering three different data communication technologies: 3G, LTE and WiFi. As shown, the cloud-only result is heavily dependent on the type of the wireless network. There are two key observations. First, the mobile-only results in lower end-to-end latency for all three types of connections. Second, although the mobile-only consumes less energy than transferring data via LTE or 3G, the cloud-only, when using WiFi, is the most energy-efficient approach. Therefore, previous work [127] proposes dynamic CNN computation partitioning at layer granularity by analyzing the CNN topology, computation, and communication characteristics to improve energy efficiency and performance. However, the proposed hybrid techniques are data invariant and do not change the partitioning decision per input when the CNN topology and wireless network remain unchanged.

4.3 SIEVE

To elevate CNN computation partitioning, we take inspiration from traditional speculation-recovery techniques and propose *SIEVE* which is an intelligent hybrid deep learning cloud-edge system for improving energy efficiency and performance in comparison to running complex CNNs on edge devices or offloading the entire computation to cloud. First, we introduce our speculative hybrid deep learning system and a new approach for dividing the labor between the server and mobile. After that, each component of the runtime system is described.

4.3.1 System Overview

SIEVE is based on the hypothesis that inference energy on edge devices can be substantially reduced with an introspective software system that is both data and environmentally aware. As reported in prior work, the availability of high speed networks have a large impact on whether offloading to the cloud is feasible and efficient [127, 130]. But, we also believe the characteristics of individual inputs play an important role in how and where inference should be performed. A simpler, lighter weight CNN is often capable of rendering accurate inferences for a significant fraction of the inputs. Thus, an intelligent runtime system can examine the characteristics of the data to help partition the computation. SIEVE is designed to partition computation between a lightweight version of the CNN that runs on the edge device (speculative inference), and the original CNN on

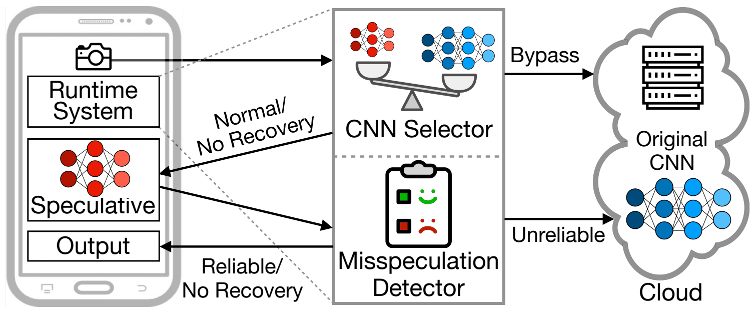


Figure 4.2: Overview of SIEVE. The camera provides the input to the runtime system to chooses between the speculative CNN on the device and the original CNN on the cloud. The misspeculation detector examines speculative CNN outputs and triggers the original CNN for recovery of faulty ones.

the cloud, and to detect misspeculations on the edge device that must be replayed in the cloud. By supporting data-aware partitioning, SIEVE enables a new level of efficiency gains that are not possible with environmental-only partitioning methods [127, 130].

Figure 4.2 shows the major parts of a SIEVE system including a camera, runtime system, mobile hardware and cloud server. At the beginning, the camera on the mobile device provides the input to the runtime system, which consists of a CNN selector and a misspeculation detector. Then, the CNN selector evaluates user preferences (accuracy requirements and privacy), environment (network speed, available hardware, and server load), and the most recent speculation results to dynamically route each input through either the original CNN on the cloud or the speculative CNN on the device. The main purpose of this CNN selection process is to minimize the energy and latency while meeting the accuracy requirements of the user. After CNN selection, if the speculative CNN is selected, its outputs are sent to the misspeculation detector unit to identify untrustworthy answers and initiate replay on the cloud. When the original CNN is selected, SIEVE bypasses the speculative CNN and misspeculation detector and sends the inputs to the cloud and downloads the results as with traditional cloud offloading.

SIEVE produces energy savings in two ways. First, successful speculation on the edge device enables a lighter weight CNN to perform the inference and avoid data transfer costs. Second, direct inference on the cloud because data transfer is actually cheaper than edge inference and predicted misspeculation costs. Conversely, SIEVE requires more energy during misspeculations

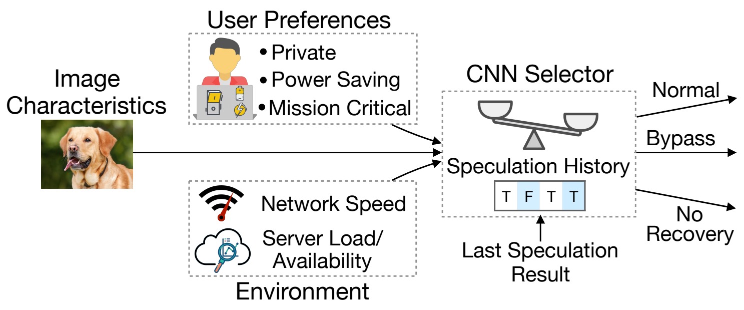


Figure 4.3: CNN selector decides based on the input, user preferences, environmental conditions and speculation history.

as inferences must occur on both the edge (speculative) and cloud (replay). Thus, it is important to manage speculation carefully to ensure that it is profitable. We found in our evaluation that only modest success rates are necessary for speculation to be useful, >60% is generally profitable. In our results, we achieved a minimum of 80% success, which was well above the threshold.

Offline Training: During the training phase, SIEVE designs and trains the speculative CNN and configures the runtime system units using the topology and parameters of the original CNN as well as characteristics of the mobile hardware, cloud servers, and communication networks.

4.3.2 CNN Selection

Figure 4.3 shows the first major component of SIEVE runtime system, which is the CNN selector. It tries to run most of the inputs speculatively on the device, however, there are various factors that have effects on the final selection decision. It estimates and compares the efficiency of data communication and local hardware to find the proper computation partitioning. Moreover, it takes into account environmental conditions and regulates the number of requests to the server. In addition, it considers user preferences by providing knobs to enable and disable different components of SIEVE. Furthermore, the recent speculation history and temporal distance of images is considered to identify similar ones.

Input Data and Environmental Conditions: To make the correct decision, SIEVE must estimate energy consumption and latency of the data transfer to the cloud (E_t, L_t) and execution on the mobile device (E_m, L_m). Additionally, it requires the probability of initiating recomputation by the misspeculation detector (P_r).

SIEVE uses techniques from Huang et al. [131, 132] to estimate E_t and L_t based on the size of the data, communication technology and bandwidth. Since the computation on the cloud GPU is much faster than the mobile GPU, the latency of the computation on the cloud is neglected in the estimation of L_t. In addition, it trains a regression model, same as Neurosurgeon [127], to estimate E_m and L_m based on the number and size of input and output feature maps as well as the characteristics of each layer including layer type, stride, kernel and group size for convolutional and pooling layers. These models are CNN invariant. Hence they are trained once for a set of CNNs and wireless connections, then they could be used for different CNN applications during runtime. P_r is calculated by comparing the output difference of the speculative CNN and original CNN on the validation set, which is explained in more detail in Section 4.4.

During runtime, it compares E_t (L_t) and $E_m + E_t \times P_r$ ($L_m + L_t \times P_r$) to pick between the original and speculative CNNs for minimum energy (latency). In addition, if the difference of these two numbers becomes larger than a defined threshold in a way that all inputs are sent to the original CNN, it concludes that the speculative CNN is not proper for the current wireless network speed and replaces it with another available topology.

Furthermore, it measures the round-trip time for each request and compares it with E_t and L_t estimations. A big difference between these two numbers indicates a high load on the server. Consequently, SIEVE temporarily turns off the misspeculation detector to eliminate recovery requests and keep the computation on the device. At the same time, it monitors the load on the server to find an opportunity to get back to its normal operation. This mechanism sacrifices some accuracy to maintain the constraints of time-sensitive applications when the server response time is low.

User Preferences: In addition to previous factors, CNN selector considers user preferences and provides a knob for the user to switch among mission critical, power saving, private and normal mode dynamically. The mission critical mode disables the speculative CNN and misspeculation detector and offloads every input on the cloud to maximize the accuracy and robustness. On the other hand, the private mode keeps the entire computation on the device and prevents any offloading to the cloud to increase privacy and security. And, the power saving mode relaxes the speculative CNN and the misspeculation detector to sacrifice a marginal amount of accuracy to obtain further energy and latency gains. Finally, the normal mode leaves the other units unchanged.

Speculation History: Since the input of SIEVE comes from the camera of a smartphone, if the temporal distance of two inputs is very low, it is likely that they are similar images from the same view. Consequently, CNN selector measures the temporal distance of two inputs and compares it with a predetermined threshold to find the similar consecutive images. This strategy results in two main benefits:

First, if the first image is sent to the server by the misspeculation detector for recovery and the original CNN provides the same answer as the speculative CNN, SIEVE will process the second image on the device without misspeculation detection and recovery, which provides energy and latency savings.

Second, if the first image is misclassified by the speculative CNN and the original CNN provides a different output during recovery, SIEVE will bypass the speculative CNN and misspeculation detector for the second image and send it directly to the server for energy and latency reduction.

4.3.3 Misspeculation Detection

The second major component of the SIEVE runtime system is the misspeculation detector. Since the speculative CNN is a simplified version of the original CNN and trades off some accuracy for better energy efficiency and latency on the edge device, it is necessary to detect and recover errors to bridge this accuracy gap to achieve the same accuracy as the baseline.

For classification tasks, CNNs convert an input instance (x) to an output vector of K elements (K is the number of classes). Using a softmax ($\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$) activation function after the output layer provides the estimated probability of that the correct output is j for $j = 1, ..., K$ ($P_j = P(y = j|x)$). Consequently, $\max\limits_{1 \leqslant j \leqslant K} P_j$ represents the confidence level of the CNN in the final output.

Figure 4.4 illustrates the confidence of wrong outputs for LeNet-5 [93] CNN, compressed to

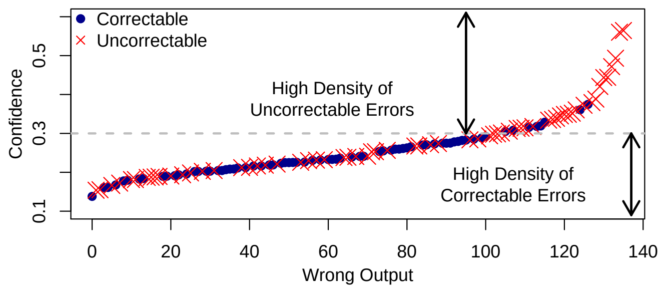


Figure 4.4: Confidence of wrong outputs for 4-bit compressed LeNet-5 on MNIST test set. For low confidences, the density of correctable errors is higher than uncorrectable ones.

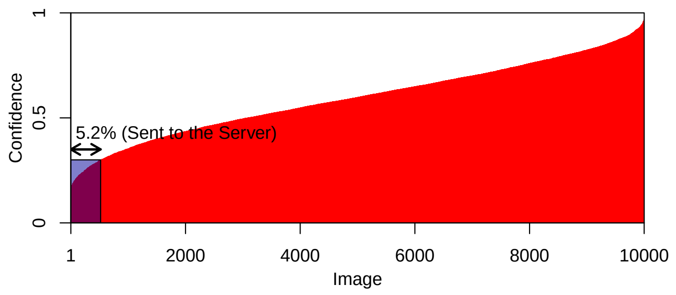


Figure 4.5: Confidence of 4-bit compressed LeNet-5 on MNIST test set. The confidence threshold for indicating correctable errors results in a limited number of recomputations.

4 bits per weights and activations, on MNIST [82] test set for handwritten digit recognition. The circles represent correctable errors, which are wrong on the compressed CNN but correct on the original CNN. And, the crosses indicate uncorrectable errors, which are wrong on both CNNs. If we pick a threshold equal to 0.3, shown by the dashed line, the density of correctable errors is much higher than the uncorrectable ones under that threshold.

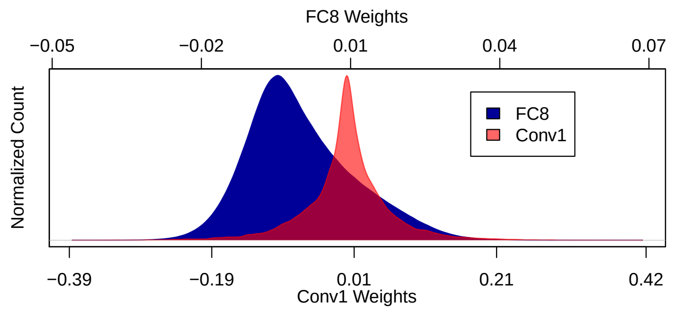
On the other hand, Figure 4.5 shows the confidence of the same CNN on the entire test set, and the 10000 images of the dataset are sorted from low to high confidence. As we can see, only 5.2% of the images result in outputs with less than 0.3 confidence. Hence, it is possible to recover most of the correctable errors by initiating the recomputation on the server for a small portion of inputs.

SIEVE determines the confidence distributions of the speculative CNN for correct, and correctable and uncorrectable wrong outputs on the validation set during training and saves them in the misspeculation detector. During runtime, the misspeculation detector defines a confidence threshold by examining those distributions and considering the output target quality to minimize the number of recomputations. Then, it compares the confidence of the speculative CNN with that threshold for each inference to detect and recover possible errors and meet the output target quality.

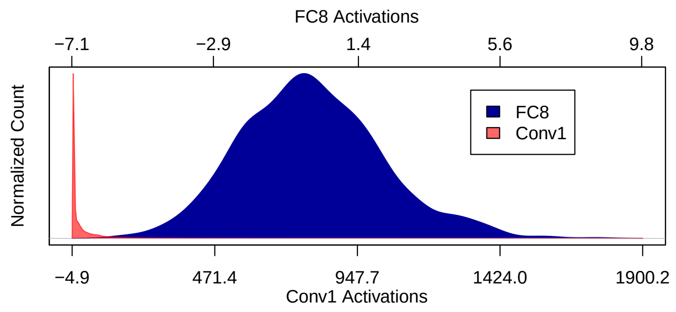
4.4 Speculative CNN Design

An important component of SIEVE is deriving a lightweight speculative CNN from the original CNN and making sure it performs faster and more efficiently than the mobile-only and cloud-only approaches. There are various well-studied methods for reducing the computation and memory accesses of CNNs for small or no accuracy reductions, such as pruning [49], compression [47, 11], fixed-point computation [54] and precision reduction [33]. Although all these techniques could be employed to design the speculative CNN, we prefer to minimize in-depth hardware modifications, rigorous CNN modifications and fine-tuning iterations. Hence, our system combines the compression and custom floating-point representation techniques to design and implement an efficient and high-performance speculative CNN on the mobile. To keep hardware modifications at the minimum level of complexity while gaining considerable improvements, SIEVE takes advantage of custom floating-point formats for compressing weights and activations stored in the memory, but performs the computation in full precision.

There are three advantages for this approach. First, since the numbers will be reformatted to single-precision floating-point before the functional unit, it is not necessary to use the same representation format for all numbers in the memory. Hence, weights and activations of different layers could be compressed differently based on their required precision as explained in prior work [133, 37, 134]. Second, the compression and decompression are basically reformatting between two different floating-point representations. Therefore, the hardware compressor and decompressor become very cheap and easy to implement. Third, since the computation is precise, each layer could be compressed very aggressively to minimize the memory accesses. In this section, first the opportunities for compressing CNNs are investigated. Then, we explain how SIEVE



(a) Weight distribution



(b) Activation distribution

Figure 4.6: Distribution of activations and weights in AlexNet. Using single-precision variables is extravagant. In addition, the range of numbers in different layers varies, so they need different optimized representations.

prunes the design space and derives the speculative CNN from the original CNN. Lastly, we describe the hardware of the speculative CNN, including the compressor and decompressor.

4.4.1 Compression

Conventional hardware usually uses IEEE 754 32-bit base-2 floating-point variables for CNN applications. These variables consist of three parts: one sign (s) bit , eight exponent (e) and 23 mantissa (m) bits. In addition, the exponent uses a bias (b) equal to +127 to represent the range from -126 to +127 by unsigned integer format (0 and 255 are interpreted specially). The value of each number is computed as $(-1)^s \times 2^{(e-b)} \times (1 + \sum_{i=1}^{23} m_{23-i} \times 2^{-i})$. Therefore, this format covers the huge range from -3.4×10^{38} to $+3.4 \times 10^{38}$.

However, the maximum and minimum of numbers even in very deep NNs are much smaller than the range provided by single-precision floating-points. Figure 4.6 shows the distribution of

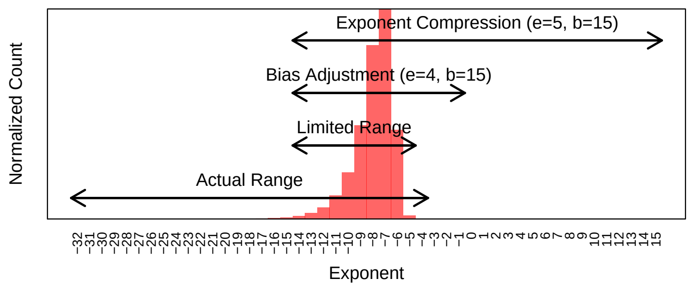


Figure 4.7: Exponent histogram for AlexNet FC8 weights. Bias adjustment reduces the bitwidth from 8 to 4.

weights and activations for the first convolutional (Conv1) and last fully-connected (FC8) layer of AlexNet. The comparison of these four distributions results in three important observations. First, the 32-bit floating-point is not the most suitable format to represent CNN numbers efficiently. Second, the range of numbers in different layers is very different from each other. For instance, the range of weights in Conv1 is about 7 times bigger than FC8. Third, the weights and activations of the same layer have very different distributions. In conclusion, it is necessary to pick the most representative format for weights and activations of each layer separately to maximize the compression rate.

It is very time consuming to sweep the entire design space and find the optimum number of bits for weights and activation of layers separately. The exponent of floating-point numbers is the part that determines the range of changes. Hence, defining the proper bitwidth for exponents is vital for accuracy maintenance. Figure 4.7 shows the histogram of weight exponents for FC8 of AlexNet, which change from -4 to -32. However, more than 99.5% of exponents are between -5 and -14. Due to the natural error resiliency of CNNs [33], we can easily omit the outliers and reduce the bitwidth e_{bw} from 8 to 5, which covers the range between -14 to +15. Yet there is no need to represent numbers higher than -4. To maximize the compression, SIEVE uses bias adjustment for exponents. Since the histogram is not symmetric around zero, a bias equal to $2^{e_{bw}-1}-1$ is wasteful. The optimum bias is determined by $b = 1 - \min|e|$. Then, there will be more room for bitwidth reduction, $e_{bw} = \lceil \log_2 (\max|e| - b - 2) \rceil$. For FC8, adjusting the bias to 15 reduces e_{bw} to 4 bits for a final range of -1 to -14.

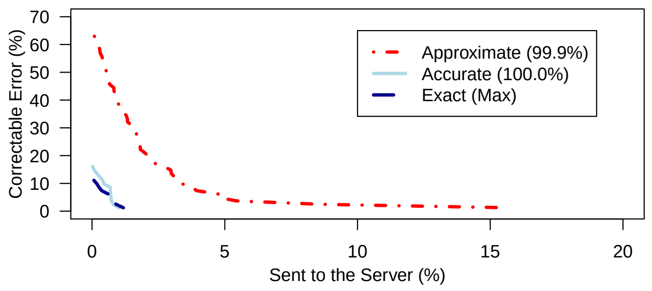


Figure 4.8: Correctable errors vs. recovery rate for 3 compressed LeNet-5s. Reducing the size increases the recovery rate, but might be efficient for high-speed connections.

To pick the best representation for each layer, SIEVE runs the original CNN on the validation set to gather exponent histograms. Then, for each set of numbers, it finds the best e_{bw}, called e_{max}, and $bias$ deterministically. After that, it sweeps e_{bw} from e_{max} to 2 bits, computes the $bias$ that maximizes the coverage and stops whenever the coverage is below 50%. For each iteration, it sweeps the m_{bw} from 6 to 0 bits, tests the CNN on the validation set while the other weights and activations are using full precision, and stops whenever the normalized accuracy is below 99.8%. Since the search space is considerably pruned, and there is no need for retraining or fine-tuning, the design space exploration is fast enough to derive the speculative CNN in a few hours on a single GPU even for huge CNNs on large datasets.

After that, our system picks the best configuration for each layer and finalizes the speculative CNN design. Each explored CNN has a compressed layer and is a trade-off between efficiency and accuracy. It is important to note that even combining layers of two 100% accurate compressed CNNs will not necessarily result in a 100% accurate CNN with two layers compressed, because of the aggregation of precision losses.

SIEVE chooses three different networks to use for different wireless connection speeds: exact, accurate and approximate. The exact model contains the most accurate layers from the explored CNNs. The accurate one, contains the most compressed layers with the accuracy not less than 100%. And the approximate one is the same as the accurate one, but with a threshold of 99.9%. If there is no configuration to meet the threshold for a layer, that will use the most accurate explored configuration.

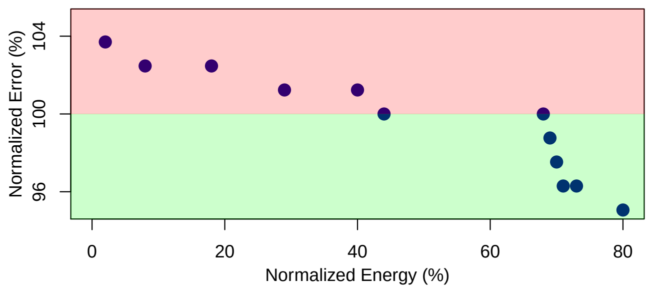


Figure 4.9: Efficiency vs. error Pareto frontier by combining result of 3 compressed LeNet-5s. Provides different design choices with different trade-offs.

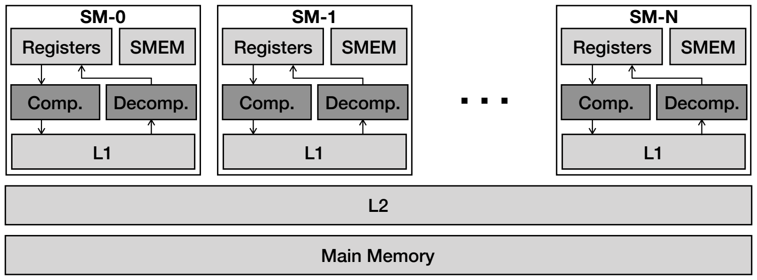


Figure 4.10: Compressor and decompressor are located before the register file to keep the data in the entire memory hierarchy compressed.

Figure 4.8 shows the trade-off between the number of correctable errors relative to the uncorrectable ones and the percentage of inputs that are recomputed on the server for the exact, accurate and approximate speculative CNN on MNIST test set. Based on the trade-offs, the approximate speculative CNN is suitable for the situations when the wireless communication is efficient, so we can make the speculative CNN as compressed as possible by increasing the recovery frequency. The CNN selector uses latency and energy models to convert these trade-offs to a single Pareto frontier based on the wireless communication speed. Figure 4.9 shows this Pareto for the same CNNs as Figure 4.8 using WiFi Internet connection. This Pareto helps the CNN selector to pick the most efficient speculative CNN which meets the output quality target.

4.4.2 Hardware

Prior work [48] identifies that the key bottleneck for CNN execution on GPUs is the on-chip memory bandwidth. Hence, SIEVE keeps the weights and activations of the speculative CNN com-

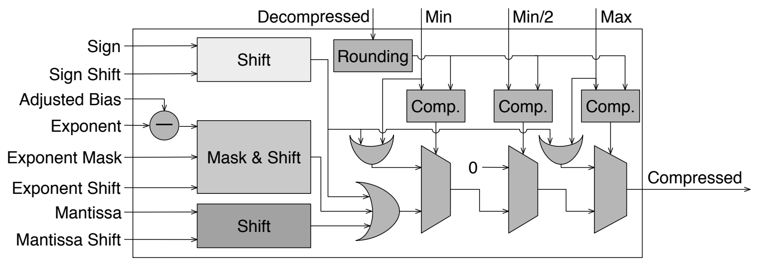


Figure 4.11: Compressor engine.

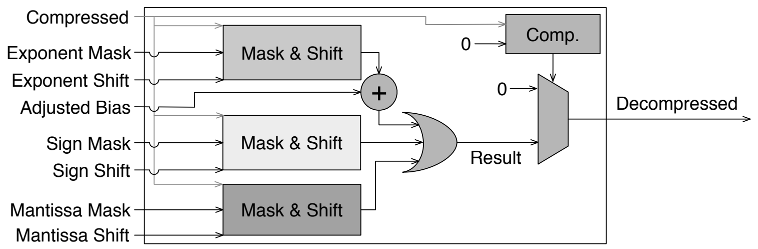


Figure 4.12: Decompressor engine.

pressed in the entire memory hierarchy, and decompresses them right before writing to the register file for computation. Since on-chip memory data compression at this level requires very frequent data reformatting, a low-overhead hardware implementation of the compression/decompression mechanism is mandatory. Figure 4.10 shows where the compressor and decompressor units are located in the memory hierarchy of a GPU. The compressor (decompressor) unit contains the same number of compressor (decompressor) engines as the number of load/store units in the SM. And, each compressor (decompressor) engine is capable of processing one compressed (decompressed) value per cycle.

Compressor Engine: Figure 4.11 shows the implementation of a compressor engine in detail. This engine is responsible for both bitwidth reduction and bias adjustment. The output format is dynamically configurable to use the same engine for different layers of the speculative CNN. In addition to the 32-bit decompressed number, it is fed by several constant values including Min, Min/2, Max, Sign Shift, Adjusted bias, Exponent Mask, Exponent Shift and Mantissa Shift. All these constants are computed based on the target compression format by a constant calculator unit, which is shared among all compressor units.

Decompressor Engine: Figure 4.12 demonstrates the hardware design of a decompressor en-

Table 4.1: CNN information for different benchmarks.

Network	**Dataset**	**Acc.**	**Input Size**	**MACs (G)**
LeNet-5 [93]	MNIST [82]	99.19%	$1 \times 28 \times 28$	0.002
ConvNet [92]	CIFAR10 [85]	82.12%	$3 \times 32 \times 32$	0.01
NIN-CIFAR10 [136]	CIFAR10	89.57%	$3 \times 32 \times 32$	0.2
AlexNet [91]	ImageNet	56.90%	$3 \times 227 \times 227$	0.7
SqueezeNet_1.0 [137]	ImageNet	57.67%	$3 \times 227 \times 227$	0.7
NIN-ImageNet [136]	ImageNet [84]	56.34%	$3 \times 224 \times 224$	1.1
GoogLeNet [138]	ImageNet	68.92%	$3 \times 224 \times 224$	1.6
ResNet-18 [139]	ImageNet	66.62%	$3 \times 224 \times 224$	1.8
VGG-16 [4]	ImageNet	68.35%	$3 \times 224 \times 224$	15.5

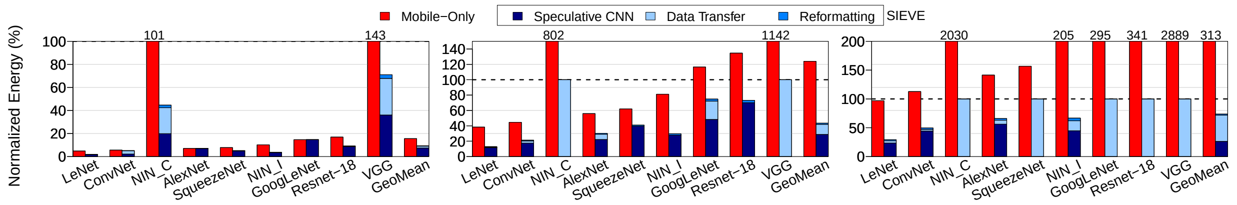
gine, which has similar characteristics to the compressor engine. In addition, all decompressor engines share a constant calculator unit in the same way as the compressor engines.

Most of the computation in these two engines is limited to logical and, logical or, shift, integer add and subtract, comparison and multiplex units, which are all cheap in terms of power and area. To measure the overhead, we implemented and synthesized the compressor and decompressor unit for an NVIDIA Tegra K1 mobile processor [135] using the ARM Artisan IBM SOI 45 nm library. It has an area overhead of 0.06mm^2 (0.05%), and an active power consumption of 0.23W (2.06%). Moreover, the compression/decompression takes less than one clock cycle and could be implemented as an additional pipeline stage. Considering the low branch misprediction rate of CNN computation, the performance overhead is negligible.

4.5 Evaluation

4.5.1 Experimental Methodology

Benchmarks: We evaluate SIEVE using a benchmark suite of nine CNNs using three wireless connection technologies: 3G, LTE and WiFi. A brief description of each benchmark is presented in Table 4.1. For each row, the first column presents the name of the original CNN. Second column is related to the dataset for providing training, validation and test data. We use a randomly sampled subset of the training set as the validation set to configure our system (same size as the test set).



(a) 3G (b) LTE (c) WiFi

Figure 4.13: Energy consumption of SIEVE vs. the mobile-only approach for 3G, LTE and WiFi normalized by cloud-only results.

Third column contains the top-1 accuracy of the trained original CNNs obtained from Caffe Model Zoo [140]. Fourth column shows the input size of each CNN, which is the amount of data to be sent to the cloud in the case of cloud-only or recovery. And the last column is the number of floating-point multiply and add (MAC) operations required to process a single input, which is a good indicator of the complexity of the CNN.

Hardware platforms: The NVIDIA Jetson TK1 embedded development kit is used as our mobile platform. It is built around NVIDIA Tegra K1 SoC: a quad-core ARM A15, a Kepler mobile GPU with a single streaming multiprocessor and a 2 GB DDR3L memory. In addition, for the server side, we use an NVIDIA TITAN X GPU [141].

Software framework: We use Caffe [95], an open-source deep learning library, and cuDNN [142], NVIDIA's GPU-accelerated library for DNNs, to build and test SIEVE.

SIEVE could be configured to prioritize efficiency (performance) improvements for maximum gains, while still obtains latency (energy) savings. To demonstrate the supremacy of SIEVE, we compare it with both cloud-only and mobile-only.

4.5.2 Energy Saving

Figure 4.13 shows the energy consumption of SIEVE compared to the mobile-only and the cloud-only methods for different wireless connection speeds. In each plot, each group of bars is dedicated to one of the nine benchmarks. In each group, the left bar represents the result of running the original CNN on the mobile (mobile-only) and the right bar is dedicated to the energy breakdown of SIEVE. Both bars are normalized by the cloud-only energy consumption result. The bottom part of the right bars is related to the amount of energy consumed for running the speculative CNN on the mobile device. The middle part is dedicated to uploading the input data to the server and downloading the final result for two situations: when the CNN selector predicts that the data network is more efficient than the speculative CNN, or when the misspeculation detector triggers the recovery phase. The top shows the energy consumption of the decoder and encoder (reformatting).

On average, SIEVE reduces the energy consumption of the cloud-only approach by 91%, 57% and 26% for 3G, LTE and WiFi, respectively. In contrast, the mobile-only approach increases the energy consumption by 24% and 213% in comparison to when the cloud-only method uses LTE and WiFi connection, respectively. In addition, for 3G connection, SIEVE results in 7% more energy savings than the mobile-only.

As demonstrated, SIEVE is capable of decreasing the energy of the cloud-only by partitioning the computation between the speculative and original CNNs, except when the complexity of the baseline CNN is much higher than the input dimensions. For example, based on Table 4.1, the number of MACs per input size for NIN_CIFAR10, ResNet-18, GoogLeNet and VGG-16 is one and two orders of magnitude higher than other benchmarks. In other words, the complexity of the baseline CNN for these two benchmarks is much higher than the input dimensions. Consequently, when the internet connection is efficient (WiFi), the CNN selector correctly predicts that the data transfer over the network connection for these benchmarks is more efficient than executing the speculative CNN. Another important observation is that the average energy consumption of reformatting never exceeds 5% of the total system energy.

Another interesting observation is that SIEVE gets the same energy as the mobile-only for AlexNet with 3G connection. The reason is that the number of MACs per input for AlexNet is the third lowest after LeNet-5 and ConvNet. However, the baseline accuracy of AlexNet is much lower than those two other CNNs. Hence, the same compression rate for AlexNet results in a higher recovery rate than the other two CNNs, and these recoveries are relatively expensive for 3G connection. Therefore, the CNN selector correctly chooses the original CNN to run as the speculative CNN in this situation.

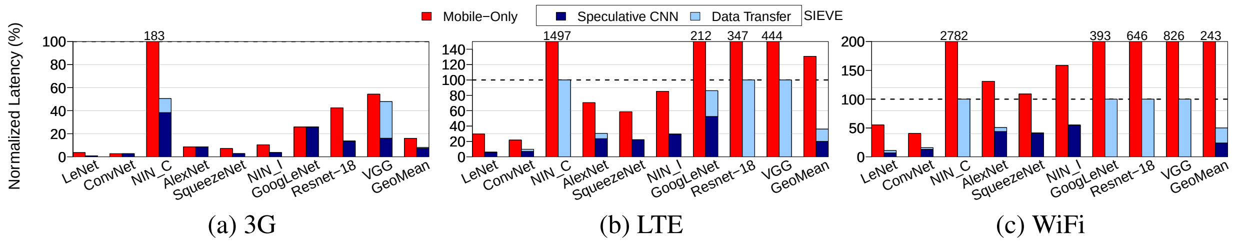


Figure 4.14: Latency of SIEVE vs. the mobile-only approach for 3G, LTE and WiFi normalized by cloud-only results.

4.5.3 Latency Improvement

As explained in Section 4.5.2, another approach to take advantage of SIEVE is configuring the system for latency improvements. This configuration forces the CNN selector to prioritize performance over efficiency for picking the proper speculative CNN topology and selecting between the speculative and original CNNs on various platforms with different hardware resources and data connection speeds. In this section we show that SIEVE is capable of improving the performance in comparison to both cloud-only and mobile-only approaches.

Figure 4.14 shows the latency of SIEVE versus the mobile-only approach normalized by the end-to-end latency of the cloud-only technique in the same format as Figure 4.13. Each right bar has two partitions. The bottom part indicates the latency of executing the speculative CNN on the device. And, the top part is dedicated to the end-to-end latency of sending one input to the cloud, processing that on the cloud GPU and downloading the final result for the cases that CNN selector or misspeculation detector activates the original CNN. On average, SIEVE improves the latency by $12.3\times$, $2.8\times$ and $2.0\times$ for 3G, LTE and WiFi, respectively.

SIEVE managed to reduce the latency for all benchmarks in comparison to the mobile-only approach. On average, the mobile-only increases the latency by 31% and 143% for LTE and WiFi, respectively. Even for 3G, SIEVE achieves 8% more latency reduction than the mobile-only.

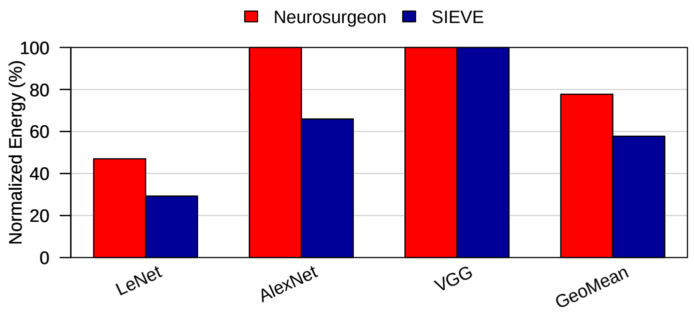


Figure 4.15: Energy improvements by SIEVE vs. Neurosurgeon normalized by cloud-only with WiFi.

4.5.4 Comparison to Prior Work

In this section, we compare SIEVE with Neurosurgeon [127] to show benefits of input-variant dynamic partitioning. Then, the comparison with DeftNN data fission [48] is presented to demonstrate additional compression opportunities because of the cloud recovery mechanism. And finally, we apply our technique on top of Scalpel node pruning [52] to show that our aggressive precision reduction is orthogonal to mobile-only compression techniques, such as pruning. For all experiments, we compare energy of SIEVE with other techniques using WiFi connection. Performance improvements follow the same trend and are not shown because of space limitations.

Neurosurgeon dynamically finds the interesting partitioning points within a CNN at layer granularity to reduce the data transfer to cloud by pushing as much computation as possible onto the mobile device for performance and efficiency gains.

Figure 4.15 compares SIEVE and Neurosurgeon for three of our benchmarks that are shared with Neurosurgeon. Each bar is normalized by the result of the cloud-only approach. In each group, the left bar represents Neurosurgeon, and the right bar is related to SIEVE results. On average, SIEVE outperforms Neurosurgeon by 20% more energy improvements. Both systems successfully indicate that the input size of VGG-16 is very small relative to the amount of computation required for an inference of VGG-16. Hence, they find WiFi connection fast and efficient enough to transfer this relatively small input to the cloud for energy savings over the mobile-only approach.

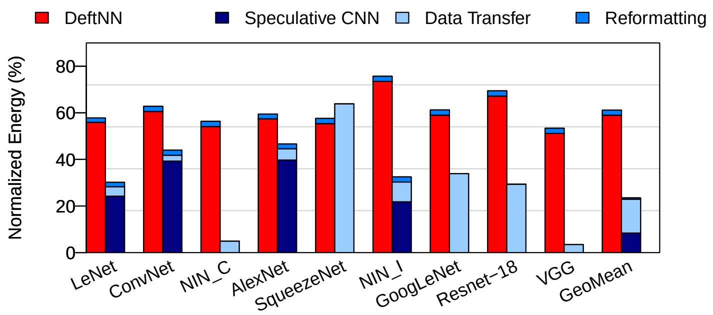


Figure 4.16: Energy improvements by SIEVE vs. DeftNN data fission normalized by full precision mobile-only.

For LeNet-5, Neurosurgeon picks the mobile-only computation. Since it does not employ any mechanism to optimize the part of the computation which is remained on the mobile device, SIEVE provides higher savings by enabling aggressive precision reduction of the speculative CNN. For AlexNet, Neurosurgeon is not able to partition the computation properly and offloads everything on the cloud. However, SIEVE keeps most of the computation on the device by utilizing a compressed and energy-efficient speculative CNN.

DeftNN demonstrates that GPU on-chip memory bandwidth is a key DNN execution bottleneck and proposes near-compute data fission to scale down the on-chip data movement requirements by efficiently packing on-chip memory.

Figure 4.16 compares the energy savings from non-aggressive precision reduction of applying DeftNN data fission versus aggressive precision reduction, which is enabled by SIEVE. Each bar is normalized by the result of mobile-only full precision baseline CNN. In each group, the left bar represents energy breakdown of DeftNN, which consists of running DeftNN on the device and reformatting overheads for compression and decompression. The right bar shows SIEVE breakdown in the same format as Figure 4.13. On average, running speculative CNNs designed by SIEVE requires 51% less energy than the CNNs designed by DeftNN, and the entire SIEVE results in 38% more savings with 100% accuracy. Hence, SIEVE is capable of utilizing more compressed and energy-efficient CNNs on the mobile device by relying on the cloud for recovery.

Scalpel customizes DNN pruning to the underlying hardware. For GPU, it proposes node pruning to reduce computation without sacrificing the dense matrix format.

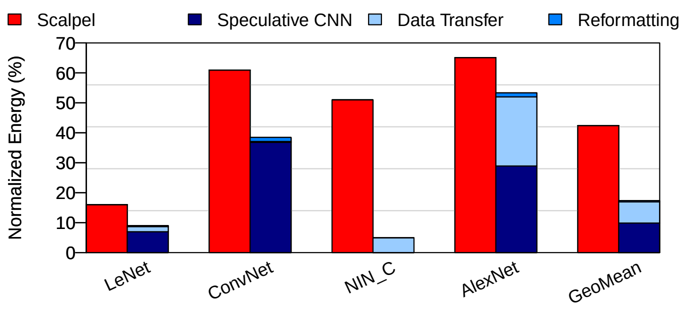


Figure 4.17: Energy of pruned SIEVE vs. pruned mobile-only normalized by unpruned mobile-only.

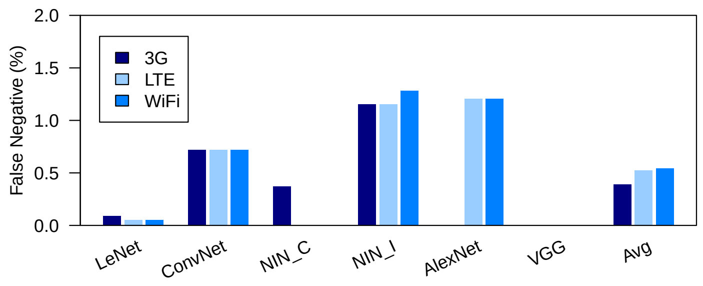


Figure 4.18: The false negative of misspeculation detector.

We apply our technique on four node-pruned CNNs, which are shared between us and Scalpel. Figure 4.17 shows the result of node-pruned SIEVE in comparison to mobile-only node-pruned CNNs from Scalpel. Each bar is normalized by the result of mobile-only unpruned original CNN. In each group, the left bar shows Scalpel results, and the right bar is related to the energy breakdown of pruned SIEVE. On average, Scalpel reduces energy by 42% on the mobile device, and SIEVE improves this result by 25% of additional energy savings without any accuracy loss. Consequently, our intelligent hybrid technique could use aggressive precision reduction and speculative execution with cloud recovery on top of mobile-only compression techniques, including pruning, for further savings.

4.5.5 Misspeculation Detector

To evaluate the misspeculation detector, SIEVE is configured for energy optimization. On average, it detects 6.3%, 3.8% and 3.4% of outputs as potential faulty results and sends them to the cloud to

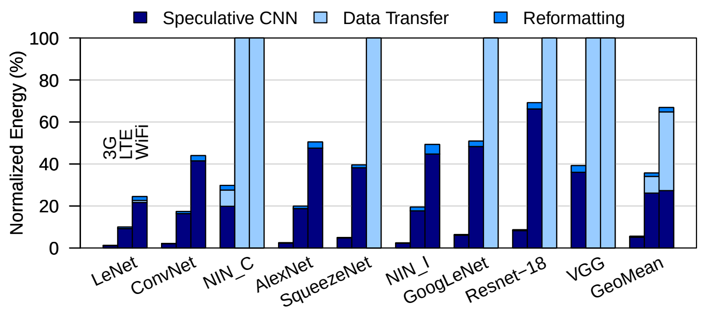


Figure 4.19: Energy improvement of SIEVE compared to the cloud-only by relaxing the target output quality to maximum 1% additional error.

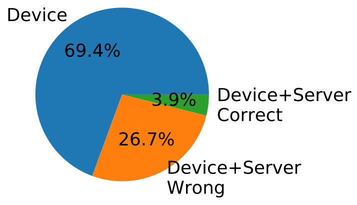
be recomputed on the original CNN for 3G, LTE and WiFi, respectively.

Figure 4.18 shows the false negative rate of the misspeculation detector. A false negative happens when a complex input, which could be classified wrongly by the speculative CNN and correctly by the original CNN, is not sent to the original CNN for recovery. The average false negative rate is 0.7%, 0.8% and 0.4% for 3G, LTE and WiFi, respectively. SIEVE keeps the false negative rate of all benchmarks below 2.0% to limit the accuracy loss in a compensable range. The lost accuracy is compensated by speculation exclusive inputs and the low false positive rates. The average false positive for all connections is below 0.5%.

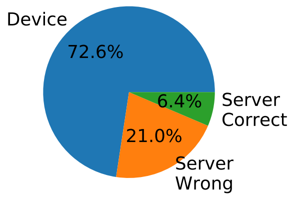
4.5.6 Approximate speculative CNN

As explained in Section 4.4, there are different speculative CNNs available in the system. Hence, the CNN selector is capable of choosing the best speculative CNN based on the available hardware and data network resources, as well as target output quality. In other words, the CNN selector unit is an available knob in the system to dynamically choose the target output quality. Therefore, there is an opportunity for the user to sacrifice a marginal amount of accuracy to obtain further energy and latency gains.

Figure 4.19 presents the energy breakdown of SIEVE with the output accuracy relaxed to 99% of the baseline accuracy normalized to the cloud-only method. In comparison to the 100% accurate SIEVE, the average energy saving for 3G, LTE and WiFi is increased by 3.7%, 7.7% and 6.8% for



(a) Without History



(b) With History

Figure 4.20: Speculation history for consecutive images helps SIEVE to reduce server activations for ConvNet.

0.69%, 0.54% and 0.38% additional error.

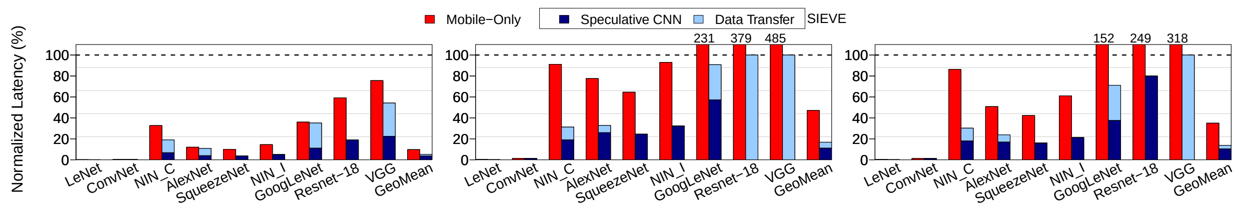
4.5.7 Speculation History

To demonstrate the effect of considering the speculation history for consecutive images, we augment CIFAR10 dataset by randomly cropped and scaled images from the dataset, which emulate images with low temporal distances from the same view. Then, the augmented dataset is used to evaluate the ConvNet SIEVE system with and without speculation history. For similar images, if the first image is run on the speculative CNN without recovery, speculation history will not have any effect on the CNN selection of the second image. Hence, Figure 4.20 shows the distribution of the host hardware for the second image in the cases that the first image requires recovery.

Without the history, 100% of inputs require speculative CNN activation and 30.6% of them require recovery. However, only 3.9% of these complex inputs result in a correct output after recovery. Speculation history helps CNN selector to bypass the speculative CNN and misspeculation detector for 27.4% of inputs and run the rest of the inputs on device with disabled misspeculation detector. Consequently, the correct recoveries increases by 2.5% while server and device activation decrease by 3.2% and 27.4%, respectively.

4.5.8 Realistic Model

In the evaluation section we use a simple model for transferring data to the cloud for both SIEVE and the cloud-only approach. This simple model ignores compression and decompression of the



(a) 3G (b) LTE (c) WiFi

Figure 4.21: Using realistic model to compare latency of SIEVE and the mobile-only approach for 3G, LTE and WiFi normalized by cloud-only results.

transferred data, handshakes required for uploading and downloading data, etc. We simply assume the uncompressed bitmap image is transferred to the server. However, the real data transfer model is more complicated. In this section, we use a more realistic model, which considers the latency of handshakes and assumes a $4\times$ lossless compression of input images before uploading to the cloud.

Figure 4.21 shows the latency of SIEVE versus the mobile-only approach normalized by the end-to-end latency of the cloud-only technique in the same format as Figure 4.14. On average, SIEVE improves the latency by $19.2\times$, $6.0\times$ and $7.3\times$ for 3G, LTE and WiFi, respectively. SIEVE managed to reduce the latency for all benchmarks in comparison to the mobile-only approach. On average, it achieves 5%, 30% and 21% more latency reduction than the mobile-only for 3G, LTE and WiFi, respectively.

4.6 Related Work

Mobile Cloud Computing (MCC) is widely used to overcome the power, memory and compute resources' constraints in mobile systems [125, 143, 144]. MAUI [121] implements a fine-grained energy-aware code offloading at methods granularity. COMET [145] develops a generalized offloading technique for applications with no offloading logic using Distributed Shared Memory (DSM) system. Qian et al. [146] introduce Jade, which offloads computations to the server based on program and device status. Odessa [147] offloads computations based on parallelism capabilities and shows that their performance is about the same as offline partitioning performance. CloneCloud [148] uses both static analysis and dynamic profiling in order to improve energy and execution time of unmodified programs using thread level offloading. By popularization of machine learning (ML) applications on mobile devices, researchers have designed ML specific MCC infrastructures [149, 150] which can be divided into two categories:

Static: Ran et al. [151] characterize mobile-only and cloud-only approaches in terms of accuracy, latency and energy consumption. Hauswald et al. [152] evaluate different configurations of computer-vision classification pipeline and shows that running the feature extraction stage on the mobile device would be the best configuration.

Dynamic: Qi et al. [153] have developed an object detection system that switches the stream of video data between cloud and mobile based on the quality of network connection. They have an online model selector in their system, which selects between different pre-trained versions of DNN models based on the destination hardware features. MCDNN [126] introduces Approximate Model Scheduling (AMS) to manage heterogeneous DNN requests across both mobile and cloud devices by trading off 1-4% of classification accuracy for higher resource utilization. DDNN [130] is a hierarchical distributed design which has multiple exit points. Neurosurgeon [127] can partition the computation to optimize its latency or mobile power consumption. In either case, it considers different parameters, such as quality of the network, DNN model architecture and destination hardware, in order to partition the computation.

SIEVE benefits from a dynamic partitioning algorithm. The key difference of SIEVE design is that it uses input-variant partitioning and treats the cloud server as a backup resource. As a result, it performs an aggressive compression on the model and generates a cost-effective model (speculative CNN). In the case of error occurrence, accuracy will be preserved by offloading the computation to the server.

4.7 Conclusion

CNN execution has traditionally been offloaded to the cloud, but there is a trend toward bringing CNNs to edge devices with the emergence of powerful and efficient processors on mobile devices.

However, the complexity of CNNs is growing much faster than the compute power and battery life of these devices. In this chapter, we introduce *SIEVE*, a novel *hybrid cloud-edge deep learning system*. It creates a heavily compressed CNN through aggressive precision reduction to enable speculative inferences on the device. However, for a subset of inputs, the heavily compressed CNN cannot yield confident predictions and is often wrong. Hence, a dynamic partitioning technique is employed to push most of the computation to this speculative CNN while the data transfer to original CNN is limited to recovery on low-confidence inferences. Compared to the cloud-only approach, SIEVE reduces the energy consumption by an average of 91%, 57% and 26% and increases the performance by an average of $12.3\times$, $2.8\times$ and $2.0\times$ for 3G, LTE and WiFi connection with 100% accuracy of baseline.

CHAPTER 5

Extension of Cloud-Edge Hybrid Systems

5.1 Introduction

Since the computational complexity of DNNs is becoming prohibitive, recent research proposed new DNN architectures, such as MobileNets [154], which are specifically designed for efficient inference, while offering the same accuracy level as complex models. Although these models are much more efficient than conventional DNNs, they still use a single model for the entire dataset. Hence, we propose M&MNet to extend the intelligent cloud-edge collaboration idea from Chapter 4 to these efficient DNNs. M&MNet uses a lightweight software runtime to distribute the computation between the original network called Master and a pair of simpler networks called Minions.

Our evaluation on a benchmark suite of eight CNNs shows that M&MNet on average increases mobile execution performance by $3.9\times$, and $1.9\times$ and reduces the energy consumption by 53.5%, and 39.5% for LTE and WiFi connections in comparison to the cloud-only approach, without any accuracy loss.

This work is a collaboration with Salar Latifi, and Pedram Zamirai.

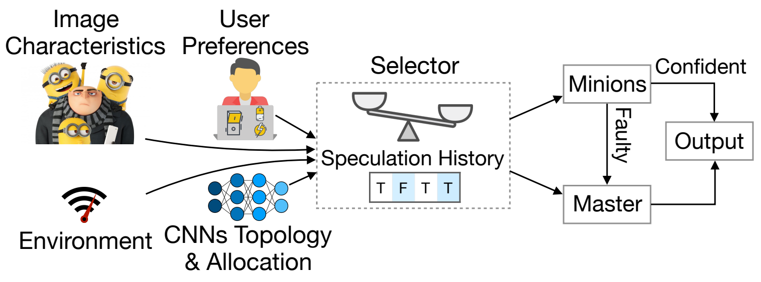


Figure 5.1: M&MNet overview. Selector decides based on the input, CNN architectures, user preferences, environmental conditions and speculation history to run speculatively on Minions or use the Master.

5.2 M&MNet

M&MNet is designed to partition computation between a pair of lightweight CNNs (Minions), and the original CNN (Master), and to detect misspeculations of Minions that must be replayed using Master. In addition, M&MNet decides to place the Master on the cloud server or device to minimize the total latency. Minions and a misspeculation detector are always located on the device. By supporting data-aware partitioning, M&MNet enables a new level of performance gains that are not possible with environmental-only partitioning methods [127, 130].

Figure 5.1 shows the major parts of an M&MNet system including a selector, Minions and Master. At the beginning, the camera on the mobile device provides the input to the runtime system. Then, the selector evaluates user preferences (accuracy requirements and privacy), environment (network speed, available hardware, and server load), CNNs topology and allocation and the most recent speculation results to dynamically route each input through either Master or the speculative Minions. The main purpose of this selection process is to minimize the latency while meeting the accuracy requirements of the user. After selection, if the Minions are selected, their outputs are sent to the misspeculation detector unit to identify untrustworthy answers and initiate replay on Master. When Master is selected, M&MNet bypasses Minions and misspeculation detector and processes the inputs directly on Master.

M&MNet produces performance improvement in two ways. First, allocating Master on the proper host to minimize its latency. Second, successful speculation of Minions enables lighter

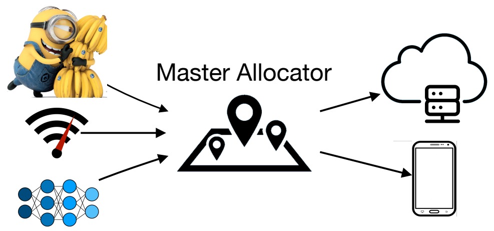


Figure 5.2: Deciding between on-device computing and cloud offloading.

weight CNNs to perform the inference on the edge and avoids data transfer (cloud-only) or original complex CNN execution (mobile-only) costs. However, M&MNet results in higher latency during misspeculations as inferences must occur on Minions and Master. Thus, it is important to manage speculation carefully to ensure that it is profitable.

Offline Training: During the training phase, M&MNet designs and trains the Minions and configures the runtime system units using the topology and parameters of the original CNN as well as characteristics of the mobile hardware, cloud servers, and communication networks.

5.2.1 Master Allocation

As explained in Section 4.2.1, even for a fixed CNN topology, there is no single approach between mobile-only and cloud-only which always works best. Hence, as shown in Figure 5.2, M&MNet dynamically determines the best host for Master based on its topology, input characteristics, current network connection speed and available hardware on the device.

To make the correct decision, M&MNet must estimate latency of the data transfer to the cloud (L_t) and execution on the mobile device (L_m). M&MNet uses techniques from Huang et al. [131, 132] to estimate L_t based on the size of the data, communication technology and bandwidth. Since the computation on the cloud GPU is much faster than the device, the latency of the computation on the cloud is neglected in the estimation of L_t. In addition, it trains a regression model, same as Neurosurgeon [127], to estimate L_m based on the number and size of input and output feature maps as well as the characteristics of each layer including layer type, stride, kernel and group size for convolutional and pooling layers. These models are CNN invariant. Hence they are trained once for a set of CNNs and wireless connections, then they could be used for different CNN applications during runtime. During runtime, M&MNet compares L_t and L_m to pick between on-device execution and cloud offloading for minimum latency.

In case of offloading Master on the cloud, M&MNet measures the round-trip time for each request and compares it with L_m estimations. If the measured latency is higher than L_m, it indicates a high load on the server. Consequently, M&MNet temporarily keeps Master computation on the device. At the same time, it monitors the load on the server to find an opportunity to get back to its normal operation.

5.2.2 Selection

The left part of Figure 5.1 shows one of the major components of M&MNet runtime system in detail, which is the selector. It tries to run most of the inputs speculatively on Minions, however, there are various factors that have effects on the final selection decision. It estimates and compares the latency of data communication and local hardware to find the proper computation partitioning. Moreover, it takes into account environmental conditions and number of requests to the server. In addition, it considers user preferences by providing knobs to enable and disable different components of M&MNet. Furthermore, the recent speculation history and temporal distance of images is considered to identify similar ones.

Input Data and Environmental Conditions: M&MNet estimates latency of Master (L_{Master}) and Minions ($L_{Minion\_1}, L_{Minion\_2}$) based on the regression models explained in Section 5.2.1. To make the correct selection decision, it additionally requires the probability of initiating recomputation by the misspeculation detector (P_r). P_r is calculated by comparing the output difference of Minions and the original CNN on the validation set, which is explained in more detail in Section 5.2.3.

During runtime, it compares L_{Master} and $L_{Minion\_1} + L_{Minion\_2} + L_{Master} \times P_r$ to pick between Master and Minions for minimum latency. In addition, if the difference of these two numbers becomes larger than a defined threshold in a way that all inputs are sent to the Master, it concludes that Minions are not proper for the current wireless network speed and replaces them with other available topologies.

User Preferences: In addition to previous factors, the selector considers user preferences and provides a knob for the user to switch among mission critical, power saving, private and normal mode dynamically. The mission critical mode disables Minions and misspeculation detector and offloads every input on Master to maximize the accuracy and robustness. On the other hand, the private mode keeps the entire computation on the device and prevents any offloading to the cloud to increase privacy and security. And, the power saving mode relaxes Minions and the misspeculation detector to sacrifice a marginal amount of accuracy to obtain further energy and latency gains. Finally, the normal mode leaves the other units unchanged.

Speculation History: Since the input of M&MNet comes from the camera of a smartphone, if the temporal distance of two inputs is very low, it is likely that they are similar images from the same view. Consequently, the selector measures the temporal distance of two inputs and compares it with a predetermined threshold to find the similar consecutive images. This strategy results in two main benefits:

First, if the first image is sent for recovery by the misspeculation detector and Master provides the same answer as Minions, M&MNet will process the second image on Minions without misspeculation detection and recovery, which provides further latency and energy savings. Second, if the first image is misclassified by Minions and Master provides a different output during recovery, M&MNet will bypass Minions and misspeculation detector for the second image and send it directly to the Master for latency and energy reduction.

5.2.3 Misspeculation Detection

The next major component of the M&MNet runtime system is the misspeculation detector. Since Minions are less complex than the original CNN and trade off some accuracy for better latency and energy efficiency, it is necessary to detect and recover errors to bridge this accuracy gap and

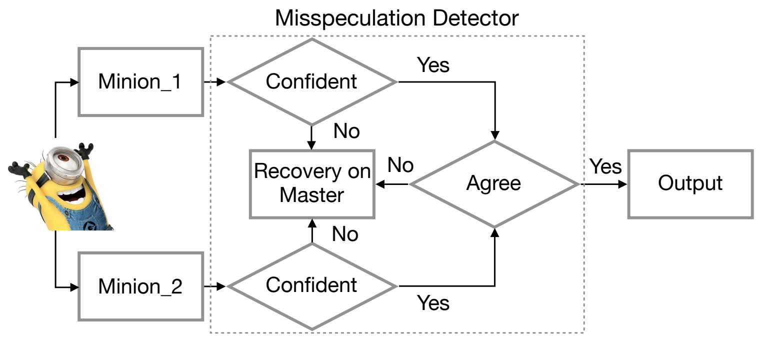


Figure 5.3: Minions and misspeculation detector.

achieve the same accuracy as the baseline.

M&MNet determines the confidence distributions of Minions for correct, and correctable and uncorrectable wrong outputs on the validation set during training and saves them in the misspeculation detector. During runtime, the misspeculation detector defines a confidence threshold by examining those distributions and considering the output target quality to minimize the number of recoveries.

Figure 5.3 shows the runtime of misspeculation detector and Minions. Misspeculation detector compares the confidence of Minions with the calculated threshold for each inference. If one of the Minions is not confident in its output or the output of two Minions does not agree, possible error is detected and sent to the Master for recovery to maintain the output target quality.

5.3 Minion Design and Pairing

An important component of M&MNet is deriving lightweight speculative Minions from original CNNs and making sure they perform faster and more efficiently than the mobile-only and cloud-only approaches. There are various well-studied methods for reducing the computation and memory accesses of CNNs for small or no accuracy reductions, such as pruning [49], compression [11], fixed-point computation [54] and precision reduction [33]. Although all these techniques could be employed to design Minions, we prefer to eliminate hardware modifications and rigorous CNN modifications.

Table 5.1: MobileNet width reduction.

Configuration	**Acc.**	**MFLOPs**	**Params**
Mobilenet_V1_1.0_224	71.0%	569	4.2M
Mobilenet_V1_0.75_224	68.3%	325	2.6M
Mobilenet_V1_0.5_224	63.2%	149	1.3M
Mobilenet_V1_0.25_224	49.7%	41	0.5M

Hence, M&MNet uses two optimization techniques from MobileNets [154] to design, train and implement high-performance and efficient Minions on the mobile.

In addition to creating a pool of candidate Minions from original CNNs, it is necessary to design an automatic mechanism to pick the most proper pair of Minions for each Master.

5.3.1 CNN Width Reduction

The first optimization technique to construct smaller and less computationally expensive models is using an extra hyperparameter, called width multiplier (α), during training. The width multiplier is responsible for scaling the width (number of kernels) of a CNN uniformly at each layer. For a given convolutional layer and width multiplier, the computational cost changes as following:

$$K \times K \times \alpha M \times \alpha N \times H \times W \tag{5.1}$$

And for a depthwise convolutional layer, the computational cost changes as following:

$$K \times K \times \alpha M \times H \times W \tag{5.2}$$

$K \times K$ indicates the kernel size. M is the number of input feature maps, and N is the number of output feature maps. $H \times W$ represents the size of the output feature map.

Table 5.1 shows the accuracy, computational complexity and size trade offs of shrinking the MobileNet_V1 architecture by sweeping the width multiplier from 0.25 to 1.0 by steps of 0.25.

Table 5.2: MobileNet input downsampling.

Configuration	Acc.	MFLOPs	Params
Mobilenet_V1_1.0_224	71.0%	569	4.2M
Mobilenet_V1_1.0_192	69.9%	418	4.2M
Mobilenet_V1_1.0_160	68.0%	290	4.2M
Mobilenet_V1_1.0_128	65.2%	186	4.2M

5.3.2 Input Downsampling

The second optimization method to reduce the computational cost of a CNN is input downsampling by using another extra hyperparameter, called resolution multiplier (ρ). Resolution multiplier is applied to the input image, which results in scaling internal representation of every subsequent layer by the same multiplier. For a given convolutional layer, width multiplier and resolution multiplier, the computational cost changes as following:

$$K \times K \times \alpha M \times \alpha N \times \rho H \times \rho W \tag{5.3}$$

And for a depthwise convolutional layer, the computational cost changes as following:

$$K \times K \times \alpha M \times \rho H \times \rho W \tag{5.4}$$

Table 5.2 shows the accuracy, computational complexity and size trade offs for applying input downsampling on Mobilenet_V1 and sweeping the input size from 128 to 224 by steps of 32.

5.3.3 Minion Pairing

Both width reduction and input downsampling could be applied to any CNN of our model pool to create new simpler CNNs as Minions. However, applying both techniques on the entire model pool results in a huge Minion pool which has two main problems. First, each Minion requires training from scratch, which increases the training time of the entire system. Second, the search space for pairing Minions becomes huge and prohibitive. Hence, M&MNet uses a subset of original CNNs to create Minions. This subset contains the original CNNs for which the on-device latency is lower

Table 5.3: CNN information for different benchmarks.

Network	**Acc.**	**Latency**	**Input Size**
MnasNet [155]	74.08%	19.4ms	224×224
Mobilenet_V1 [154]	71.0%	24ms	224×224
NASNet mobile [156]	73.9%	56ms	224×224
Inception_V3 [157]	77.9%	249ms	299×299
Inception_ResNet_V2 [158]	77.5%	422ms	299×299
Inception_V4 [158]	80.1%	486ms	299×299
ResNet_V2_101 [159]	76.8%	526ms	299×299
NASNet large [156]	82.6%	1170ms	331×331

than the latency of offloading their computation on the cloud using the fastest available network connection speed.

After limiting the size of Minion pool, M&MNet uses the validation set to evaluate each pair of candidate Minions for each Master. The sum of latency of these two Minions should be smaller than Master to result in performance improvement and leave some room for the extra latency of recoveries. In addition, the final M&MNet design should result in a moderate recovery rate to allow performance and efficiency improvement.

5.4 Evaluation

5.4.1 Experimental Methodology

Benchmarks: We evaluate M&MNet using a benchmark suite of eight CNNs on the ImageNet [84] dataset using two wireless connection technologies: LTE and WiFi. A brief description of each benchmark is presented in Table 5.3. For each row, the first column presents the name of the original CNN. Second column contains the top-1 accuracy of the trained original CNNs obtained from TensorFlow Lite Hosted Models. Third column presents the model latency of processing a single input on Google Pixel 3, which is a good indicator of the complexity of the CNN. And the last column shows the input size of each CNN, which is the amount of data to be sent to the cloud in the case of cloud-only or recovery. We use a randomly sampled subset of the training set as the validation set to configure our system (same size as the test set).

Hardware platforms: The Google Pixel 3 running on Android 10 is used as our mobile platform. It is using Qualcomm's Snapdragon 845 SoC with 4GB of RAM, and Qualcomm Kryo 385 octa-core CPU which is based on ARM A75 processor. In addition, for the server side, we use NVIDIA TITAN X GPUs.

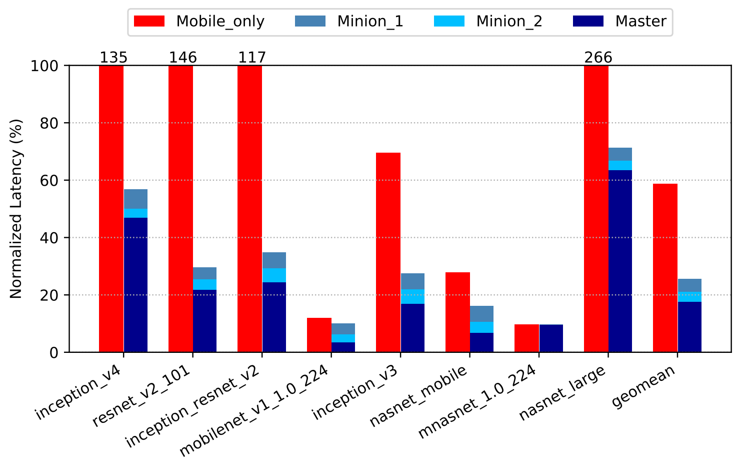
Software framework: We use the open-source library of TensorFlow, and cuDNN, NVIDIA's GPU-accelerated library for DNNs, to train and build the CNN models. After the models are trained, they are converted to an appropriate format which can be used by TensorFlow Lite to deploy them on mobile devices.

Energy estimation: We estimate the energy of running MobileNet and MnasNet models and their compressed versions as Minion, based on the methodology used in previous work [160].

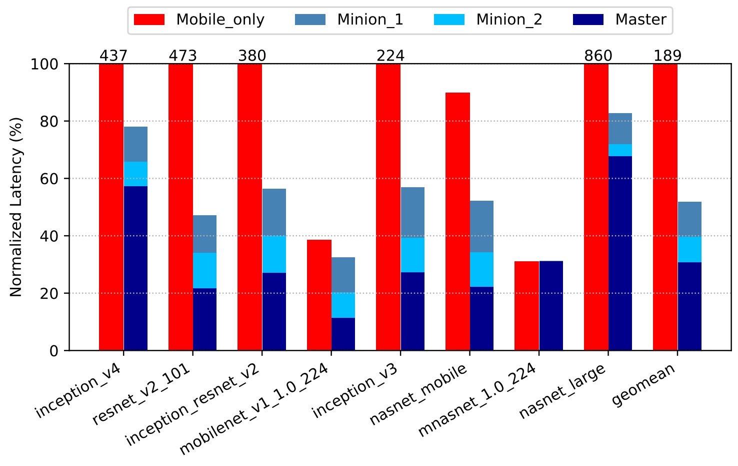
M&MNet could be configured to prioritize efficiency (performance) improvements for maximum gains, while still obtaining latency (energy) savings. To demonstrate the supremacy of M&MNet, we compare it with both cloud-only and mobile-only.

5.4.2 Latency Improvement

Figure 5.4 shows the execution latency of M&MNet compared to the mobile-only and the cloud-only methods for different wireless connection speeds. In each plot, each group of bars is dedicated to one of the eight benchmarks. In each group, the left bar represents the result of running the original CNN on the mobile (mobile-only) and the right bar is dedicated to the latency breakdown of M&MNet. Both bars are normalized by the cloud-only latency numbers. The upper two parts of the right bars is related to the amount of time consumed for running each Minion CNN in Minions on the mobile device. The bottom part is dedicated to running Master CNN which depending on where it is allocated can be equal to data transfer overhead, or running Master CNN on the mobile device. Data transfer overhead is also composed of uploading the input data to the server and downloading the final result. This happens in the situation that the data network is efficient and Master allocator designates the server for Master computation and also the misspeculation detector



(a) LTE



(b) WiFi

Figure 5.4: Latency of M&MNet vs. the mobile-only approach for LTE and WiFi normalized by cloud-only results.

triggers the recovery phase.

On average, M&MNet reduces the latency of the cloud-only approach by 74.5% and 48.3% for LTE and WiFi, respectively. And, reduces the latency of the mobile-only approach by 33.1% and 137.5% using LTE and WiFi connection, respectively. All benchmarks in Figure 5.4 are offering the exact accuracy level as baseline networks in Table 5.3, evaluated on the test set. Table 5.4 also presents the configuration of M&MNet for the LTE network connection.

Figure 5.4 also demonstrates that in four of the benchmarks including Inception_V4, ResNet_V2_101, Inception_ResNet_V2, and NASNet_large, mobile-only method results in higher latency com-

Table 5.4: M&MNet lookup table

Master	Minion_1	Minion_2
Inception_V4	Mobilenet_1.0_224	MnasNet_1.0_160
ResNet_V2_101	MnasNet_1.0_192	MnasNet_0.75_224
Inception_ResNet	MnasNet_1.0_224	Mobilenet_1.0_192
Mobilenet_V1	MnasNet_1.0_128	MnasNet_1.0_96
Inception_V3	MnasNet_1.0_224	Mobilenet_1.0_192
NASNet mobile	MnasNet_1.0_160	MnasNet_1.0_128
NASNet large	MnasNet_1.0_224	Mobilenet_0.75_224

pared to the cloud-only solution regardless of network speed. Based on Table 5.3, these benchmarks have higher accuracy compared to other benchmarks, and as a result, their network architecture is more complex and requires a longer latency to execute them on mobile device. Therefore, cloud-only approach is more efficient in the case of these benchmarks. On the other hand, as presented in Figure 5.4, M&MNet is able to reduce the average latency of these benchmarks even compared to cloud-only approach. This gives us the opportunity to benefit from the higher accuracy of the more complex networks, while offering a lower latency compared to both mobile- and cloud-only solutions.

M&MNet improves the runtime of all benchmarks in both LTE and WiFi except MnasNet_1.0_224 model. In this benchmark, the computation latency of M&MNet is equal to the mobile-only method. MnasNet_1.0_224 is the smallest network with the lowest latency in our benchmark suite according to Table 5.3. Therefore, when we are generating the Minion CNNs for this benchmark, if the compression rate is not high enough, due to the assumption of serial execution of Minion networks, the overall latency becomes higher than executing the Master network alone on mobile. However, if the compression rate is too high, the accuracy gap between Minion and Master becomes too high. This results in a higher rate of recovery phase which increases the overall latency. As a result, CNN selector correctly decides to only run Master which results in the same latency number as the mobile-only approach.

In addition, if we compare the latency reduction between LTE and WiFi in Figure 5.4a and Figure 5.4b, we can see higher rates of improvement in LTE connection. M&MNet reduces the

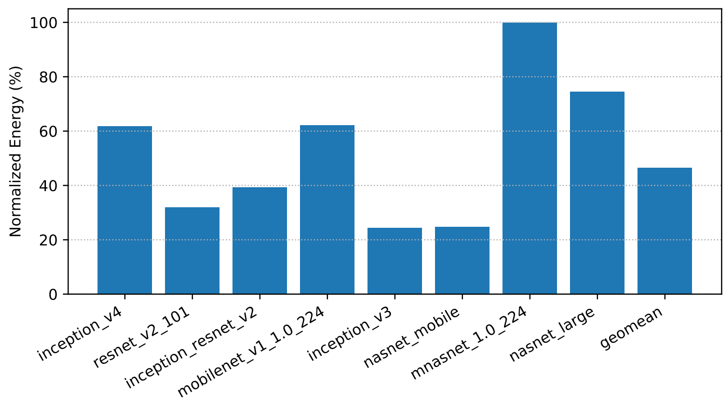


Figure 5.5: Energy consumption of M&MNet compared to the most efficient baseline out of mobile-only or cloud-only for LTE.

runtime by reducing the number of accesses required to the cloud, which as discussed in Section 4.2.1 can be expensive. Therefore, the slower the network connection is, the overhead of data transfer becomes higher, and there is a bigger opportunity for latency improvement by deploying M&MNet. For the rest of the evaluation section, we will focus on the results based on LTE network connection which has a wider availability. Performance improvement of WiFi follows the same trend and is omitted because of space limitations.

5.4.3 Energy Saving

The latency improvements discussed in Section 5.4.2 also translates to lower energy consumption of running CNNs on mobile platforms. Figure 5.5 shows the normalized overall energy consumption of M&MNet on LTE connection, compared to the most efficient baseline out of mobile- or cloud-only, depending on whichever consumes lower energy per benchmark.

Based on Figure 5.5, M&MNet managed to reduce the energy consumption for all benchmarks except MnasNet_1.0_224, which is explained in Section 5.4.2. On average, the proposed solution decreases the energy usage of the mobile device by 53.5% compared to the traditional solutions.

Table 5.5: CNN information for different quantized benchmarks.

Network	**Acc.**	**Latency**	**Input Size**
Mobilenet_V1_quant [161]	70.0%	13ms	224 × 224
Inception_V1_quant [138]	70.1%	39ms	224 × 224
Inception_V2_quant [157]	73.5%	59ms	299 × 299
Inception_V3_quant [162]	77.5%	148ms	299 × 299
Inception_V4_quant [158]	79.5%	268ms	299 × 299

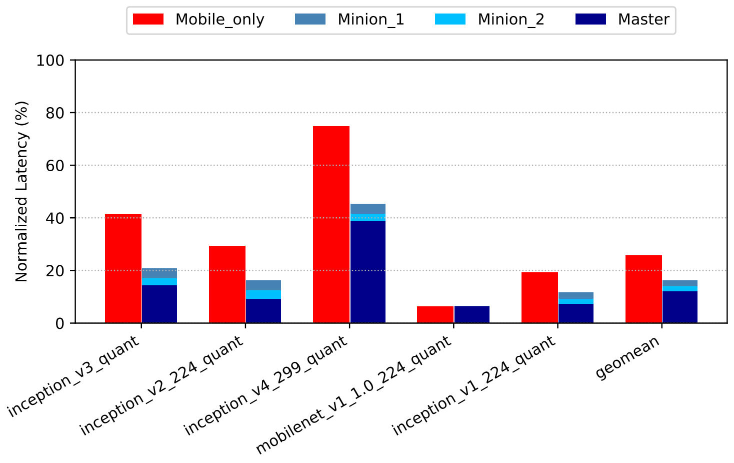


Figure 5.6: Latency of quantized M&MNet vs. the mobile-only approach for LTE normalized by cloud-only results.

5.4.4 Comparison to Network Quantization

In this section, we apply our technique on top of quantized networks to show that CNN computation partitioning is orthogonal to optimization techniques, including model compression, that are proposed for more efficient mobile execution. For this goal, we use a quantization technique that trains the baseline network using 8-bit fixed-point weights. Recent works [161] show that this training method results in a similar accuracy level to baseline network, while having a much smaller memory footprint and higher performance. Quantization technique is selected as an example, while it can be replaced by other model compression techniques such as pruning.

Table 5.5 displays the benchmark suite used in this experiment. All model options for Minion and Master are compressed by quantizing entire weights of the network to 8-bit fixed-point values.

Figure 5.6 presents the latency breakdown of quantized M&MNet. All bar graphs are normalized to cloud-only approach for LTE network connection. As demonstrated, we can see ad-

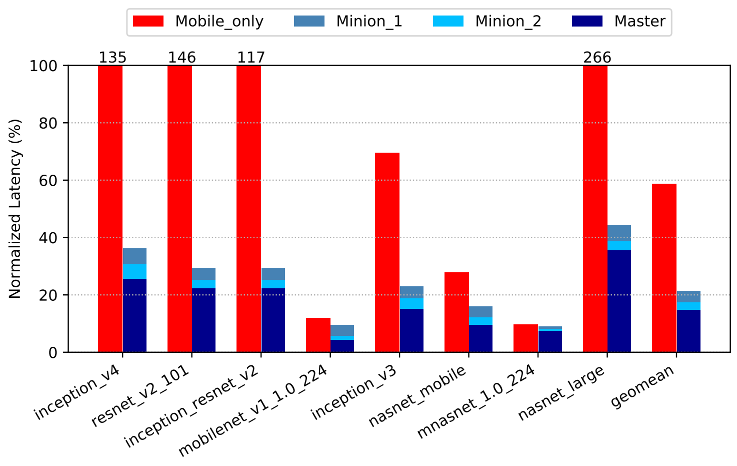


Figure 5.7: Latency improvement of M&MNet compared to the cloud-only by relaxing the target output quality to maximum 1% additional error.

ditional latency reduction by partitioning quantized CNN computation for all benchmarks except MobileNet_V1_quant, which can be explained similar to MnasNet_1.0_224 in Section 5.4.2.

Overall, CNN computation partitioning can introduce an extra 83.8% and 9.5% latency reduction on quantized models for cloud-only and mobile-only solutions, respectively.

5.4.5 Approximate speculative CNN

As explained in Section 5.3, there are different Minion designs available in the system. Hence, the CNN selector is capable of choosing the best options based on the available hardware and data network resources, as well as target output quality. In other words, the CNN selector unit is an available knob in the system to dynamically choose the target output quality. Therefore, there is an opportunity for the user to sacrifice a marginal amount of accuracy to obtain further energy and latency gains.

Figure 5.7 presents the latency breakdown of M&MNet with the output accuracy relaxed to 99% of the baseline accuracy normalized to the cloud-only method. In comparison to the 100% accurate M&MNet, the latency improvement for LTE is increased by 4.1% for 0.7% additional error.

5.5 Conclusion

CNN execution has traditionally been offloaded to the cloud, but there is a trend toward bringing CNNs to edge devices with the emergence of powerful and efficient processors on mobile devices. However, the complexity of CNNs is growing much faster than the compute power and battery life of these devices. In this chapter, we introduce *M&MNet*, a novel *hybrid cloud-edge deep learning system*. It creates heavily compressed CNNs to enable speculative inferences on the device. However, for a subset of inputs, the heavily compressed CNNs cannot yield confident predictions and are often wrong. Hence, a dynamic partitioning technique is employed to push most of the computation to these speculative CNNs while the data transfer to original CNN is limited to recovery on low-confidence inferences. Compared to the cloud-only approach, M&MNet reduces the energy consumption by an average of 53.5% and 39.5% and increases the performance by an average of $3.9\times$ and $1.9\times$ for LTE and WiFi connection with 100% accuracy of baseline.

CHAPTER 6

An Input-Driven Synergistic Deep Learning System

6.1 Introduction

In addition to the growing popularity of CNN algorithms across many application domains, it is observed that a single CNN implementation could be shared among various applications for their vision tasks. Therefore, Web service providers can employ an approach of providing CNN as a general datacenter service to be used by different products [42]. This approach helps Web service companies to reduce the development overhead by centralizing CNN implementations and unifying CNN optimization points. For example, the Google Brain project is a deep-learning service that is used in over 30 different product teams across the company [163].

However, CNN inference queries require significant amounts of compute resources in comparison to traditional text-based web services. Hence, there has been significant research interest to leverage hardware [39] and software [11] techniques to accelerate deep learning on various platforms, such as GPUs [48], CPUs [54], manycore co-processors [164], FPGAs [165] and custom ASICs [27] to increase performance and energy efficiency.

All the aforementioned techniques are input-invariant and accelerate a single CNN for the entire dataset. However, CNNs are usually overparameterized and most of the inputs do not require their entire computational power to produce an accurate output. CNNs with early exits [15, 16] take

This work is a collaboration with Salar Latifi.

advantage of the observation that features learned at earlier stages of a deep network can be used to correctly infer a subset of the data population. By terminating the prediction of these samples at earlier stages, they reduce the total required computation for inference.

However, CNNs with early exits cannot be automatically applied to off-the-shelf CNNs and require machine learning expertise and manual design to address two main concerns. First, the features in the early layers are not extracted to be directly used by the classifier. Second, the features in different layers have different scales. Hence, adding early classifiers interferes with later classifiers, which results in accuracy loss. Conversely, we hypothesize that additional performance gains could be achieved through input-variant acceleration approaches while eliminating the manual labor by employing multiple off-the-shelf CNNs in a system, instead of one, and offloading most of the computation on the simple ones.

Conventional ensemble methods activate multiple NNs and combine their outputs to obtain accuracy improvements for an extra latency and energy consumption [72, 73, 74, 75]. We explore an opposing technique by taking inspiration from traditional speculation-recovery approaches and present *Dynamic Duo (DD)*, which is input-variant and dynamically distributes CNN computation between a synergistic pair of a big and a little CNNs to achieve maximum performance and energy efficiency while maintaining the target output quality. DD chooses a simple and less accurate CNN as the little CNN and a complex and more accurate CNN as the big one. The little CNN is chosen in a way that it is capable of performing inferences correctly for a significant fraction of inputs while requiring less computation per input instance. To recover from unreliable outputs wherein the little CNN can only provide a low-confidence output, the big CNN is selectively invoked.

This chapter focuses on servers because DD requires multiple CNNs available in the memory to achieve performance gains, and these memory requirements could be beyond mobile devices' limits. DD has three main advantages. First, replacing the complex CNN by a simple one for most of the input instances reduces the server load and improves the throughput and average response time. Second, the server load reduction releases hardware resources to serve more users on the cloud platform. Third, the average latency and energy consumption of the system is improved in comparison to single CNN approaches because the excessive complexity of CNNs for marginal accuracy improvements is eliminated.

The contributions of this chapter are as following:

- We propose an intelligent hybrid CNN computation partitioning technique to achieve performance and efficiency gains. DD takes advantage of the synergy between a pair of CNNs and offloads most of the computation on the little one. The activation of the big CNN is limited to recovering unreliable outputs of the little one.

- We develop a systematic methodology to find the most efficient synergistic pair of CNNs and determine proper thresholds based on the distribution of outputs' confidence learned during training, for minimum recovery activation while meeting the output target quality.

- We utilize a lightweight confidence probe, which dynamically applies predefined confidence thresholds on the little CNN output to detect potential faulty outputs and send them to the original CNN for recovery. Surprisingly, DD is robust to moderate misprediction rates of the confidence probe because of inputs that are classified correctly by the simple model but misclassified by the complex one.

- We provide a set of heuristics to limit the performance overhead of searching for synergistic pairs and confidence thresholds, which makes our technique applicable to huge model pools containing numerous CNN models in a feasible time.

Our evaluation on a benchmark suite of 21 CNNs on ImageNet [84] dataset shows that DD on average improves throughput by $2.1\times$ and reduces average inference latency to 62% on an NVIDIA TITAN X GPU in comparison to the traditional CNN as a service approach, while maintaining the accuracy of the baseline.

6.2 Background and Motivation

6.2.1 NN Ensembles

A well-known approach to increase NN accuracy is using ensembles [77, 76, 70, 71], which train multiple NNs on different subsets of the training set and combine their results during inference. There are two major classes of methods to construct ensemble NNs. The first category, including bagging [76] and boosting [77], change the distribution of the original training data to create explicitly different training sets for different NNs. The second category, such as mixture of experts (MoE) [78], train different NN experts on different subsets of the training data to create implicitly different training sets [79]. The combination of multiple learners can reduce the impact of a single learner getting stuck in local optima or suffering from overfitting. It can also expand the space of the representable hypothesis. The net result is that ensembles can provide higher prediction accuracy than a single NN.

Although NN ensembles result in higher accuracies, they worsen the problem of low performance and high energy consumption. Ensemble methods require the activation of more than one NN per input instance, which can multiplicatively increase the computation per inference. Even with MoEs, subsets of training data should not be mutually exclusive to avoid limited local generalization of experts, which results in complex gating NNs for input partitioning and activation of multiple NNs during inference [81].

6.2.2 Input-Driven Synergy

Although ensemble methods combine NNs to improve accuracy for higher cost, it is not necessarily the only way to take advantage of the synergy among CNNs. As demonstrated in Figure 2.3, the size and accuracy of CNNs are changing drastically over the years. However, even the simplest model classifies more than half of the input instances correctly. Hence, different input instances show various behavior patterns, and it might not be required to activate the most complex model for every single input to achieve the highest possible accuracy.

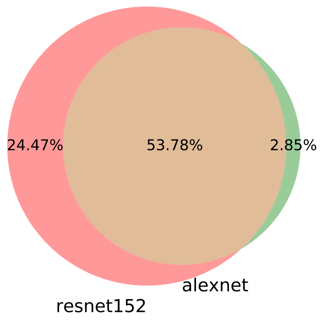


Figure 6.1: Venn diagram of correct predictions for ResNet-152 and AlexNet. Only 24.47% of inputs require the complex model. Ideally partitioning the input data between these two CNNs results in improving ResNet-152 accuracy by 2.85%.

Figure 6.1 shows the Venn diagram of predictions for ResNet-152 and AlexNet with top-1 accuracy of 78.25% and 56.63%, respectively, on the ImageNet test set. Based on the diagram, 53.78% of inputs are classified correctly by both CNNs (*common corrects*). In addition, 18.9% of inputs result in wrong answers regardless of the complexity of CNNs. And more surprisingly, 2.85% of inputs are classified correctly by the simple model but misclassified by the complex one (*odd corrects*). Thus, only 24.47% of inputs require the complex model and the remaining inputs could be processed by the simple one. Moreover, partitioning the input dataset to two subsets and activating the proper CNN for each input could improve the top-1 accuracy of ResNet-152 by 2.85%. The common correct predictions and the additional accuracy benefit the system in two ways. First, even by using a non-ideal partitioner with a moderate misclassification rate, it is possible to offload part of the computation on a simpler CNN while achieving the same accuracy as the complex one. Second, there is room to further decrease (less than 24.47%) the activation of ResNet-152 and maintain its accuracy.

To expand the idea of synergistic CNN pairs, we investigate 21 different CNNs on ImageNet dataset. For each experiment, a pair of a larger more accurate CNN (big) and a smaller less accurate (little) CNN are chosen to form the similar diagram as Figure 6.1. Figure 6.2 shows the distribution of the size of odd subsets for $\binom{21}{2}$ possible pairs. The x-axis shows the big CNNs from the least

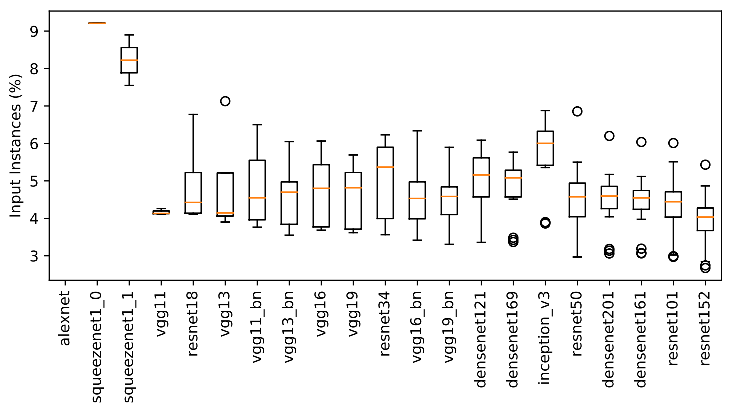


Figure 6.2: Distribution of the size of odd subsets for pairing different CNNs out of 21 CNNs on ImageNet test set. Even the least synergistic pair contains 2.7% of odd corrects.

accurate on the left to the most accurate one on the right. Since the little CNN should be smaller and less accurate than the big one, it is not possible to pair each big CNN with all other 20 CNNs. Consequently, there is no candidate for pairing with AlexNet, and the first column of the plot is empty. As demonstrated, the odd subset could contain from 2.7% to 9.2% of inputs, which indicates that there is some inherent potential synergy between any two CNNs out of these 21 single CNNs.

6.3 Dynamic Duo

We take inspiration from traditional ensemble NNs and speculation-recovery techniques to propose *Dynamic Duo (DD)*, which is an intelligent synergistic deep learning system for improving performance in comparison to traditional CNN as a service system, which runs a single complex CNN on servers per request. First, we introduce our input-driven synergistic deep learning system and a new approach for exploiting the synergy between CNNs. After that, DD's design and runtime are described in detail.

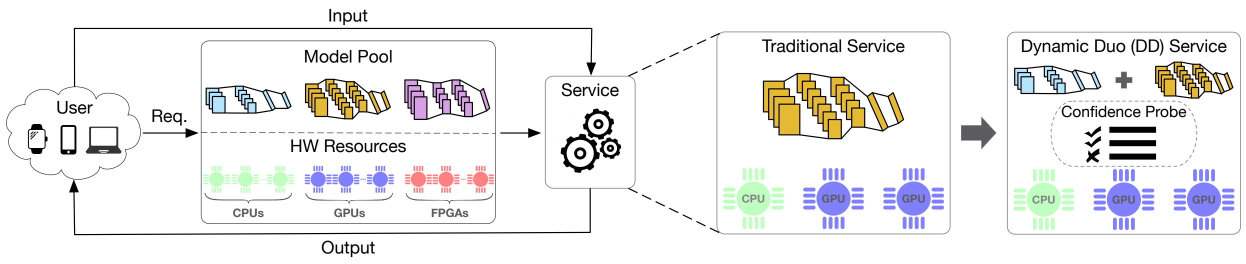


Figure 6.3: Overview of DD in comparison to the traditional approach. DD replaces the single CNN with a pair of a big and a little CNN and employs the proper strategy to detect and recover unreliable outputs of the little one.

6.3.1 Synergistic Deep Learning System

Due to CNNs' high computational demands, many approaches have emerged to improve their performance and efficiency while maintaining the accuracy, such as CNN hardware accelerators [20, 23, 24], pruning [49], compression [47], fixed-point computation [54] and precision reduction [33]. However, these approaches are input-invariant and use the same set of optimizations for the entire dataset without considering the diversity of input instances. In contrast, DD finds the optimum CNN pair inside a model pool to process most of the inputs fast and energy-efficiently on the simpler CNN and uses the complex one only for recovering the faulty results.

Overview: Figure 6.3 compares the overview of DD and a traditional CNN as a service system. In traditional approach, the server selects the most efficient CNN among existing trained models to meet the target output quality. After that, it allocates the minimum hardware resources to run the model and satisfy the maximum response time restrictions. On the other hand, DD replaces the single CNN with a pair of a big and a little CNN to improve performance and reduce energy consumption while maintaining the minimum required accuracy considering the available hardware resources. In addition, it adds a confidence probe to the system and configures it with the proper strategy to detect and recover the unreliable outputs of the little CNN.

Design: Based on Section 6.2.2, a key component to exploit the inherent synergy between a pair of CNNs is utilizing a proper input partitioning mechanism to offload most of the computation on the little CNN. The partitioner requires to be activated for each input instance to select the proper CNN for processing that input. If the little CNN is activated, the system will require to examine

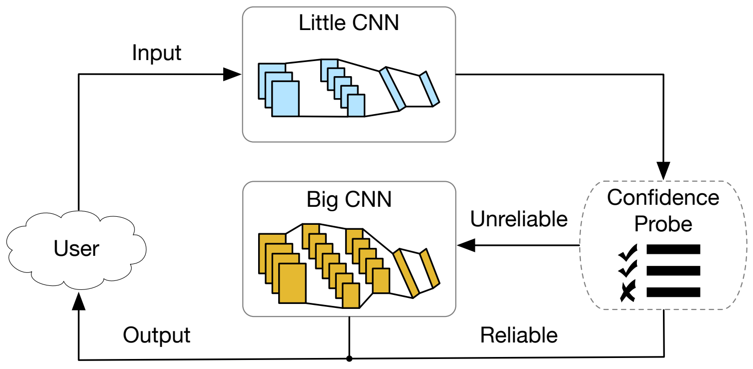


Figure 6.4: Runtime of DD. All inputs are run on the little CNN. The confidence probe examines all outputs and triggers the big CNN for recovering the unreliable ones.

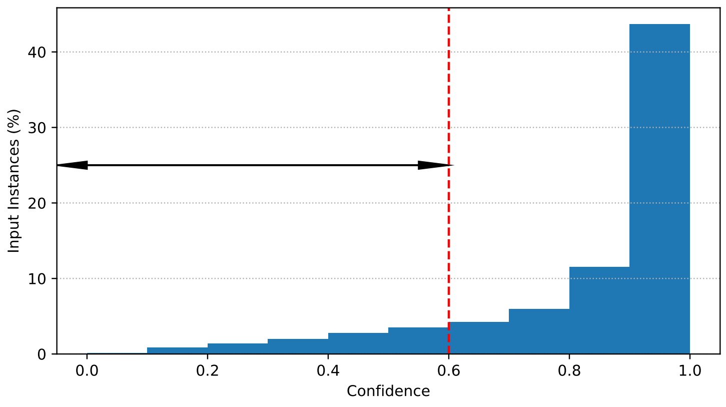
its output reliability and activate the big CNN for recovering the unreliable outputs. Since inputs of CNN applications are usually non-trivial images, the partitioning task itself requires a CNN to achieve acceptable results. Considering the complexity and overhead of designing, training and running a partitioner per CNN pair, and the fact that an efficient DD design tends to run most of the inputs on the little CNN, we decide to eliminate the partitioner and keep the little CNN and the confidence probe always active. Consequently, the key component to exploit the synergy using DD is utilizing a lightweight and accurate-enough confidence probe.

6.3.2 Runtime

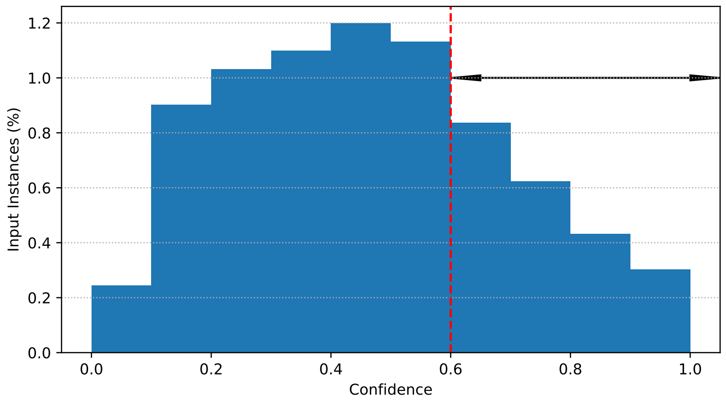
Figure 6.4 shows the runtime overview of DD. For each input instance, DD activates the little and efficient CNN, which is simpler than the big CNN and sacrifices the accuracy to achieve higher performance. To bridge this accuracy gap, the confidence probe evaluates the reliability of the little CNN output and sends the faulty ones to the big CNN for recovery.

As explained in Section 6.3.1, a lightweight and accurate confidence probe is necessary to achieve performance improvements using DD. Since DD design eliminates the partitioner and keeps the little CNN always active, we examine its output vector to predict the reliability of the final output.

Output Confidence: For classification tasks, CNNs convert an input instance (x) to an output vector of K elements (K is the number of classes). Using a softmax ($\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$) activation function



(a) Histogram of common corrects



(b) Histogram of complex corrects

Figure 6.5: Confidence distribution of Inception-v3 when it is paired with ResNet-152. Using a threshold of 0.6 separates most of the common corrects from the complex ones.

after the output layer provides the estimated probability of that the correct output is j for $j = 1, ..., K$ $(P_j = P(y = j|x))$. Consequently, $\max\limits_{1 \leqslant j \leqslant K} P_j$ represents the confidence level of the CNN in the final output.

To explore the idea of analyzing the confidence of the little CNN as an indicator of the output reliability, we use Inception-v3 and ResNet-152 as the little and big CNNs of a DD system, respectively. Then, the output confidence of Inception-v3 is measured for all 50000 images of ImageNet test set.

Figure 6.5 shows the confidence distribution of Inception-v3 for different data subsets of ImageNet test set. Figure 6.5a shows the histogram of confidences for common corrects, which could

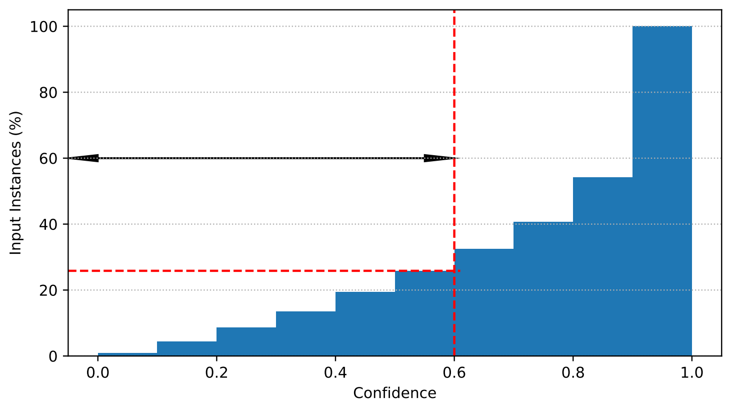


Figure 6.6: Cumulative histogram of all inputs. The threshold of 0.6 results in offloading 74% of inputs on the little CNN and a recovery rate of 26%.

be classified correctly by both CNNs and do not need recovery. On the other hand, Figure 6.5b demonstrates the histogram of confidences for complex corrects, which are classified correctly by the big CNN and wrongly by the little one, so they need the activation of the big CNN for recovery.

As shown, Inception-v3 classifies most of the common corrects with high confidence. In contrast, the classification confidence for complex corrects is low. Comparison of these two plots suggests that activating Inception-v3 as the little CNN and using a confidence threshold equal to 0.6 on the final output could distinguish these two types of correct outputs in most of the cases.

There are two types of outputs that can cause problems for this algorithm: low-confidence common corrects and high-confidence complex ones. Low-confidence common corrects result in unnecessary recoveries, which reduces performance and energy improvements but does not have any effect on the final accuracy of the system. High-confidence complex corrects result in false negatives for the confidence probe and degrade the final accuracy. However, as explained in Section 6.2.2, high-confidence odd corrects could compensate for this accuracy degradation.

Figure 6.6 shows the cumulative histogram of confidences for all 50000 inputs. As demonstrated, the threshold of 0.6 on the output confidence results in the solo activation of Inception-v3 for 74% of inputs and activating ResNet-152 only for 26% of inputs to recover the unreliable outputs instead of running ResNet-152 for the entire dataset.

However, it is not necessary to limit the system to a single confidence threshold. In addition

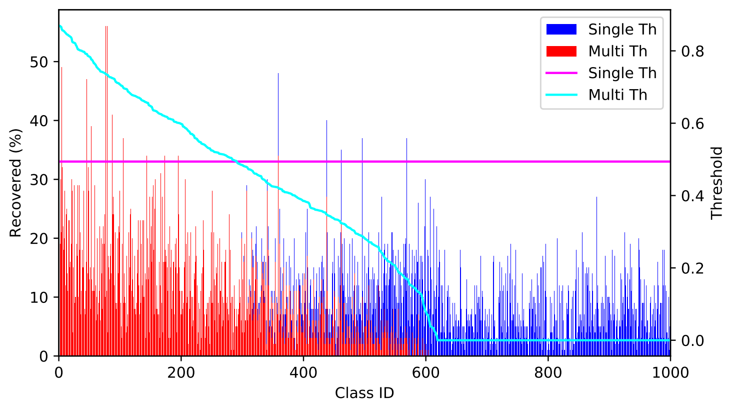


Figure 6.7: Switching from single threshold to 1000 thresholds reduces the recovery rate by 6.72%

to the confidence, there is another valuable piece of information provided by the little CNN during runtime, which is the predicted class. Hence, it is possible to define a confidence threshold per class. Figure 6.7 shows the threshold per class ID for both single threshold and multi-threshold scenarios in the case of pairing ResNet-152 and Inception-v3. Class IDs are sorted from the highest threshold on the left to the lowest one on the right. In addition to thresholds, this figure demonstrates the recovery rate per class ID for both scenarios to achieve the same accuracy as ResNet-152. As illustrated, switching to multi-threshold reduces the total recovery rate from 19.12% to 12.40%. Another interesting point is that for 381 out of 1000 classes there is no recovery required at all.

confidence probe: Confidence probe is the dynamic and input-driven part of the system. During runtime, for each input instance, the lightweight confidence probe compares the output confidence of the little CNN with predetermined thresholds to detect possible errors and activate the big CNN for recovery.

6.4 Automatic System Setup

When a service request arrives, it is necessary to allocate hardware resources and load proper CNN models and parameters to process the incoming inputs. DD is configured once and remains the same for all inputs unless a service request with a new target output quality arrives. Based on

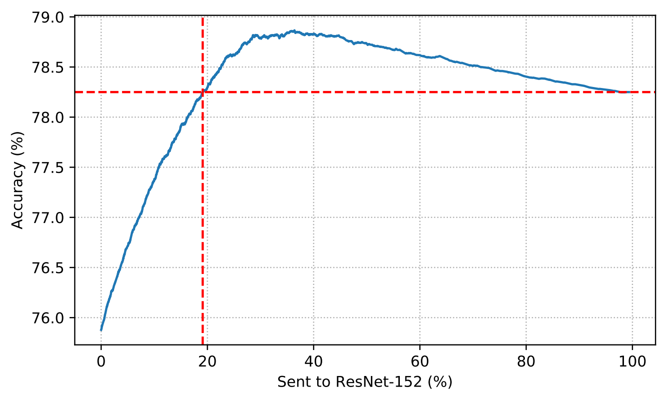


Figure 6.8: Accuracy vs recomputation rate for pairing ResNet-152 and Inception-v3. Less than 20% of inputs require recovery to achieve the same accuracy as ResNet-152.

Figure 6.3, there are two main additional steps for setting up a DD system: CNN pair selection and confidence probe configuration.

6.4.1 CNN Pair Selection

To achieve performance improvements by replacing a single CNN with DD, it is crucial to find the most synergistic pair of CNNs existing in the model pool, which is able to meet the target output accuracy.

During training, DD trains each single CNN separately. Then, evaluates them on the validation set and saves the accuracy numbers and confidence distributions in the system. During runtime, when a service request with a new target output quality arrives, DD estimates the final accuracy for different recomputation rates based on the accuracy of the big and little CNNs and forms plots similar to Figure 6.8 for all possible pairs.

Figure 6.8 shows the final accuracy for different recomputation rates when pairing ResNet-152 and Inception-v3. The y-axis shows the top-1 accuracy of DD and the x-axis shows the percentage of inputs which are sent to the big CNN for recovery. As shown, it is possible to achieve the same accuracy as ResNet-152 (78.25%) by solo activation of Inception-v3 for more than 80% of inputs.

Although it is possible to pick the optimum point for a fixed pair of CNNs, it is not possible to compare the performance of two different pairs without converting the recomputation probability

(P_r) to latency (L) and energy (E) numbers.

Latency and Energy Estimation: DD trains a regression model, same as Neurosurgeon [127], to estimate energy and latency based on the number and size of input and output feature maps as well as the characteristics of each layer including layer type, stride, kernel and group size for convolutional and pooling layers, in addition to available hardware resources. These models are CNN-invariant. Hence, they are trained once for a set of CNNs, then they could be used for different CNN applications during runtime.

To select the most efficient pair, DD computes $E_{pair} = E_{little} + E_{big} \times P_r$ and $L_{pair} = L_{little} + L_{big} \times P_r$ for each pair. After that, it chooses the pair which maximizes the latency and energy reduction while meeting the output target quality.

6.4.2 Confidence Probe Configuration

Single Threshold: Since the confidence distribution of each CNN is saved in the system, it is possible to convert back P_r to a confidence threshold on the little CNN after choosing the optimal pair. For example, for Figure 6.8, the optimal P_r of 19.1% is converted to confidence threshold of 0.49 using the output confidence distribution of Inception-v3.

Multi-Threshold: When the optimal pair is selected, finding one threshold per class to minimize the recovery rate while maintain the accuracy of the big CNN is a multiple-choice knapsack problem [166], and DD uses the following heuristic to solve it. First, it finds the optimal threshold per class which maximizes the number of correct predictions of that class. Then, it linearly reduces all thresholds by equal steps until it achieves the point that the accuracy of DD is the same as the accuracy of the big CNN. This threshold reduction reduces the number of recoveries and increases the overall performance and efficiency.

6.4.3 Optimized CNN Pair Selection

The naive way for CNN pairing and confidence threshold selection is to exhaustively search the entire design space for each request. The exhaustive search requires running the confidence thresh-

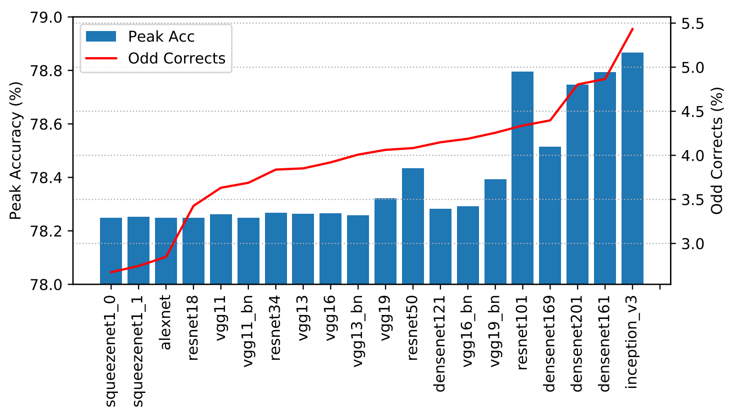


Figure 6.9: Peak accuracies and size of odd subsets for pairing ResNet-152 with 20 other CNNs. The odd subset size predicts the complementariness.

old selection algorithm for all $\binom{N}{2}$ possible pairs (N is the number of existing CNNs in the model pool). Nowadays, the number of available CNNs for a specific task is increasing rapidly which results in huge design spaces. Hence, the overhead of the exhaustive search becomes prohibitive. For example, there are 101 models available on [167] for image classification on ImageNet and 40 of them are introduced in 2019. Consequently, it is necessary to utilize heuristics to limit the search space.

As explained in Section 6.2.2, there are two subsets of data which are beneficial for the synergy between a pair of CNNs: common corrects and odd corrects. The odd corrects plus common corrects equals to all correct predictions of the little CNN, which does not reveal any information about the synergy between the little and big CNNs. In addition, the size of the common subset is highly correlated to the accuracy of the little CNN. On the other hand, the size of the odd subset is sensitive to both accuracy of the little CNN and how it complements the accuracy of the big one. Consequently, the size of the odd subset could be measured to indicate the potential synergy between a pair of CNNs.

To investigate this idea, we use the same 21 CNNs as Section 6.2.2. For this experiment, ResNet-152 is fixed as the big CNN and other 20 CNNs are swept as the little one. Then, for each pair, the peak accuracy of DD is measured on ImageNet test set by sweeping the confidence threshold. Figure 6.9 shows the peak accuracy as well as the size of the odd subset for each pair.

Table 6.1: CNN information for different benchmarks.

Model	Parameters	FLOPs	Top-1 Acc.
AlexNet [91]	244 MB	727 MFLOPs	56.63 %
SqueezeNet1-0 [137]	5 MB	837 MFLOPs	58.00 %
SqueezeNet1-1 [137]	5 MB	360 MFLOPs	58.18 %
VGG-11 [4]	531 MB	8 GFLOPs	68.87 %
VGG-11-bn [4]	531 MB	8 GFLOPs	70.41 %
VGG-13 [4]	532 MB	12 GFLOPs	69.98 %
VGG-13-bn [4]	532 MB	12 GFLOPs	71.62 %
VGG-16 [4]	553 MB	16 GFLOPs	71.63 %
VGG-16-bn [4]	553 MB	16 GFLOPs	73.48 %
VGG-19 [4]	575 MB	20 GFLOPs	72.36 %
VGG-19-bn [4]	575 MB	20 GFLOPs	74.22 %
ResNet-18 [139]	47 MB	2 GFLOPs	69.64 %
ResNet-34 [139]	87 MB	4 GFLOPs	73.27 %
ResNet-50 [139]	102 MB	4 GFLOPs	76.01 %
ResNet-101 [139]	178 MB	8 GFLOPs	77.31 %
ResNet-152 [139]	240 MB	11 GFLOPs	78.25 %
Inception-v3 [157]	95 MB	6 GFLOPs	75.88 %
DenseNet-121 [168]	32 MB	3 GFLOPs	74.47 %
DenseNet-161 [168]	114 MB	8 GFLOPs	77.15 %
DenseNet-169 [168]	56 MB	3 GFLOPs	75.63 %
DenseNet-201 [168]	79 MB	4 GFLOPs	76.93 %

The x-axis represents the name of little CNNs, which is sorted from the smallest odd subset on the left to the largest one on the right. As shown, the peak accuracies of DD systems with bigger odd subsets are usually higher than other ones. Consequently, the size of the odd subset could be used to predict how a pair of CNNs would complement each other.

Heuristic: For each output target quality adjustment, DD chooses the smallest CNN, which has higher or equal accuracy as the target output accuracy, from the model pool as the big CNN. Then, it uses the size of the odd subsets to indicate the top synergistic candidates for the little CNN. Finally, CNN pairing and confidence threshold selection are run only for the top candidates to find the most efficient little CNN and the corresponding confidence thresholds.

6.5 Evaluation

6.5.1 Experimental Methodology

Benchmarks: We evaluate DD using a benchmark suite of 21 CNNs on ImageNet dataset. The original validation set is divided into two subsets of 25000 images to be used as a validation set for configuring DD and a test set for evaluation. A brief description of each benchmark is presented in Table 6.1. For each row, the first column presents the name of the CNN. Second column shows the amount of memory required to keep the parameters of the model, which indicates the size of the CNN. Third column is the number of floating-point operations (FLOPs) required to process a single input, which is limited to floating-point multiply and adds and indicates the computational complexity of the model. The last column contains the top-1 accuracy of trained CNNs obtained from pyTorch Torchvision package [169]. The end-to-end inference latency of each CNN is affected by memory access patterns and the way the CNN gets scheduled on the actual hardware in addition to FLOPs.

Software frameworks: We use PyTorch [170], an open-source machine learning library for Python based on Torch, cuDNN [142], NVIDIA's GPU-accelerated library for DNNs, and BigHouse [171], an open-source infrastructure for simulating datacenter systems, to implement and test our system.

Hardware platform: The NVIDIA TITAN X GPU [141] is used as the processor on the server to evaluate our approach. Although we use 32-bit floating-point representation for the evaluation, it is possible to apply precision reduction to both baseline CNNs and DDs. Since precision reduction is orthogonal to our approach, the performance improvements will have the same trend as full-precision results.

6.5.2 Latency Improvement

Figure 6.10 shows the latency of DD normalized by the latency of the traditional CNN as a service method for different benchmarks. For each experiment, DD uses the heuristic explained in

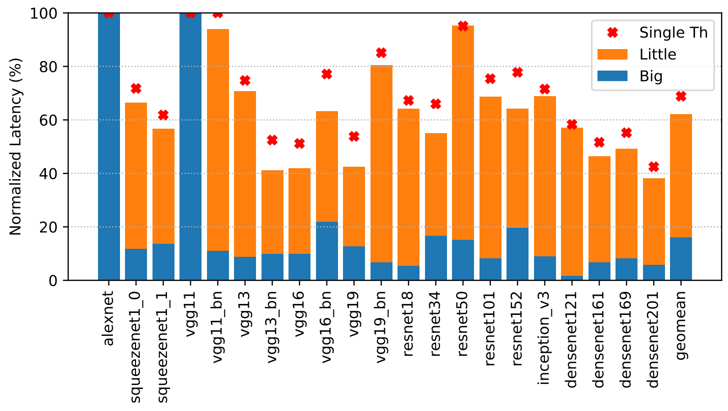


Figure 6.10: Average latency of DD with heuristic search for 100% of baseline accuracy.

Section 6.4.3 to find the most synergistic pair in the model pool, which contains all CNNs from Table 6.1. The output target quality of each experiment is determined by the traditional single CNN approach. Since the heuristic chooses the smallest network with higher or equal accuracy as the output target quality for the big CNN of DD, for each experiment, the big CNN is the same network as the baseline single CNN shown on the x-axis. Each bar demonstrates the latency breakdown of DD. The bottom part is related to the normalized average latency of the big CNN on the entire test set, which is the normalized average recovery time. And, the top part shows the normalized latency of the little CNN. In addition, the cross indicates the normalized latency in the case of single threshold DD.

On average, multi-threshold DD reduces the latency of the traditional method to 62% while the target output quality is met. In addition, it increases the average latency savings by 7% in comparison to the single threshold scenario. Since the training and validation sets are representative of the test set, and the odd corrects could compensate for accuracy degradations, DD meets the target output quality for all 21 benchmarks. DD is capable of finding highly efficient and accurate-enough solutions by pairing faster little CNNs with higher recovery rates and slower little CNNs with more relaxed recovery policies.

As demonstrated, DD decreases the latency for all benchmarks except AlexNet and VGG-11. Based on Table 6.1, AlexNet is the smallest and least accurate CNN, which requires the minimum FLOPs for each inference. Therefore, DD is unable to find a CNN in the model pool as a candidate

Table 6.2: Single threshold DD configuration.

Big CNN	Little CNN	Confidence Threshold
AlexNet	AlexNet	0.00
SqueezeNet1-0	AlexNet	0.23
SqueezeNet1-1	AlexNet	0.25
VGG-11	VGG-11	0.00
VGG-11-bn	VGG-11-bn	0.00
VGG-13	VGG-11-bn	0.04
VGG-13-bn	VGG-11-bn	0.45
VGG-16	VGG-11-bn	0.42
VGG-16-bn	ResNet-18	0.63
VGG-19	VGG-11-bn	0.48
VGG-19-bn	ResNet-34	0.37
ResNet-18	VGG-11	0.23
ResNet-34	VGG-11-bn	0.53
ResNet-50	VGG-13	0.85
ResNet-101	ResNet-50	0.48
ResNet-152	ResNet-101	0.44
Inception-v3	VGG-19-bn	0.39
DenseNet-121	VGG-19-bn	0.20
DenseNet-161	ResNet-50	0.43
DenseNet-169	VGG-19-bn	0.44
DenseNet-201	ResNet-50	0.39

for the little CNN to be paired with AlexNet and reduce its latency. For VGG-11, there is a big gap between its accuracy and smaller CNNs, however the latency gap is much smaller. Consequently, pairing it with smaller CNNs results in considerable accuracy losses without performance gains. DD successfully spots this limitation of the model pool and runs the single CNN for this case.

Table 6.2 shows the configuration of single threshold DD for different benchmarks in details. In each row, the first column demonstrates the big CNN. The second column shows the little CNN paired with the big one. And, the last column shows the threshold on the classification confidence of the little CNN to activate the big one for recovering the outputs with confidences lower than the threshold.

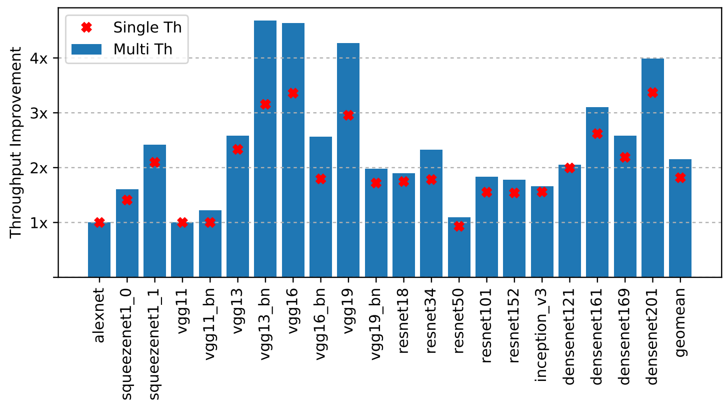


Figure 6.11: Throughput of DD with heuristic search vs. the traditional approach for 100% of baseline accuracy.

6.5.3 Datacenter Throughput Improvement

Improving the average latency per inference by DD, which is shown in Figure 6.10, provides opportunities to increase the datacenter throughput. The goal is to maintain the same response time as baseline while processing more queries per second (QPS), hence, improving the datacenter throughput. The server load is increased for both baseline and DD to the point that the response time values match the baseline latency. Then, we compute the throughput gains by comparing the QPS values of these two solutions. In this section, we deploy the DD designs with 100% of baseline accuracy, and each benchmark is evaluated in isolation in order to prevent the requests from different benchmarks from interfering with each other.

The BigHouse [171] infrastructure is used to evaluate datacenter behaviors. Bighouse uses the statistical models of task service times and user queries to simulate the expected performance of datacenters in a more reasonable time compared to using detailed micro-architectural models. To collect service times for each benchmark, we run each design multiple times with different inputs and derive the overall distribution of their latencies. We also use Google web search query distribution [172] for the query inter-arrival rate.

Figure 6.11 presents throughput gain of both single threshold and multi-threshold DD for each individual benchmark. Based on these results, DD achieves higher performance in comparison to the traditional approach whereas throughput gains change in a range of $1.0\times$ to $4.7\times$, averaging in $2.1\times$ for multi-threshold and $1.8\times$ for single threshold. If DD was used to reduce the response time of the baseline by $K\times$, the maximum throughput improvement would be $K\times$. However, the goal is to maintain the same response time as baseline while processing more queries per second. Hence, the throughput could improve more than $K\times$.

The highest improvements are achieved by VGG-13-bn and VGG-16 benchmarks. In these benchmarks, the difference between latencies of the big and little CNNs is high, and the recovery frequency is relatively low. Therefore, there are more opportunities for throughput improvement. On the other hand, no gains are observed for AlexNet and VGG-11. These two benchmarks are explained as in Section 6.5.2. Another interesting observation is that for single threshold ResNet-50, although DD obtains 7% reduction in latency as shown in Figure 6.10, the throughput gain is insignificant. The main source of this result is that it needs to activate both little and big CNNs for nearly 54% of queries. Therefore, more than half of the requests are experiencing tail latencies leading to increased average of waiting times, hence, higher response times.

6.5.4 Approximation

Since DD design has more degrees of freedom than the traditional approach, the pair selection algorithm and confidence thresholds could be used as knobs to dynamically choose target output qualities other than the accuracy of a single CNN. Therefore, there is an opportunity for the user to sacrifice a marginal amount of accuracy to obtain further performance and energy gains.

Figure 6.12 presents the latency breakdown of multi-threshold DD with the output accuracy relaxed to 99% of the baseline accuracy in the same format as Figure 6.10. In comparison to the 100% accurate DD, the average latency is reduced further by 5% for 0.85% additional error.

6.5.5 Heuristic Coverage

An exhaustive search in a model pool of 21 single CNNs results in analyzing $\binom{21}{2} = 210$ pairs of CNNs. Our heuristic chooses the big CNN based on the output target quality and limits the number of options for the little CNN. We found that in our model pool, limiting little CNN search to top-3

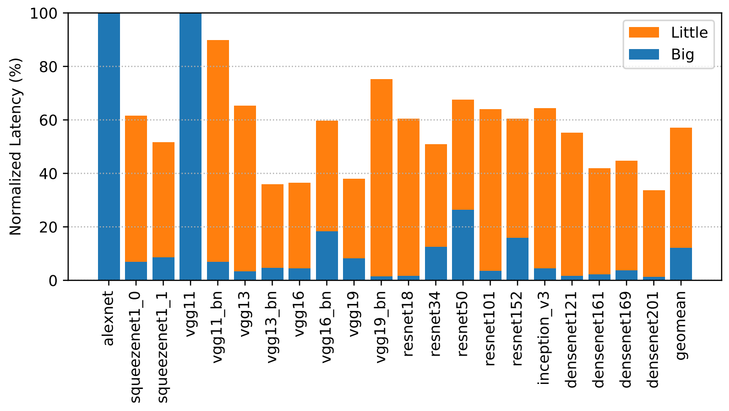


Figure 6.12: Average latency of DD with heuristic search vs. the traditional approach for 99% of baseline accuracy.

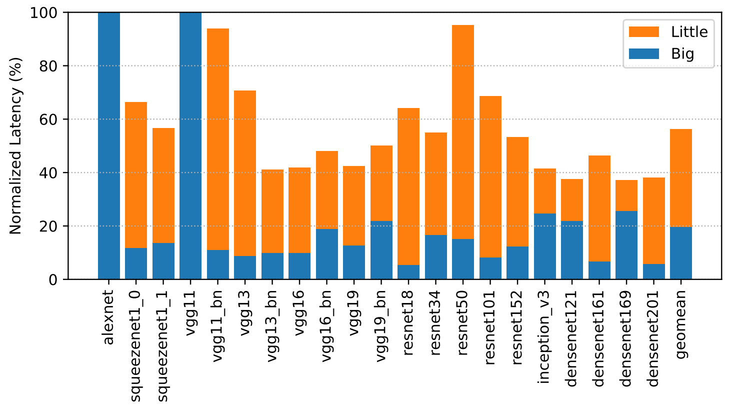


Figure 6.13: Average latency of DD with exhaustive search vs. the traditional approach for 100% of baseline accuracy.

CNNs with highest odd subset sizes results in CNN pairs with negligible performance degradation in comparison to exhaustive search.

Figure 6.13 shows the result of the same experiment as Figure 6.10 while using exhaustive search instead of the heuristic. The full design exploration results in a 6% additional improvement of the normalized latency. Another interesting point is that for 15 out of 21 cases, the heuristic finds the exact same CNN pair and confidence thresholds as the exhaustive search.

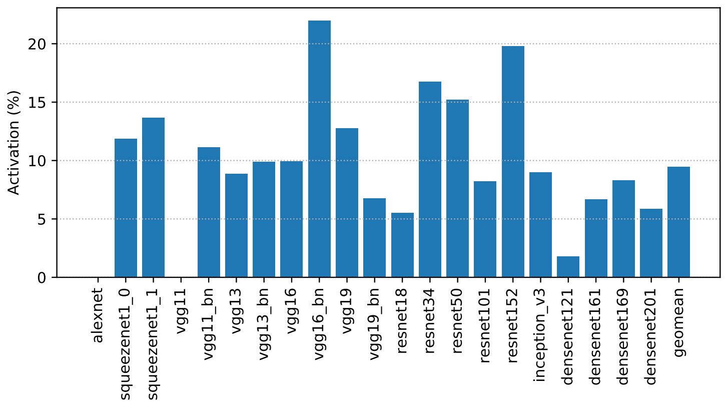


Figure 6.14: The percentage of outputs that are indicated by the confidence probe as being potentially incorrect and sent to the big CNN for recovery.

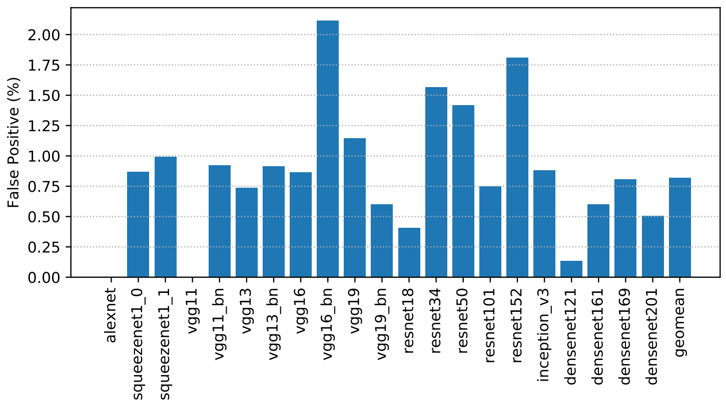


Figure 6.15: False positives of the confidence probe.

6.5.6 Confidence Examination

Figure 6.14 shows the percentage of outputs that the confidence probe detects them as potential faulty results and sends them to the big CNN for recomputation. The average recovery rate is 9.5%. The recovery rate for AlexNet and VGG-11 is zero because DD automatically decides to use the single CNN approach for these benchmarks as explained in Section 6.5.2. Hence, the confidence probe remains idle. For all benchmarks, the recovery rate is below 25%, which means more than 75% of the inferences are successfully offloaded on the little CNN.

Figure 6.15 shows the false positive rate of the confidence probe, which means the percentage of odd corrects that are mistakenly sent to the big CNN for recovery. The average false positive

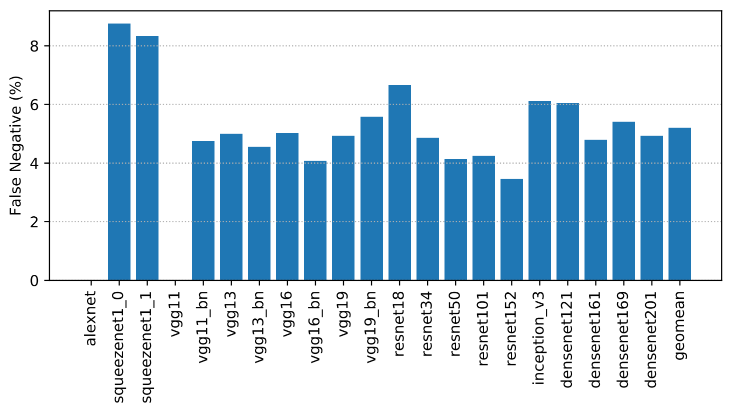


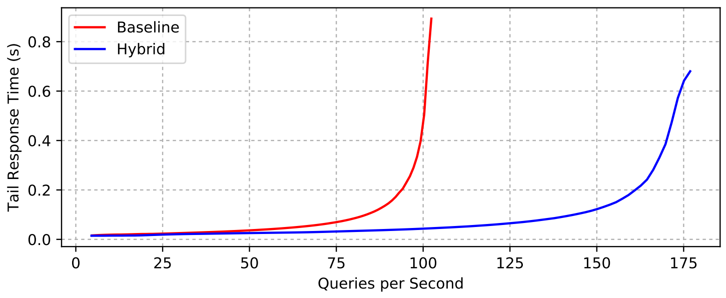
Figure 6.16: False negatives of the confidence probe.

rate is 0.8%. The maximum false positive rate is 2.1%, and these low rates result in running most of the odd corrects on the little CNN, which can increase the final accuracy of DD. Hence, there is more room for relaxing other design restrictions to achieve higher performance while maintaining baseline accuracy.

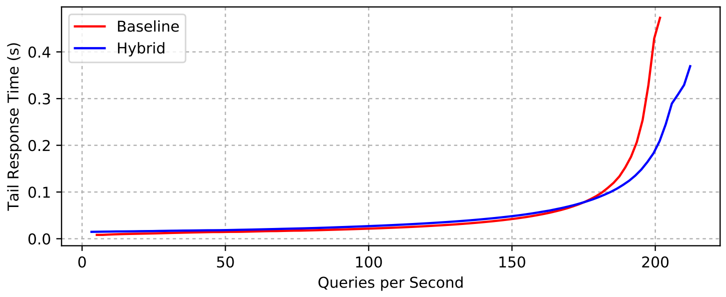
Figure 6.16 shows the false negative rate of the confidence probe, which means the percentage of complex corrects that are not sent to the big CNN for recovery. DD keeps the false negative rate of all benchmarks below 8.8% to limit the accuracy loss in a compensable range. The lost accuracy is compensated by odd corrects. Since the big CNN is a more complex model, the cases that the big one is correct and the small one is wrong are more common than when the small one is correct and the big one is wrong. Hence, false positive is much lower than false negative. The low false positive rates result in running most of the odd corrects on the little CNN and improving the final accuracy.

6.5.7 Tail Response Time with Load Variation

In order to produce reliable outputs for all users in DD, a number of requests are required to go through both little and big CNNs, which face a higher latency for computation than the traditional single CNN approach and lead to higher tail response times. To evaluate the effect of recomputations for each benchmark, we use BigHouse infrastructure and collect the 95^{th} percentile of



(a) DenseNet-121



(b) ResNet-50

Figure 6.17: Comparison of tail response time behavior under various server loads for traditional approach and DD with 100% of baseline accuracy.

response times for different server loads.

Based on the results of this experiment, two kinds of behavior is observed in the benchmarks. Figure 6.17 presents an example for each group: DenseNet-121 and ResNet-50. In these graphs, x-axis represents the load on the server (QPS) and y-axis shows the corresponding tail response time. To analyze these results, first, we need to consider the factors that affect the response time including computation latency and waiting time. When the load on the server is low, the waiting time is negligible and the response time depends on the computation latency. Therefore, the 95^{th} percentile response time results depend on the average rate of required recomputations. As a result, for benchmarks that require less than 5% of requests to go through both little and big CNNs, such as DenseNet-121, the tail response time of DD is lower than the baseline.

On the other hand and with heavy traffic, we observe that the tail response time of DD is lower for all benchmarks. When the load on the server is increased, the waiting time plays a vital role on the response time whereas the computation latency is still the same as the case with low traffic. Considering that the waiting time of request is correlated to the average computation latency of active queries, and the average computation latency is lower for DD as presented in Section 6.5.2, DD results in lower average of waiting times. The gap between waiting times of traditional approach and DD is further increased by increasing the traffic. As a result, lower tail response time is achieved by DD in higher loads.

6.6 Conclusion and Future Work

Traditional CNN as a service approaches employ either a single complex CNN or an ensemble of multiple CNNs for all inputs of an application without considering the inherent diversity of input instances. However, the complexity of CNNs is growing fast, and inference queries require a significant amount of compute resources in comparison to traditional text-based web services. In this chapter, we introduce *Dynamic Duo (DD)*, a novel *input-driven synergistic deep learning system*. It pairs an efficient less accurate CNN (little) with a complex more accurate CNN (big) and offloads most of the computation on the little one. However, for a subset of the inputs, the little CNN cannot yield confident predictions and is often wrong. Hence, a dynamic confidence examination technique is employed to probe the outputs of the little CNN and invoke the big one only for recovering the low-confidence inferences. Compared to the traditional approach, DD improves the datacenter throughput by $2.1\times$ and reduces the average inference latency by 38% on an NVIDIA TITAN X GPU with 100% accuracy of the baseline.

Adding more CNNs to the system is left for future work. Using confidence threshold on all smaller networks and activating multiple networks, instead of two, in cases which require recovery might reduce the average latency savings. Moreover, it makes the tail response time worse. One possible solution is applying multiple thresholds on the confidence of the smallest CNN and choosing only one of the big CNNs for recovery based on the confidence, which means a DD system with a fixed little CNN and multiple choices for the big CNN during runtime. The other challenge is expanding the pairing heuristic to determine a group of most synergistic CNNs because the exhaustive search grows exponentially by adding more CNNs. In addition, a theoretical way to choose confidence thresholds could reduce the training time further and help to build systems with more than two CNNs.

Expanding DD to other types of networks, such as recurrent neural networks, and other applications is another interesting direction for feature work. We chose CNNs because they use a softmax layer at the end to produce probabilities for different classes, and we exploit these numbers to predict the situations that require recovery.

CHAPTER 7

Double-Shot Neural Network Pruning

7.1 Introduction

In recent years, DNNs have received lots of attention and achieved dramatic accuracy improvements in many tasks. This success relies on DNNs with millions or even billions of parameters, and the availability of GPUs with very high computation capability. In addition to inference, training of these huge models to get reasonable performance is very time-consuming as well. For example, one forward pass of ResNet50 [139] model requires 4 GFLOPs of computation and training requires 10^{18} FLOPs, which takes 14 days on one state-of-the-art NVIDIA M40 GPU [173].

As larger NNs with more layers and nodes are getting popular, reducing their memory and computational complexity becomes critical, especially for some real-time applications, such as online learning, virtual reality, augmented reality and smart wearable devices. Many NNs are overparameterized [174, 175], which enables the opportunity of compressing each layer [175, 11, 176] or the entire network [177] to reduce their size and complexity. These compression techniques are necessary for alleviating the fundamental challenges in deploying deep learning systems on portable devices with limited resources.

A well-known category of network compression technique is network pruning methods, which are a set of techniques for removing weights, filters, neurons, or other structures from NNs. Pruning relies on the fact that NNs are often found to be highly overparameterized, and redundant parameters can be eliminated while preserving accuracy [178, 49].

Most of the existing methods in prior work attempt to find a sparse network from the pretrained reference network either based on a saliency criterion [179, 178, 49] or using sparsity enforcing penalties to augment the loss function [180, 181]. However, these methods include pruning as a part of an iterative optimization procedure, which results in several rounds of expensive prune – fine-tune. In addition, these traditional pruning approaches require heuristic design choices with additional hyperparameters, which makes them non-trivial to extend to new architectures and tasks for a nonexpert user.

While pruning has been a standard practice for model compression, some recent efforts start empirically linking it to a potential approach for more efficient training. Training sparse pruned networks directly from scratch often fails and results in networks underperforming their dense counterparts [182, 11]. However, the latest series of works [183, 184] reveal that dense, randomly-initialized networks contain small sparse networks (winning tickets), which can match the accuracy of original networks when trained in isolation. However, there is a major gap between the existence of winning tickets and more efficient training, since winning tickets were only identified by iteratively pruning a fully trained dense network.

To alleviate the complexity and inefficiency of applying pruning techniques and training pruned sparse networks, SNIP [185] introduces single-shot pruning, which uses a saliency criterion that identifies important connections of a network for a given task in a data-dependent way before training. Then, the sparse pruned network is trained in the standard way. Hence, it eliminates the need for pretraining, complex pruning schedules and additional hyperparameters. It does not require machine learning expertise and can be applied to various architectures with no modifications.

Although SNIP claims that it retains connections which are indeed essential for a given task by focusing only on architectural features of the network and forming the pruning algorithm based on randomly initialized weights, we show that there is some opportunity to improve the performance of SNIP by augmenting the architectural insights with information about the importance of weight values. We propose a double-shot noniterative pruning approach, which translates the global sparsity rate to layer-wise local sparsity rates using SNIP, then, prunes the trained dense network utilizing the local sparsity rate and connection importance metrics, and at the end, trains the sparse network in the same way as the dense one.

Although our double-shot pruning method requires pretraining of the dense network in addition to the training of the sparse network, it is still a noniterative pruning approach and does not require any complex pruning schedule. Since our method requires no additional hyperparameters, and both training of the dense and the sparse network are performed in the standard way, machine learning expertise is not required for applying it to various architectures and different tasks.

We evaluate our method using seven different CNN architectures on MNIST and CIFAR-10 classification datasets. Furthermore, we investigate the effect of five different connection importance metrics on the performance of double-shot pruning. The most proper metric, on average, improves the accuracy of SNIP by 0.79% while maintaining the same sparsity level for all tested architectures.

7.2 Background and Motivation

7.2.1 Neural Network Pruning

Since, NNs are usually overparameterized, pruning techniques are proposed to eliminate unnecessary weights [178, 186, 49, 182]. These approaches are capable of reducing parameter-counts by more than 90% without accuracy loss. It is possible to take advantage of these network size reductions for improving energy efficiency and performance of inference [49, 50, 187, 188, 189].

Training a NN is an optimization problem, which could be formulated as follows:

$$\begin{gathered}\min_{\mathbf{w}} L(\mathbf{w};\mathcal{D})) = \min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n}\ell(\mathbf{w};(\mathbf{x}_i,\mathbf{y}_i)))), \\ s.t. \quad \mathbf{w}\in\mathbb{R}^m.\end{gathered} \tag{7.1}$$

$\mathcal{D} = \{(\mathbf{x}_i,\mathbf{y}_i)\}_{i=1}^{n}$ represents the training dataset, $\mathbf{w}$ is the set of NN parameters, m is the total number of parameters, and ℓ is the standard loss function (e.g., cross-entropy). It is possible to add

$\|\mathbf{w}\|_0 \leq k$ to the above formula as a constraint to formulate pruning as a constrained optimization problem. k indicates the sparsity rate, which means the number of non-zero parameters, and $\| \cdot \|_0$ is the standard L_0 norm.

There are two major groups of approaches to achieve a sparse NN by optimizing the above problem. The first group prunes a NN based on some saliency criteria [49, 190, 178, 186]. These methods measure the sensitivity of the loss with respect to the activations or weights, and try to prune redundant neurons or connections. Since these criteria are heavily dependent on weight values, these methods are iterative (i.e., require pretraining and many expensive iterations of pruning and fine-tuning.) and prohibitively slow.

The second group of methods augment the loss function with sparsity enforcing penalty terms [180, 191, 192, 181]. Then, during training, back-propagation effectively penalizes the weights, and weights below a certain threshold may be deleted. These methods usually result in lower sparsity than saliency-based techniques.

Both groups of methods require machine learning expertise to define the saliency criteria, sparsity enforcing penalty terms, additional hyperparameters and training schedule. In addition, they require manual effort to tune hyperparameters to obtain acceptable results, which makes them non-trivial to be extended to new architectures and different tasks. Furthermore, the architectures uncovered by pruning approaches are harder to train from scratch and result in lower accuracy than the original networks [183].

7.2.2 Single-Shot Pruning

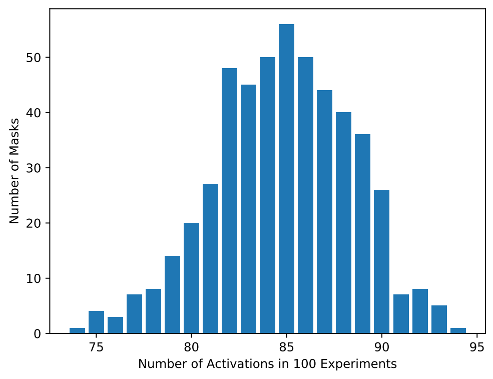
To overcome these shortcoming, recent prior work proposes single-shot network pruning (SNIP), which prunes a given network once at initialization before training [185]. It uses a saliency criterion which identifies structurally important connections for the given task. Not only it eliminates the need for pretraining and the complex pruning schedule, but also it is robust to architecture variations. After pruning, it trains the sparse network in the standard way. Therefore, this method does not require the expensive prune – retrain cycles.

SNIP measures the importance of each connection independently of its weight by introducing masks $\mathbf{c} \in \{0,1\}^m$, which indicate whether each connection ($\mathbf{w}$) is present or not. Hence, the constrained optimization problem is modified as follows:

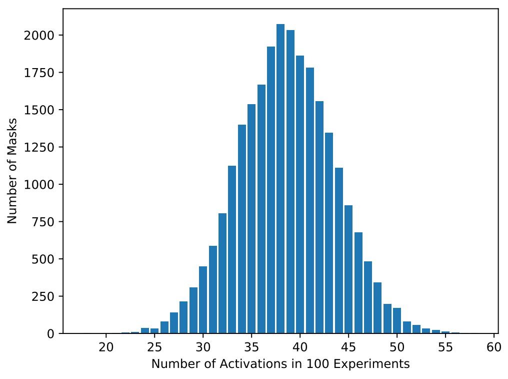
$$\begin{gathered}\min_{\mathbf{c},\mathbf{w}} L(\mathbf{c} \odot \mathbf{w}; \mathcal{D})) = \min_{\mathbf{c},\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell(\mathbf{c} \odot \mathbf{w}; (\mathbf{x}_i, \mathbf{y}_i)))), \\ s.t. \quad \mathbf{w} \in \mathbb{R}^m, \quad \mathbf{c} \in \{0,1\}^m, \quad \|\mathbf{c}\|_0 \leq k, \end{gathered} \tag{7.2}$$

where $\odot$ represents the Hadamard product. Then, it uses one forward-backward pass and automatic differentiation to compute $\partial L/\partial c_j$ for all j st once, which approximates the influence of connections on the loss. At the end, the top-k connections are retained and the remaining are pruned.

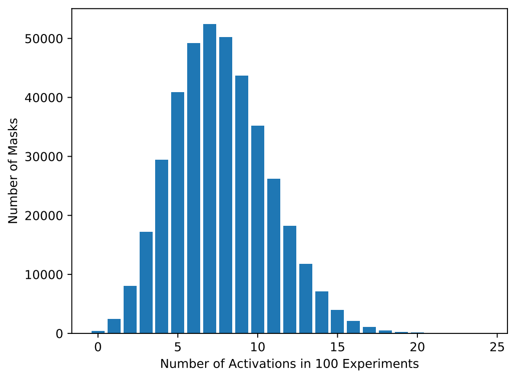
Figure 7.1 shows the histogram of active masks for 100 experiments using SNIP to prune LeNet-5 on MNIST dataset. For each experiment, weights are randomly initialized using variance scaling initialization [193], then, SNIP is used to prune 90% of the weights. After that, each mask is indicated as active (1) or inactive (0). After running the experiments, the masks of these 100 experiments are aggregated. Then, for each connection, we compute the number of experiments in which the mask of that specific connection is active, which could be a number between 0 and 100. At the end, the histogram of mask activations per layer is plotted. As shown in the figure, there is no single mask which is active for all 100 experiments. And, most of the masks are active in less than 50% of experiments. Hence, although the network architecture, dataset and sparsity rate are fixed, different weight initializations result in different pruned networks using SNIP. This in contrast to the claim that SNIP retains connections which are indeed essential for the given task. Consequently, we hypothesize that it is possible to enhance the performance of SNIP by paying attention to both connection importance based on the network architecture and weight values.



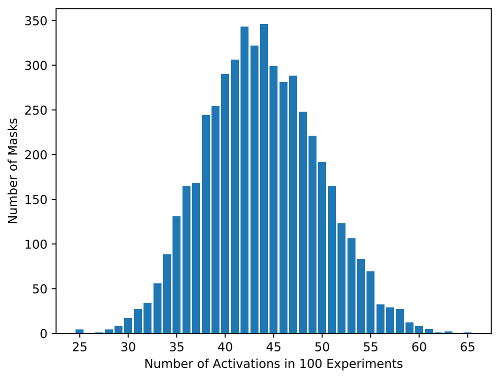
(a) Conv1



(b) Conv2

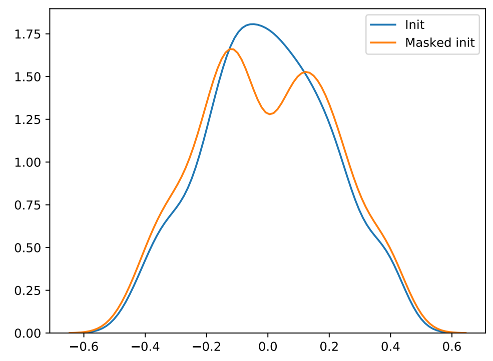


(c) FC1

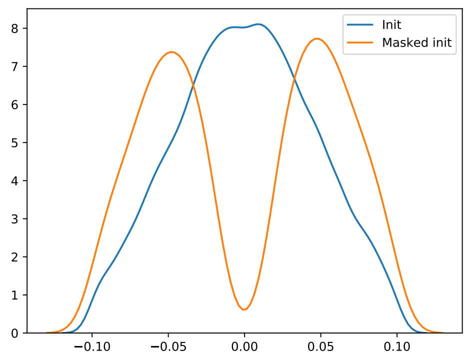


(d) FC2

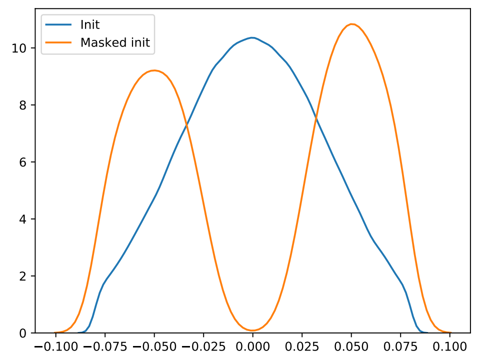
Figure 7.1: Histogram of active masks for 100 different weight initializations of LeNet-5 on MNIST dataset using SNIP. Changing weights changes the connections pruned by SNIP.



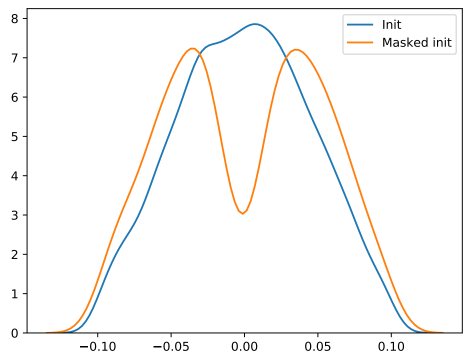
(a) Conv1



(b) Conv2



(c) FC1

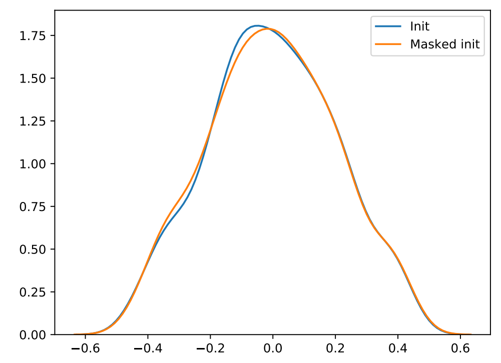


(d) FC2

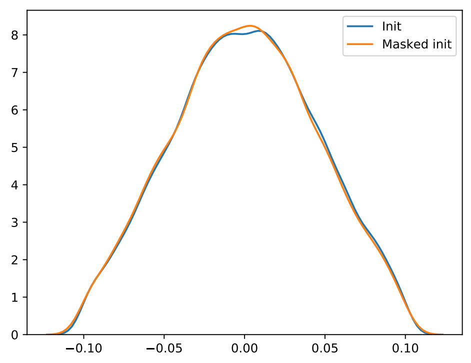
Figure 7.2: Distribution of weights before and after applying SNIP on different layers of LeNet-5 on MNIST dataset.

7.3 Single-Shot vs. Random Pruning

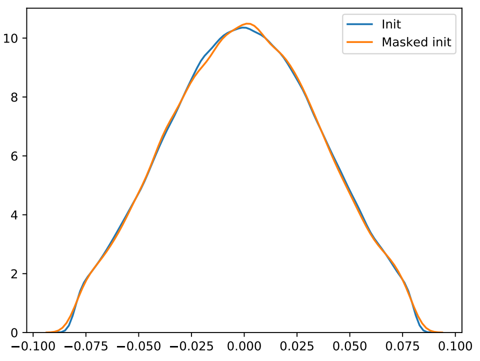
Figure 7.2 shows the distribution of weights using variance scaling initialization before and after applying SNIP with 90% sparsity rate on LeNet-5 network and MNIST dataset. As demonstrated, applying SNIP has a significant effect on the distribution of initialized weights.



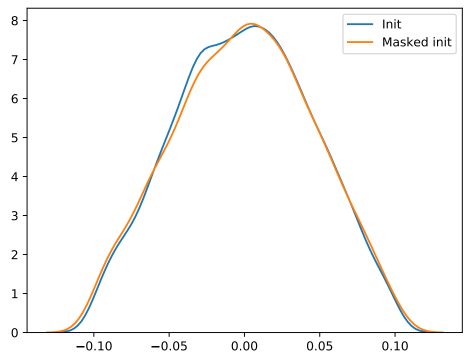
(a) Conv1



(b) Conv2



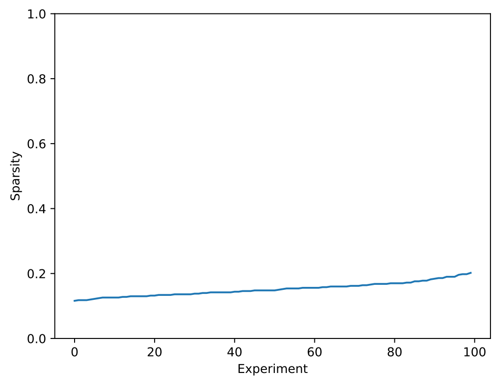
(c) FC1



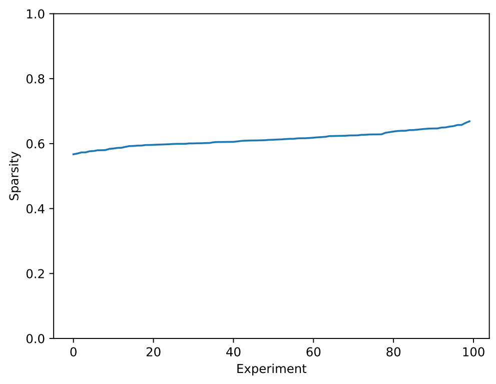
(d) FC2

Figure 7.3: Distribution of weights before and after applying random pruning on different layers of LeNet-5 on MNIST dataset.

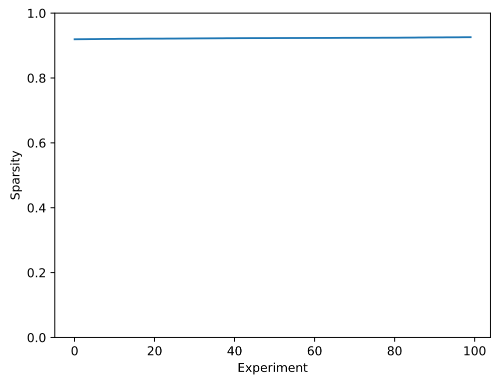
On the other hand, Figure 7.3 shows the distribution of weights for the same network, dataset, weight initialization and sparsity rate but using random pruning instead of SNIP. The main advantage of random pruning in comparison to SNIP is maintaining the same distribution of weight initialization, but the main disadvantage is completely ignoring the network architectural insights extracted by SNIP.



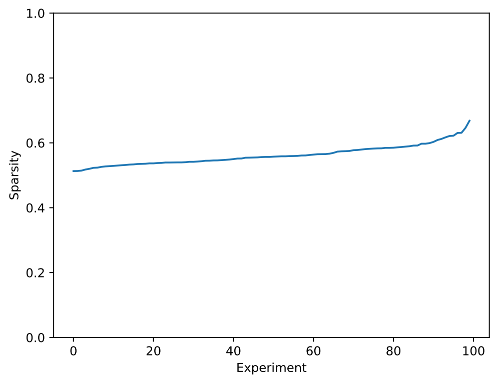
(a) Conv1



(b) Conv2



(c) FC1



(d) FC2

Figure 7.4: Histogram of active masks for 100 different weight initializations of LeNet-5 on MNIST dataset using SNIP. Changing weights changes the connections pruned by SNIP.

To augment the random pruning approach with some architectural insights, it is important to analyze SNIP further and discover the effect architectural features of the network on the final sparse network produced by SNIP. Based on Figure 7.1, we concluded that the position of connections in the network are not important to SNIP, and one specific connection in a unique network might be either pruned or retained for various random weight initializations. In contrast, if we look at the sparsity rate of each layer for the same 100 experiments, sorted from the lowest sparsity in the left to the highest sparsity in the right, as shown in Figure 7.4, it could be observed that the changes in weight initialization do not change the sparsity rate of each layer drastically. Comparing plots of different layers in Figure 7.4 results in two major observations.

First, although the global sparsity rate of the entire network is 90%, the local sparsity rate of various layers is very different from each other. For example, the first layer (Conv1), which is the closest to the raw input and contains important information, is on average pruned only 15%. On the other hand, the first fully connected layer (FC1), which is in the middle of the network, is on average pruned more than 92%. It is important to consider that FC1 is the largest layer of the network, and the extra 2% of pruned parameters in this layer let other layers be less than 90% sparse while the entire network is 90% pruned.

Second, although the masks themselves change a lot during these 100 experiments based on Figure 7.1, the variance of the sparsity rate for each layer is not high. Hence, SNIP is capable of defining local sparsity rates per layer based on the architectural features and the global sparsity rate of the entire network regardless of weight initialization. Consequently, we obtain local sparsity rates by applying SNIP, and then, apply local random pruning instead of global random pruning to more fairly evaluate the connection selection algorithm of SNIP per layer versus random pruning.

Figure 7.5 compares the classification error of LeNet-5 on MNIST dataset using global random, local random and SNIP pruning approaches for 90%, 95% and 98% sparsity rates. As demonstrated, switching from global random to local random by borrowing local sparsity rates from SNIP results in a considerable error reduction and produces very comparable results to SNIP. Hence, one of the key advantages of SNIP over pure random pruning is breaking down the global sparsity

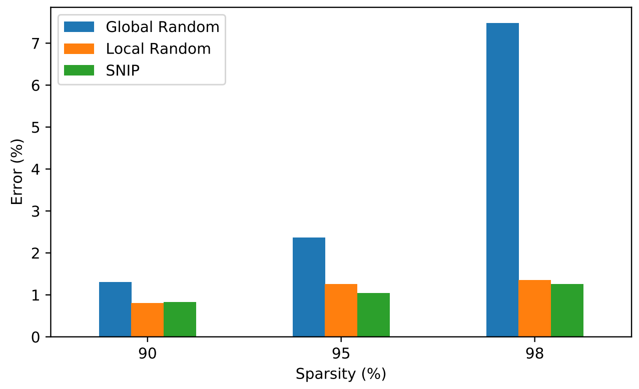


Figure 7.5: Classification error of pruned LeNet-5 on MNIST dataset using global and local random pruning and SNIP for different sparsity rates.

rate of the entire network and forming local sparsity rates for individual layers. Consequently, we hypothesize that it should be possible to improve the performance of SNIP by borrowing local sparsity rates as a conclusion of architectural features of the network, augment them with some insights from weight values and form a new pruning approach.

7.4 Double-Shot Pruning

Since we want to augment SNIP with some insights from weight values, it is not possible to structure the entire approach based on untrained randomly initialized networks. In other words, the information about weight values should be obtained from a trained or at least partially trained network because initial weights are random and does not necessarily reveal any useful information about the final values of weights. To satisfy this purpose, we propose a double-shot pruning approach, which prunes a network based on both architectural information and weight values as follows.

First the weights are randomly initialized using variance scaling initialization. Then, a mini-batch of training data is sampled and provided to SNIP to compute gradients of loss with respect to the masks. After that, the top-k (global sparsity rate provided by the user) gradients are considered as potentially important connections, and the remaining connections are indicated as potentially unimportant ones. These classification of connections helps our algorithm to break down the global sparsity rate properly and find the local sparsity rate per layer ($k_{l_1}, k_{l_2}, ..., k_{l_n}$). Then, the dense network is normally and fully trained. And, at the end, an importance metric is used to measure the importance of each connection based on its weight value. For each layer l_j, the importance value of its connections are sorted from low to high, the top-k_{l_j} connections with highest importance values are retained, and the remaining connections are pruned. As the final step, the pruned sparse network is regularly trained another time while the weights are initialized using the final values of the trained dense network.

Although our double-shot pruning requires training of the dense network as pretraining and training of the sparse network as the final training instead of a single training, it is still not an iterative pruning approach and does not require any complex pruning schedule. Our method requires no additional hyperparameters, and both training of the dense and the sparse network are performed in the standard way. In addition, it is robust to architecture variations and can be applied to various architectures with no modifications.

7.4.1 Connection Importance Metrics

A key part of double-shot pruning approach is defining a metric to measure the importance of each connection in the trained dense network based on weight values and decide to keep or prune the connection. We investigate five different importance metrics to figure out which of them are proper candidates to complement SNIP. Each metric is defined as a function of the weight values both at initialization ($\mathbf{w_i}$) and after training the dense network ($\mathbf{w_f}$). For example, one well-known importance metric is the final weight magnitude, which means the weights with large final magnitude will be kept and the remaining will be pruned. This metric could be formulated as $|\mathbf{w_f}|$.

Figure 7.6 shows our five different connection importance metrics along with their associated equations: magnitude, movement, magnitude_increase, weighted_movement and weighted_magnitude_increase. The first three metrics are introduced in [194], and the last two are designed by us based on some inspiration from [195]. While magnitude pruning retains the connections with the maximum dis-

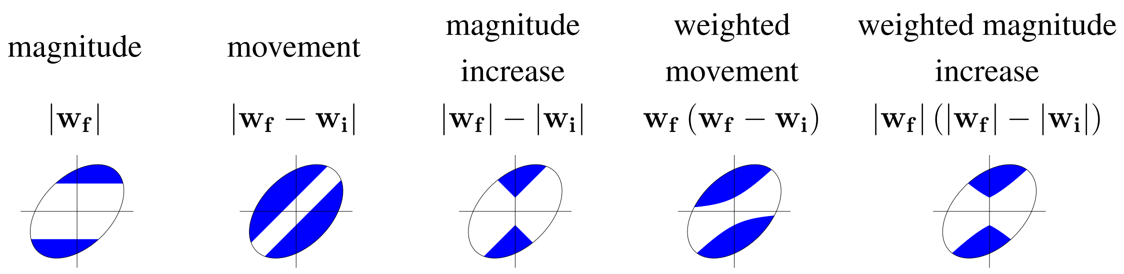

magnitude
$|\mathbf{w_f}|$
movement
$|\mathbf{w_f} - \mathbf{w_i}|$
magnitude increase
$|\mathbf{w_f}| - |\mathbf{w_i}|$
weighted movement
$\mathbf{w_f}\left(\mathbf{w_f} - \mathbf{w_i}\right)$
weighted magnitude increase
$|\mathbf{w_f}|\left(|\mathbf{w_f}| - |\mathbf{w_i}|\right)$

Figure 7.6: Name, formula and 2D visualization of connection importance metrics. The x axis represents $\mathbf{w_i}$ and the y axis is $\mathbf{w_f}$. Weights with the largest scores (colored regions) are kept, and weights with the smallest scores (white regions) are pruned.

Benchmark	Dataset	Dense Error (%)	SNIP Error (%)	Sparsity (%)
LeNet-5	MNIST	1.03	1.26	98.00
AlexNet	CIFAR-10	13.48	15.21	90.00
VGG	CIFAR-10	6.94	7.32	95.00

Table 7.1: Benchmark information.

tance from 0, movement and magnitude_increase select the connections with weights moving away from 0. The main difference between magnitude_increase and movement is that movement is more likely to keep weights which change sign than magnitude_increase. The last two metrics are designed as hybrid metrics to combine both distance from 0 and moving away from 0. For each metric, in addition to its equation, the visualization of the function as a set of decision boundaries in a 2D space is illustrated. Each function segments the 2D space into regions corresponding to mask values of one and zero. The ellipse represents the area occupied by the initial and final weights from a given layer. Connections with the largest importance scores (colored regions) are kept, and weights with the smallest scores (white regions) are pruned.

7.5 Evaluation

We evaluate our double-shot pruning technique with various importance metrics, which are summarized in Figure 7.6, using seven network architectures obtained from [185]. Table 7.1 demonstrates the detailed information of each benchmark. The first column shows the network architec-

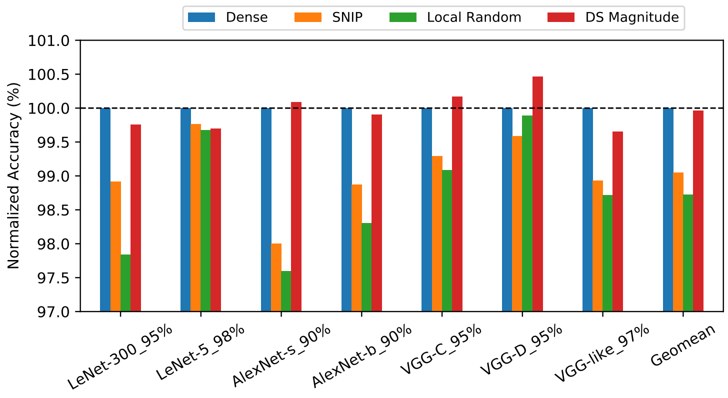


Figure 7.7: Comparison of dense, SNIP, local random and double-shot with importance metrics based on the distance from 0. Everything is normalized by the accuracy of the dense network.

ture. The second column is the dataset used for training and inference. The third column contains the absolute classification error values for trained dense networks. The fourth one is the absolute classification error values of the sparse networks after applying SNIP. And, the last column shows the global sparsity rate of each sparse network.

Figure 7.7 compares double-shot with importance metrics based on the distance from 0 (magnitude), single-shot (SNIP and local random) and dense network. Each group of bars represents the result of a single benchmark from 7.1. Each bar shows the accuracy of the classification task normalized by the accuracy of the dense network. As shown, local random pruning, which obtains the sparsity rates of layers from SNIP, provides comparable results to SNIP. On average it reduces the normalized accuracy by only 0.32% in comparison to SNIP. On the other hand, double-shot magnitude improves the accuracy of both SNIP and the dense network for AlexNet-s, VGG-C and VGG-D, but performs worse than SNIP for LeNet-5. On average it improves the normalized accuracy by 0.91% in comparison to SNIP.

Figure 7.8 compares double-shot with importance metrics based on moving away from 0 (movement and magnitude increase), single-shot (SNIP) and dense network in the same way as Figure 7.7. As demonstrated, double-shot movement improves the normalized accuracy of LeNet-300, LeNet-5, AlexNet-b and VGG-D in comparison to SNIP but performs worse than SNIP for AlexNet-s, VGG-C and VGG-like. On average, it improves the normalized accuracy of SNIP by

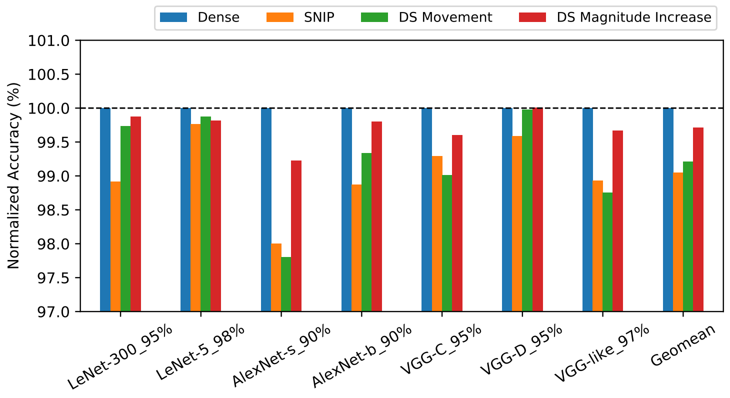


Figure 7.8: Comparison of dense, SNIP and double-shot with importance metrics based on moving away from 0. Everything is normalized by the accuracy of the dense network.

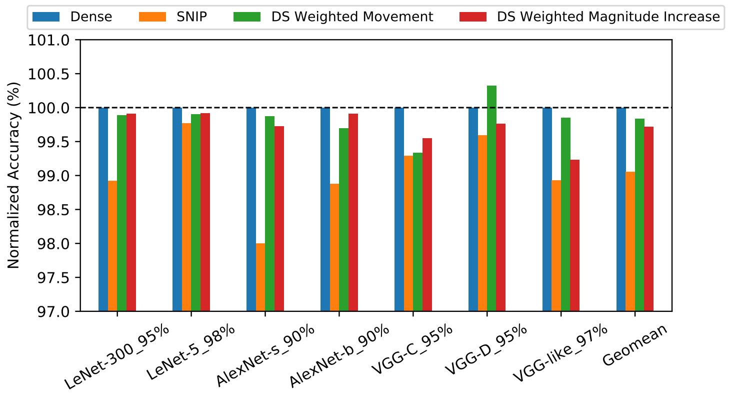


Figure 7.9: Comparison of dense, SNIP and double-shot with hybrid importance metrics. Everything is normalized by the accuracy of the dense network.

0.16%. On the other hand, double-shot magnitude increase results in better normalized accuracy than SNIP for all seven benchmarks and on average, improves SNIP result by 0.66%.

Figure 7.9 compares double-shot with hybrid importance metrics (weighted movement and weighted magnitude increase), which combine moving away from 0 and the distance from 0, single-shot (SNIP) and dense network in the same way as Figure 7.7. As shown, both hybrid double-shot approaches improve the normalized accuracy of all seven benchmarks in comparison to SNIP. On average, weighted movement and weighted magnitude increase improve the normalized accuracy of SNIP by 0.79% and 0.66%, respectively. An interesting observation is that double-shot weighted movement performs even better than the dense network for VGG-D and increases its normalized accuracy by 0.32%.

In conclusion, out of five introduced importance metrics, three of them (magnitude increase, weighted movement and weighted magnitude increase) result in double-shot approaches with higher accuracy than SNIP for all seven benchmarks. Among these three metrics, using weighted movement for double-shot pruning results in the highest average accuracy. Hence, we pick weighted movement as the importance metric of our double-shot pruning metric to ensure it supports different network architectures and various tasks.

7.6 Related Work

Neural network pruning [178, 49] rely on the fact that networks are usually overparameterized [196, 197, 198, 199, 50]. Traditional pruning approaches can be divided to two categories [200]: 1) augmenting the loss function with sparsity enforcing penalties; and 2) pruning based on some saliency criterion. Methods from the first category [180, 191, 192] augment the loss function with some sparsity enforcing penalty terms. Then, during training, back-propagation effectively penalizes the weight values. After that, weights below a certain threshold will be pruned. The second category of methods includes the sensitivity of the loss with respect to the activations [179] or the weights [201, 178, 186]. Traditional pruning approaches are heavily dependent on the weight values, which requires many iterations of pruning and fine-tuning and makes them prohibitively slow.

In recent years, extreme sparsity rates, without loss in accuracy, are achieved by using magnitude of the weights as the criterion [49, 190, 181, 202]. The magnitude criterion still requires machine learning expertise to heavily tune hyperparameters and are non-trivial to be extended to new architectures and different tasks. Other main drawbacks of magnitude based approaches are the reliance on pretraining and the expensive iterative prune – fine-tune cycles. [203] extends the soft weight sharing in [204] to obtain a sparse network. [205] uses learned dropout rates to prune the network.

Single-shot pruning [185, 206] approaches try to replace the traditional iterative and complex pruning approaches by faster and easier-to-apply methods for nonexpert users, which will improve the applicability and popularity of network pruning while having the potential for faster and more efficient training. SNIP [185] eliminates pretraining, the complex pruning schedule and iterative optimization procedure of the traditional pruning approaches by pruning the network once at initialization prior to training and using saliency criterion based on connection sensitivity. In addition, it does not require additional hyperparameters and the sparse network is trained in the standard way. The lottery ticket hypothesis tackles the fact that the sparse pruned architectures are difficult to train from scratch, which would improve training performance [183, 207, 208]. The lottery ticket hypothesis proposes that the dense networks contain sparse networks that (when trained in isolation) reach accuracy comparable to the original network. The main shortcoming of [183] is that it uses iterative pruning to find the winning ticket, which is counterintuitive to the purpose of efficient training. To alleviate this problem, drawing early-bird tickets [209] proposes a method to identify the winning tickets at very early training stages. However, it still requires modifications to the standard training procedure and is not trivial to apply for nonexpert users.

SNIP completely ignores the importance of weight values by forming the pruning algorithm based on randomly initialized weights and assumes that architectural features of the network are sufficient for identifying essential connections. In contrast, our double-shot noniterative pruning approach improves the result of SNIP by combining the insights from both architectural features and weight values, while keeping the network pruning fast, simple, easy-to-apply and not requiring additional hyperparameters.

7.7 Conclusion

In this chapter, we propose a double-shot pruning approach, which is simple, fast and easy-to-apply. It augments the architectural insights from the single-shot pruning method with information about weight values to prune irrelevant connections for a given task in a noniterative way. We evaluate our method using seven network architectures on two classification datasets (MNIST and CIFAR-10). After investigating the effect of five different connection importance metrics on the performance of double-shot pruning, we show that the most proper metric, on average, improves the accuracy of SNIP by 0.79% while maintaining the same global sparsity rate for all networks.

CHAPTER 8

Summary and Conclusion

Deep Learning algorithms are the main focus of modern machine learning systems. As data volumes keep growing, it has become customary to train complex neural networks with hundreds of millions of parameters to obtain state-of-the-art accuracy. To get around the costly computations associated with large models and data, the community is increasingly investing in two major categories of techniques: input-invariant and input-variant. Input-invariant methods, including network pruning, take advantage of hardware and software techniques to accelerate a single DNN for the entire dataset on various platforms. On the other hand, input-variant approaches rely on the fact that DNNs are overprovisioned, and the entire computational power of the model is not required for most of the inputs to produce accurate final outputs. Consequently, these techniques employ a combination of simple less accurate and complex more accurate models to dynamically adjust the complexity of the DNN based on input difficulty.

Although both categories contain numerous approaches to improve performance and efficiency of DNNs, most of them require in depth hardware, software or network architecture modifications, which are not trivial for nonexpert users to implement and deploy. To overcome these shortcomings, this thesis proposes techniques to improve performance and applicability of both input-invariant and input-variant methods.

In Chapter 3, we propose a system of *data specialized neural networks (dsNNs)* to increase energy efficiency of NNs. A multi-dsNN consists of multiple simple and specialized NNs, which are customized for different subsets of the input space. In contrast to conventional NN ensembles, multi-dsNN selects and activates a single dsNN per input instance dynamically. Compared to conventional monolithic NNs, a 2-way multi-dsNN reduces the energy consumption by an average of 61%, 53% and 55% and increases the performance by an average of $2.6\times$, $1.9\times$ and $1.8\times$ on microcontroller, mobile CPU and mobile GPU, respectively, with 100% accuracy of the baseline.

In Chapter 4, we introduce *SIEVE*, a novel *hybrid cloud-edge deep learning system*. It creates a heavily compressed CNN through aggressive precision reduction to enable speculative inferences on the device. A dynamic partitioning technique is employed to push most of the computation to this speculative CNN while the data transfer to original CNN is limited to recovery on low-confidence inferences. Compared to the cloud-only approach, SIEVE reduces the energy consumption by an average of 91%, 57% and 26% and increases the performance by an average of $12.3\times$, $2.8\times$ and $2.0\times$ for 3G, LTE and WiFi connection, respectively, with 100% accuracy of baseline.

In Chapter 5, we introduce *M&MNet*, another *hybrid cloud-edge deep learning system*. Same as SIEVE, it dynamically distributes the DNN computation between the server and mobile device. However, instead of precision reduction, it deploys a pair of lightweight and efficient CNNs on the device for speculation. Compared to the cloud-only approach, M&MNet reduces the energy consumption by an average of 53.5% and 39.5% and increases the performance by an average of $3.9\times$ and $1.9\times$ for LTE and WiFi connection, respectively, with 100% accuracy of baseline.

In Chapter 6, we introduce *Dynamic Duo (DD)*, a novel *input-driven synergistic deep learning system*. It pairs an efficient less accurate CNN (little) with a complex more accurate CNN (big) and offloads most of the computation on the little one. A dynamic confidence examination technique is employed to probe the outputs of the little CNN and invoke the big one only for recovering the low-confidence inferences. Compared to the traditional approach, DD improves the datacenter throughput by $2.1\times$ and reduces the average inference latency by 38% on an NVIDIA TITAN X GPU with 100% accuracy of the baseline.

Finally, in Chapter 7, we propose a double-shot pruning approach, which is simple, fast and easy-to-apply. It augments the architectural insights from the single-shot pruning method with information about weight values to prune irrelevant connections for a given task in a noniterative way. We evaluate our method using seven network architectures on two classification datasets (MNIST and CIFAR-10). After investigating the effect of five different connection importance metrics on the performance of double-shot pruning, we show that the most proper metric, on average, improves the accuracy of SNIP by 0.91% while maintaining the same global sparsity rate for all networks.

BIBLIOGRAPHY

[1] X. Huang, J. Baker, and R. Reddy, "A historical perspective of speech recognition," *Communications of the ACM*, vol. 57, no. 1, pp. 94–103, 2014.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Computer Vision and Pattern Recognition*, 2014.

[3] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1440–1448, 2015.

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

[5] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the devil in the details: Delving deep into convolutional nets," *arXiv preprint arXiv:1405.3531*, 2014.

[6] T. Warren, "The story of cortana, microsoft's siri killer," tech. rep., Retrieved 5/20/2014, from http://www. theverge. com/2014/4/2/5570866/cortana-windows-phone-8-1-digital-assistant, 2014.

[7] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, pp. 160–167, ACM, 2008.

[8] Z.-Q. Liu, J.-H. Cai, and R. Buse, *Handwriting recognition: soft computing and probabilistic approaches*, vol. 133. Springer, 2012.

[9] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[10] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8595–8598, IEEE, 2013.

[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[12] "Apple's massive new data center set to host nuance tech."

[13] "Google supercharges machine learning tasks with tpu custom chip."

[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[15] G. Huang, D. Chen, T. Li, F. Wu, L. Van Der Maaten, and K. Q. Weinberger, "Multi-scale dense convolutional networks for efficient prediction," *arXiv preprint arXiv:1703.09844*, vol. 2, 2017.

[16] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, IEEE, 2016.

[17] N. J. Harvey, J. Dunagan, M. Jones, S. Saroiu, M. Theimer, and A. Wolman, "Skipnet: A scalable overlay network with practical locality properties," 2002.

[18] J. A. Anderson, *An introduction to neural networks*. MIT press, 1995.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[20] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[21] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, IEEE Press, 2016.

[22] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 262–263, IEEE, 2016.

[23] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[24] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[25] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 380–392, IEEE, 2016.

[26] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[27] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[28] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "An ultra low-power hardware accelerator for automatic speech recognition,"

[29] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *Acoustics, speech and signal processing (icassp), 2014 ieee international conference on*, pp. 4087–4091, IEEE, 2014.

[30] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE Computer Society, 2012.

[31] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 356–367, IEEE Computer Society, 2012.

[32] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pp. 1–4, IEEE, 2011.

[33] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[34] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas,"

[35] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 535–547, ACM, 2017.

[36] X. Lei, A. W. Senior, A. Gruenstein, and J. Sorensen, "Accurate and compact large vocabulary speech recognition on mobile devices.," Citeseer, 2013.

[37] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, ACM, 2017.

[38] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 1–13, IEEE, 2016.

[39] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622, IEEE, 2014.

[40] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 13–26, ACM, 2017.

[41] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[42] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 27–40, ACM, 2015.

[43] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 255–266, IEEE Press, 2016.

[44] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[45] P. Chi, S. Li, Z. Qi, P. Gu, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of ISCA*, vol. 43, 2016.

[46] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 393–405, IEEE Press, 2016.

[47] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 27–40, ACM, 2017.

[48] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 786–799, ACM, 2017.

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

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

[51] P. Nakkiran, R. Alvarez, R. Prabhavalkar, and C. Parada, "Compressing deep neural networks using a rank-constrained topology," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[52] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, ACM, 2017.

[53] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, *et al.*, "C ir cnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 395–408, ACM, 2017.

[54] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," Citeseer.

[55] V. Vanhoucke, M. Devin, and G. Heigold, "Multiframe deep neural networks for acoustic modeling," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 7582–7585, IEEE, 2013.

[56] M. Courbariaux, I. Hubara, C. D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training neural networks with weights and activations constrained to+ 1 or-,"

[57] A. Lavin, "Fast algorithms for convolutional neural networks," *arXiv preprint arXiv:1509.09308*, 2015.

[58] M. F. Stollenga, J. Masci, F. Gomez, and J. Schmidhuber, "Deep networks with internal selective attention through feedback connections," in *Advances in Neural Information Processing Systems*, pp. 3545–3553, 2014.

[59] A. Almahairi, N. Ballas, T. Cooijmans, Y. Zheng, H. Larochelle, and A. Courville, "Dynamic capacity networks," *arXiv preprint arXiv:1511.07838*, 2015.

[60] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in Neural Information Processing Systems*, pp. 1269–1277, 2014.

[61] C. Feichtenhofer, H. Fan, J. Malik, and K. He, "Slowfast networks for video recognition," *arXiv preprint arXiv:1812.03982*, 2018.

[62] D. Stamoulis, T.-W. R. Chin, A. K. Prakash, H. Fang, S. Sajja, M. Bognar, and D. Marculescu, "Designing adaptive neural networks for energy-constrained image classification," in *Proceedings of the International Conference on Computer-Aided Design*, p. 23, ACM, 2018.

[63] N. K. Jayakodi, A. Chatterjee, W. Choi, J. R. Doppa, and P. P. Pande, "Trading-off accuracy and energy of deep inference on embedded systems: A co-design approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2881–2893, 2018.

[64] E. Park, D. Kim, S. Kim, Y.-D. Kim, G. Kim, S. Yoon, and S. Yoo, "Big/little deep neural network for ultra low power inference," in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pp. 124–132, IEEE Press, 2015.

[65] X. Wang, Y. Luo, D. Crankshaw, A. Tumanov, F. Yu, and J. E. Gonzalez, "Idk cascades: Fast deep learning by learning not to overthink," *arXiv preprint arXiv:1706.00885*, 2017.

[66] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, pp. I–511, IEEE, 2001.

[67] A. Angelova, A. Krizhevsky, V. Vanhoucke, A. Ogale, and D. Ferguson, "Real-time pedestrian detection with deep network cascades," 2015.

[68] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: optimizing neural network queries over video at scale," *arXiv preprint arXiv:1703.02529*, 2017.

[69] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor, "Scaling video analytics on constrained edge nodes," *arXiv preprint arXiv:1905.13536*, 2019.

[70] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[71] J. Gama and P. Brazdil, "Cascade generalization," *Machine Learning*, vol. 41, no. 3, pp. 315–343, 2000.

[72] A. C. Lorena, A. C. De Carvalho, and J. M. Gama, "A review on the combination of binary classifiers in multiclass problems," *Artificial Intelligence Review*, vol. 30, no. 1, pp. 19–37, 2008.

[73] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, "Machine learning: a review of classification and combining techniques," *Artificial Intelligence Review*, vol. 26, no. 3, pp. 159–190, 2006.

[74] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1, pp. 1–39, 2010.

[75] L. K. Hansen and P. Salamon, "Neural network ensembles," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 10, pp. 993–1001, 1990.

[76] L. Breiman, “Bagging predictors,” *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[77] R. E. Schapire, “The strength of weak learnability,” *Machine learning*, vol. 5, no. 2, pp. 197–227, 1990.

[78] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, “Adaptive mixtures of local experts,” *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[79] S. Masoudnia and R. Ebrahimpour, “Mixture of experts: a literature survey,” *Artificial Intelligence Review*, pp. 1–19, 2014.

[80] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer,” *arXiv preprint arXiv:1701.06538*, 2017.

[81] Y. Bengio, “Deep learning of representations: Looking forward,” in *International Conference on Statistical Language and Speech Processing*, pp. 1–37, Springer, 2013.

[82] Y. LeCun, C. Cortes, and C. J. Burges, “The mnist database of handwritten digits,” 1998.

[83] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, “Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android,” in *Proceedings of the 2016 ACM on Multimedia Conference*, pp. 1201–1205, ACM, 2016.

[84] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, IEEE, 2009.

[85] A. Krizhevsky, V. Nair, and G. Hinton, “The cifar-10 dataset,” 2014.

[86] T. Joachims, “A probabilistic analysis of the rocchio algorithm with tfidf for text categorization.,” tech. rep., DTIC Document, 1996.

[87] J. A. Blackard, *Comparison of neural networks and discriminant analysis in predicting forest cover types*. Colorado State University, 1998.

[88] D. Lewis, “Reuters-21578 text categorization collection data set,” 1998.

[89] A. M. d. J. C. Cachopo, *Improving methods for single-label text categorization*. PhD thesis, Universidade Técnica de Lisboa, 2007.

[90] M. Craven, A. McCallum, D. PiPasquo, T. Mitchell, and D. Freitag, “Learning to extract symbolic knowledge from the world wide web,” tech. rep., DTIC Document, 1998.

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

[92] A. Krizhevsky, “cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks,” 2012.

[93] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[94] Z. Lin, R. Memisevic, and K. Konda, "How far can we go without convolution: Improving fully-connected networks," *arXiv preprint arXiv:1511.02580*, 2015.

[95] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[96] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[97] J. Zhu, H. Zou, S. Rosset, and T. Hastie, "Multi-class adaboost," *Statistics and its Interface*, vol. 2, no. 3, pp. 349–360, 2009.

[98] Y. Freund and R. Schapire, "A short introduction to boosting," *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.

[99] Z.-H. Zhou, J. Wu, and W. Tang, "Ensembling neural networks: many could be better than all," *Artificial intelligence*, vol. 137, no. 1, pp. 239–263, 2002.

[100] H. Schwenk and Y. Bengio, "Boosting neural networks," *Neural Computation*, vol. 12, no. 8, pp. 1869–1887, 2000.

[101] H. R. Lourenço, O. C. Martin, and T. Stutzle, "Iterated local search," *International series in operations research and management science*, pp. 321–354, 2003.

[102] P. Brazdil, C. G. Carrier, C. Soares, and R. Vilalta, *Metalearning: Applications to data mining*. Springer Science & Business Media, 2008.

[103] M. M. Islam, X. Yao, S. S. Nirjon, M. A. Islam, and K. Murase, "Bagging and boosting negatively correlated neural networks," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 38, no. 3, pp. 771–784, 2008.

[104] A. J. Sharkey, *Combining artificial neural nets: ensemble and modular multi-net systems*. Springer Science & Business Media, 2012.

[105] Z. Wang, R. Schapire, and N. Verma, "Error-adaptive classifier boosting (eacb): Exploiting data-driven training for highly fault-tolerant hardware," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 3884–3888, IEEE, 2014.

[106] E. Alpaydin and C. Kaynak, "Cascading classifiers," *Kybernetika*, vol. 34, no. 4, pp. 369–374, 1998.

[107] C. Kaynak and E. Alpaydin, "Multistage cascading of multiple classifiers: One man's noise is another man's data," in *ICML*, pp. 455–462, 2000.

[108] L. Bourdev and J. Brandt, “Robust object detection via soft cascade,” in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 2, pp. 236–243, IEEE, 2005.

[109] H. Schneiderman, “Feature-centric evaluation for efficient cascaded object detection,” in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 2, pp. II–29–II–36 Vol.2, June 2004.

[110] C. Ferri, P. Flach, and J. Hernández-Orallo, “Delegating classifiers,” in *Proceedings of the twenty-first international conference on Machine learning*, p. 37, ACM, 2004.

[111] J. Ortega, M. Koppel, and S. Argamon, “Arbitrating among competing classifiers using learned referees,” *Knowledge and Information Systems*, vol. 3, no. 4, pp. 470–490, 2001.

[112] L. Todorovski and S. Džeroski, “Combining multiple models with meta decision trees,” *Principles of Data Mining and Knowledge Discovery*, pp. 69–84, 2000.

[113] Y. Bengio, “Deep learning of representations: Looking forward,” in *Proceedings of the First International Conference on Statistical Language and Speech Processing*, SLSP’13, (Berlin, Heidelberg), pp. 1–37, Springer-Verlag, 2013.

[114] D. Eigen, M. Ranzato, and I. Sutskever, “Learning factored representations in a deep mixture of experts,” *arXiv preprint arXiv:1312.4314*, 2013.

[115] M. I. Jordan and R. A. Jacobs, “Hierarchical mixtures of experts and the em algorithm,” *Neural computation*, vol. 6, no. 2, pp. 181–214, 1994.

[116] E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup, “Conditional computation in neural networks for faster models,” *arXiv preprint arXiv:1511.06297*, 2015.

[117] K. Cho and Y. Bengio, “Exponentially increasing the capacity-to-computation ratio for conditional computation in deep learning,” *arXiv preprint arXiv:1406.7362*, 2014.

[118] L. Denoyer and P. Gallinari, “Deep sequential neural network,” *arXiv preprint arXiv:1410.0510*, 2014.

[119] M. Halpern, Y. Zhu, and V. J. Reddi, “Mobile cpu’s rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction,” in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 64–76, IEEE, 2016.

[120] “Artificial intelligence tech in snapdragon 835: personalized experience created by machine learning.”

[121] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, ACM, 2010.

[122] “Apple machine learning journal.”

[123] "Natural language understanding team."

[124] "Alexa machine learning."

[125] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[126] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 123–136, ACM, 2016.

[127] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 615–629, ACM, 2017.

[128] Z. Wu, T. Nagarajan, A. Kumar, S. Rennie, L. S. Davis, K. Grauman, and R. Feris, "Blockdrop: Dynamic inference paths in residual networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8817–8826, 2018.

[129] "Nvidia jetson tk1 development kit: Bringing gpuaccelerated computing to embedded systems."

[130] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pp. 328–339, IEEE, 2017.

[131] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An in-depth study of lte: effect of network protocol and application behavior on performance," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 363–374, 2013.

[132] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g lte networks," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 225–238, ACM, 2012.

[133] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, p. 23, ACM, 2016.

[134] A. Delmas, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, and A. Moshovos, "Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how," *arXiv preprint arXiv:1803.03688*, 2018.

[135] T. NVIDIA, "K1: A new era in mobile computing," *Nvidia, Corp., White Paper*, 2014.

[136] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[137] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[138] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[139] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[140] "Caffe model zoo."

[141] "Nvidia titan x."

[142] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[143] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future generation computer systems*, vol. 29, no. 1, pp. 84–106, 2013.

[144] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *International Conference on Mobile Computing, Applications, and Services*, pp. 59–79, Springer, 2010.

[145] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently.," in *OSDI*, vol. 12, pp. 93–106, 2012.

[146] H. Qian and D. Andresen, "Reducing mobile device energy consumption with computation offloading," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pp. 1–8, IEEE, 2015.

[147] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 43–56, ACM, 2011.

[148] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, pp. 301–314, ACM, 2011.

[149] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[150] F. Nan and V. Saligrama, "Adaptive classification for prediction under a budget," in *Advances in Neural Information Processing Systems*, pp. 4727–4737, 2017.

[151] X. Ran, H. Chen, Z. Liu, and J. Chen, “Delivering deep learning to mobile devices via offloading,” in *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pp. 42–47, ACM, 2017.

[152] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, “A hybrid approach to offloading mobile image classification,” in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 8375–8379, IEEE, 2014.

[153] B. Qi, M. Wu, and L. Zhang, “A dnn-based object detection system on mobile cloud computing,” in *Communications and Information Technologies (ISCIT), 2017 17th International Symposium on*, pp. 1–6, IEEE, 2017.

[154] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.

[155] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, “Mnasnet: Platform-aware neural architecture search for mobile,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

[156] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[157] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[158] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[159] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *European conference on computer vision*, pp. 630–645, Springer, 2016.

[160] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, *et al.*, “Chamnet: Towards efficient network design through platform-aware model adaptation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11398–11407, 2019.

[161] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

[162] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018.

[163] R. MCMILLAN, “Inside the artificial brain that's remaking the google empire,” 2014.

[164] A. Viebke and S. Pllana, "The potential of the intel (r) xeon phi for supervised deep learning," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pp. 758–765, IEEE, 2015.

[165] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[166] H. Kellerer, U. Pferschy, and D. Pisinger, "Multidimensional knapsack problems," in *Knapsack problems*, pp. 235–283, Springer, 2004.

[167] "Image classification on imagenet."

[168] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[169] "Pytorch torchvision package."

[170] "Pytorch."

[171] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 35–45, IEEE, 2012.

[172] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 319–330, ACM, 2011.

[173] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proceedings of the 47th International Conference on Parallel Processing*, pp. 1–10, 2018.

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

[175] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, "Predicting parameters in deep learning," in *Advances in neural information processing systems*, pp. 2148–2156, 2013.

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

[177] C. Li, H. Farkhoor, R. Liu, and J. Yosinski, "Measuring the intrinsic dimension of objective landscapes," *arXiv preprint arXiv:1804.08838*, 2018.

[178] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, pp. 598–605, 1990.

[179] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," in *Advances in neural information processing systems*, pp. 107–115, 1989.

[180] Y. Chauvin, "A back-propagation algorithm with optimal use of hidden units," in *Advances in neural information processing systems*, pp. 519–526, 1989.

[181] M. A. Carreira-Perpinán and Y. Idelbayev, ""learning-compression" algorithms for neural net pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8532–8541, 2018.

[182] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[183] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[184] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.

[185] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," *arXiv preprint arXiv:1810.02340*, 2018.

[186] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in neural information processing systems*, pp. 164–171, 1993.

[187] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

[188] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[189] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

[190] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances in neural information processing systems*, pp. 1379–1387, 2016.

[191] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Generalization by weight-elimination with application to forecasting," in *Advances in neural information processing systems*, pp. 875–882, 1991.

[192] M. Ishikawa, "Structural learning with forgetting," *Neural networks*, vol. 9, no. 3, pp. 509–521, 1996.

[193] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[194] H. Zhou, J. Lan, R. Liu, and J. Yosinski, "Deconstructing lottery tickets: Zeros, signs, and the supermask," in *Advances in Neural Information Processing Systems*, pp. 3597–3607, 2019.

[195] V. Sanh, T. Wolf, and A. M. Rush, "Movement pruning: Adaptive sparsity by fine-tuning," *arXiv preprint arXiv:2005.07683*, 2020.

[196] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *arXiv preprint arXiv:1611.03530*, 2016.

[197] B. Neyshabur, R. Tomioka, and N. Srebro, "In search of the real inductive bias: On the role of implicit regularization in deep learning," *arXiv preprint arXiv:1412.6614*, 2014.

[198] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, *et al.*, "A closer look at memorization in deep networks," *arXiv preprint arXiv:1706.05394*, 2017.

[199] Y. Bengio, N. L. Roux, P. Vincent, O. Delalleau, and P. Marcotte, "Convex neural networks," in *Advances in neural information processing systems*, pp. 123–130, 2006.

[200] R. Reed, "Pruning algorithms-a survey," *IEEE transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.

[201] E. D. Karnin, "A simple procedure for pruning back-propagation trained neural networks," *IEEE transactions on neural networks*, vol. 1, no. 2, pp. 239–242, 1990.

[202] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," *arXiv preprint arXiv:1704.05119*, 2017.

[203] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," *arXiv preprint arXiv:1702.04008*, 2017.

[204] S. J. Nowlan and G. E. Hinton, "Simplifying neural networks by soft weight-sharing," *Neural computation*, vol. 4, no. 4, pp. 473–493, 1992.

[205] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational dropout sparsifies deep neural networks," *arXiv preprint arXiv:1701.05369*, 2017.

[206] M. S. Zhang and B. Stadie, "One-shot pruning of recurrent neural networks by jacobian spectrum evaluation," *arXiv preprint arXiv:1912.00120*, 2019.

[207] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," *arXiv preprint arXiv:1911.11134*, 2019.

[208] A. Morcos, H. Yu, M. Paganini, and Y. Tian, "One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers," in *Advances in Neural Information Processing Systems*, pp. 4932–4942, 2019.

[209] H. You, C. Li, P. Xu, Y. Fu, Y. Wang, X. Chen, Y. Lin, Z. Wang, and R. G. Baraniuk, "Drawing early-bird tickets: Towards more efficient training of deep networks," *arXiv preprint arXiv:1909.11957*, 2019.