EFFICIENT METHODS AND HARDWARE FOR DEEP LEARNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Song Han

September 2017

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Bill Dally, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Fei-Fei Li

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The future will be populated with intelligent devices that require inexpensive, low-power hardware platforms. Deep neural networks have evolved to be the state-of-the-art technique for machine learning tasks. However, these algorithms are computationally intensive, which makes it difficult to deploy on embedded devices with limited hardware resources and a tight power budget. Since Moore's law and technology scaling are slowing down, technology alone will not address this issue. To solve this problem, we focus on efficient algorithms and domain-specific architectures specially designed for the algorithm. By performing optimizations across the full stack from application through hardware, we improved the efficiency of deep learning through smaller model size, higher prediction accuracy, faster prediction speed, and lower power consumption.

Our approach starts by changing the algorithm, using "Deep Compression" that significantly reduces the number of parameters and computation requirements of deep learning models by pruning, trained quantization, and variable length coding. "Deep Compression" can reduce the model size by 18× to 49× without hurting the prediction accuracy. We also discovered that pruning and the sparsity constraint not only applies to model compression but also applies to regularization, and we proposed dense-sparse-dense training (DSD), which can improve the prediction accuracy for a wide range of deep learning models. To efficiently implement "Deep Compression" in hardware, we developed EIE, the "Efficient Inference Engine", a domain-specific hardware accelerator that performs inference directly on the compressed model which significantly saves memory bandwidth. Taking advantage of the compressed model, and being able to deal with the irregular computation pattern efficiently, EIE improves the speed by 13× and energy efficiency by 3,400× over GPU.

Acknowledgments

First and foremost, I would like to thank my Ph.D. advisor, Professor Bill Dally. Bill has been an exceptional advisor and I have been very fortunate to receive his guidance for the five years of Ph.D. journey. In retrospect, I learned from Bill how to define a problem in year one and two, solve this problem in year three and four, and spread the discovery in year five. In each step, Bill gave me extremely visionary advice, most generous support, and most sincere and constructive feedback. Bill's industrial experience made his advice insightful beyond academic research contexts. Bill's enthusiastic of impactful research greatly motivated me. Bill's research foresight, technical depth, and commitment to the students is a valuable treasure for me.

I would also thank my co-advisor, Professor Mark Horowitz. I met Mark in my junior year and I was encouraged by him to pursue a Ph.D. After coming to Stanford, I had the unique privilege to have access to Mark's professional expertise and brilliant thinking. Mark offered me invaluable advice and diligently guided me through challenging problems. He taught me to perceive the philosophy. I feel so fortunate to have Mark be my co-advisor.

I gave my sincere thanks to Professor Fei-Fei Li. She is my first mentor in computer vision and deep learning. Her ambition and foresight ignited my passion for bridging the research in deep learning and hardware. Sitting on the same floor with Fei-Fei and her students spawned many research spark. I sincerely thank Fei-Fei's students Andrej Karpathy, Yuke Zhu, Justin Johnson, Serena Yeung and Olga Russakovsky for the insightful discussions that helped my interdisciplinary research between deep learning and hardware.

I also thank Professor Christos Kozyrakis, Professor Kunle Olukotun, Professor Subhasish Mitra and Dr. Ofer Shacham for the fatalistic course offerings that nurtured me in the field of computer architecture and VLSI systems. I would thank my friends and lab mates in the CVA group: Milad Mohammadi, Subhasis Das, Nic McDonald, Albert Ng, Yatish Turakhia, Xingyu Liu, Huizi Mao, and also the CVA interns Chenzhuo Zhu, Kaidi Cao, Yujun Liu. It was a pleasure working together with you all.

It has been an honor to work with many great collaborators outside Stanford. I would like to thank Professor Kurt Keutzer, Forrest Iandola, Bichen Wu and Matthew Moskewicz for teaming up on the SqueezeNet project. I would like to thank Jensen Huang for the ambitious encouragements and

the generous GPU support. I really enjoyed the collaborations with Jeff Pool, John Tran, Peter Vajda, Manohar Paluri, Sharan Narang and Greg Diamos. Thank you all and looking forward to future collaborations. Also many thanks to Steve Keckler, Jan Kautz, Andrew Ng, Rob Fergus, Yangqing Jia, Liang Peng, Yu Wang, Song Yao and Yi Shan for many insightful and valuable discussions.

And I give sincere thanks to my family, Mom and Dad. I can't forget the encouragements I get from you when I came to US thousands of miles away from home, and I can't accomplish what I did without your love. Thank you for nurturing me and set me a great role model. And to my grandparents, thank you for the influence you gave me when I was young.

Finally, I would like to thank the funding support from Stanford Vice Provost for Graduate Education and Rambus Inc. through the Stanford Graduate Fellowship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural networks (DNNs) have shown significant improvements in many AI applications, including computer vision [5], natural language processing [10], speech recognition [9], and machine translation [11]. The performance of DNN is improving rapidly: the winner of ImageNet challenge has increased the classification accuracy from 84.7% in 2012 (AlexNet [2]) to 96.5% in 2015 (ResNet-152 [5]). Such exceptional performance enables DNNs to bring artificial intelligence to far-reaching applications, such as in smart phones [12], drones [13], and self-driving cars [14].

However, this accuracy improvement comes at the cost of high computational complexity. For example, AlexNet takes 1.4GOPS to process a single 224×224 image, while ResNet-152 takes 22.6GOPS, more than an order of magnitude more computation. Running ResNet-152 in a self-driving car with 8 cameras at 1080p 30 frames/sec requires the hardware to deliver $22.6GOPS \times 30fps \times 8 \times 1920 \times 1280/(224 \times 224) = 265$ Teraop/sec computational throughput; using multiple neural networks on each camera will make the computation even larger. For embedded mobile devices that have limited computational resources, such high demands for computational resource become prohibitive.

Another key challenge is energy consumption: because mobile devices are battery-constrained, heavy computations will quickly drain the battery. The energy cost per 32b operation in a 45nm technology ranges from $3pJ$ for multiplication to $640pJ$ for off-chip memory access [15]. Running a larger model needs more memory references, and each memory reference requires two orders of magnitude more energy than an arithmetic operation. Large DNN models do not fit in on-chip storage and hence require costlier DRAM accesses. To a first-order approximation, running a 1-billion connection neural network, for example, at 30Hz would require $30Hz \times 1G \times 640pJ = 19.2W$ just for DRAM accesses, which is well beyond the power envelope of a typical mobile device.

Despite the challenges and constraints, we have witnessed rapid progress in the area of efficient deep learning hardware. Designers have designed custom hardware accelerators specialized for neural networks [16–23]. Thanks to specialization, these accelerators tailored the hardware architecture given the computation pattern of deep learning and achieved higher efficiency compared with CPUs and GPUs. The first wave of accelerators efficiently implemented the computational primitives for neural networks [16, 18, 24]. Researchers then realized that memory access is more expensive and critically needs optimization, so the second wave of accelerators efficiently optimized memory transfer and data movement [19–23]. These two generations of accelerators have made promising progress in improving the speed and energy efficiency of running DNNs.

However, both generations of deep learning accelerators treated the algorithm as a black box and focused on only optimizing the hardware architecture. In fact, there is plenty of room at the top by optimizing the algorithm. We found that DNN models can be significantly compressed and simplified before touching the hardware; if we treat these DNN models merely as a black box and hand them directly to hardware, there is massive redundancy in the workload. However, existing hardware accelerators are optimized for uncompressed DNN models, resulting in huge wastes of computation cycles and memory bandwidth compared with running on compressed DNN models. We therefore need to co-design the algorithm and the hardware.

In this dissertation, we co-designed the algorithm and hardware for deep learning to make it run faster and more energy-efficiently. We developed techniques to make the deep learning workload more efficient and compact to begin with and then designed the hardware architecture specialized for the optimized DNN workload. Figure 1.1 illustrates the design methodology of this thesis. Breaking the boundary between the algorithm and the hardware stack creates a much larger design space with many degrees of freedom that researchers have not explored before, enabling better optimization of deep learning.

On the algorithm side, we investigated how to simplify and compress DNN models to make them less computation and memory intensive. We aggressively compressed the DNNs by up to 49× without losing prediction accuracy on ImageNet [25, 26]. We also found that the model compression algorithm removes the redundancy, prevents overfitting, and serve as a suitable regularization method [27].

From the hardware perspective, a compressed model has great potential to improve speed and energy efficiency because it requires less computation and memory. However, the model compression algorithm makes the computation pattern irregular and hard to parallelize. Thus we designed customized hardware for the compressed model, tailoring the data layout and control flow to model compression. This hardware accelerator achieved 3,400 × better energy efficiency than GPU and an order of magnitude better than previous accelerators [28]. The architecture has been prototyped on FPGA and applied to accelerate speech recognition systems [29].

1.1 Motivation

"Less is more"

— Robert Browning, 1855

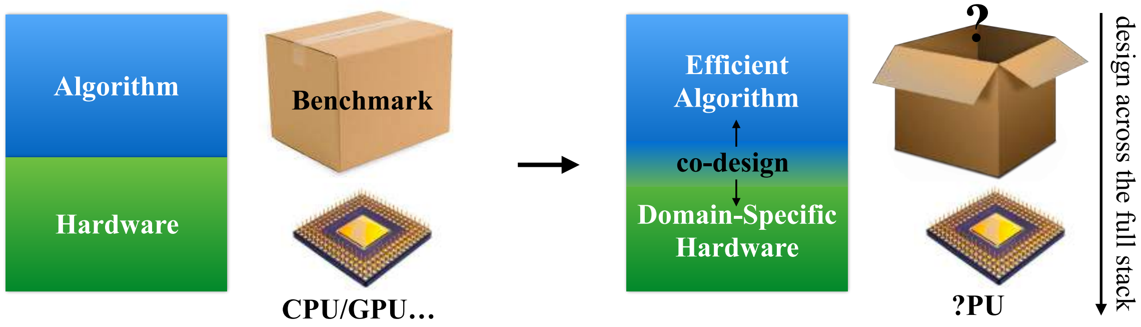


Figure 1.1: This thesis focused on algorithm and hardware co-design for deep learning. This thesis answers the two questions: what **methods** can make deep learning algorithm more efficient, and what is the best **hardware** architecture for such algorithm.

The philosophy of this thesis is to make neural network inference **less** complicated and make it **more** efficient through algorithm and hardware co-design.

Motivation for Model Compression: First, a smaller model means less overhead when exporting models to clients. Take autonomous driving for example; Tesla periodically copies new models from their servers to customers' cars. Smaller models require less communication in such over-the-air (OTA) updates, making frequent updates more feasible. Another example is the Apple Store: mobile applications above 100 MB will not download until a user connects to Wi-Fi. As a result, a new feature that increases the binary size by 100MB will receive much more scrutiny than one that increases it by 10MB. Thus, putting a large DNN model in a mobile application is infeasible.

The second reason is inference speed. Many mobile scenarios require low-latency, real-time inference, including self-driving cars and AR glasses, where latency is critical to guarantee safety or user experience. A smaller model helps improve the inference speed on such devices: from the computational perspective, smaller DNN models require fewer arithmetic operations and computation cycles; from the memory perspective, smaller DNN models take less memory reference cycles. If the model is small enough it can fit in the on-chip SRAM, which is faster to access than off-chip DRAM memory.

The third reason is energy consumption. Running large neural networks requires significant memory bandwidth to fetch the weights — this consumes considerable energy and is problematic for battery-constrained mobile devices. As a result, iOS 10 requires iPhones to be plugged into chargers while performing photo analysis. Memory access dominates energy consumption. Smaller neural networks require less memory access to fetch the model, saving energy and extending battery life.

The fourth reason is cost. When deploying DNNs on Application-Specific Integrated Circuits (ASICs), a sufficiently small model can be stored on-chip directly. As smaller models require less on-chip SRAM, this permits a smaller ASIC die thus making the chip less expensive.

Smaller deep learning models are also appealing when deployed in large-scale data centers as cloud AI. Future data center workloads would be populated with AI applications, such as Google Cloud Machine Learning and Amazon Rekognition. The cost of maintaining such large-scale data centers is tremendous. Smaller DNN models reduce the computation of the workload and take less energy to run. This helps to reduce the electricity bill and the total cost of ownership (TCO) of running a data center with deep learning workloads.

A byproduct of model compression is that it can remove the redundancy during training and prevents overfitting. The compression algorithm automatically selects the optimal set of parameters as well as their precision. It additionally regularizes the network by avoiding capturing the noise in the training data.

Motivation for Specialized Hardware: Though model compression reduces the total number operations deep learning algorithms require, the irregular pattern caused by compression hinders the efficient acceleration on general-purpose processors. The irregularity limited the benefits of model compression, and we achieved only 3× energy efficiency improvement on these machines. The potential saving is much larger: $1 - 2$ orders of magnitude comes from model compression, another two orders of magnitude come from DRAM $\Rightarrow$ SRAM. The compressed model is small enough to fit in about 10MB of SRAM (verified with AlexNet, VGG-16, Inception-V3, ResNet-50, as discussed in Chapter 4) rather than having to be stored in a larger capacity DRAM.

Why is there such a big gap between the theoretical and the actual efficiency improvement? The first reason is the inefficient data path. First, running on compressed models requires traversing a sparse tensor, which has poor locality on general-purpose processors. Secondly, model compression incurs a level of indirection for the weights, which requires dedicated buffers for fast access. Lastly, the bit width of an aggressively compressed model is not byte aligned, which results in serialization and de-serialization overhead on general-purpose processors.

The second reason for the gap is inefficient control flow. Out-of-order CPU processors have complicated front ends attempting to speculate the parallelism in the workload; this has a costly consequence (flushing the pipeline) if any speculation is wrong. However, once narrowed down to deep learning workloads, the computation pattern is known to the processor ahead of time. Neither branch prediction nor caching are needed, and the execution is deterministic, not speculative. Therefore, such speculative units are wasteful in out-of-order processors.

There are alternatives, but they are not perfect. SIMD units can amortize the instruction overhead among multiple pieces of data. SIMT units can also hide the memory latency by having a pool of threads. These architectures prefer the workload to be executed lockstep and in a parallel manner. However, model compression leads to irregular computation patterns and makes it hard to parallelize, causing divergence problem on these architectures.

While previously proposed DNN accelerators [19–21] can efficiently handle the dense, uncompressed DNN model; they are unable to handle the aggressively compressed DNN model due to different computation patterns. There is an enormous waste of computation and memory bandwidth for

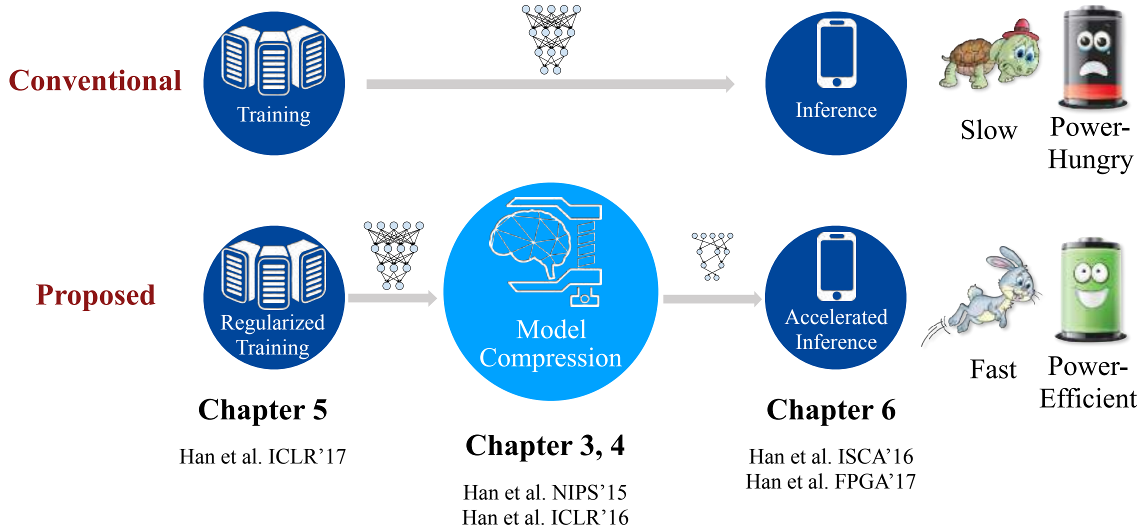


Figure 1.2: Thesis contributions: regularized training, model compression, and accelerated inference.

previous accelerators running the uncompressed model. Previously proposed sparse linear algebra accelerators [30–32] do not address weight sharing, extremely narrow bits, or the activation sparsity, the other benefits of model compression. These factors motivate us to build a specialized hardware accelerator that can operate efficiently on a deeply compressed neural network.

1.2 Contribution and Thesis Outline

We optimize the efficiency of deep learning with a top-down approach from algorithm to hardware. This thesis proposes the techniques for regularized training ⇒ model compression ⇒ accelerated inference, illustrated in Figure 1.2. The contributions of this thesis are:

- A model compression technique, called Deep Compression, that consists of pruning, trained quantization and variable length coding, which can compress DNN models by $18-49\times$ while fully preserving the prediction accuracy.
- A regularization technique, called Dense-Sparse-Dense (DSD) Training, that can regularize neural network training and prevent overfitting to improve the accuracy for a wide range of CNNs, RNNs, and LSTMs. The DSD model zoo is available *online*.
- An efficient hardware architecture, called "Efficient Inference Engine" (EIE), that can perform inference on the sparse, compressed DNNs and save a significant amount of memory bandwidth. EIE achieved $13\times$ speed up and $3,400\times$ better energy efficiency than a GPU.

All these techniques center around exploiting the sparsity in neural networks, shown in Figure 1.3.

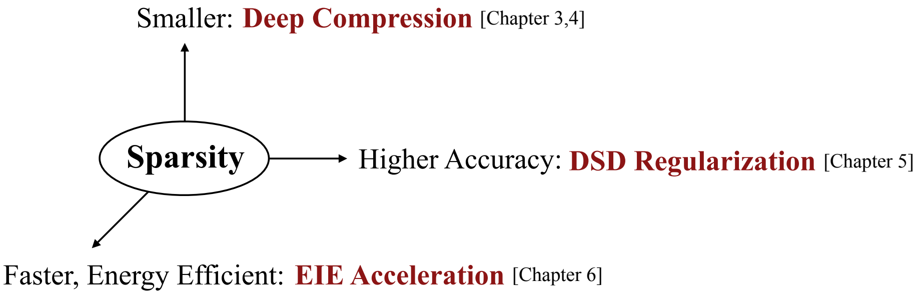

Smaller: Deep Compression [Chapter 3,4]
Sparsity
Higher Accuracy: DSD Regularization [Chapter 5]
Faster, Energy Efficient: EIE Acceleration [Chapter 6]

Figure 1.3: We exploit sparsity to improve the efficiency of neural networks from multiple aspects.

Chapter 2 provides the background for the deep neural networks, datasets, training system, and hardware platform that we used in the thesis. We also survey the related works in model compression, regularization, and hardware acceleration.

Chapter 3 describes the pruning technique which reduces the number of parameters of deep neural networks, thus reducing the computation complexity and memory requirements. We also introduce the iterative retraining methods to fully recover the prediction accuracy, together with hardware efficiency consideration for pruning techniques. The content of this chapter is based primarily on Han *et al.* [25].

Chapter 4 describes the trained quantization technique to reduce the bit width of the parameters in deep neural networks. Combining pruning, trained quantization, and variable length coding, we propose "Deep Compression" that can compress deep neural networks by an order of magnitude without losing accuracy. The content of this chapter is based primarily on Han *et al.* [26].

Chapter 5 explains another benefit of pruning, which is to regularize deep neural networks and prevent overfitting. We propose dense-sparse-dense training (DSD) that periodically prunes and restores the connections, which serves as a regularizor to improve the optimization performance. The content of this chapter is based primarily on Han *et al.* [27].

Chapter 6 presents the "Efficient Inference Engine" (EIE) to efficiently implement deep compression. EIE is a hardware accelerator that performs decompression and inference simultaneously and accelerates the resulting sparse matrix-vector multiplication with weight sharing. EIE takes advantage of the compressed model, which significantly saves memory bandwidth. EIE is also able to deal with the irregular computation pattern efficiently. As a result, EIE achieved significant speedup and energy efficiency improvement over GPU. The content of this chapter is based primarily on Han *et al.* [28] and briefly on Han *et al.* [29].

In **Chapter 7** we summarize the thesis and discuss the future work for efficient deep learning.

Chapter 2

Background

In this chapter, we first introduce what is deep learning, how it works, and its applications. Then we introduce the neural network architectures we experimented with, the datasets, and the frameworks we use to train the architectures on the datasets. After this introduction, we describe previous work in compression, regularization, and acceleration.

Deep learning uses deep neural networks to solve machine learning tasks. Neural networks consist of a collection of neurons and connections. A *neuron* receives many inputs from predecessor neurons and produces one output. The output is a weighted sum of the inputs followed by the neuron's activation function, which is usually nonlinear. Neurons are organized as *layers*. Neurons in the same layer are not connected. Neurons with no predecessor are called input neurons, neurons with no successor are called output neurons. If the number of layers between the input neuron and the output neuron is large, then it is called deep neural network. There is no strict definition, but in general, with more than eight layers it is considered "deep" [2]. Modern deep neural networks can have hundreds of layers [5]. Neurons are wired through *connections*. Each connection transfers the output of a neuron i to the input of another neuron j. Each connection has a weight w_{ij} that will be multiplied with the activation, which will increase or decrease the signal. This weight will be adjusted during the learning process, and this process is called *training*.

Gradient descent is the most common technique for training deep neural networks. It is a first-order optimization method by calculating the gradient of the loss function over the variable and moving the variable in the negative direction of the gradient. The step size is proportional to the absolute value of the gradient. The ratio between the step size and the absolute value of the gradient is called *learning rate*. Calculating the gradient is the key step when performing gradient descent, which is based on the *back-propagation* algorithm. To calculate the gradient with back-propagation, we need to first calculate each layer's activation by performing a feed-forward pass from the input neuron to the output neuron. This forward pass is also called $inference$, the output of inference could either be a continuous value in regression problems, or a discrete value

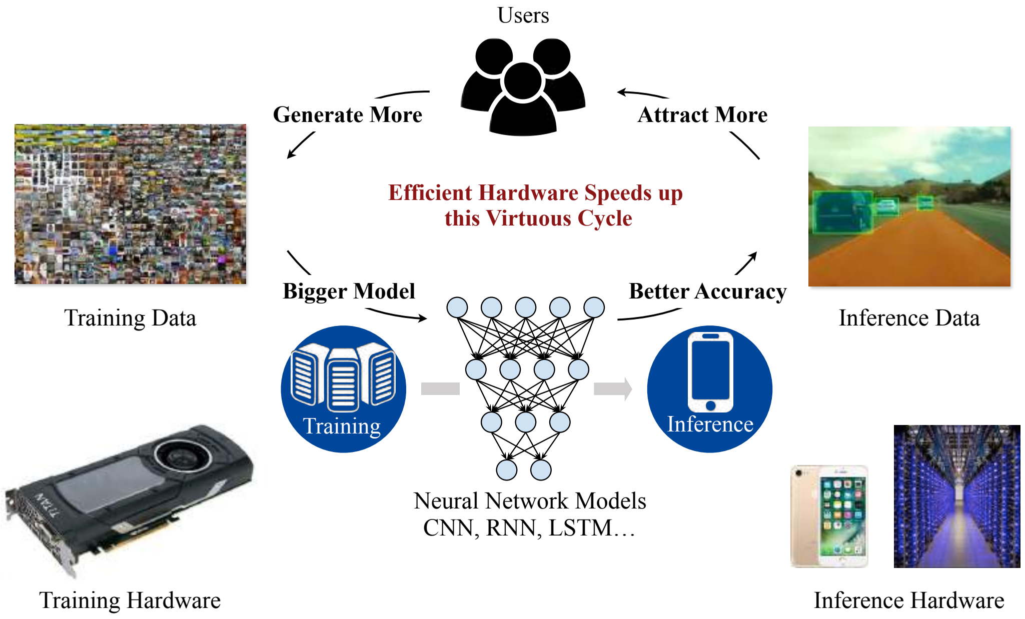


Figure 2.1: The basic setup for deep learning and the virtuous loop. Hardware plays an important role speeding up the cycle.

in classification problems. The inference result could be correct or wrong, which is quantitatively measured by the loss function. Next, we calculate the gradient of the loss function for each neuron and each weight. The gradients are calculated iteratively from the output layer to the input layer according to the chain rule. Then we update the weights with gradient descent $w_{i,j}^{t+1} = w_{i,j}^{t} - \alpha \frac{\partial L}{\partial w_{i,j}}$. Such feed-forward, back-propagation, and weight update constitute one training iteration. It usually takes hundreds of thousands of iterations to train a deep neural network. Training ResNet-50 on ImageNet, for example, takes 450,450 iterations.

Figure 2.1 summarizes the setup of deep learning. Training on the left, inference on the right, model in the middle. There is a virtuous loop with user, data and neural network models. More users will generate more training data (it could be images, speech, search histories or driving actions). The performance of DNNs scales with the amount of training data. With a larger amount of training data, we can train larger models without overfitting, resulting in higher inference accuracy. Better accuracy will attract more users, which will generate more data...This is a positive feedback forming a virtuous cycle. Hardware plays an important role in this cycle. At training time, efficient hardware can improve the productivity of designing new models; deep learning researcher can quickly iterate over different model architectures. At inference time, efficient hardware can improve the user experience by reducing the latency and achieving real-time inference; efficient hardware can also reduce the cost. For example, running DNNs on cheap mobile devices. In sum, hardware makes this virtuous cycle turn faster.

Deep learning has a wide range of applications. The performance of computer vision tasks greatly benefited from deep learning by replacing hand-crafted features with features automatically extracted from deep neural networks [2]. With recent techniques such as batch normalization [33] and residual blocks [5], we can train even deeper neural networks and the image classification accuracy can surpass human beings [5]. These advancements has spawned many vision-related applications, such as self-driving cars [34], medical diagnostics [35] and video surveillance [36]. Deep learning techniques have made significant advancements in generative models, which have improved the efficiency and quality of compressed sensing [37], super resolution [38]. Generative models have born many new applications such as image style transfer [39], visual manipulation [40] and image synthesis [41]. Recurrent neural networks have the power to model sequences of data and greatly improved the accuracy of speech recognition [42], natural language processing [43], and machine translation [11]. Deep reinforcement learning have made progress in game playing [44], visual navigation [45], device placement [46], automatic neural network architecture design [47] and robotic grasping [48]. The big-bang of deep learning applications highlight the importance of improving the efficiency of deep learning computation, as will be discussed in this thesis.

2.1 Neural Network Architectures

In this section, we give an overview of different types of neural networks, including multi-layer perceptron (MLP), convolutional neural network (CNN), and recurrent neural network (RNN). MLP consists of many fully-connected layers each followed by a non-linear function. In a MLP, each neuron from $layer_i$ is connected to $layer_{i+1}$, and the computation boils down to matrix-vector multiplication. MLP accounted for more than 61% of Google TPU's workload [49]. Convolutional Neural Network (CNN) takes advantage of the spatial locality of the input signal (such as images) and shares the weights in space, which makes it invariant to translations of the input. Such weight sharing makes the number of weight much smaller compared to fully-connected layer with the same input/output dimensions. From the hardware perspective, the CNN architecture have good data locality since the kernel can be reused across different places; CNNs are usually computation bounded. Recurrent Neural Network (RNN) captures the temporal information of the input signal (such as speech) and share the weights in time. As time stamp gets longer, RNNs are prone to suffer from the gradient explosion or gradient vanishing problem. Long Short-Term Memory networks (LSTMs) [50] is a popular variant of RNN. LSTM solves the gradient vanishing problem by enabling uninterrupted gradient flow with the hidden cell. From the hardware perspective, RNN and LSTM have a low ratio of operations per weight, which is less efficient for hardware because computation is cheap but fetching the data is expensive. RNNs and LSTMs are usually memory bounded. LSTM accounts for 29% of the workload in TPU [49].

We used the following neural network architectures to evaluate the techniques we proposed for

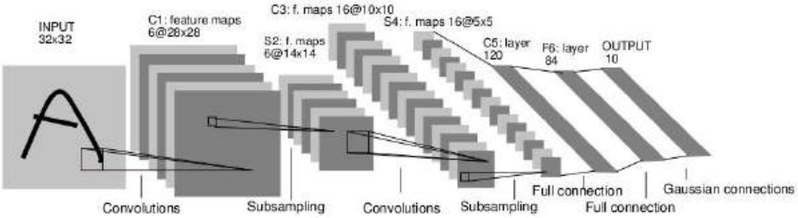


Figure 2.2: Lenet-5 [1] Architecture.

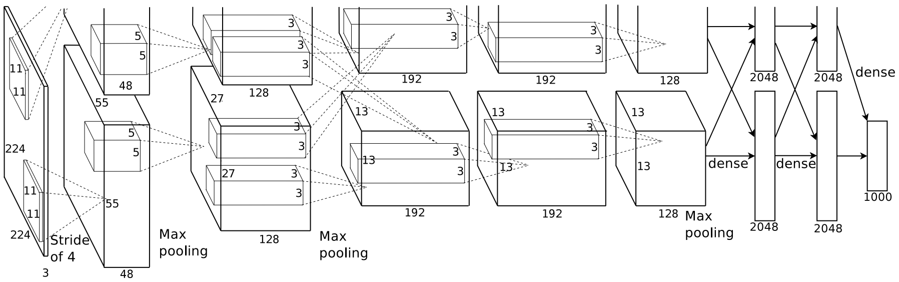


Figure 2.3: AlexNet [2] Architecture.

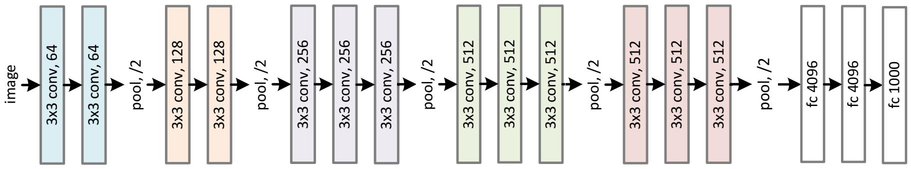


Figure 2.4: VGG-16 [3] Architecture.

model compression and regularization; we also used them as a benchmark to evaluate the performance on different hardware platforms.

LeNet-300-100 [1] (1998) is a fully connected network with two hidden layers harboring 300 and 100 neurons each. LeNet-5 [1] (1998) is a convolutional network which has two convolutional layers and two fully connected layers. Both of them are designed for hand written digits recognition.

AlexNet [2] (2012) significantly decreased the error rate of image classification compared with previous approaches based on hand crafted features. AlexNet has 61 million parameters across five convolutional layers and three fully connected layers. The AlexNet Caffe model achieved a top-1 accuracy of 57.2% and a top-5 accuracy of 80.3% on ImageNet. The convolution layer of AlexNet has three different kernel sizes: 11×11, 5×5, and 3×3.

VGGNet [3] (2014) has 138 million parameters across 13 convolutional layers and three fully connected layers. The VGG-16 Caffe model achieved a top-1 accuracy of 68.5% and a top-5 accuracy of 88.7% on ImageNet. All the convolutional layers of VGG-16 have the same kernel sizes of 3×3.

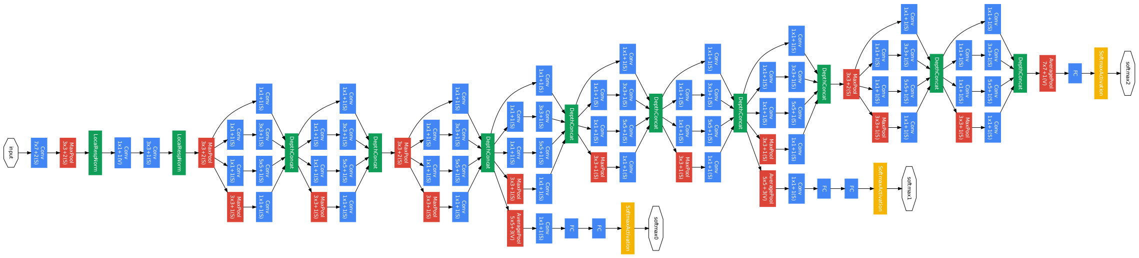


Figure 2.5: GoogleNet [4] Architecture.

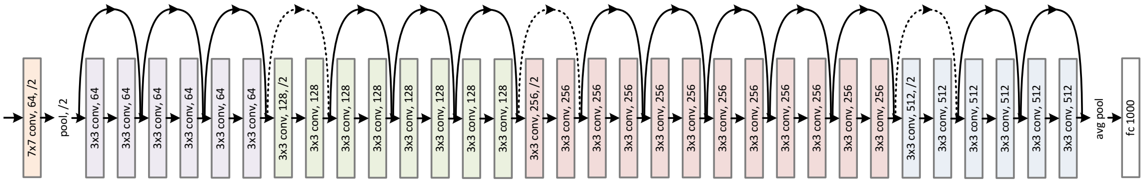


Figure 2.6: ResNet [5] Architecture.

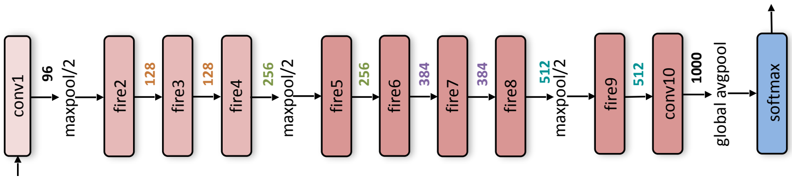


Figure 2.7: SqueezeNet [6] Architecture.

VGG-16 has a strong generalization ability, and the ImageNet pre-trained model is widely used in image classification, detection and segmentation tasks.

GoogleNet (Inception-V1) [51] (2014) is very parameter-efficient. It has 7 million parameters across 57 convolutional layers and only one fully connected layer. GoogleNet has nine inception modules. Each inception module consists of four branches with 1×1, 3×3, 5×5 convolutions and down-sampling. Two auxiliary loss layers inject loss from the intermediate layers and prevent gradient vanishing. At inference time, the auxiliary layers can be removed. The GoogleNet Caffe model achieved a top-1 accuracy of 68.9% and a top-5 accuracy of 89.0% on ImageNet. We also used the Inception-V3 model in our deep compression experiments. Compared with Inception-V1, the 5×5 convolutions are replaced with two 3×3 convolutions, separable kernels came into place, and batch normalization is added in Inception-V3. The pre-trained Inception-V3 PyTorch model achieved a top-1 accuracy of 77.45% and a top-5 accuracy of 93.6% on ImageNet.

ResNet [5] (2015) proposed the residual block with bypass layer, which allows the gradient to flow more easily, even with deeper layers. ResNet-50 has 25.5 million parameters across 49 convolution layers and one fully-connected layer. Each residual block element-wise adds the current feature map

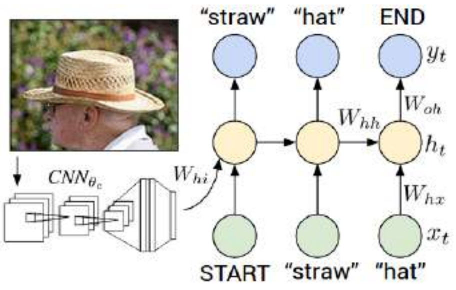


Figure 2.8: NeuralTalk [7] Architecture.

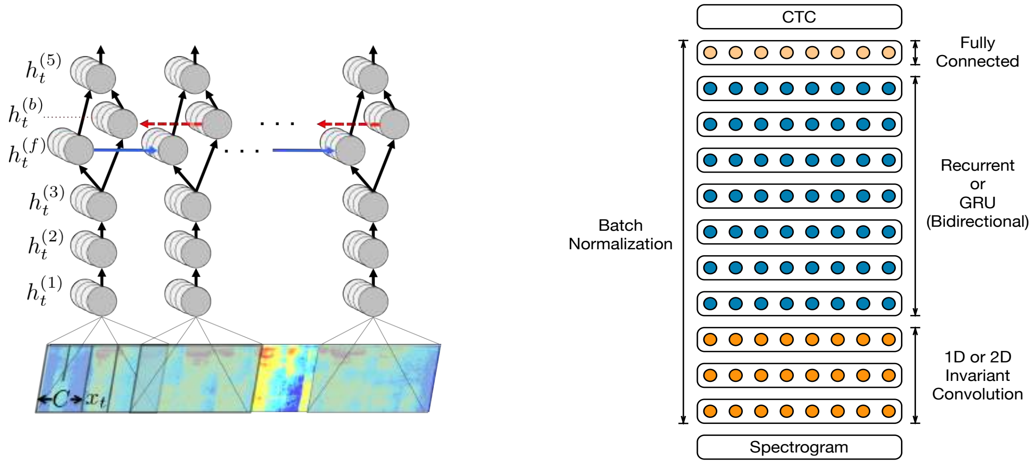


Figure 2.9: DeepSpeech1 [8] (Left) and DeepSpeech2 [9] (Right) Architecture.

with the feature map from the previous residual block. There is also a bottleneck layer with 1×1 convolution that shields a large number of channels for the more expensive 3×3 layer. The pre-trained ResNet-50 PyTorch model achieved a top-1 accuracy of 76.1% and a top-5 accuracy of 92.9% on ImageNet. ResNet-50 is the most popular version of the ResNet family balancing computational complexity and prediction accuracy.

SqueezeNet [6] (2016) targets extremely compact model sizes for mobile applications. It has only 1.2 million parameters but achieved an accuracy similar to AlexNet. SqueezeNet has 26 convolutional layers and no fully connected layer. The last feature map goes through a global pooling and forms a 1000-dimension vector to feed the softmax layer. SqueezeNet has eight "Fire" modules. Each fire module contains a squeeze layer with 1×1 convolution and a pair of 1×1 and 3×3 convolutions. The SqueezeNet caffemodel achieved a top-1 accuracy of 57.4% and a top-5 accuracy of 80.5% on ImageNet. SqueezeNet is widely used in mobile applications in which model size is a large constraint. Our model compression technique can further decrease the SqueezeNet model size by 10× without losing accuracy.

NeuralTalk [7] (2014) is a Long Short Term Memory (LSTM) for generating image captions. It uses the feature generated by VGG-16 to feed an LSTM to generate the captions. It first embeds the image into a 4096 dimension vector by a convolutional neural network (VGG-16). The image embedding feeds the LSTM as input. At each time-step, the LSTM outputs one word of the caption sentence. The baseline NeuralTalk model we used for experiment is downloaded from NeuralTalk Model Zoo. It has 6.8 million parameters with the following dimensions: $W_e : 4096 \times 512, W_{lstm} : 1025 \times 2048, W_d : 512 \times 2538 and W_s : 2538 \times 512$. The baseline BLEU1-4 score is [57.2, 38.6, 25.4 16.8].

DeepSpeech [8] (2014) is a bidirectional recurrent neural network for speech recognition. It is a five-layer network with one bi-directional recurrent layer. It has 8 million parameters in total. DeepSpeech 2 [9] (2015) improved on DeepSpeech 1 and the model size is much larger. It has seven bi-directional recurrent layers with approximately 67 million parameters, around eight times larger than the DeepSpeech 1 model. The DeepSpeech family replaced the previous hybrid NN-HMM model and adopted end-to-end training for automatic speech recognition task.

We extensively experimented our model compression and regularization techniques on the above network architectures that have covered MLP, CNN, RNN, and LSTM.

2.2 Datasets

We used different datasets for a variety of machine learning tasks to test the performance of model compression and regularization techniques. The datasets we used in the experiments include MNIST, Cifar-10, and ImageNet for image classification; Flickr 8K for image caption; and TIMIT and WSJ for speech recognition.

MNIST is a dataset for handwritten digits [52] with 60,000 training images and 10,000 test images. There are ten classes with ten digits, and the image size is 28×28. Each image is grayscale. This dataset is relatively easy and small, taking the model only a few minutes to train. We used MNIST only for prototyping, and larger datasets to further verify each idea.

Cifar-10 is a dataset of color images [53]. It has 50,000 training images and 10,000 test images. There are ten classes, and the image size is 32×32. The dataset is slightly more difficult than MNIST but the model still only required a few hours to train. We used Cifar-10 for ablation studies when we needed to repeat a group of similar experiments many times.

ImageNet is a large-scale dataset for ILSVRC challenge [54]. The training dataset contains 1000 categories and 1.2 million images. The validation dataset contains 50,000 images, 50 per class. The classification performance is reported using Top-1 and Top-5 accuracy. Top-1 accuracy measures the proportion of correctly-labeled images. If one of the five labels with the largest probability is a correct label, then this image is considered to have a correct label for Top-5 accuracy. We used the ImageNet dataset to measure the performance of model compression and regularization.

Flickr-8k is a dataset that includes 8K images obtained from the Flickr website [55]. Each image comes with five natural language descriptions. We used Flickr-8k to measure the performance of the pruning technique and the DSD training technique on image captioning tasks. The BLEU [56] score is used to measure the correlation between the generated caption against ground truth caption. We also used visualization to measure the image caption performance qualitatively.

TIMIT is an acoustic-phonetic continuous speech corpus [57]. It contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences. We used TIMIT to test the performance of the pruning technique on LSTM and speech recognition.

WSJ is a speech recognition data set from the Wall Street Journal published by the Linguistic Data Consortium. It contains 81 hours of speech, and the validation set includes 1 hour of speech. The test sets are WSJ'92 and WSJ'93, which contains 1 hour of speech combined. We used the WSJ data set to test the performance of DSD training on DeepSpeech and DeepSpeech2 for speech recognition.

2.3 Deep Learning Frameworks

Directly programming a multi-core CPU or a GPU can be difficult, but luckily the neural network computation can be abstracted into a few basic operations such as convolution, matrix multiplication, etc. Using these operations that are already highly optimized for high-performance hardware, users only need to focus on high-level neural network architectures rather than low-level implementations. Deep learning frameworks provide these abstractions of neural networks computation, so programmers only need to write a description of the computation, and the deep learning frameworks then efficiently map the computation to high-performance hardware.

We used the Caffe [58] and PyTorch [59] framework for our experiments. Caffe was an early deep learning framework made by the Berkeley Vision and Learning Center. Users only need to write highly abstracted network architecture descriptions and specify the hyper-parameters. Caffe also offers a comprehensive "model zoo" that contains popular pre-trained models. PyTorch is a recent and more flexible framework for deep learning which supports dynamic graph. We used PyTorch for the latest experiments on compressing Inception-V3 and ResNet-50. Both Caffe and PyTorch use the cuDNN library [60] for GPU acceleration.

We used a NVIDIA-DIGITS development box [61] as training hardware. It contains four NVIDIA Maxwell TitanX GPUs with 12GB of memory per GPU. It has 64GB DDR4 memory and Core i7-5930K 6-core 3.5GHz desktop processor. Thanks to the 3TB hard drive configured in RAID5 and the 512GB PCI-E M.2 SSD cache for RAID, we can achieve high disk I/O bandwidth to load the training data.

2.4 Related Work

Given the importance of deep neural networks, there has been enormous research trying to optimize their performance. This section will briefly review the previous work that our research is built on. We first look at prior attempts at network compression, then improved accuracy through regularization, and finally tries to speed up inference using hardware acceleration.

2.4.1 Compressing Neural Networks

Neural networks are typically over-parameterized, and there is significant redundancy for deep learning models [62]. This results in a waste of both computation and memory. There are two directions to make the network smaller: having fewer weights and having lower precision (fewer bits per weight).

There have been proposals to reduce the number of parameters. An early approach was Optimal Brain Damage [63] and Optimal Brain Surgeon [64], which reduced the number of connections based on the Hessian of the loss function. Denton et al. [65] exploited the linear structure of the neural network by finding an appropriate low-rank approximation to reduce the number of parameters. Singular Value Decomposition (SVD) and Tucker Decomposition can also decrease the number of weights [66]. There have been architectural innovations to reduce the number of neural network parameters, such as replacing fully-connected layers with convolutional layers or replacing the fully-connected layer with global average pooling. The Network in Network architecture [67] and GoogleNet [51] achieve state-of-the-art results by adopting this idea.

There also have been proposals to reduce the precision and bit width. Vanhoucke *et al.* [68] explored a fixed-point implementation with 8-bit integer (vs 32-bit floating point) activations. Abwer et al. [69] quantized the neural network using L2 error minimization. Hwang et al. [70] proposed an optimization method for neural network with ternary weights and 3-bit activations. Gong et al. [71] compressed deep neural networks using vector quantization. HashedNets [72] reduced the parameters' bit width by using a hash function to randomly group connection weights into hash tables. With above techniques, the precision of the weight can be reduced, and each weight can be represented with fewer bits.

Our work in Chapter 3, like optimal brain damage, works by pruning network connections, but it carefully chooses the connections to drop enabling it to work on much larger neural networks with no accuracy degradation. Chapter 4 describes how to we minimize the space used for weights, by combining ideas of reduced precision and binning the weights. Again by adaptively choosing the bins, this compression has minimal effect on accuracy. Finally, we combined the ideas of reducing the number of connections and reducing the bit width per connection, which born the Deep Compression algorithm that can significantly compress neural networks without accuracy loss.

Recent Progress of Model Compression. Our work has spawned a lot of efforts in pruning and model compression. New heuristics have been proposed to select the important connections and recover the accuracy. Loss-approximating Taylor expansion [73] proposed gradient-based importance metrics used for pruning. Anwar et al. [74] select pruning candidate by hundreds of random evaluations. Yang et al. [75] select pruning candidate weighted by energy consumption. He et al. [76] explored LASSO regression based channel selection and achieved 2x theoretical speed-up on ResNet and Xception with 1.4%, 1.0% accuracy loss, respectively. Early pruning [77] and dynamic pruning [78] explored how to integrate pruning with re-training better, and save the retraining time. Recent work also discussed structured pruning at different granularities, for example, pruning at the granularity of rows and columns after lowering [79], pruning at the granularity of channels instead of individual weights [73, 80–82]. Anwar et al. [83] explored the structured sparsity at various scales, including channel-wise, kernel-wise and intra-kernel strided sparsity. Mao et al. [84] provided a detailed design space exploration and trade-off analysis for different pruning granularities: element-wise, row-wise, kernel-wise and channel-wise.

As for quantization, more aggressive compression pushed the bit width even narrower, with 2-bit weight [85], ternary weight [86], or binary weight [87]. There are also efforts to quantize both the weight and the activation to a lower precision [88–90]. When both the weight and activation is binarized, multiplication and reduction can be replaced with XNOR and pop count. Alternatively, Miyashita et al. [91] experimented with logarithmically quantized weights, where multiplications can be replaced with shifts, which is cheap.

In industry, Facebook adopted "Deep Compression" for deploying DNNs in mobile phones, powering a mobile AR application debuted in Facebook Developer Conference 2017 [92]. Intel labs used iterative quantization method to shrink to bit width [93]. Baidu adopted Deep Compression to compress deep learning models in the mobile Apps (credit card recognition, face recognition) before shipping the Apps to Apple Store.

2.4.2 Regularizing Neural Networks

The large parameter space of modern DNN requires regularization to prevent overfitting, and help convergence. The most commonly used one is weight decay, which is achieved by adding the L1 or L2 norm of the weight to the loss function. This additional loss term helps the system find solutions with small weight values, but are often not sufficient to find optimal solutions.

To help with the regularization, researchers have tried different approaches to reduce the parameter space. Dropout [94] and Dropconnect [95] select a *random* set of activations or connections during training time, and they use full network for forward path. In each training iteration, a different set of weights and activations got trained, preventing complex co-adaptations on training data.

Batch normalization [96] is a modern technique for regularizing neural networks, where the activations are normalized with a mean of zero and a standard deviation of one in a mini-batch. Batch normalization constraints the dynamic range that the activations can take. Since each mini-batch is shuffled, a training example is seen in conjunction with random examples in the mini-batch, and SGD no longer produces deterministic values for a given training example. This helps the generalization of the network and prevents capturing the noise in the training data. Batch normalization reduces the need for Dropout to prevent over-fitting.

Chapter 5 describes DSD, our regularization approach to improve the model's generalization ability. Like the Dropout and DropConnect, it removes network connections to reduce the solution space, but chooses weights which can be deleted with minimal effect in a *deterministic* manner. Besides, after retraining this smaller model, DSD restores the model to its full size and further tune the model with a lower learning rate. This procedure works well on top of the standard regularization method, which already uses dropout and batch normalization.

Concurrently with our publication of DSD, a similar approach using iterative hard thresholding [97] to regularize neural networks was also published. It performs hard thresholding to drop connections with small activations and fine-tune the other significant filters, then re-activate the frozen connections and train the entire network. It also demonstrates better accuracy on Cifar and ImageNet.

2.4.3 Specialized Hardware for Neural Networks

While hardware DNN accelerators are relatively a new area, there has already been three waves of designs. The first accelerators only looked at the data flow, ignoring the memory energy. The second wave tried to address memory energy. Both the first and the second wave treat the application as a black box. The third wave (our work), looks at joint hardware/software optimization.

The first wave of accelerators include CNP [16], Neuflow [17], DC-CNN [24] and Convolution Engine [18]. This line of work proposed customized logic to efficiently map convolution to hardware with more parallelism and flexibility. The second wave of accelerators focused on optimizing memory transfer and data movement. As modern neural networks get larger, researchers realized that memory access and moving data is more critical than arithmetic. Among these accelerators, DianNao [19] implements an array of multiply-add units to map large DNNs onto its core architecture. It has customized on-chip buffer to minimize DRAM traffic. DaDianNao [20] and ShiDianNao [21] eliminate the DRAM access by having all weights on-chip (eDRAM or SRAM). In both architectures, the weights are uncompressed and stored in the dense format. Eyeriss [22] proposed a row-stationary data flow to maximize data reuse and minimize the memory reference. TPU [49] used 8-bit fixed point quantization that can save the memory bandwidth. Prime [98] and ISAAC [99] proposed to replace DRAM or eDRAM memory and run neural network in memristors. RedEye [100] alleviates the readout circuitry and data traffic by moving the neural network computation to analog domain.

Both waves of DNN accelerators didn't touch the application and treated the workload as a black box. We found that by opening the box and applying model compression, the memory access can be greatly reduced. However, previous accelerators focused on accelerating dense, uncompressed models, which limits their utility to execute compressed models. Without model compression, it is only possible to fit very small neural networks, such as Lenet-5, in on-chip SRAM [21]. These accelerators cannot exploit either form of sparsity and must expand the network to dense form before operation [19, 20], and neither can exploit weight sharing.

There are many research efforts on optimizing sparse matrix-vector multiplication (SPMV) with specialized accelerators. Zhuo et al. [30] proposed a FPGA-based design on Virtex-II Pro for SPMV. Their design outperforms general-purpose processors, but the performance is limited by memory bandwidth. Fowers et al. [31] proposed a novel sparse matrix encoding and a FPGA-optimized architecture for SPMV. With lower bandwidth, it achieves 2.6× and 2.3× higher power efficiency over CPU and GPU, respectively, while having lower performance due to lower memory bandwidth. Dorrance et al. [32] proposed a scalable SMVM kernel on Virtex-5 FPGA. It outperforms GPU counterparts with 38-50× improvement in energy efficiency. However, a compressed network is not efficient on these sparse linear algebra accelerators either. Previous SPMV accelerators can only exploit the static weight sparsity but are unable to exploit dynamic activation sparsity nor weight sharing. Cnvlutin [101] and Minerva [102] can only take and can exploit the activation sparsity, but not weight sparsity nor weight sharing.

Given the constraints of the first and second wave of DNN accelerators, we propose the third wave that performs algorithm and hardware co-design. Chapter 6 describes "Efficient Inference Engine" (EIE), our sparse neural network accelerator. Like the prior work, it exploits the structured data flow of DNN applications and uses customized buffers and memory hierarchy wisely to reduce memory energy. Unlike previous approaches, it uses the sparsity of the application (exploited by Deep Compression) to substantially reduce the required memory and computation. Customizing the hardware for the sparse, compressed neural network dramatically improves its performance over prior approaches.

Recent Trend of DNN Accelerators. After we published our sparse neural network accelerator EIE in ISCA'2016, sparse neural network accelerators become very popular in both academia and industry. In academia, many neural network accelerators appeared that also takes advantage of sparsity. Cambricon-X [103] exploits the weight sparsity by finding the activations that correspond only to non-zero weights. SCNN [104] exploits both weight and activation sparsity by doing the outer product of the two, and also support sparse convolution layers. Cnvlutin2 [105] improved on top of Cnvlutin [101] and supports both weight and activation sparsity to skip the ineffectual computation. ESE [29] used EIE as the basic building block to support sparse RNNs and LSTMs. Li et al. [106] took advantage of sparsity on a coarse granularity and accelerated sparse CNN on FPGA.

In industry, hardware support for sparse neural networks has been rapidly adopted. NVIDIA's recently announced deep learning accelerator (XAVIER DLA) supports sparse weight decompression [107]. NEC developed middleware that incorporates sparse matrix structures to simplify the use of machine learning [108]. Intel labs accelerate CNNs using low-precision and sparsity [85]. In Google's TPU, "sparsity will have high priority in future designs" [49].

Chapter 3

Pruning Deep Neural Networks

3.1 Introduction

Modern deep neural networks have many parameters to provide enough model capacity, making them both computationally and memory intensive. In addition, conventional neural networks fix the architecture before training starts; thus, training cannot improve the architecture. Moreover, the large number of parameters may lead to overfitting. Identifying the right model capacity and removing redundancies are crucial for computational efficiency and accuracy.

To address these problems, we developed a pruning method to remove redundant and keep useful neural network connections, which can decrease the computational and storage requirements for performing inference. The key challenge is how to preserve the original prediction accuracy after pruning the model.

Our pruning method removes redundant connections and learns only important connections (Figure 3.1). In this example, there are three layers. Before pruning, layer i and layer $i + 1$ are densely connected. After pruning, layer i and layer $i + 1$ are sparsely connected. When all the synapses associated with a neuron are pruned, the neuron is also pruned. Pruning turns a dense neural network into a sparse neural network, reducing the number of parameters and computations while fully preserving prediction accuracy. Pruning improves inference speed and also reduces the energy required to run such large networks, permitting use on battery-constrained mobile devices. Pruning also facilitates the storage and transmission of mobile applications which incorporate deep neural networks.

After an initial training phase, we prune the DNN model by removing all connections whose weight is lower than a threshold. This pruning converts a dense layer to a sparse layer. This first phase learns the topology of the networks, noting important connections while removing unimportant connections. We then retrain the sparse network so that the remaining connections can compensate for the removed connections. We then retrain the sparse network so the remaining connections can compensate for

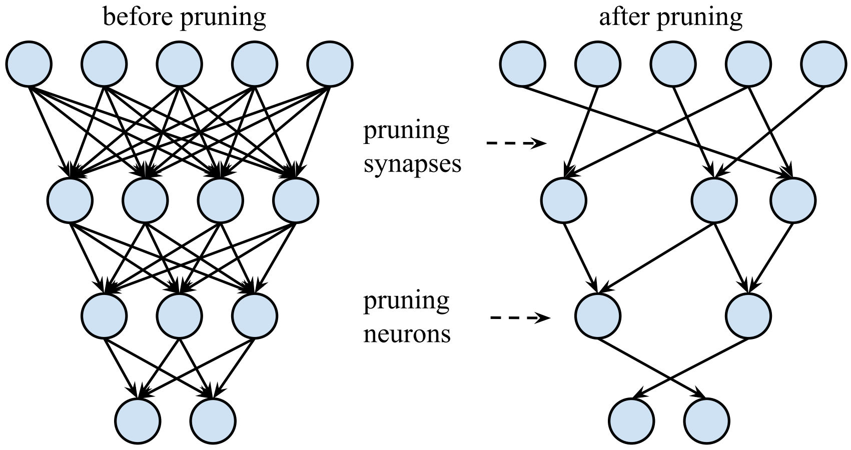


Figure 3.1: Pruning the synapses and neurons of a deep neural network.

the connections that have been removed. The phases of pruning and retraining may be repeated iteratively to further reduce network complexity. In effect, this training process learns the network connectivity in addition to the weights — this parallels the human brain development [109] [110], where excess synapses formed in the first few months of life are gradually "pruned", with neurons losing little-used connections while preserving the functionally important connections.

On the ImageNet dataset, the pruning method reduced the number of parameters of AlexNet by a factor of 9× (61 to 6.7 million), without incurring accuracy loss. Similar experiments with VGG-16 found that the total number of parameters can be reduced by 13× (138 to 10.3 million), again with no loss of accuracy. We also experimented with the more efficient fully-convolutional neural networks: GoogleNet (Inception-V1), SqueezeNet, and ResNet-50, which have zero or very thin fully connected layers. From these experiments we find that they share very similar pruning ratios before the accuracy drops: 70% of the parameters in those fully-convolutional neural networks can be pruned. GoogleNet is pruned from 7 million to 2 million parameters, SqueezeNet from 1.2 million to 0.38 million, and ResNet-50 from 25.5 million to 7.47 million, all with no loss of Top-1 and Top-5 accuracy on Imagenet.

In the following sections, we provide solutions on how to prune neural networks and how to retrain the pruned model to recover prediction accuracy. We also demonstrate the speedup and energy efficiency improvements of the pruned model when run on commodity hardware.

3.2 Pruning Methodology

Our pruning method employs a three-step process: training connectivity, pruning connections, and retraining the remaining weights. The last two steps can be done iteratively to obtain better compression ratios. The process is illustrated in Figure 3.2 and Algorithm 1.

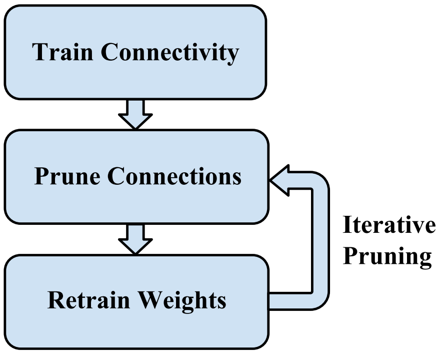


Figure 3.2: The pipeline for iteratively pruning deep neural networks.

The process begins by learning the connectivity via normal network training, as shown in the **Train Connectivity** block in Algorithm 1. Unlike conventional training, however, the purpose of this step is not learning the final values of the weights, but rather learning which connections are important. We use a simple heuristic for determining the importance of a weight, which is the absolute value: if the absolute value is small, then we consider this weight unimportant.

The second step is to prune the connections with low-absolute values, as shown in the **Prune Connectivity** block in Algorithm 1. All connections with weights below a threshold are removed from the network, converting a dense network into a sparse network (Figure 3.1).The threshold is a hyper-parameter that depends on where one wants to fall on the trade-off curve between compression ratio and prediction accuracy, which is discussed in Section 3.4. We use a mask to implement model pruning; weights below the threshold have a mask of zero, while those above have a mask of one. By taking the dot product between the original weight tensor and the mask tensor, the unimportant connections are set to zero, meaning that they are pruned.

The final step retrains the network to learn the final weights for the remaining sparse connections, as shown in the **Retrain Weights** block in Algorithm 1. This step is critical: if the pruned network is used without retraining, the accuracy is significantly impacted. Steps 2 and 3 can be repeated iteratively to provide a better compression ratio.

Learning the correct connections is an iterative process. Pruning followed by retraining is one iteration; after many such iterations, the minimum number of connections can be found. Figure 3.3 compares direct and iterative pruning: though both prune the network to 70% sparsity, iterative pruning eliminates a smaller proportion of weights each time and is compensated for having multiple such iterations. In Figure 3.3's example, $iteration_1$ prunes the network to 30% sparsity, $iteration_2$ prunes the network to 50% sparsity, and $iteration_3$ prunes the network to 70% sparsity. The last code block in Algorithm 1 explains how to implement iterative pruning. Every time we increase the threshold by $\delta[iter]$ to prune more parameters. $\delta[iter]$ is set to meet the required pruning ratio of this iteration.

```
Algorithm 1: Pruning Deep Neural Networks
  Initialization: $W^{(0)}$ with $W^{(0)} \sim N(0, \Sigma)$, $iter = 0$.
  Hyper-parameter: $threshold$, $\delta$.
  Output: $W^{(t)}$.
  ———————————— Train Connectivity ————————————
  while not converged do
      $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
      $t = t + 1$;
  end
  ———————————— Prune Connections ————————————
  // initialize the mask by thresholding the weights.
  $Mask = \mathbb{1}(|W| > threshold)$;
  $W = W \cdot Mask$;
  ———————————— Retrain Weights ————————————
  while not converged do
      $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
      $W^{(t)} = W^{(t)} \cdot Mask$;
      $t = t + 1$;
  end
  ———————————— Iterative Pruning ————————————
  $threshold = threshold + \delta[iter + +]$;
  goto Pruning Connections;
```

Iterative pruning can achieve a better compression ratio than direct pruning. Without loss of accuracy, the iterative pruning method can boost the pruning rate from 5× to 9× on AlexNet compared with single-step direct pruning. Each iteration is a greedy search in that we find the best connections. We also experimented with probabilistically pruning parameters based on their absolute value, but this gave worse results with respect to accuracy.

After pruning connections, neurons with zero input or output connections may be safely pruned. This pruning is then furthered by removing all connections to or from pruned neurons. The retraining phase automatically arrives at the result where dead neurons will have both zero input and output connections. Dead neurons occur due to gradient descent and regularization. A neuron that has zero input or output connections will not contribute to the final loss, making the gradient zero for its output or input connections. Only the regularization term will push the weights to zero. Thus, the dead neurons will be automatically removed after connections are removed.

The training hyper-parameters need to be adjusted at retraining time. Starting with the learning rate adjustment, let LR_1 be the learning rate at the *beginning* of the Train Connectivity phase and LR_2 be the learning rate at the *end* of the Train Connectivity phase. Note that usually, $LR_1 > LR_2$. Let LR be the learning rate at the retraining phase. A practical recipe for adjusting the learning

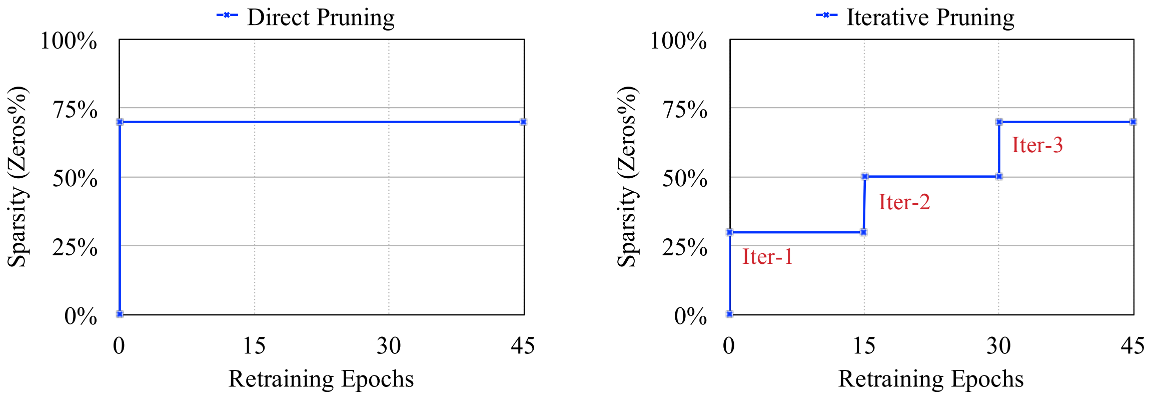


Figure 3.3: Pruning and Iterative Pruning.

rate at the *Retrain Weights* step is to use a learning rate according to Equation 3.1.

$$LR_1 > LR_{retrain} > LR_2. \tag{3.1}$$

Since the weights are already settled to a good local minimum after the initial Train Connectivity phase, the learning rate of the final retraining step needs to be smaller than training from scratch. Because pruning moved the weights away from the original local minimum, the learning rate should be larger than at the end of the Train Connectivity phase. Thus, we pick a learning rate in between. In practice, reducing LR_1 by one or two orders of magnitude generally works well. This equation is a practical guide for the learning rate hyper-parameter search.

Dropout [94] is widely used to prevent overfitting. In dropout, each neuron is probabilistically dropped during training but comes back during inference. In pruning, parameters are deterministically dropped forever after pruning and have no chance to come back during both training and inference. When retraining a pruned model, the dropout ratio must be adjusted to account for the change in model capacity. For a pruned model, as the parameters become sparser, the classifier will select the most informative predictors and thus have much less prediction variance, which reduces over-fitting. Because pruning already reduced model capacity, the retraining dropout ratio should be smaller.

Quantitatively, let C_i be the number of connections in layer i, C_{io} for the original network, C_{ir} for the network after retraining, and let N_i be the number of neurons in layer i. Since dropout works on neurons and C_i varies quadratically with N_i, according to Equation 3.2, the dropout ratio after pruning the parameters should follow Equation 3.3, where D_o represents the original dropout rate, D_r represents the dropout rate during retraining. For dropout that works on the convolutional layer, such as GoogleNet and SqueezeNet, we didn't change the dropout ratio.

$$C_i = N_i N_{i-1} \tag{3.2}$$

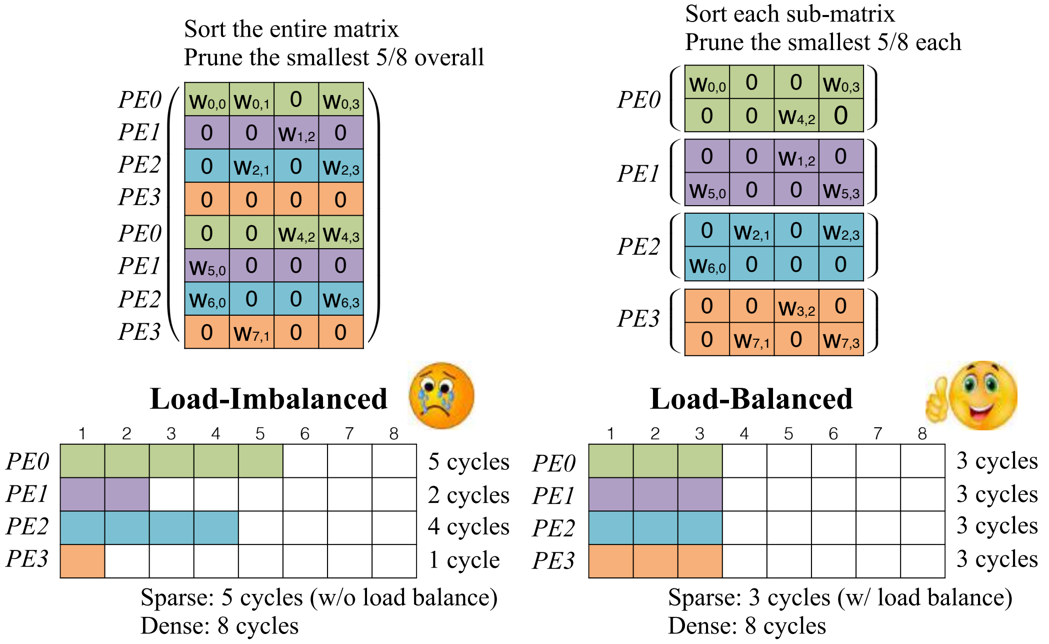


Figure 3.4: Load-balance-aware pruning saves processing cycles for sparse neural network.

$$D_r = D_o\sqrt{\frac{C_{ir}}{C_{io}}} \tag{3.3}$$

During retraining, it is better to retain the weights that are initialized from the Train Connectivity phase for the connections that survived pruning, rather than from zero initialization or random initialization. This is because neural networks contain fragile co-adapted features [111]: gradient descent is able to find a good solution when the network is initially trained but not after re-initializing some layers and then retraining them. Thus, keeping the surviving parameters attains better accuracy when retraining pruned layers.

So far we have discussed the basic techniques to prune deep neural networks and to retrain the sparse model to recover accuracy. Next, we describe advanced pruning techniques from the hardware efficiency perspective.

3.3 Hardware Efficiency Considerations

Pruning makes a dense weight tensor sparse. While this reduces model size and computation, it also reduces the regularity of the remaining computation, which makes it more difficult to parallelize in hardware, as we will see in Chapter 6. It is beneficial to regularize the way we exploit sparsity. The section describes two issues that can be addressed by considering them during pruning.

Load-Balance-Aware Pruning. Modern hardware architecture has many parallel processors.

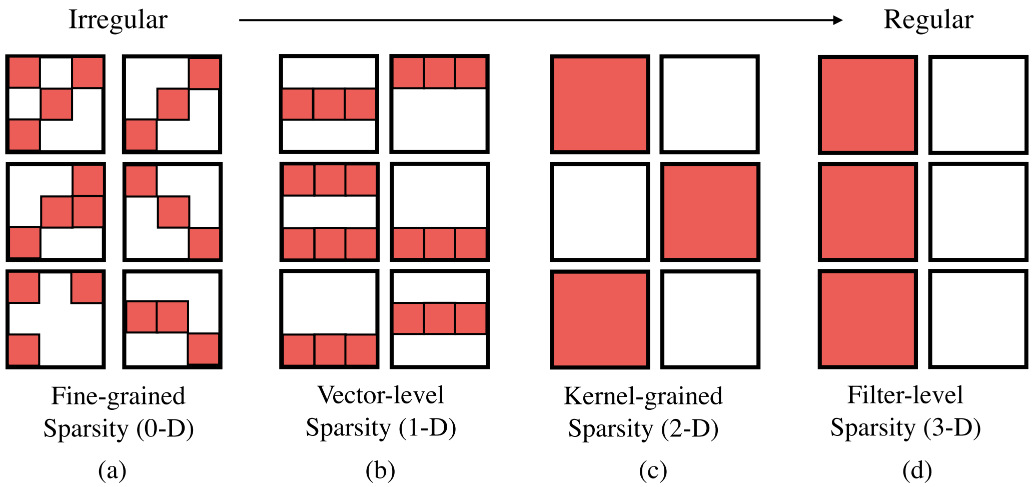

Irregular
Regular
Fine-grained Sparsity (0-D)
(a)
Vector-level Sparsity (1-D)
(b)
Kernel-grained Sparsity (2-D)
(c)
Filter-level Sparsity (3-D)
(d)

Figure 3.5: Pruning at different granularities: from un-structured pruning to structured pruning.

In hardware accelerators, processors are also called processing elements (PEs). Each PE handles part of the workload in parallel, which makes the computation faster than executing the workload in series. Pruning leads to the potential problem of unbalanced non-zero weight distributions when parallelizing the computation on multiple PEs, leading to a situation in which the number of multiply-accumulation operations performed by every PE is different. Consequently, PEs with fewer computation tasks must wait until the PE with the most computation tasks finishes. The workload imbalance over different PEs may cause a gap between the real performance and the peak performance of hardware.

Load-balance-aware pruning is designed to solve this problem and obtain a hardware-friendly sparse network. Load-balance-aware pruning is implemented by adding a constraint to pruning so that the sub-matrices that are allocated to each PE yield the same sparsity ratio. Thus, we ensure an even distribution of non-zero weights among PEs. In the end, all PEs have the same amount of workload, which leads to less idle cycles.

Load-balance-aware pruning is illustrated in Figure 3.4. There are four PEs and the matrix is divided into four sub-matrices for parallel processing. Elements with the same color belongs to the same sub-matrix. The target sparsity level is 5/8 (3/8 non-zeros) in the example. With conventional pruning, the elements in **the whole matrix are sorted** and the smallest 5/8 are pruned. As a result, PE_0 has five non-zero weights while PE_3 has only one. The total processing time is restricted to the longest one, which is five cycles.

With load-balance-aware pruning, however, the elements in **each sub-matrix are sorted**, and the smallest 5/8 for each individual sub-matrix are pruned. As a result, *all* the PEs have 3/8 non-zero weights (three elements in this example); thus, only three cycles are necessary to carry out the operation. Compared with conventional pruning, load-balance-aware pruning produces the same number of non-zero weights, but load-balance-aware pruning needs fewer computation cycles.

Pruning with Structure. While load-balance-aware pruning keeps all processors equally busy, they all need to fetch the non-zero weights in the sparse network (Figure 3.5(a)). We could make this access more regular by pruning by rows, kernels, or filters instead of individual elements.

Pruning at the granularity of rows is shown in Figure 3.5(b). Figure 3.5(c) shows pruning at the granularity of two-dimension kernels; every time a 2D kernel is pruned and it saves the entire 2D convolution on the corresponding feature map. Figure 3.5(d) shows pruning at the granularity of three-dimension filters. Pruning one 3D filter means reducing one output channel, and the remaining model is still a dense tensor.

There is a trade-off for different grain sizes. An advantage of coarse-grained pruning is index saving. We need an index to represent the location of non-zero entries. For fine-grained sparsity, we need one index for **each** non-zero weight. While for kernel-level sparsity, we need one index for **every nine** non-zero weights (assume 3×3 kernel). This is a 9× storage savings on indices.

The downside of coarse-grained pruning is that we can prune less connections with the same accuracy. For example, without any structure, we can prune GoogleNet to only 29% non-zero connections; if enforcing 1-D structure, we need 37% non-zero connections; if pruning 2D kernels, this number rises to 44%. Still, we can save 56% of the 2D convolutions, and the advantage is that the remaining 2D convolutions are still dense, which makes the computation pattern more regular. Structured pruning therefore buys us computational regularity at the cost of the pruning rate.

The storage ratio might either go up or go down with structured pruning, since we have more non-zero parameters but less index overhead for coarse-grained sparsity. For GoogleNet, 1D pruning leads to the best compression ratio compared with 2D pruning and fine-grained pruning.

3.4 Experiments

We implemented network pruning in Caffe [112]. Caffe was modified to add a mask that disregards pruned parameters during network operation for each weight tensor. We carried out the experiments on Nvidia TitanX and GTX980 GPUs.

We pruned six representative networks: Lenet-300-100 and Lenet-5 on MNIST, together with AlexNet, VGG-16, Inception-V1, SqueezeNet, ResNet-18 and ResNet-50 on ImageNet. The network parameters and accuracy[1] before and after pruning are shown in Table 3.1.

3.4.1 Pruning for MNIST

We first experimented on the MNIST dataset with the LeNet-300-100 and LeNet-5 networks [1]. After pruning, the network is retrained with 1/10 of the original network's initial learning rate. Table 3.2 and Table 3.3 show that pruning reduces the number of connections by 12× on these networks. For each layer of the network, the table shows (left to right) the original number of weights, the number

[1] The reference model is from Caffe model zoo; accuracy is measured without data augmentation

Table 3.1: Summary of pruning deep neural networks.

Network	Top-1 Error	Top-5 Error	Parameters	Pruning Rate
LeNet-300-100	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	**22K**	**12×**
LeNet-5	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	**36K**	**12×**
AlexNet	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9×**
VGG-16	31.50%	11.32%	138M	
VGG-16 Pruned	31.34%	10.88%	**10.3M**	**13×**
GoogleNet	31.14%	10.96%	7.0M	
GoogleNet Pruned	31.04%	10.88%	**2.0M**	**3.5×**
SqueezeNet	42.56%	19.52%	1.2M	
SqueezeNet Pruned	42.26%	19.34%	**0.38M**	**3.2×**
ResNet-50	23.85%	7.13%	25.5M	
ResNet-50 Pruned	23.65%	6.85%	**7.47M**	**3.4×**

Table 3.2: Pruning Lenet-300-100 reduces the number of weights by 12× and computation by 12×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
fc1	235K	470K	38%	8%	8%
fc2	30K	60K	65%	9%	4%
fc3	1K	2K	100%	26%	17%
total	266K	532K	46%	**8%**	**8%**

of floating point operations to compute that layer's activations, the average percentage of activations that are non-zero, the percentage of non-zero weights after pruning, and the percentage of actually required floating point operations. One multiplication operation plus one add operation count as two floating point operations.

An interesting byproduct is that network pruning detects visual attention regions. Figure 3.6 shows the sparsity pattern of the first fully connected layer of LeNet-300-100; the matrix size is 784×300. The sparsity pattern of the matrix has 28 bands, each band's width is 28, corresponding to the 28×28 input pixels. The colored regions of the figure (indicating non-zero parameters) correspond to the center of the image. Because digits are written in the center of the image, these center parameters are important. The graph is sparse on the left and right, corresponding to the less important regions on the top and bottom of the image. After pruning, the neural network finds the center of the image more important, and thus the connections to the peripheral regions are more heavily pruned.

Table 3.3: Pruning Lenet-5 reduces the number of weights by 12× and computation by 6×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1	0.5K	576K	82%	66%	66%
conv2	25K	3200K	72%	12%	10%
fc1	400K	800K	55%	8%	6%
fc2	5K	10K	100%	19%	10%
total	431K	4586K	77%	**8%**	**16%**

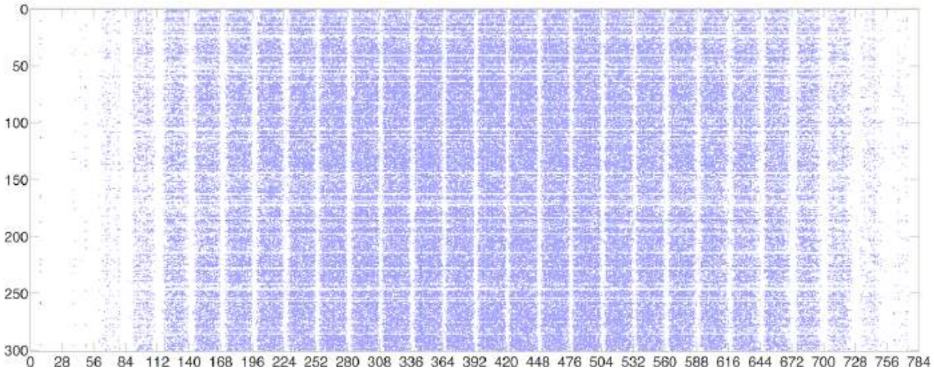


Figure 3.6: Visualization of the sparsity pattern.

3.4.2 Pruning for ImageNet

Beyond the small MNIST dataset, we further examined the performance of pruning on the ImageNet ILSVRC-2012 dataset, which is much larger and hence the result more convincing.

AlexNet. We first pruned AlexNet [2]. We used AlexNet Caffe model as the reference model, which achieved a top-1 accuracy of 57.2% and a top-5 accuracy of 80.3%. After pruning, the whole network is retrained with 1/100 of the original network's initial learning rate. Table 3.4 shows that AlexNet can be pruned to 1/9 of its original size without impacting accuracy, and the amount of computation can be reduced by 3×. The layer-wise pruning statistics are shown in Table 3.4

VGG-16. Given the promising results on AlexNet, we also looked at a larger and more accurate network, VGG-16 [113], on the same ILSVRC-2012 dataset. VGG-16 has far more convolutional layers but still only three fully-connected layers. Following a similar methodology, we aggressively pruned both convolutional and fully-connected layers to realize a significant reduction in the number of weights (Table 3.5). The convolutional layers are pruned to about 30% non-zeros.

The VGG-16 network as a whole has been reduced to 7.5% of its original size (13× smaller). In particular, note that the two largest fully-connected layers can each be pruned to less than 4% of their original size. This reduction is critical for real-time image processing, where there is little reuse of fully connected layers across images (unlike batch processing during training).

Both AlexNet and VGG-16 have three bulky fully connected layers, which occupy more than 90% of the total weights. Those fully connected layers have a great deal of redundancy and can be pruned

Table 3.4: Pruning AlexNet reduces the number of weights by 9× and computation by 3×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10%
total	61M	1.5B	54%	**11%**	**30%**

Table 3.5: Pruning VGG-16 reduces the number of weights by 12× and computation by 5×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1_1	2K	0.2B	53%	58%	58%
conv1_2	37K	3.7B	89%	22%	12%
conv2_1	74K	1.8B	80%	34%	30%
conv2_2	148K	3.7B	81%	36%	29%
conv3_1	295K	1.8B	68%	53%	43%
conv3_2	590K	3.7B	70%	24%	16%
conv3_3	590K	3.7B	64%	42%	29%
conv4_1	1M	1.8B	51%	32%	21%
conv4_2	2M	3.7B	45%	27%	14%
conv4_3	2M	3.7B	34%	34%	15%
conv5_1	2M	925M	32%	35%	12%
conv5_2	2M	925M	29%	29%	9%
conv5_3	2M	925M	19%	36%	11%
fc6	103M	206M	38%	4%	1%
fc7	17M	34M	42%	4%	2%
fc8	4M	8M	100%	23%	9%
total	138M	30.9B	64%	**7.5%**	**21%**

by an order of magnitude. We also wanted to examine fully-convolutional neural networks. We picked GoogleNet (Inception-V1), SqueezeNet, and ResNet-50 as representative, fully-convolutional neural networks on which to experiment.

GoogleNet. We experimented with GoogleNet Inception-v1 model, which has five inception modules (each with four branches) and only one fully connected layer. Table 3.6 shows the layer-wise statistics. Pruning GoogleNet reduces the number of weights by 3.4× and computation by 4.5×. It is unsurprising that the pruning ratio of GoogleNet is smaller than AlexNet and VGG-16 because convolutional layers dominate GoogleNet, and convolutional layers are much more efficient than fully connected layers. Nevertheless, most convolutional layers in GoogleNet can be pruned away 70%, with only 30% being non-zero. The first few convolutional layers generating lower-level features have less room to be pruned.

Table 3.6: Pruning GoogleNet reduces the number of weights by 3.5× and computation by 5×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1/7x7_s2	9K	236M	45%	50%	50%
conv2/3x3_reduce	4K	26M	62%	50%	23%
conv2/3x3	111K	694M	27%	40%	25%
inception_3a/1x1	12K	19M	67%	38%	10%
inception_3a/3x3_reduce	18K	29M	72%	30%	20%
inception_3a/3x3	111K	173M	51%	30%	22%
inception_3a/5x5_reduce	3K	5M	79%	30%	15%
inception_3a/5x5	13K	20M	51%	30%	24%
inception_3a/pool_proj	6K	10M	56%	40%	20%
inception_3b/1x1	33K	51M	32%	40%	22%
inception_3b/3x3_reduce	33K	51M	57%	30%	10%
inception_3b/3x3	221K	347M	21%	30%	17%
inception_3b/5x5_reduce	8K	13M	68%	30%	6%
inception_3b/5x5	77K	120M	18%	30%	20%
inception_3b/pool_proj	16K	26M	22%	40%	7%
inception_4a/1x1	92K	36M	34%	30%	7%
inception_4a/3x3_reduce	46K	18M	68%	30%	10%
inception_4a/3x3	180K	70M	18%	30%	20%
inception_4a/5x5_reduce	8K	3M	80%	30%	5%
inception_4a/5x5	19K	8M	28%	30%	24%
inception_4a/pool_proj	31K	12M	22%	30%	8%
inception_4b/1x1	82K	32M	69%	30%	7%
inception_4b/3x3_reduce	57K	22M	74%	30%	21%
inception_4b/3x3	226K	89M	50%	30%	22%
inception_4b/5x5_reduce	12K	5M	81%	30%	15%
inception_4b/5x5	38K	15M	41%	30%	24%
inception_4b/pool_proj	33K	13M	42%	30%	12%
inception_4c/1x1	66K	26M	60%	30%	13%
inception_4c/3x3_reduce	66K	26M	60%	30%	18%
inception_4c/3x3	295K	116M	40%	30%	18%
inception_4c/5x5_reduce	12K	5M	58%	30%	12%
inception_4c/5x5	38K	15M	37%	30%	17%
inception_4c/pool_proj	33K	13M	35%	30%	11%
inception_4d/1x1	57K	22M	33%	30%	10%
inception_4d/3x3_reduce	74K	29M	44%	30%	10%
inception_4d/3x3	373K	146M	23%	30%	13%
inception_4d/5x5_reduce	16K	6M	64%	30%	7%
inception_4d/5x5	51K	20M	23%	30%	19%
inception_4d/pool_proj	33K	13M	19%	30%	7%
inception_4e/1x1	135K	53M	26%	30%	6%
inception_4e/3x3_reduce	84K	33M	67%	30%	8%
inception_4e/3x3	461K	181M	26%	30%	20%
inception_4e/5x5_reduce	17K	7M	83%	30%	8%
inception_4e/5x5	102K	40M	17%	30%	25%
inception_4e/pool_proj	68K	26M	21%	30%	5%
inception_5a/1x1	213K	21M	38%	30%	6%
inception_5a/3x3_reduce	133K	13M	56%	30%	11%
inception_5a/3x3	461K	45M	27%	30%	17%

inception_5a/5x5_reduce	27K	3M	68%	30%	8%
inception_5a/5x5	102K	10M	26%	30%	20%
inception_5a/pool_proj	106K	10M	22%	30%	8%
inception_5b/1x1	319K	31M	16%	30%	7%
inception_5b/3x3_reduce	160K	16M	27%	30%	5%
inception_5b/3x3	664K	65M	18%	30%	8%
inception_5b/5x5_reduce	40K	4M	35%	30%	5%
inception_5b/5x5	154K	15M	21%	30%	10%
inception_5b/pool_proj	106K	10M	12%	30%	6%
loss3/classifier	1,024K	2M	100%	20%	2%
total	7M	3.2B	40%	29%	20%

SqueezeNet. SqueezeNet [6] is another fully-convolutional neural network for which we examined the effect of pruning. We experimented with SqueezeNet-v1.0, which has eight fire modules. SqueezeNet has no fully connected layers; instead, it uses a global average pooling layer after the last convolutional layer to provide a 1000-dimension vector. SqueezeNet is a very efficient model even before pruning: with only 1.2 million parameters, it is 50× smaller than AlexNet but achieves the same accuracy. After pruning, only 0.4 million parameters are needed, 150× less than AlexNet.

Similar to GoogleNet, SqueezeNet is dominated by convolutional layers, which can be pruned about 3×. Some 1x1 convolutions have less potential to be pruned, but they contribute very little to either the number of parameters or the number of arithmetic operations.

Table 3.7: Pruning SqueezeNet reduces the number of weights by 3.2× and computation by 3.5×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1	14K	348M	51%	80%	80%
fire2/conv1x1_1	2K	9M	89%	75%	39%
fire2/conv1x1_2	1K	6M	63%	70%	62%
fire2/conv3x3_2	9K	56M	50%	30%	19%
fire3/conv1x1_1	2K	12M	93%	70%	35%
fire3/conv1x1_2	1K	6M	74%	70%	65%
fire3/conv3x3_2	9K	56M	43%	30%	22%
fire4/conv1x1_1	4K	25M	75%	70%	30%
fire4/conv1x1_2	4K	25M	52%	70%	52%
fire4/conv3x3_2	37K	223M	33%	30%	16%
fire5/conv1x1_1	8K	12M	83%	70%	23%
fire5/conv1x1_2	4K	6M	58%	70%	58%
fire5/conv3x3_2	37K	54M	42%	30%	17%
fire6/conv1x1_1	12K	18M	79%	70%	29%
fire6/conv1x1_2	9K	13M	43%	50%	40%
fire6/conv3x3_2	83K	121M	24%	30%	13%
fire7/conv1x1_1	18K	27M	73%	50%	12%
fire7/conv1x1_2	9K	13M	52%	70%	51%
fire7/conv3x3_2	83K	121M	36%	30%	15%
fire8/conv1x1_1	25K	36M	69%	70%	25%

fire8/conv1x1_2	16K	24M	28%	50%	34%
fire8/conv3x3_2	147K	215M	12%	30%	8%
fire9/conv1x1_1	33K	11M	76%	50%	6%
fire9/conv1x1_2	16K	6M	18%	70%	54%
fire9/conv3x3_2	147K	50M	11%	30%	5%
conv_final	512K	230M	100%	20%	2%
total	1.2M	1.7B	47%	31%	28%

ResNet-50. We pruned the state-of-the-art convolutional neural network architecture ResNet. ResNet consists of many residual blocks, each with a bypass layer that provides a shortcut for gradient propagation. Pruning reduces ResNet-50's weights by 3.4× and computation by 6.25× (Table 3.8). Again we used the iterative pruning strategy, with 3 pruning iterations in total. The retraining step took 90 epochs, which is the same as the initial training time.

Different from other architectures, ResNet-50 has bypass layers, which introduce the element-wise add operation after each residual block. This bypass changes the activation sparsity: in Table 3.8, the activation density for the *last* convolutional layer of each residual block shows the density *after* the element-wise add operation, which directly feeds to the next convolutional layer. Another special case is the activation after layer4.1.conv3, which is completely dense. This is because the last convolutional layer is followed by a global average pooling. Averaging over several sparse elements results in a dense value, which destroys the activation sparsity.

GoogleNet, SqueezeNet and ResNet are all fully-convolutional neural networks, but they still can be pruned without losing accuracy; their pruning ratios are all similar, with about 30% of the parameters non-zero (GoogleNet: 29%, SqueezeNet: 31%, ResNet: 29%)

Table 3.8: Pruning ResNet-50 reduces the number of weights by 3.4× and computation by 6.25×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1	9K	236B	90%	50%	50%
layer1.0.conv1	4K	26B	58%	40%	36%
layer1.0.conv2	37K	231B	64%	30%	18%
layer1.0.conv3	16K	103B	59%	30%	19%
layer1.0.shortcut	16K	103B	59%	40%	36%
layer1.1.conv1	16K	103B	48%	30%	18%
layer1.1.conv2	37K	231B	51%	30%	14%
layer1.1.conv3	16K	103B	75%	30%	15%
layer1.2.conv1	16K	103B	49%	30%	22%
layer1.2.conv2	37K	231B	44%	30%	15%
layer1.2.conv3	16K	103B	80%	30%	13%
layer2.0.conv1	33K	206B	41%	40%	32%
layer2.0.conv2	147K	231B	58%	33%	14%
layer2.0.conv3	66K	103B	50%	30%	17%
layer2.0.shortcut	131K	206B	50%	30%	24%
layer2.1.conv1	66K	103B	61%	30%	15%

layer2.1.conv2	147K	231B	56%	30%	18%
layer2.1.conv3	66K	103B	66%	30%	17%
layer2.2.conv1	66K	103B	54%	30%	20%
layer2.2.conv2	147K	231B	55%	30%	16%
layer2.2.conv3	66K	103B	58%	30%	16%
layer2.3.conv1	66K	103B	49%	30%	17%
layer2.3.conv2	147K	231B	41%	30%	15%
layer2.3.conv3	66K	103B	59%	30%	12%
layer3.0.conv1	131K	206B	32%	40%	23%
layer3.0.conv2	590K	231B	62%	30%	10%
layer3.0.conv3	262K	103B	47%	30%	18%
layer3.0.shortcut	524K	206B	47%	30%	18%
layer3.1.conv1	262K	103B	48%	30%	14%
layer3.1.conv2	590K	231B	43%	30%	14%
layer3.1.conv3	262K	103B	52%	30%	13%
layer3.2.conv1	262K	103B	41%	30%	15%
layer3.2.conv2	590K	231B	40%	30%	12%
layer3.2.conv3	262K	103B	50%	30%	12%
layer3.3.conv1	262K	103B	36%	30%	15%
layer3.3.conv2	590K	231B	39%	30%	11%
layer3.3.conv3	262K	103B	48%	30%	12%
layer3.4.conv1	262K	103B	34%	30%	14%
layer3.4.conv2	590K	231B	35%	30%	10%
layer3.4.conv3	262K	103B	40%	30%	11%
layer3.5.conv1	262K	103B	31%	30%	12%
layer3.5.conv2	590K	231B	36%	30%	9%
layer3.5.conv3	262K	103B	32%	30%	11%
layer4.0.conv1	524K	206B	23%	30%	10%
layer4.0.conv2	2M	231B	38%	30%	7%
layer4.0.conv3	1M	103B	41%	30%	12%
layer4.0.shortcut	2M	206B	41%	30%	10%
layer4.1.conv1	1M	103B	27%	30%	12%
layer4.1.conv2	2M	231B	32%	30%	8%
layer4.1.conv3	1M	103B	56%	30%	10%
layer4.1.conv1	1M	103B	21%	30%	17%
layer4.1.conv2	2M	231B	37%	30%	6%
layer4.1.conv3	1M	103B	100%	30%	11%
fc	2M	4K	100%	20%	20%
total	25.5M	8G	56%	29%	16%

3.4.3 Pruning RNNs and LSTMs

Having demonstrated that the pruning technique works well on CNNs, we also evaluated our pruning techniques on RNNs and LSTMs. We applied our model pruning to NeuralTalk [7], an LSTM for generating image descriptions. It uses a CNN as an image feature extractor and a LSTM to generate

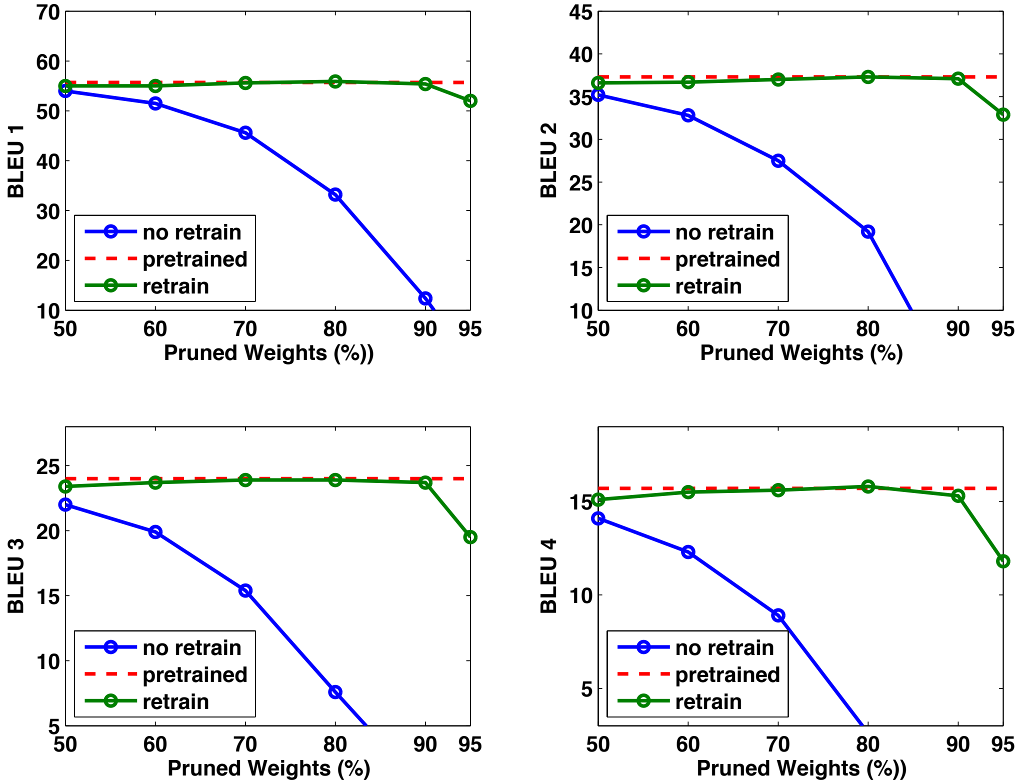


Figure 3.7: Pruning the NeuralTalk LSTM reduces the number of weights by 10×.

captions. To show that LSTMs can be pruned, we fixed the CNN weights and pruned only the LSTM weights. The baseline NeuralTalk model we used is downloaded from the NeuralTalk Model Zoo.

In the pruning step, we pruned all layers except W_s, the word embedding lookup table, to only 10% non-zeros. We retrained the remaining sparse network using the same weight decay and batch size as the original paper. We measured the BLEU score before and after retraining. The BLEU score measures the similarity of the generated caption to the ground truth caption.

Figure 3.7 shows the trade-off curve of the BLEU score and the ratio of pruned weights. The dashed red line shows the baseline dense model's BLUE score; the blue curve shows the BLEU score without retraining; the green curve shows the BLEU score after retraining. Comparing the blue curve and the green curve, we find that retraining plays a very important role in recovering accuracy. Seeing the green line itself, we find that not until pruning away 90% of the parameters does the BLEU score begins to drop sharply.

The BLEU score is not the only quality metric of an image captioning system. We visualize the captions generated by the pruned model (to 10% density) in Figure 3.8 and compare the quality. The first line is the caption generated by the baseline dense model, and the second is generated by the

Baseline: a white bird is flying over water.

Pruned: a white bird is flying over water.

Baseline: a basketball player in a white uniform is playing with a ball.

Pruned: a basketball player in a white uniform is playing with a basketball.

Baseline: a brown dog is running through a grassy field.

Pruned: a brown dog is running through a grassy area.

Baseline: a man is riding a surfboard on a wave.

Pruned: a man in a wetsuit is riding a wave on a beach.

Figure 3.8: Pruning the NeuralTalk LSTM does not hurt image caption quality.

sparse model. The pruned model sometimes produces the same caption (the first image), sometimes produces a different word to describe the same thing (the second and third image), or comes up with completely new captions (the last image).

3.5 Speedup and Energy Efficiency

After pruning, the storage requirements of DNNs are smaller, which reduces the off-chip DRAM access, or even completely avoids off-chip DRAM access if all weights can be stored on chip. On-chip memory takes orders of magnitude less energy and a fewer number of cycles to access, which makes it possible for faster and more energy efficient inference.

Performance of DNNs, especially the fully-connected layers, depends on whether multiple images can be run through the DNN at the same time (batch processing) or run one at a time. Batching images allows for the weights to be reused, which often improves the performance, but also increases the latency to get the result from the first image.

We are considering latency-focused applications running on mobile devices, such as pedestrian detection on an embedded processor inside an autonomous vehicle. These applications demand real-time inference and minimal latency and would be significantly hindered by waiting for batch assembly. Thus, when benchmarking the performance and energy efficiency, we emphasize the case when the batch size equals to 1.

We benchmarked the fully connected layers of AlexNet and VGG-16 to see the effect of pruning on performance and energy. In the non-batching case, the activation matrix is a vector, so the computation boils down to dense/sparse matrix-vector multiplication for the original/pruned model, respectively.

We compared three different kinds of off-the-shelf hardware: the NVIDIA GeForce GTX Titan

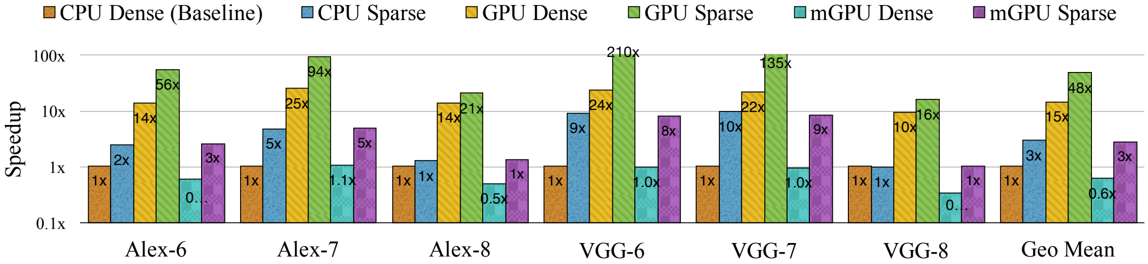


Figure 3.9: Speedup of sparse neural networks on CPU, GPU and mobile GPU with batch size of 1.

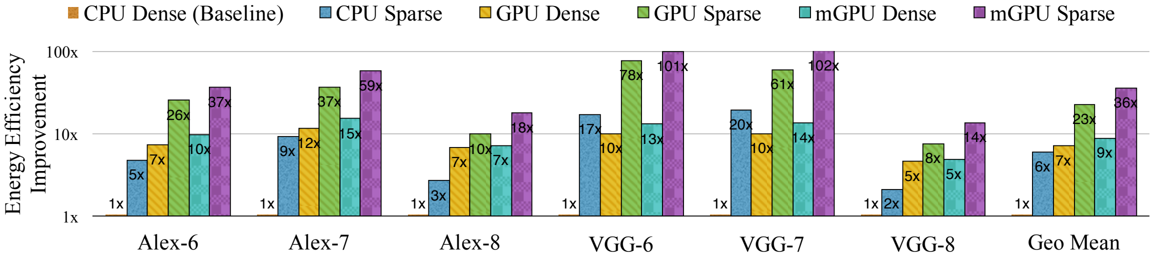


Figure 3.10: Energy efficiency improvement of sparse neural networks on CPU, GPU and mobile GPU with batch size of 1.

X and the Intel Core i7 5930K as desktop processors (same package as NVIDIA Digits Dev Box) and NVIDIA Tegra K1 as a mobile processor. To run the benchmark on a GPU, we used cuBLAS GEMV for the original dense layer. For the pruned sparse layer, we stored the sparse matrix in the CSR format and used the cuSPARSE CSRMV kernel, which is optimized for sparse matrix-vector multiplication on GPU. To run the benchmark on CPU, we used MKL CBLAS GEMV for the original dense model and MKL SPBLAS CSRMV for the pruned sparse model. To avoid variance when measuring the latency, we measured the time spent on each layer for 4096 input samples, and averaged the time regarding each input sample. For GPU, the time consumed by `cudaMalloc` and `cudaMemcpy` was not counted.

To compare power consumption between different systems, it is important to measure power in a consistent manner [114]. For our analysis, we are comparing the pre-regulation power of the entire application processor (AP) / SOC and DRAM combined. On CPU, the benchmark is running on a single socket with a single Haswell-E class Core i7-5930K processor. The CPU socket and DRAM power are as reported by the `pcm-power` utility provided by Intel. For GPU, we used the `nvidia-smi` utility to report the power of Titan X. For mobile GPU, we use a Jetson TK1 development board and measured the total power consumption with a power-meter. We assume 15% AC to DC conversion loss, 85% regulator efficiency, and 15% power consumed by peripheral components [115] to report the AP+DRAM power for Tegra K1.

Figure 3.9 shows the speedup of pruning on different hardwares. There are 6 columns for each

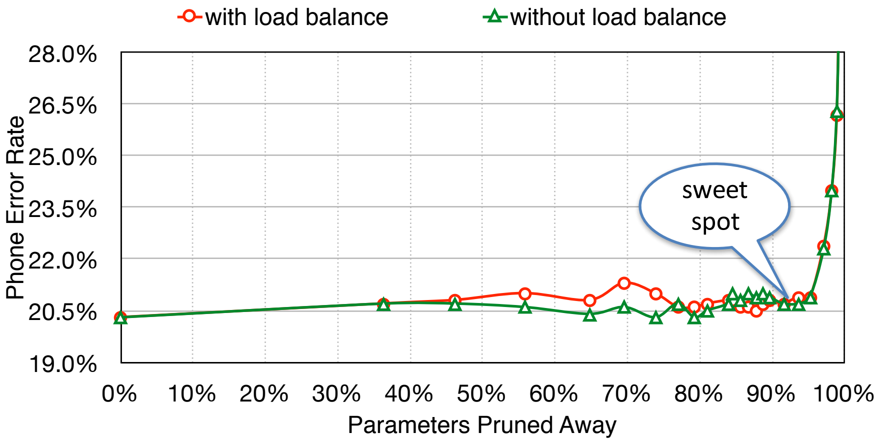


Figure 3.11: Accuracy comparison of load-balance-aware pruning and original pruning.

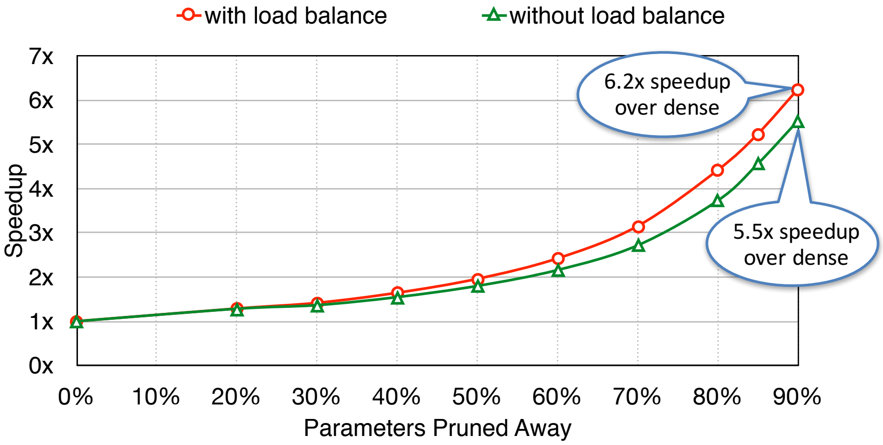


Figure 3.12: Speedup comparison of load-balance-aware pruning and original pruning.

benchmark, showing the computation time of CPU / GPU / TK1 on dense / pruned networks. The time is normalized to CPU. When batch size is equal to 1, the pruned network layer obtained 3× to 4× speedup over the dense network on average because it has a smaller memory footprint and alleviates the data transferring overhead, especially for large matrices that are unable to fit into the caches. For example, VGG16's FC6 layer, the largest layer in our experiment, contains $25088 \times 4096 \times 4\ Bytes \approx 400MB$ data, which is much larger than the capacity of a typical L3 cache (usually 16MB).

In latency-tolerating applications such as off-line image processing, batching improves memory locality when weights could be reused in matrix-matrix multiplication. In this scenario, the pruned network no longer has an advantage and resulted in a slowdown of 2× to 4× for the batch $= 64$ case.

This is because the expensive memory fetch is amortized by more computation, and the irregularity hurts more than the memory could save for the sparse case.

Figure 3.10 illustrates the energy efficiency of pruning on different hardware. We multiply power consumption with computation time to get energy consumption, then normalize to CPU to get energy efficiency. When batch size equals to 1, the pruned network layer consumes 3× to 7× less energy over the dense network on average. Reported by `nvidia-smi`, GPU utilization is 99% for both dense and sparse cases.

Load-balance-aware pruning can preserve the prediction accuracy and achieve better speedup. To show that load-balance-aware pruning still preserves the prediction accuracy similarly to standard pruning, we experimented with the speech recognition LSTM on the TIMIT dataset [57]. As demonstrated in Figure 3.11, the accuracy margin between two pruning methods is within the variance of the pruning process itself. As we prune away more parameters, the error rate goes up. Both pruning methods can prune away 90% of the parameters before the error rate increases.

Load-balance-aware pruning can improve hardware utilization and achieve higher speedup than the basic pruning method. Figure 3.12 compares speedup with and without load-balance-aware pruning as measured on FPGA [29]. In both cases, speedup increases as more parameters are pruned away. Comparing the red and green lines, we find that load-balance-aware pruning achieved better speedup at all sparsity levels. With load-balance-aware pruning, the sparse model pruned to 10% non-zeros achieved 6.2× speedup over the dense baseline, while without load-balance-aware pruning only 5.5× speedup is achieved. The advantage of load-balance-aware pruning is that it can improve the hardware efficiency from the workload perspective without changing the hardware architecture.

3.6 Discussion

The trade-off between pruning ratio and prediction accuracy is shown in Figure 3.13. The more parameters are pruned away, the lower the prediction accuracy becomes. We experimented with L1 and L2 regularization (with and without retraining) together with iterative pruning to give five trade-off lines. Comparing the solid and the dashed lines, the importance of retraining is clear: without retraining, accuracy begins dropping much sooner — with 1/2 of the original connections, rather than with 1/10 of the original connections. It is interesting to see that we have the "free lunch" of reducing 2× the connections without losing accuracy even without retraining, while with retraining we can reduce connections by 9×.

L1 or L2 Regularization. Choosing the correct regularization affects the performance of pruning and retraining. L1 regularization penalizes non-zero parameters, resulting in more parameters near zero. L1 regularization gives better accuracy than L2 after pruning (dotted blue and purple lines) since it pushes more parameters closer to zero. However, a comparison of the yellow and green lines shows that L2 outperforms L1 after retraining since there is no benefit to further pushing values

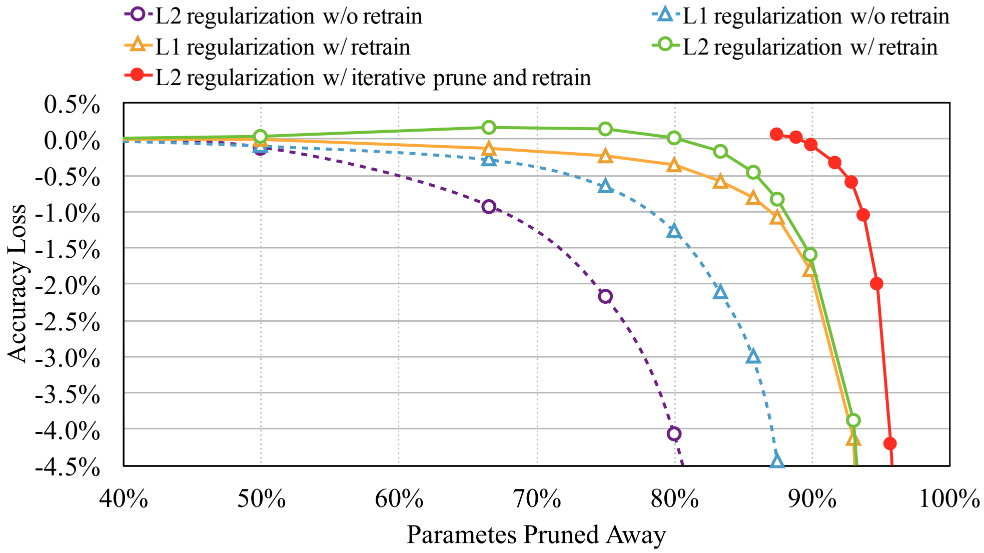


Figure 3.13: Trade-off curve for parameter reduction and loss in top-5 accuracy.

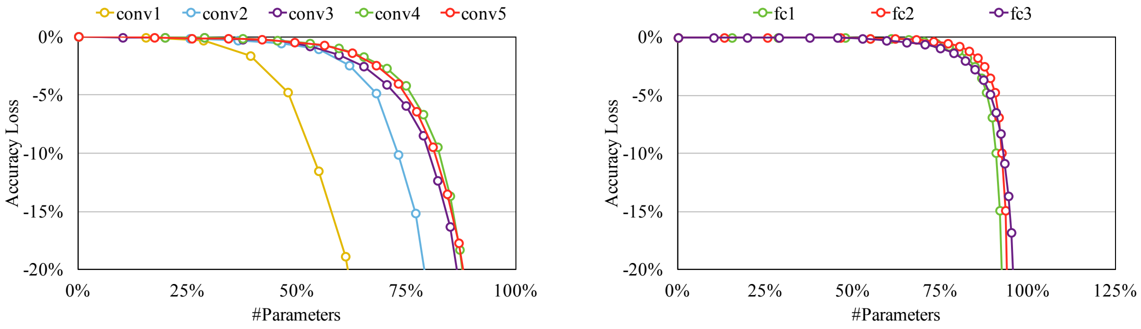


Figure 3.14: Pruning sensitivity for CONV layer (left) and FC layer (right) of AlexNet.

towards zero. One extension is to use L1 regularization for pruning and then L2 for retraining, but this did not perform better than simply using L2 for both phases. Parameters from one mode do not adapt well to the other.

The greatest gain comes from iterative pruning (solid red line with solid circles). Here we take the pruned and retrained network (solid green line with circles), then prune and retrain the network again. The leftmost dot on this curve corresponds to the point on the green line at 80% (5× pruning) pruned to 8×. There is no accuracy loss at 9×, and not until 10× does the accuracy begin to drop sharply. There are two green points that achieve slightly better accuracy than the original model. This accuracy improvement is due to pruning finding the right capacity of the network and reducing overfitting.

Sensitivity Analysis. Both convolutional and fully-connected layers can be pruned but with different sensitivity. Figure 3.14 shows the sensitivity of each layer to network pruning. The convolutional layers (on the left) are more sensitive to pruning than the fully connected layers (on the right). The first convolutional layer, which interacts with the input image directly, is most sensitive to pruning. We suspect this sensitivity is because the input layer has only three channels and extracts low-level features, thus having less redundancy than the other convolutional layers. In general, convolutional layers can be pruned around 3× and fully connected layers can be pruned more than 10×.

Since our pruning technique was published in NIPS'2015, neural network pruning has been the subject of multiple academic papers and industry R&D. These related works are described in Chapter 2.

3.7 Conclusion

We have presented a pruning method to reduce the number of connections of deep neural networks without affecting accuracy. Our pruning method, motivated in part by how learning works in the human brain, operates by learning which connections are important, pruning the unimportant connections, and then retraining the remaining sparse network. We highlight the power of our pruning method through experiments with AlexNet, VGG-16, GoogleNet, SqueezeNet, and ResNet-50 on ImageNet, which revealed that both fully connected layers and convolutional layers can be pruned: the number of connections of convolutional layers was reduced by 3×, and fully connected layers by 10× without loss of accuracy. With the NeuralTalk experiment on Flickr-8K, we find LSTMs can also be pruned by 10×. Pruning reduces the amount of computation required by deep neural networks. However, this gain is hard to exploit with modern CPU/GPU. We will design customized hardware to exploit sparsity in Chapter 6.

Chapter 4

Trained Quantization and Deep Compression

4.1 Introduction

This chapter introduces a trained quantization technique for compressing deep neural networks, which when combined with the pruning technique introduced in the previous chapter, creates "Deep Compression" [26], a model compression pipeline for deep neural networks. Deep Compression consists of pruning, trained quantization, and variable-length coding, and it can compress deep neural networks by an order of magnitude without losing the prediction accuracy. This large compression enables machine learning applications to run on mobile devices.

"Deep Compression" is a three-stage pipeline (Figure 4.1) to reduce the model size of deep neural networks in a manner that preserves the original accuracy. First, we prune the network by removing the redundant connections, keeping only the most informative connections (described in Chapter 3). Next, the weights are quantized and multiple connections share the same weight. Thus, only the codebook (effective weights) and the indices need to be stored; each parameter can be represented with much fewer bits. Finally, we apply variable-length coding (Huffman coding) to take advantage of the non-uniform distribution of effective weights and use variable length encoding to represent the weights in a lossless manner.

Our most important insight is that pruning and trained quantization can compress the network without interfering with each other, thus leading to a surprisingly high compression rate. Deep Compression makes the required storage so small (a few megabytes) that all weights can be cached on-chip instead of going to off-chip DRAM, which is slow and energy consuming. The deep compression technique is the foundation of the efficient inference engine (EIE) to be discussed in Chapter 6, which achieved significant speedup and energy efficiency improvement by taking advantage of the

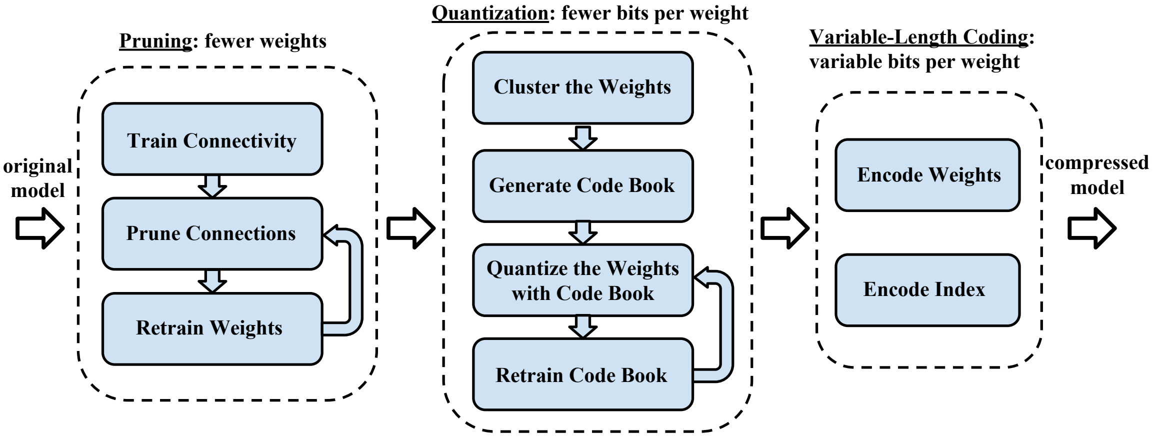


Figure 4.1: Deep Compression pipeline: pruning, quantization and variable-length coding.

compressed model.

4.2 Trained Quantization and Weight Sharing

Trained quantization and weight sharing compress the pruned network by reducing the number of bits required to represent each weight. We limit the number of effective weights we need to store by having multiple connections share the same weight and then fine-tune those shared weights.

Weight sharing is illustrated in Figure 4.2. Suppose we have a layer that has four input neurons and four output neurons, and the weight matrix is 4×4. On the top left of the figure is the 4×4 weight matrix, and on the bottom left is the 4×4 gradient matrix. The weights are quantized to 4 bins (denoted with four colors); all the weights in the same bin share the same value. Thus for each weight, we need to store only a small index into a table of shared weights. During the SGD update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from the previous iteration. For pruned AlexNet, we can quantize to 8-bits (256 shared weights) for each convolutional layers and 5-bits (32 shared weights) for each fully-connected layer without any loss of accuracy. With 4-bits for convolutional layers and 2-bits for fully-connected layers, we observed only 2% loss of accuracy.

To calculate the compression rate, given k clusters, we only need $log_2(k)$ bits to encode the index. In general, for a network with n connections and each connection represented with b bits, constraining the connections to have only k shared weights will result in a compression rate of:

$$r = \frac{nb}{nlog_2(k) + kb}. \tag{4.1}$$

For example, Figure 4.2 shows the weights of a single layer neural network with four input units and four output units. There are $4 \times 4 = 16$ weights originally, but there are only 4 shared weights:

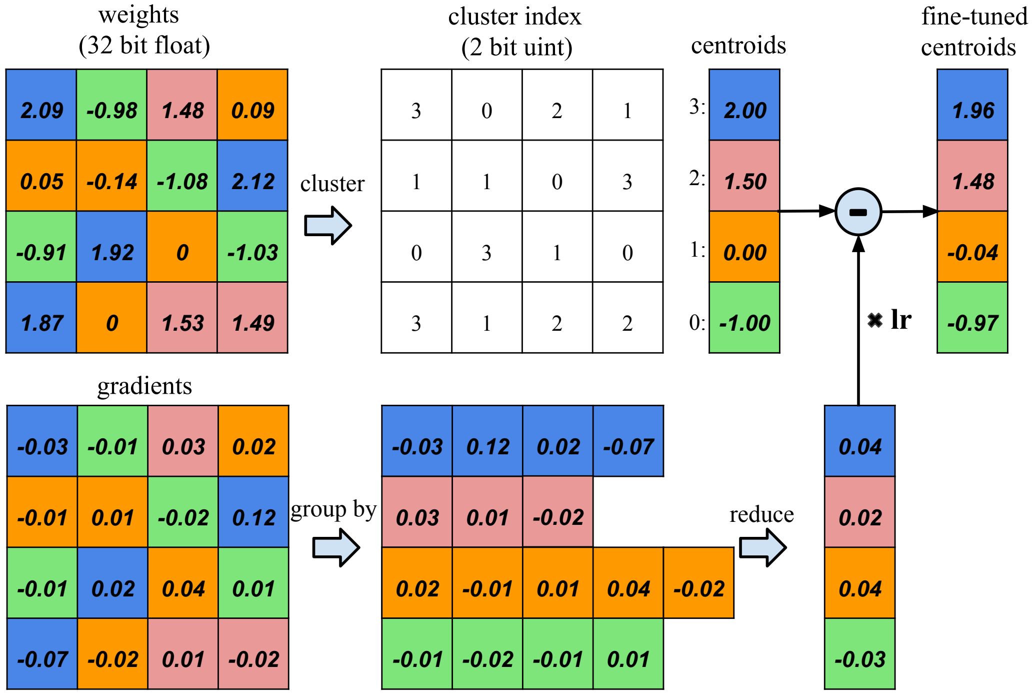


Figure 4.2: Trained quantization by weight sharing (top) and centroids fine-tuning (bottom).

similar weights are grouped to share the same value. Originally we need to store 16 weights, each with 32 bits, now we need to store only four effective weights (blue, green, red and orange), each with 32 bits, together with 16 2-bit indices giving a compression rate of $16 * 32/(4 * 32 + 2 * 16) = 3.2$

We use k-means clustering to identify the shared weights for each layer of a trained network, so that all the weights that fall into the same cluster will share the same weight. Weights are not shared across layers. We partition n original weights $W = \{w_1, w_2, ..., w_n\}$ into k clusters $C = \{c_1, c_2, ..., c_k\}$, $n \gg k$, as to minimize the within-cluster sum of squares (WCSS):

$$\underset{C}{\arg\min} \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2 . \tag{4.2}$$

Different from HashNet [72] where weight sharing is determined by a hash function before the networks see any training data, our method determines weight sharing after a network is fully trained so that the shared weights approximate the original network.

Centroid initialization impacts the quality of clustering and thus affects the network's prediction accuracy. We examine three initialization methods: Forgy(random), density-based, and linear initialization. In Figure 4.4 we plotted the original weights' distribution of conv3 layer in AlexNet (CDF in blue, PDF in red). The weights form a bimodal distribution after network pruning. On the bottom of the figure plots the effective weights (centroids) with three different initialization methods

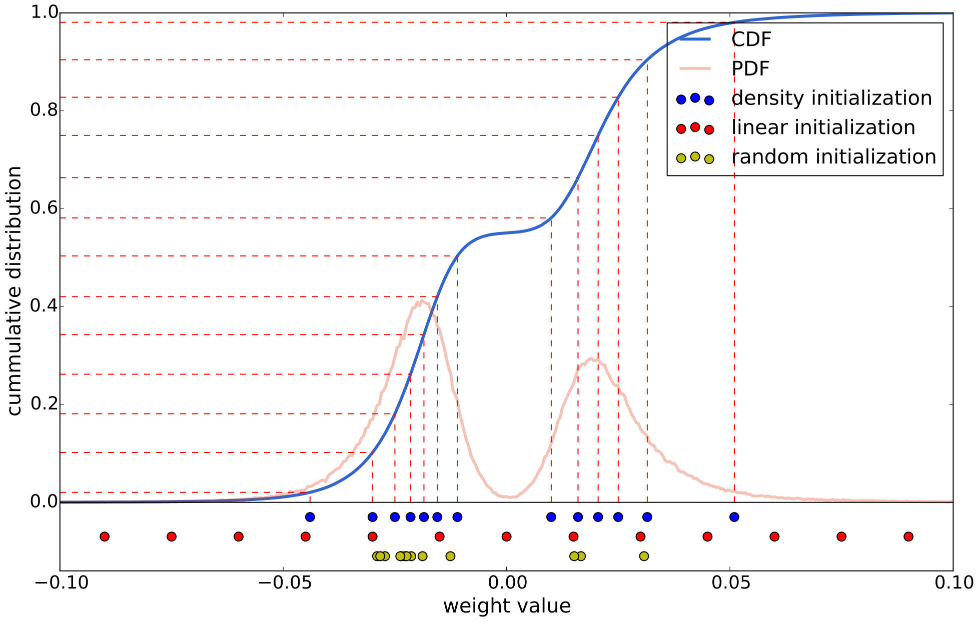


Figure 4.3: Different methods of centroid initialization: density-based, linear, and random.

(shown in blue, red and yellow). In this example, there are 13 clusters.

Forgy (random) initialization randomly chooses k observations from the data set and uses these as the initial centroids. The initialized centroids are shown in yellow. Since there are two peaks in the bimodal distribution, the Forgy method tends to concentrate around those two peaks.

Density-based initialization linearly spaces the CDF of the weights in the y-axis, then finds the horizontal intersection with the CDF, and finally finds the vertical intersection on the x-axis, which becomes a centroid (blue dots). This method makes the centroids denser around the two peaks, but more scattered than the Forgy method.

Linear initialization linearly spaces the centroids between the [min, max] of the original weights. This initialization method is invariant to the distribution of the weights and is the most scattered compared with the former two methods.

Large weights play a more important role than small weights [25], but there are fewer of these large weights. Thus, for both Forgy initialization and density-based initialization, very few centroids have large absolute values, which results in a poor representation of these few large weights. Linear initialization does not suffer from this problem. The experiment section compares the accuracy of different initialization methods after clustering and fine-tuning, showing that linear initialization works best.

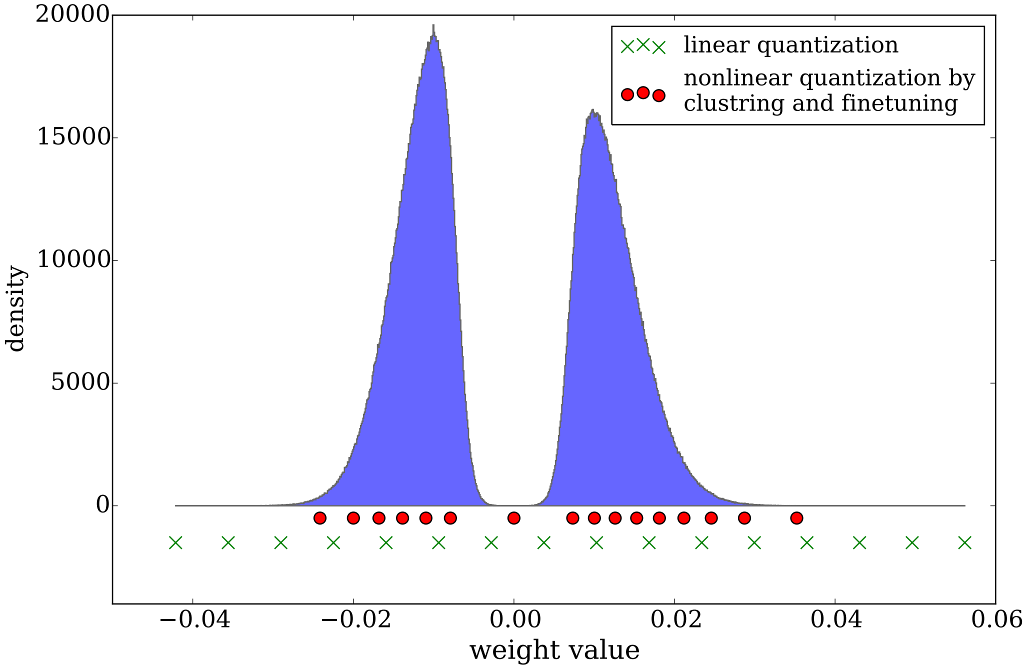


Figure 4.4: Distribution of weights and codebook before (green) and after fine-tuning (red).

During the feed forward phase and back-propagation phase of trained quantization, the weight is fetched from the lookup table. Thus, there is one level of indirection. An index of the shared weight table is stored for each connection. During back-propagation, the gradient for each shared weight is calculated and used to update the shared weight. This procedure is shown in Figure 4.2.

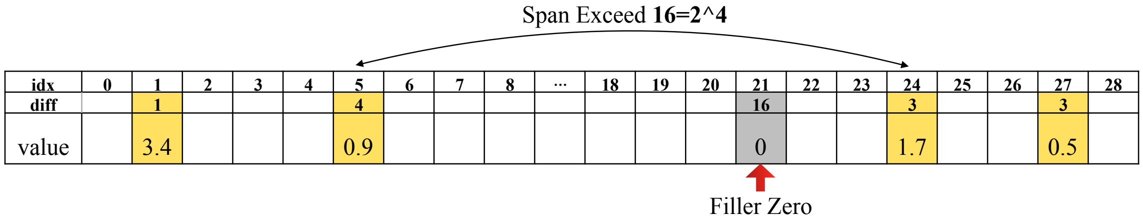
We denote the loss by $\mathcal{L}$, the weight in the ith column and jth row by W_{ij}, the centroid index of element W_{ij} by I_{ij}, the kth centroid of the layer by C_k. By using the indicator function $\mathbb{1}(.)$, the gradient of the centroids is calculated as:

$$\frac{\partial\mathcal{L}}{\partial C_k} = \sum_{i,j} \frac{\partial\mathcal{L}}{\partial W_{ij}} \frac{\partial W_{ij}}{\partial C_k} = \sum_{i,j} \frac{\partial\mathcal{L}}{\partial W_{ij}} \mathbb{1}(I_{ij} = k) \tag{4.3}$$

4.3 Storing the Meta Data

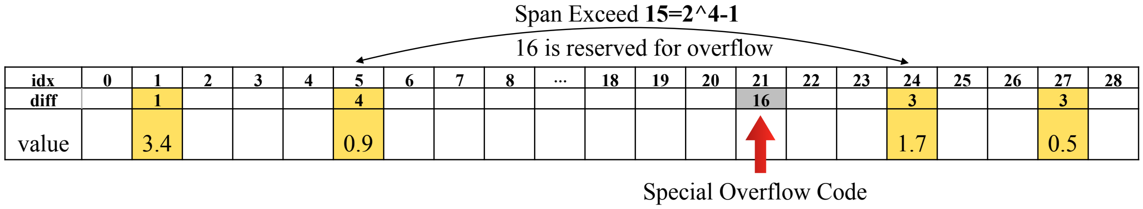
Both pruning and trained quantization produce meta data. Pruning makes the weight matrix sparse, and thus extra space is needed to store the indices of non-zero weights. Quantization needs extra storage for a codebook. Each layer has its code book. Here we quantitatively analyze the overhead of these meta data.

We store the sparse structure that results from pruning with weight itself and an index. We minimize the number of bits used to represent the index by storing the relative index (the difference)



idx	0	1	2	3	4	5	6	7	8	···	18	19	20	21	22	23	24	25	26	27	28
diff		1				4								16			3			3	
value		3.4				0.9								0			1.7			0.5	

Figure 4.5: Pad a filler zero to handle overflow when representing a sparse vector with relative index.



idx	0	1	2	3	4	5	6	7	8	···	18	19	20	21	22	23	24	25	26	27	28
diff		1				4								16			3			3	
value		3.4				0.9											1.7			0.5	

Figure 4.6: Reserve a special code to indicate overflow when representing a sparse vector with relative index.

instead of the absolute index, so that we can encode this difference in fewer bits. In practice, four bits are sufficient, which can represent a maximum index difference of 16. Considering a common sparsity of 10%, on average every ten weights has a non-zero weight, so 16 is a good upper bound. When we need an index difference larger than the bound, there are two methods to handle overflow:

1. The zero padding solution (Figure 4.5): when the index difference exceeds 16, the largest 4-bit (as an example) unsigned number, we add a filler zero. The advantage of this solution is that there is no waste of the relative index, and no special hardware logic is needed to handle the special ticker code. The disadvantage is that filler zero will result in wasted computation cycles.

2. The special code solution (Figure 4.6): we reserve the last code (16 in this case) as a ticker to represent overflow. The next relative index will be based on this ticker. In this scenario, the maximum relative index is only 15, which is the disadvantage of this method: one special code is wasted, and special logic is needed to deal with the special code, making the computation harder to parallelize. The advantage of this method is the opposite as the first method: there are no wasted computation cycles.

Both methods have pros and cons. In practice, we found the overflow is rare (given 10% sparsity, on average every ten entries has a non-zero, and 16 is a good upper bound for relative index), so the wasted cycles are negligible. We adopted solution 1 in our compression algorithm.

The codebook is the meta data required by trained quantization. We quantized each layer separately, so each layer needs its codebook. However, this extra storage is very small. When using

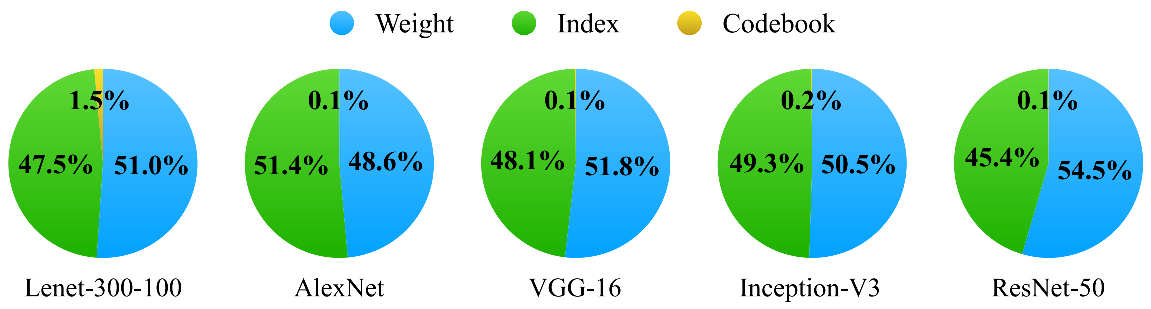


Figure 4.7: Storage ratio of weight, index, and codebook.

4-bit quantization, the codebook has only 16 entries. Say we have 50 layers in total, we need to store only 800 entries; each entry is 4 Bytes with a total of 3KB.

Figure 4.7 shows the breakdown of the three different components when compressing LeNet-300-100, AlexNet, VGG-16, Inception-V3 and ResNet-50. The green part is the useful data (weights), and the blue and yellow are meta-data (index and codebook). The total proportion of the meta-data is roughly half.

4.4 Variable-Length Coding

So far we have visited the first two stages of Deep Compression: pruning for *fewer weights*, trained quantization for *fewer bits per weight*. Up to this stage, all the weights are represented with a fixed number of bit width. However, the weight distribution is non-uniform (Figure 4.8). We can use variable-length coding to further compress the model. The idea is: we can use *fewer* bits to represent those *more* frequently appearing weights, and use *more* bits to represent those *less* frequently appearing weights.

Huffman coding [116], Lempel–Ziv coding [117] and arithmetic coding [118] are all variable-length coding strategies. Without loss of generality, we pick Huffman coding for Deep Compression.

A Huffman code is an optimal prefix code commonly used for lossless data compression. It uses variable-length codewords to encode source symbols. The table is derived from the occurrence probability for each symbol. More common symbols are represented with fewer bits. Variable-length coding is lossless, so we don't have to worry about retraining or loss of accuracy.

Figure 4.8 shows the probability distribution of quantized weights and the sparse matrix index of the last fully-connected layer in AlexNet. Both distributions are not uniformly distributed: most of the quantized weights are distributed around the two peaks; the sparse matrix index has a single peak near zero. Experiments show that Huffman coding these non-uniformly distributed values can save 20%-50% of model storage. For example, before Huffman coding, the compression ratio for ResNet-50 is 13×, after Huffman coding, the compression ratio is 17×.

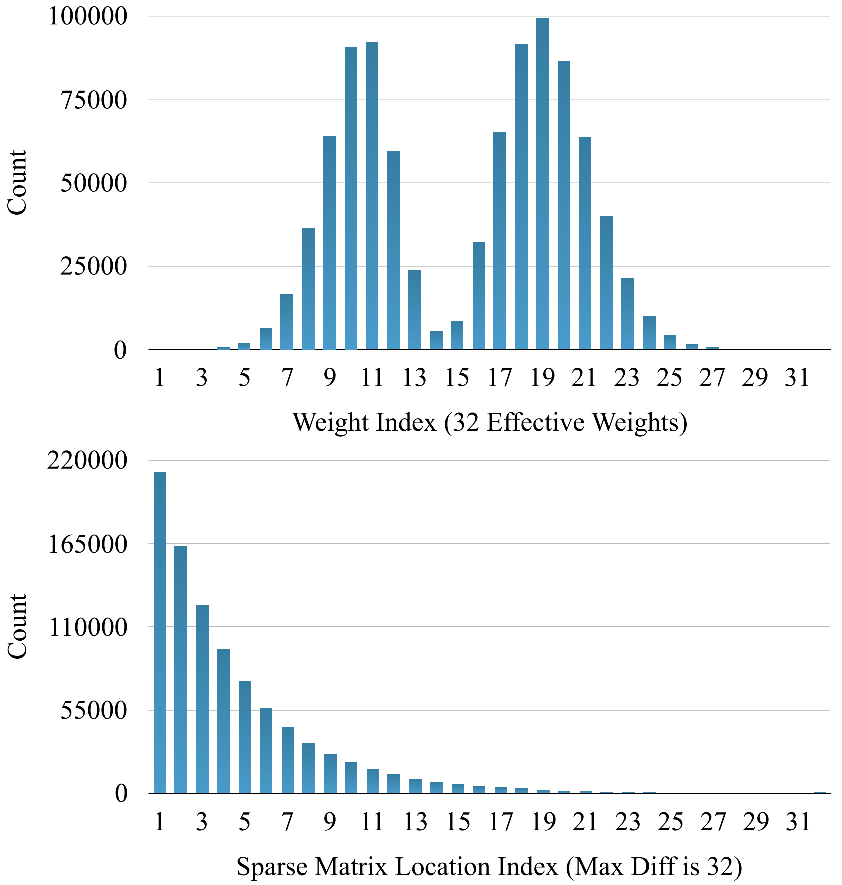


Figure 4.8: The non-uniform distribution for weight (Top) and index (Bottom) gives opportunity for variable-length coding.

4.5 Experiments

We pruned, quantized, and Huffman encoded five networks: LeNet-300-100, LeNet-5, AlexNet, VGG-16, Inception-V3 and ResNet-50. The network parameters and accuracy before and after pruning are shown in Table 4.1. The compression pipeline saves network storage by 17× to 49× across different networks without loss of accuracy. The total size of ResNet-50 decreased from 100MB to 5.8MB, which is small enough to be put into on-chip SRAM, eliminating the need to store the model in energy-consuming DRAM memory.

Deep Compression is implemented with both the Caffe framework [112] and the Pytorch framework [59]. Trained quantization and weight sharing are implemented by maintaining a codebook structure that stores the shared weight, and group-by-index after calculating the gradient of each layer. Each shared weight is updated with all the gradients that fall into that bucket. Huffman coding does not require training and is implemented off-line after all the fine-tuning is finished.

Table 4.1: Deep Compression saves 17× to 49× parameter storage with no loss of accuracy.

Network	Top-1 Error	Top-5 Error	Model Size	Compress Rate
LeNet-300-100	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40**×
LeNet-5	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39**×
AlexNet	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35**×
VGG-16	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49**×
Inception-V3	22.55%	6.44%	91 MB	
Inception-V3 Compressed	22.34%	6.33%	**4.2 MB**	**22**×
ResNet-50	23.85%	7.13%	97 MB	
ResNet-50 Compressed	23.85%	6.96%	**5.8 MB**	**17**×

Table 4.2: Compression statistics for LeNet-300-100. P: pruning, Q: quantization, H: Huffman coding.

Layer	Weight	Weight Density (P)	Weight Bits (P+Q)	Weight Bits (P+Q+H)	Index Bits (P+Q)	Index Bits (P+Q+H)	Compress Rate (P+Q)	Compress Rate (P+Q+H)
ip1	235K	8%	6	4.4	5	3.7	3.1%	2.32%
ip2	30K	9%	6	4.4	5	4.3	3.8%	3.04%
ip3	1K	26%	6	4.3	5	3.2	15.7%	12.70%
Total	266K	8%	6	5.1	5	3.7	3.1% (**32**×)	2.49% (**40**×)

Table 4.3: Compression statistics for LeNet-5. P: pruning, Q: quantization, H: Huffman coding.

Layer	Weight	Weight Density (P)	Weight Bits (P+Q)	Weight Bits (P+Q+H)	Index Bits (P+Q)	Index Bits (P+Q+H)	Compress Rate (P+Q)	Compress Rate (P+Q+H)
conv1	0.5K	66%	8	7.2	5	1.5	78.5%	67.45%
conv2	25K	12%	8	7.2	5	3.9	6.0%	5.28%
ip1	400K	8%	5	4.5	5	4.5	2.7%	2.45%
ip2	5K	19%	5	5.2	5	3.7	6.9%	6.13%
Total	431K	8%	5.3	4.1	5	4.4	3.1% (**32**×)	2.55% (**39**×)

LeNet. We first experimented on the MNIST dataset with LeNet-300-100 and LeNet-5 networks [1]. LeNet-300-100 is a fully-connected network with two hidden layers, with 300 and 100 neurons each, which achieves a 1.6% error rate on MNIST. LeNet-5 is a convolutional network that has two convolutional layers and two fully-connected layers, which achieves 0.8% error rate on the MNIST dataset.

Table 4.2 and Table 4.3 show the statistics of the compression pipeline on LeNet. The compression rate includes the overhead of the codebook and sparse indexes. Weights can be pruned to 8% non-zero,

Table 4.4: Accuracy of AlexNet with different quantization bits.

#CONV bits / #FC bits	Top-1 Error	Top-5 Error	Top-1 Error Increase	Top-5 Error Increase
32bits / 32bits	42.78%	19.73%	-	-
8 bits / 5 bits	42.78%	19.70%	0.00%	-0.03%
8 bits / 4 bits	42.79%	19.73%	0.01%	0.00%
4 bits / 2 bits	44.77%	22.33%	1.99%	2.60%

Table 4.5: Compression statistics for AlexNet. P: pruning, Q: quantization, H: Huffman coding.

Layer	Weight	Weight Density (P)	Weight Bits (P+Q)	Weight Bits (P+Q+H)	Index Bits (P+Q)	Index Bits (P+Q+H)	Compress Rate (P+Q)	Compress Rate (P+Q+H)
conv1	35K	84%	8	6.3	4	1.2	32.6%	20.53%
conv2	307K	38%	8	5.5	4	2.3	14.5%	9.43%
conv3	885K	35%	8	5.1	4	2.6	13.1%	8.44%
conv4	663K	37%	8	5.2	4	2.5	14.1%	9.11%
conv5	442K	37%	8	5.6	4	2.5	14.0%	9.43%
fc6	38M	9%	5	3.9	4	3.2	3.0%	2.39%
fc7	17M	9%	5	3.6	4	3.7	3.0%	2.46%
fc8	4M	25%	5	4	4	3.2	7.3%	5.85%
Total	61M	11%	5.4	4	4	3.2	3.7% (**27**×)	2.88% (**35**×)

and quantized to 5-8 bits. Most of the saving comes from pruning and quantization (compressed 32×), while Huffman coding gives a marginal gain (compressed an additional 1.25×).

AlexNet. We then applied Deep Compression on the ImageNet ILSVRC-2012 dataset. We use the AlexNet [2] Caffe model as the reference model, which has 61 million parameters and achieved a top-1 accuracy of 57.2% and a top-5 accuracy of 80.3%. Table 4.5 shows that AlexNet can be compressed to 2.88% of its original size without impacting accuracy. Pruning reduced the number of parameters to 11%. After trained quantization, there are 256 shared weights in each convolutional layer, which are encoded with 8 bits, and 32 shared weights in each fully-connected layer, which are encoded with only 5 bits. The relative sparse index is encoded with 4 bits. Huffman coding compressed additional 22%, resulting in 35× compression in total.

We experimented with the trade-off between compression ratio and prediction accuracy (Table 4.4). For convolution and fully-connected layer, 8/5 bit quantization has no loss of accuracy; 8/4 bit quantization, which is more hardware friendly to encode in byte aligned fashion, has a negligible loss of accuracy of 0.01%; to achieve more aggressive compression, 4/2 bit quantization resulted in 1.99% and 2.60% loss of Top-1 and Top-5 accuracy.

VGG-16. With promising results on AlexNet, we also looked at a larger, more recent network, VGG-16 [3], on the same ILSVRC-2012 dataset. VGG-16 has far more convolutional layers but still only three fully-connected layers. Following a similar methodology, we aggressively compressed both convolutional and fully-connected layers to realize a significant reduction in the number of effective

Table 4.6: Compression statistics for VGG-16. P: pruning, Q: quantization, H: Huffman coding.

Layer	Weight	Weight Density (P)	Weight Bits (P+Q)	Weight Bits (P+Q+H)	Index Bits (P+Q)	Index Bits (P+Q+H)	Compress Rate (P+Q)	Compress Rate (P+Q+H)
conv1_1	2K	58%	8	6.8	5	1.7	40.0%	29.97%
conv1_2	37K	22%	8	6.5	5	2.6	9.8%	6.99%
conv2_1	74K	34%	8	5.6	5	2.4	14.3%	8.91%
conv2_2	148K	36%	8	5.9	5	2.3	14.7%	9.31%
conv3_1	295K	53%	8	4.8	5	1.8	21.7%	11.15%
conv3_2	590K	24%	8	4.6	5	2.9	9.7%	5.67%
conv3_3	590K	42%	8	4.6	5	2.2	17.0%	8.96%
conv4_1	1M	32%	8	4.6	5	2.6	13.1%	7.29%
conv4_2	2M	27%	8	4.2	5	2.9	10.9%	5.93%
conv4_3	2M	34%	8	4.4	5	2.5	14.0%	7.47%
conv5_1	2M	35%	8	4.7	5	2.5	14.3%	8.00%
conv5_2	2M	29%	8	4.6	5	2.7	11.7%	6.52%
conv5_3	2M	36%	8	4.6	5	2.3	14.8%	7.79%
fc6	103M	4%	5	3.6	5	3.5	1.6%	1.10%
fc7	17M	4%	5	4	5	4.3	1.5%	1.25%
fc8	4M	23%	5	4	5	3.4	7.1%	5.24%
Total	138M	7.5%	6.4	4.1	5	3.1	3.2% (**31**×)	2.05% (**49**×)

weights, shown in Table4.6.

The VGG-16 network as a whole has been compressed by 49×. Pruning reduced the number of parameters to 7.5%. After trained quantization, weights in the convolutional layers are represented with 8 bits, and fully-connected layers use 5 bits, which does not affect accuracy. The two largest fully-connected layers can each be pruned to less than 1.6% of their original size. This reduction is critical for real time image processing, where the batch size is one and unlike batch processing, there is fewer weight reuse. The reduced layers will fit in an on-chip SRAM and have modest bandwidth requirements. Without the reduction, the memory bandwidth requirements are prohibitive.

We next applied Deep Compression on fully-convolutional neural networks, which is much more parameter-efficient. We picked two representative fully-convolutional neural networks: Inception-V3 and ResNet-50.

Inception-V3. The compression result of Inception-V3 [119] is shown in Table 4.7. Inception-V3 has been compressed to 4.6% of its original size, from 91MB to 4.2MB, which can very easily fit in SRAM cache. The 7x7, 7x1 and 1x7 kernels can be pruned to 10%-20% non-zero, the 3x3, 3x1 and 1x3 kernels can be pruned to 20%-30% non-zero, and the 1x1 kernels can be pruned to 20%-60% non-zero. All the layers are quantized to only 4 bits, except for the first few layers that extract low-level features and the first layer of each inception block. Huffman coding further decreased the representation of weight from 4 bits to 3.8 bits, and that of the index from 4 bits to 3.1 bits, pushing the compression ratio from 18× to 22×.

Table 4.7: Compression statistics for Inception-V3. P: pruning, Q: quantization, H: Huffman coding.

		Weight	Weight	Weight	Index	Index	Compress	Compress
Layer	Weight	Density	Bits	Bits	Bits	Bits	Rate	Rate
		(P)	(P+Q)	(P+Q+H)	(P+Q)	(P+Q+H)	(P+Q)	(P+Q+H)
Conv2d_1a_3x3	864	40%	6	5.7	4	2.2	21.5%	19.3%
Conv2d_2a_3x3	9K	59%	6	5.4	4	1.6	19.5%	13.8%
Conv2d_2b_3x3	18K	50%	6	5.5	4	1.9	16.0%	11.9%
Conv2d_3b_1x1	5K	60%	6	5.4	4	1.7	20.2%	14.8%
Conv2d_4a_3x3	138K	50%	6	5.3	4	2.0	15.6%	11.4%
5b.branch1x1	12K	60%	6	5.4	4	1.7	19.3%	14.0%
5b.branch5x5_1	9K	29%	4	3.4	4	3.0	7.6%	6.1%
5b.branch5x5_2	77K	19%	4	3.1	4	3.2	5.3%	4.0%
5b.branch3x3dbl_1	12K	30%	4	3.4	4	2.9	7.6%	6.1%
5b.branch3x3dbl_2	55K	29%	4	3.4	4	2.9	7.5%	5.9%
5b.branch3x3dbl_3	83K	20%	4	3.4	4	3.3	5.3%	4.3%
5b.branch_pool	6K	40%	6	5.4	4	2.5	13.7%	11.1%
5c.branch1x1	16K	50%	6	5.4	4	2.0	16.0%	12.0%
5c.branch5x5_1	12K	10%	4	3.5	4	3.6	3.3%	2.7%
5c.branch5x5_2	77K	19%	4	3.4	4	3.3	5.2%	4.3%
5c.branch3x3dbl_1	16K	20%	4	3.6	4	3.4	5.4%	4.6%
5c.branch3x3dbl_2	55K	20%	4	3.5	4	3.4	5.3%	4.5%
5c.branch3x3dbl_3	83K	20%	4	3.6	4	3.4	5.2%	4.5%
5c.branch_pool	16K	29%	4	3.4	4	2.9	7.6%	6.0%
5d.branch1x1	18K	40%	6	5.4	4	2.5	12.9%	10.3%
5d.branch5x5_1	14K	20%	4	3.6	4	3.3	5.5%	4.7%
5d.branch5x5_2	77K	20%	4	3.6	4	3.3	5.3%	4.4%
5d.branch3x3dbl_1	18K	10%	4	3.4	4	3.5	3.3%	2.6%
5d.branch3x3dbl_2	55K	29%	4	3.4	4	2.9	7.5%	5.8%
5d.branch3x3dbl_3	83K	20%	4	3.6	4	3.3	5.3%	4.5%
5d.branch_pool	18K	29%	4	3.4	4	2.9	7.6%	6.0%
6a.branch3x3	995K	20%	4	3.6	4	3.2	5.3%	4.3%
6a.branch3x3dbl_1	18K	10%	4	3.6	4	3.6	3.3%	2.7%
6a.branch3x3dbl_2	55K	20%	4	3.6	4	3.4	5.2%	4.5%
6a.branch3x3dbl_3	83K	10%	4	3.7	4	3.2	3.4%	2.6%
6b.branch1x1	147K	49%	6	5.3	4	2.0	15.5%	11.3%
6b.branch7x7_1	98K	20%	4	3.5	4	3.5	5.2%	4.5%
6b.branch7x7_2	115K	10%	4	3.5	4	3.5	3.2%	2.5%
6b.branch7x7_3	172K	20%	4	3.6	4	3.4	5.2%	4.4%
6b.branch7x7dbl_1	98K	20%	4	3.4	4	3.5	5.1%	4.3%
6b.branch7x7dbl_2	115K	10%	4	3.5	4	3.5	3.2%	2.6%
6b.branch7x7dbl_3	115K	19%	4	3.5	4	3.2	5.3%	4.3%
6b.branch7x7dbl_4	115K	10%	4	3.6	4	3.4	3.2%	2.5%
6b.branch7x7dbl_5	172K	10%	4	3.5	4	3.5	3.2%	2.5%
6b.branch_pool	147K	40%	6	5.5	4	2.5	12.5%	9.9%
6c.branch1x1	147K	20%	4	3.5	4	3.4	5.2%	4.4%
6c.branch7x7_1	123K	10%	4	3.4	4	3.7	3.1%	2.6%

6c.branch7x7_2	179K	10%	4	3.5	4	3.5	3.2%	2.5%
6c.branch7x7_3	215K	10%	4	3.6	4	3.5	3.2%	2.6%
6c.branch7x7dbl_1	123K	10%	4	3.3	4	3.7	3.1%	2.5%
6c.branch7x7dbl_2	179K	10%	4	3.6	4	3.5	3.2%	2.5%
6c.branch7x7dbl_3	179K	10%	4	3.6	4	3.4	3.2%	2.5%
6c.branch7x7dbl_4	179K	10%	4	3.5	4	3.4	3.2%	2.5%
6c.branch7x7dbl_5	215K	10%	4	3.5	4	3.4	3.2%	2.5%
6c.branch_pool	147K	30%	4	3.6	4	3.0	7.4%	6.1%
6d.branch1x1	147K	29%	4	3.3	4	3.0	7.4%	5.8%
6d.branch7x7_1	123K	20%	4	3.5	4	3.5	5.2%	4.4%
6d.branch7x7_2	179K	10%	4	3.6	4	3.5	3.2%	2.5%
6d.branch7x7_3	215K	10%	4	3.6	4	3.6	3.2%	2.6%
6d.branch7x7dbl_1	123K	20%	4	3.4	4	3.5	5.2%	4.4%
6d.branch7x7dbl_2	179K	10%	4	3.5	4	3.6	3.2%	2.5%
6d.branch7x7dbl_3	179K	20%	4	3.5	4	3.4	5.2%	4.4%
6d.branch7x7dbl_4	179K	20%	4	3.4	4	3.4	5.2%	4.3%
6d.branch7x7dbl_5	215K	20%	4	3.5	4	3.4	5.2%	4.4%
6d.branch_pool	147K	29%	4	3.5	4	3.0	7.5%	6.0%
6e.branch1x1	147K	20%	4	3.4	4	3.5	5.2%	4.4%
6e.branch7x7_1	147K	20%	4	3.5	4	3.5	5.2%	4.5%
6e.branch7x7_2	258K	10%	4	3.5	4	3.4	3.2%	2.5%
6e.branch7x7_3	258K	20%	4	3.4	4	3.3	5.2%	4.3%
6e.branch7x7dbl_1	147K	20%	4	3.4	4	3.5	5.1%	4.4%
6e.branch7x7dbl_2	258K	20%	4	3.7	4	3.4	5.2%	4.5%
6e.branch7x7dbl_3	258K	20%	4	3.6	4	3.3	5.3%	4.4%
6e.branch7x7dbl_4	258K	20%	4	3.7	4	3.4	5.2%	4.5%
6e.branch7x7dbl_5	258K	30%	4	3.6	4	2.8	7.6%	6.0%
6e.branch_pool	147K	30%	4	3.5	4	3.0	7.5%	6.0%
7a.branch3x3_1	147K	20%	4	3.6	4	3.5	5.2%	4.5%
7a.branch3x3_2	553K	19%	4	3.7	4	3.3	5.3%	4.4%
7a.branch7x7x3_1	147K	20%	4	3.5	4	3.5	5.1%	4.4%
7a.branch7x7x3_2	258K	10%	4	3.6	4	3.5	3.2%	2.5%
7a.branch7x7x3_3	258K	10%	4	3.6	4	3.4	3.2%	2.5%
7a.branch7x7x3_4	332K	10%	4	3.4	4	3.4	3.2%	2.5%
7b.branch1x1	410K	20%	4	3.5	4	3.5	5.1%	4.4%
7b.branch3x3_1	492K	29%	4	3.6	4	3.0	7.4%	6.1%
7b.branch3x3_2a	442K	20%	4	3.7	4	3.2	5.3%	4.4%
7b.branch3x3_2b	442K	20%	4	3.7	4	3.4	5.2%	4.5%
7b.branch3x3dbl_1	573K	29%	4	3.6	4	3.0	7.4%	6.1%
7b.branch3x3dbl_2	1548K	29%	4	3.7	4	2.7	7.5%	5.9%
7b.branch3x3dbl_3a	442K	30%	4	3.4	4	2.6	7.5%	5.6%
7b.branch3x3dbl_3b	442K	30%	4	3.6	4	2.9	7.4%	6.0%
7b.branch_pool	246K	20%	4	3.5	4	3.5	5.2%	4.4%
7c.branch1x1	655K	10%	4	3.6	4	3.7	3.0%	2.5%
7c.branch3x3_1	786K	30%	4	3.6	4	3.0	7.4%	6.1%
7c.branch3x3_2a	442K	10%	4	3.7	4	2.8	3.4%	2.4%

7c.branch3x3_2b	442K	10%	4	3.7	4	2.9	3.4%	2.4%
7c.branch3x3dbl_1	918K	29%	4	3.5	4	3.0	7.4%	5.9%
7c.branch3x3dbl_2	1548K	20%	4	3.4	4	2.8	5.5%	4.1%
7c.branch3x3dbl_3a	442K	20%	4	3.7	4	2.9	5.4%	4.2%
7c.branch3x3dbl_3b	442K	20%	4	3.7	4	3.0	5.3%	4.4%
7c.branch_pool	393K	10%	4	3.7	4	3.7	3.1%	2.6%
fc	2048K	20%	4	3.8	4	3.4	5.2%	4.6%
Total	24M	20%	4.1	3.7	4	3.1	5.7% (**18**×)	4.6% (**22**×)

ResNet-50. ResNet-50 [5] is a modern architecture that has residual blocks and only one thin fully-connected layer. The ResNet-50 network as a whole has been compressed to 5.95% of its original size, from 100MB to 5.8MB. The layer-wise breakdown is shown in Table 4.8. Weights in both the convolutional layers and the fully-connected layers can be pruned to 30% non-zero, and quantized to only 4 bits, except for the first few layers that extract low-level features. We have also experimented quantizing *all* the layers of the pruned ResNet-50 to 4-bits. We achieved [Top1, Top5] accuracy of [75.91%, 92.84%]. This is only [0.24%, 0.03%] different from the baseline accuracy, which is [76.15%, 92.87%]. In practice all layers Huffman coding further decreased the representation of weight from 4 bits to 3.5 bits, and that of the index from 4 bits to 2.8 bits, pushing the compression ratio from 13× to 17×.

Table 4.8: Compression statistics for ResNet-50. P: pruning, Q: quantization, H: Huffman coding.

Layer	Weight	Weight Density (P)	Weight Bits (P+Q)	Weight Bits (P+Q+H)	Index Bits (P+Q)	Index Bits (P+Q+H)	Compress Rate (P+Q)	Compress Rate (P+Q+H)
conv1	9K	50%	6	5.5	4	1.7	15.7%	11.24%
layer1.0.conv1	4K	40%	6	5.4	4	2.3	12.6%	9.65%
layer1.0.conv2	37K	30%	4	3.2	4	2.6	7.5%	5.47%
layer1.0.conv3	16K	30%	4	3.3	4	2.6	7.6%	5.63%
layer1.0.shortcut	16K	40%	6	5.2	4	2.3	12.6%	9.36%
layer1.1.conv1	16K	30%	4	3.5	4	2.9	7.5%	5.94%
layer1.1.conv2	37K	30%	4	3.3	4	2.8	7.5%	5.73%
layer1.1.conv3	16K	30%	4	3.4	4	2.5	7.7%	5.67%
layer1.2.conv1	16K	30%	4	3.6	4	2.9	7.5%	6.04%
layer1.2.conv2	37K	30%	4	3.5	4	2.9	7.5%	5.94%
layer1.2.conv3	16K	30%	4	3.5	4	2.3	7.7%	5.54%
layer2.0.conv1	33K	40%	6	5.4	4	2.4	12.5%	9.82%
layer2.0.conv2	147K	33%	6	5.4	4	2.6	10.5%	8.43%
layer2.0.conv3	66K	30%	4	3.3	4	2.5	7.6%	5.59%
layer2.0.shortcut	131K	30%	4	3.1	4	2.8	7.5%	5.53%
layer2.1.conv1	66K	30%	4	3.3	4	2.9	7.4%	5.80%
layer2.1.conv2	147K	30%	4	3.3	4	2.7	7.5%	5.60%
layer2.1.conv3	66K	30%	4	3.4	4	2.5	7.5%	5.52%
layer2.2.conv1	66K	30%	4	3.4	4	2.9	7.4%	5.93%

layer2.2.conv2	147K	30%	4	3.4	4	2.8	7.5%	5.86%
layer2.2.conv3	66K	30%	4	3.4	4	2.9	7.5%	5.87%
layer2.3.conv1	66K	30%	4	3.5	4	2.9	7.4%	5.98%
layer2.3.conv2	147K	30%	4	3.5	4	2.9	7.5%	5.99%
layer2.3.conv3	66K	30%	4	3.5	4	2.7	7.6%	5.83%
layer3.0.conv1	131K	40%	6	5.3	4	2.4	12.5%	9.70%
layer3.0.conv2	590K	30%	4	3.4	4	2.7	7.5%	5.73%
layer3.0.conv3	262K	30%	4	3.4	4	2.8	7.5%	5.88%
layer3.0.shortcut	524K	30%	4	3.3	4	2.9	7.5%	5.72%
layer3.1.conv1	262K	30%	4	3.3	4	3.0	7.4%	5.82%
layer3.1.conv2	590K	30%	4	3.4	4	2.8	7.5%	5.81%
layer3.1.conv3	262K	30%	4	3.3	4	2.9	7.5%	5.81%
layer3.2.conv1	262K	30%	4	3.4	4	3.0	7.4%	5.89%
layer3.2.conv2	590K	30%	4	3.4	4	2.9	7.5%	5.90%
layer3.2.conv3	262K	30%	4	3.4	4	2.9	7.5%	5.86%
layer3.3.conv1	262K	30%	4	3.4	4	3.0	7.4%	5.88%
layer3.3.conv2	590K	30%	4	3.5	4	2.9	7.5%	5.92%
layer3.3.conv3	262K	30%	4	3.4	4	2.9	7.5%	5.83%
layer3.4.conv1	262K	30%	4	3.4	4	2.9	7.4%	5.86%
layer3.4.conv2	590K	30%	4	3.5	4	2.9	7.5%	5.97%
layer3.4.conv3	262K	30%	4	3.4	4	2.9	7.5%	5.84%
layer3.5.conv1	262K	30%	4	3.3	4	2.9	7.4%	5.84%
layer3.5.conv2	590K	30%	4	3.5	4	2.8	7.5%	5.89%
layer3.5.conv3	262K	30%	4	3.4	4	2.9	7.5%	5.87%
layer4.0.conv1	524K	30%	4	3.5	4	2.9	7.5%	6.00%
layer4.0.conv2	2M	30%	4	3.4	4	2.6	7.6%	5.65%
layer4.0.conv3	1M	30%	4	3.5	4	2.9	7.5%	6.00%
layer4.0.shortcut	2M	30%	4	3.3	4	2.9	7.4%	5.83%
layer4.1.conv1	1M	30%	4	3.4	4	2.9	7.5%	5.95%
layer4.1.conv2	2M	30%	4	3.5	4	2.7	7.6%	5.80%
layer4.1.conv3	1M	30%	4	3.5	4	2.9	7.5%	6.01%
layer4.2.conv1	1M	30%	4	3.5	4	2.9	7.5%	5.99%
layer4.2.conv2	2M	30%	4	3.6	4	2.2	7.7%	5.60%
layer4.2.conv3	1M	30%	4	3.5	4	2.9	7.5%	6.00%
fc	2M	20%	4	3.3	4	3.5	5.1%	4.31%
Total	25.5M	29%	4.0	3.5	4	2.8	7.6% (**13**×)	5.95% (**17**×)

4.6 Discussion

Figure 4.9 compares the accuracy at different compression rates for pruning and quantization working together or individually. When working individually (purple and yellow lines), the accuracy of a pruned network begins to drop significantly when compressed below 8% of its original size; the accuracy of the quantized network also begins to drop significantly when compressed below 8% of its

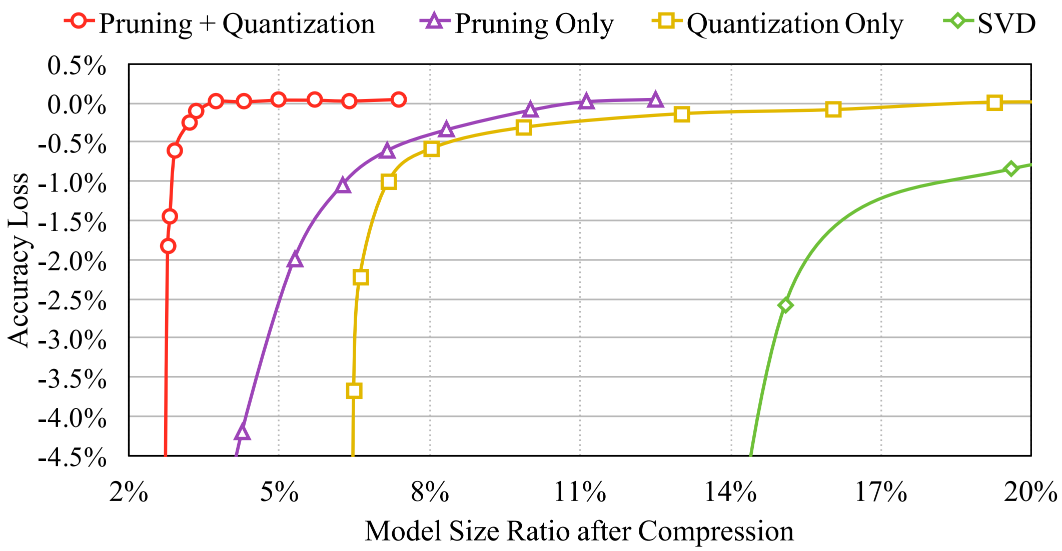


Figure 4.9: Accuracy vs. compression rates under different compression methods. Pruning and quantization works best when combined.

original size. This means pruning and quantization can independently reduce the model size, but the compression ratio is not significant. However, when combining pruning and quantization, (red line), the network can be compressed to 3% of its original size with no loss of accuracy, a very significant compression ratio when combining these techniques. Quantization works well on top of a pruned network because the unpruned model has 100% weights to be quantized, while the pruned model has only 10% to 30% of the weights that need to be quantized (others are all zero). Given the same number of centroids, the latter has less quantization error. On the far right side of Figure 4.9, we compared the compression result of SVD, which is inexpensive but has a poor compression rate.

Retraining is important for trained quantization. The solid lines and dashed lines in Figure 4.10 compares the top-1 accuracy of quantized ResNet-50 with and without retraining. For both uniform and non-uniform case, retraining can greatly improve the accuracy after quantization. For example, at 4-bits, trained quantization gives 68.30% top-1 accuracy without retraining. After retraining, this number increased to 76.17%, which caught up with the full-precision baseline accuracy. Even with 2-bits, retraining can give 69.36% top-1 accuracy for ResNet-50. This number is almost zero if not retrained.

Figure 4.10 and Table 4.9 compares the performance of uniform quantization and non-uniform quantization. Uniform quantization refers to the case when the distance between adjacent code is a constant. Trained quantization is a form of non-uniform quantization because the distance between each code is different. Uniform quantization can be handled by fixed-point arithmetic [120]. However, uniform quantization performs worse than non-uniform quantization in accuracy (Figure 4.10). For

Table 4.9: Comparison of uniform quantization and non-uniform quantization (this work) with different update methods. -c: updating centroid only; -c+l: update both centroid and label. Baseline ResNet-50 accuracy: 76.15%, 92.87%. All results are after retraining.

Quantization Method	1bit	2bit	4bit	6bit	8bit
Uniform (Top-1)	-	59.33%	74.52%	75.49%	**76.15%**
Uniform (Top-5)	-	82.39%	91.97%	92.60%	**92.91%**
Non-uniform -c (Top-1)	24.08%	68.41%	**76.16%**	76.13%	**76.20%**
Non-uniform -c (Top-5)	48.57%	88.49%	92.85%	**92.91%**	**92.88%**
Non-uniform -c+l (Top-1)	24.71%	69.36%	**76.17%**	**76.21%**	**76.19%**
Non-uniform -c+l (Top-5)	49.84%	89.03%	**92.87%**	**92.89%**	**92.90%**

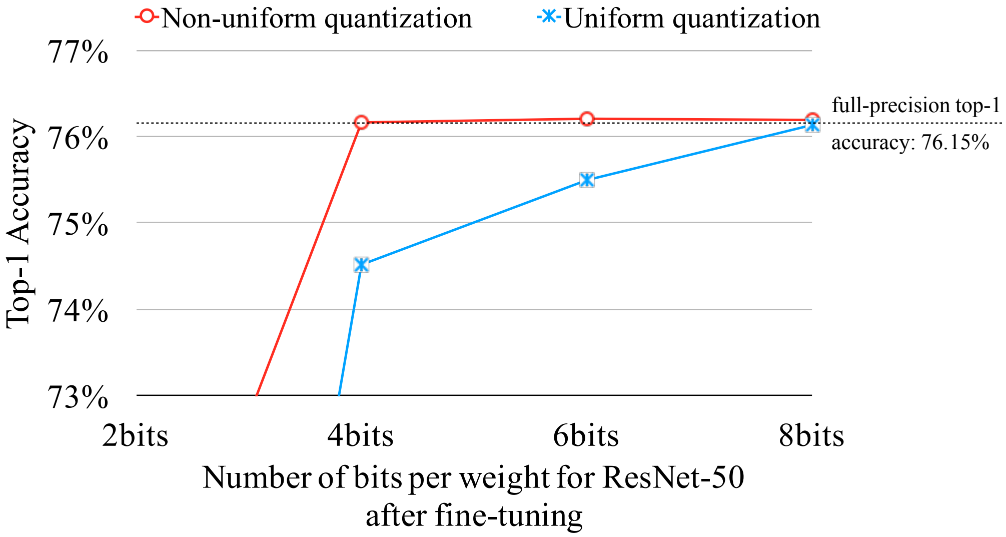


Figure 4.10: Non-uniform quantization performs better than uniform quantization.

non-uniform quantization (this work), all the layers of the baseline ResNet-50 can be compressed to **4-bits** without losing accuracy. For uniform quantization, however, all the layers of the baseline ResNet-50 can be compressed to **8 bits** without losing accuracy (at 4 bits, there are about 1.6% top-1 accuracy loss when using uniform quantization). The advantage of non-uniform quantization is that it can better capture the non-uniform distribution of the weights. When the probability distribution is higher, the distance between each centroid would be closer. However, uniform quantization can not achieve this.

Table 4.9 compares the performance two non-uniform quantization strategies. During fine-tuning, one strategy is to only update the centroid; the other strategy is to update both the centroid and the label (the label means which centroid does the weight belong to). Intuitively, the latter case has more degree of freedom in the learning process and should give better performance. However, experiments show that the improvement is not significant, as shown in the third row and the fourth row in Table

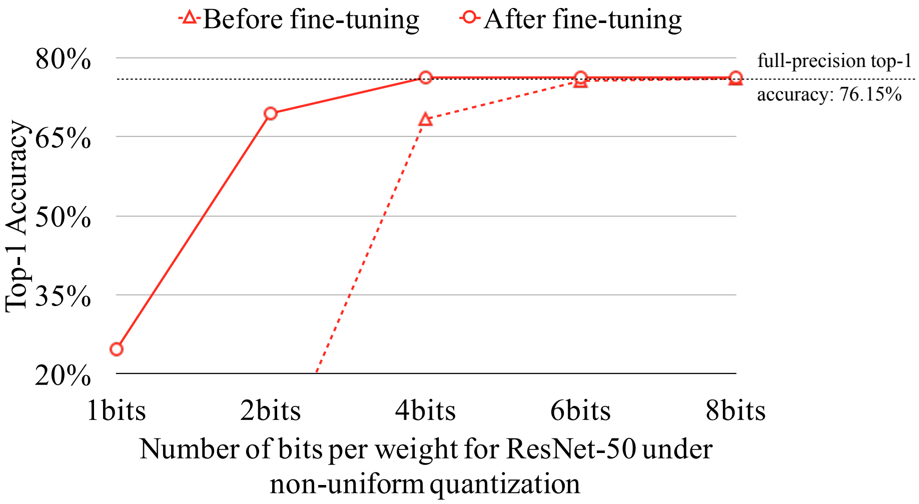


Figure 4.11: Fine-tuning is important for trained quantization. It can fully recover the accuracy when quantizing ResNet-50 to 4 bits.

4.9. So we looked into how many weights changed their label during the retraining process and found that most of the weights stayed within the group they belong to during the first Kmeans clustering process. The gradient was not large enough to nudge them to neighboring centroids. This is especially the case for fewer bits since the distance between adjacent centroid is larger than that when we have more bits. We tried increasing the learning rate by an order of magnitude, then weights began to change labels. By increasing the learning rate by two orders of magnitude, even more weights began to change labels. However, a quantized model that already have $> 50\%$ top-1 accuracy can not tolerate such large learning rate, and the optimization process fails to convergence. We later solved this problem in Trained Ternary Quantization [86] by introducing separate scaling factors for different centroid, which enables different learning for different centroids. During each SGD operation, we collect two gradients: the gradient of the latent weight adjusts the labels, and the gradient of the scaling factor adjusts the centroid. By separately learning the label and the centroid we can quantize ResNet-18 to ternary (positive, negative and zero) while losing 3% of Top-1 accuracy.

Figure 4.12 compares the accuracy of the three different initialization methods with respect to the top-1 accuracy (Left) and top-5 accuracy (Right). The network is quantized to $2 \sim 8$ bits as shown on the x-axis. Linear initialization outperforms the density initialization and random initialization in all cases except at 3 bits.

The centroids of linear initialization spread equally across the x-axis, from the min value to the max value. This initialization helps to maintain the large weights. The large weights play a more important role than smaller ones, which is also shown in network pruning by Han et al. [25]. Neither random nor density-based initialization retains large centroids. With these initialization methods,

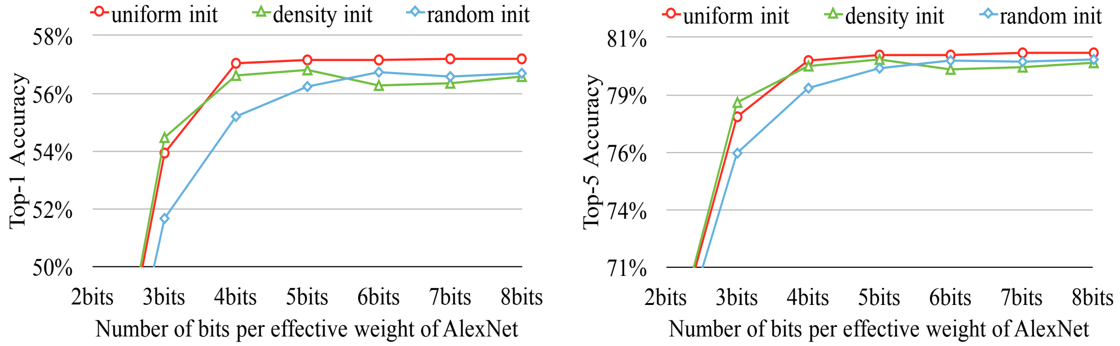


Figure 4.12: Accuracy of different initialization methods. Left: top-1 accuracy. Right: top-5 accuracy. Linear initialization gives the best result.

large weights are clustered to the small centroids because there are fewer large weights than small weights. In contrast, linear initialization gives large weights a better chance to form a large centroid, and lead to a better prediction accuracy.

Comparison with Previous Work. Previous work attempted to remove the redundancy and to compress deep neural network models. Table 4.10 compares the compression rate and the error rate. Deep Compression achieved an order of magnitude better compression ratio compared with previous work at no loss of accuracy.

Data-free pruning [121] saved 1.5× parameters with much loss of accuracy. Deep Fried Convnets [122] worked on the fully-connected layers and reduced the parameters by $2 - 3.7\times$. Collins et al. [123] reduced the parameters of AlexNet by 4×. Naively cutting the layer size and training s smaller model saves parameters but suffers from 4% loss of accuracy. SVD saves parameters but suffers from large accuracy loss, as much as 2% Top-1 accuracy loss on Imagenet. On other networks similar to AlexNet, Denton et al. [65] exploited the linear structure of convolutional neural networks and compressed the network by 2.4× to 13.4× layer wise, with 0.9% accuracy loss on compressing a single layer. Gong et al. [71] experimented with vector quantization and compressed the network by 16× to 24×, incurring 1% accuracy loss.

After Deep Compression was published in ICLR'2016, DNN model compression has been the topic of multiple academic papers and Deep Compression has been applied to industry. These related works are described in Chapter 2.

Future Work. While the *pruned* network has been benchmarked on commodity hardware platforms, the *quantized* network with weight sharing has not, because off-the-shelf cuSPARSE or MKL SPBLAS libraries do not support indirect matrix entry lookup, nor is the relative index in CSC or CSR format supported without being byte aligned. So the full advantage of Deep Compression, which makes a deep neural network fit in the cache, is not fully unveiled. A software solution

Table 4.10: Comparison with other compression methods on AlexNet.

Method	Top-1 Error	Top-5 Error	Model Size	Compress Rate
Baseline Caffemodel [124]	42.78%	19.73%	240MB	1×
Data-free pruning [121]	44.40%	-	158MB	1.5×
Fastfood-32-AD [122]	41.93%	-	131MB	2×
Fastfood-16-AD [122]	42.90%	-	64MB	3.7×
Collins & Kohli [123]	44.40%	-	61MB	4×
Naive Cut	47.18%	23.23%	55MB	4.4×
SVD [65]	44.02%	20.56%	48MB	5×
Deep Compression [26]	**42.78%**	**19.70%**	**6.9MB**	**35**×

is to write customized GPU kernels that support this. A hardware solution is to build custom ASIC architecture specialized to traverse the sparse and quantized network structure, which will be discussed in Chapter 6.

4.7 Conclusion

This chapter has presented "Deep Compression" which compresses deep neural network models by an order of magnitude without affecting the prediction accuracy. This method operates by pruning the unimportant connections, quantizing the network using weight sharing, and then applying variable-length coding. Deep Compression leads to a smaller storage requirement of deep neural networks and makes it easier to implement deep neural networks on mobile applications. With Deep Compression, the size of these networks fit into on-chip SRAM cache (5pJ/access) rather than requiring off-chip DRAM memory (640pJ/access). This potentially makes deep neural networks more energy efficient for running on mobile platforms.

Chapter 5

DSD: Dense-Sparse-Dense Training

5.1 Introduction

Modern high-performance hardware makes it easier to train complex DNN models with large model capacities. The upside of complex models is that they are very expressive and can capture highly non-linear relationships between features and outputs. The downside of such large models is that they are prone to capture the noise, rather than the intended pattern, in the training dataset. This noise does not generalize to test datasets, leading to overfitting and high variance.

However, simply reducing the model capacity would lead to the other extreme: a machine learning system that misses the relevant relationships between features and target outputs, leading to underfitting and a high bias. Thus, bias and variance are hard to optimize at the same time. To solve this problem, we propose a dense-sparse-dense training flow, DSD, for regularizing deep neural networks, preventing overfitting and achieving better accuracy.

Unlike conventional training, where all the parameters are trained at the same time, DSD training regularizes the network by periodically pruning and restoring the connections. The number of effective connections at training time is dynamically changing. Pruning the connections allows performing the optimization in a low-dimensional space and capture the robust features; restoring the connections allows increasing the model capacity. Unlike conventional training where all the weights are initialized *only once* at the beginning of training, DSD allows the connections to have *more than one* opportunity of being initialized through iterative pruning and restoring.

An advantage of DSD training is that the final neural network model still has the same architecture and dimensions as the original dense model, so DSD training does not incur any inference overhead. No specialized hardware or specialized deep learning framework is required to perform inference on DSD models. Experiments show that DSD training improves the performance of a wide range of CNNs, RNNs, and LSTMs on the tasks of image classification, caption generation, and speech recognition. On ImageNet, DSD improved the Top1 accuracy of GoogleNet by 1.1%, VGG-16 by 4.3%,

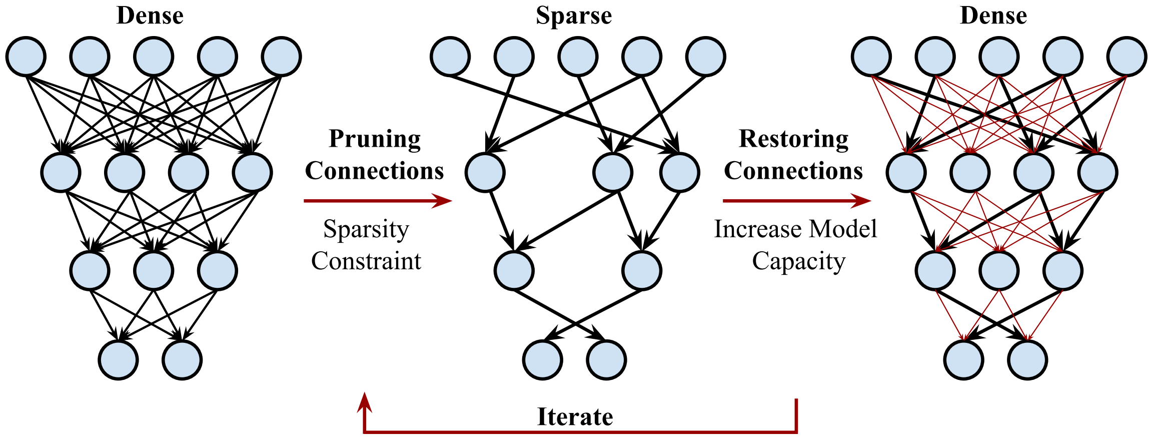


Figure 5.1: Dense-Sparse-Dense training consists of iteratively pruning and restoring the weights.

ResNet-18 by 1.2% and ResNet-50 by 1.1%, respectively. On the WSJ'93 dataset, DSD improved the DeepSpeech and the DeepSpeech2's word error rate(WER) by 2.0% and 1.1%, respectively. On the Flickr-8K dataset, DSD improved the NeuralTalk BLEU score by over 1.7.

DSD is easy to implement in practice: at training time, DSD incurs only one extra hyper-parameter: the sparsity ratio in the sparse training step. At testing time, DSD does not change the network architecture or incur any inference overhead. The consistent and significant performance gain of DSD experiments highlights the inadequacy of the current training methods for finding the global optimum. DSD training, in contrast to conventional training, effectively achieves higher accuracy. DSD models are available to download at https://songhan.github.io/DSD.

5.2 DSD Training

The DSD training employs a three-step process: dense, sparse, re-dense. Each step is illustrated in Figure 5.1 and Algorithm 2. The progression of weight distribution is plotted in Figure 5.2.

Initial Dense Training The initial dense training step learns the connection weights and their importance. In our approach, we use a simple heuristic to quantify the importance of the weights using their absolute value. Weights with large absolute value are considered important.

Optimizing a deep neural networks is a highly non-convex problem, and today it is generally solved using stochastic gradient descent (SGD), a convex optimization method. As a result, redundancy is needed to help with convergence. During optimization, redundant parameters provide multiple paths that allow easier convergence to a good local minima. If we don't have the dense training step or prune too early, the network fails to converge. The details of this experiment are discussed in Section 5.5.

Sparse Training Once the initial dense training finds a good local minima, it is safe to remove

```
Algorithm 2: DSD training
  Initialization: $W^{(0)}$ with $W^{(0)} \sim N(0, \Sigma)$
  Output: $W^{(t)}$.
  ———————— Initial Dense Phase ————————
  while not converged do
      $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
      $t = t + 1$;
  end
  ———————— Sparse Phase ————————
  // initialize the mask by sorting and keeping the Top-k weights.
  $S = sort(|W^{(t-1)}|)$;
  $\lambda = S_{k_i}$;
  $Mask = \mathbb{1}(|W^{(t-1)}| > \lambda)$;
  while not converged do
      $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
      $W^{(t)} = W^{(t)} \cdot Mask$;
      $t = t + 1$;
  end
  ———————— Final Dense Phase ————————
  while not converged do
      $W^{(t)} = W^{(t-1)} - 0.1\eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
      $t = t + 1$;
  end
  goto Sparse Phase for iterative DSD;
```

the redundant connections and perform the optimization in a low-dimensional space. The sparse training step prunes the low-weight connections and trains a sparse network. We applied the *same* sparsity to all the layers. Thus there is a single hyper parameter: the sparsity, which is the percentage of weights that are pruned to 0. For each layer W with N parameters, we sort the parameters, pick the k-th largest one $\lambda = S_k$ as the threshold where $k = N * (1 - sparsity)$, and generate a binary mask to remove all the weights smaller than λ. This process is shown in Algorithm 2.

DSD removes small weights because of the Taylor expansion. The loss function and its Taylor expansion are shown in Equation (5.1)(5.2). Because we want to minimize the increase in $Loss$ when conducting a hard thresholding on the weights, we need to minimize the first and second terms in Equation 5.2. Since we are zeroing out parameters, ΔW_i is actually $W_i - 0 = W_i$. At the local minimum where $\partial Loss/\partial W_i \approx 0$ and $\frac{\partial^2 Loss}{\partial W_i^2} > 0$, only the second order term matters. Since the second order gradient, $\partial^2 Loss/\partial W_i^2$ is expensive to calculate and W_i has a power of two, we use $|W_i|$ as the metric of pruning. Smaller $|W_i|$ means a smaller increase in the loss function.

$$Loss = f(x, W_1, W_2, W_3...) \tag{5.1}$$

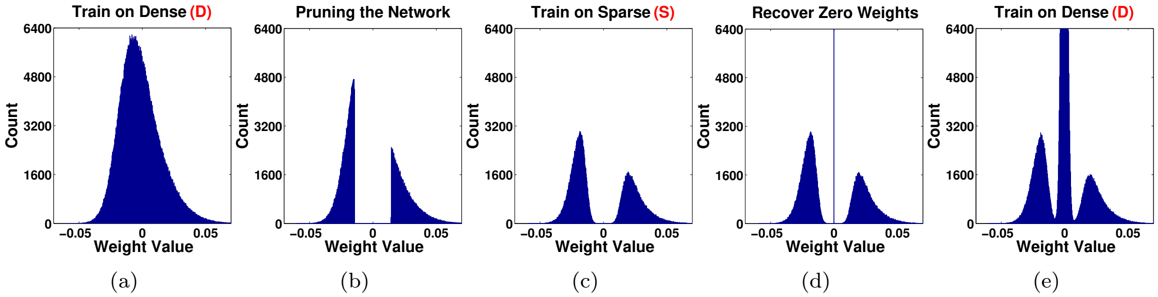


Figure 5.2: Weight distribution for the original GoogleNet (a), pruned (b), after retraining with the sparsity constraint (c), recovering the zero weights (d), and after retraining the dense network (e).

$$\Delta Loss = \frac{\partial Loss}{\partial W_i}\Delta W_i \ + \frac{1}{2}\frac{\partial^2 Loss}{\partial W_i^2}\Delta {W_i}^2 + ... \tag{5.2}$$

Retraining while enforcing the binary mask in each SGD iteration, DSD converts a dense network into a sparse network that has a known sparsity support. As the prior chapters have shown, the sparse network can fully recover or even increase the accuracy of the initial dense model.

Both model compression [25, 26] and DSD training use network pruning. The difference is that the focus of DSD training is improving the accuracy, not reducing the model size. As a result DSD training doesn't require aggressive pruning. We have found a modestly pruned network (30%-50% sparse) works well. However, model compression requires aggressively pruning the network to achieve high compression rates.

Restoring the Pruned Connections The final D step restores and re-initializes the pruned connections, making the network dense again. In this step, the previously-pruned connections are initialized from zero, and the entire network is retrained with 1/10 the original learning rate (since the sparse network is already at a good local minima, a smaller learning rate is required). Hyper parameters such as dropout and weight decay remain unchanged. By restoring the pruned connections, the final dense training increases the model capacity of the network and makes it possible to arrive at a better local minima compared with the sparse model. This restoring process also gives those pruned connections a second opportunity to initialize.

To visualize the DSD training flow, Figure 5.2 plot the progression of the weight distribution. The figure shows data from GoogleNet's inception_5b3x3 layer, and we find this progression of weight distribution very representative for VGGNet and ResNet as well. The original weight distribution is centered on zero with tails dropping off quickly. Pruning is based on the absolute values, so, after pruning, the large central region is truncated. The un-pruned network parameters adjust themselves during retraining, so in (c), the boundary becomes soft and forms a bimodal distribution. In (d), at the beginning of the re-dense training step, all the pruned weights are restored and reinitialized from zero. Finally, in (e), the restored weights are retrained together with those un-pruned weights. In

Table 5.1: Overview of the neural networks, data sets and performance improvements from DSD.

Neural Network	**Domain**	**Dataset**	**Type**	**Baseline**	**DSD**	**Abs. Imp.**	**Rel. Imp.**
GoogleNet	Vision	ImageNet	CNN	31.1%[1]	**30.0%**	1.1%	3.6%
VGG-16	Vision	ImageNet	CNN	31.5%[1]	**27.2%**	4.3%	13.7%
ResNet-18	Vision	ImageNet	CNN	30.4%[1]	**29.2%**	1.2%	4.1%
ResNet-50	Vision	ImageNet	CNN	24.0%[1]	**22.9%**	1.1%	4.6%
NeuralTalk	Caption	Flickr-8K	LSTM	16.8[2]	**18.5**	1.7	10.1%
DeepSpeech	Speech	WSJ'93	RNN	33.6%[3]	**31.6%**	2.0%	5.8%
DeepSpeech-2	Speech	WSJ'93	RNN	14.5% [3]	**13.4%**	1.1%	7.4%

[1] Top-1 error. VGG/GoogleNet baselines from the Caffe Model Zoo, ResNet from Facebook.
[2] BLEU score baseline from Neural Talk model zoo.
[3] Word error rate: DeepSpeech2 is trained with a portion of Baidu internal dataset with only max decoding to show the effect of DNN improvement.

this step, we keep the same learning rate for restored weights and un-pruned weights. As (d) and (e) show, the un-pruned weights' distribution remains the same, while the restored weights became distributed further around zero. The overall mean absolute value of the weight distribution is much smaller.

5.3 Experiments

We applied DSD training to different kinds of neural networks in different domains. We found that DSD training improved the accuracy for *all* these networks compared to the baseline networks that were not trained with DSD. The neural networks were chosen from CNNs, RNNs, and LSTMs; the datasets covered image classification, speech recognition, and caption generation. For networks trained for ImageNet, we focused on GoogleNet, VGG, and ResNet, which are widely used in research and production.

An overview of the networks, datasets and accuracy results is shown in Table 5.1. For the convolutional networks, the first layer is not pruned during the sparse phase, since it has only three channels and extracts low-level features. The sparsity is the *same* for all the other layers, including convolutional and fully connected layers. The initial learning rate at each stage is decayed the same as conventional training. The number of epochs is determined by loss converges. When the loss no longer decreases, we stop the training.

5.3.1 DSD for CNN

GoogleNet. We experimented with the BVLC GoogleNet [4] model obtained from the Caffe Model Zoo [124]. It has 13 million parameters and 57 convolutional layers. We pruned each layer (except the first) to 30% sparsity (70% non-zeros). Retraining the sparse network gave some improvement in accuracy due to regularization (Table 5.2). After the final dense training step, GoogleNet's error

Table 5.2: DSD results on GoogleNet

GoogleNet	Top-1 Err	Top-5 Err	Sparsity	Epochs	LR
Baseline	31.14%	10.96%	0%	250	1e-2
Sparse	30.58%	10.58%	30%	11	1e-3
DSD	**30.02%**	**10.34%**	0%	22	1e-4
LLR	30.20%	10.41%	0%	33	1e-5
Improve (abs)	1.12%	0.62%	-	-	-
Improve (rel)	**3.6%**	**5.7%**	-	-	-

Table 5.3: DSD results on VGG-16

VGG-16	Top-1 Err	Top-5 Err	Sparsity	Epochs	LR
Baseline	31.50%	11.32%	0%	74	1e-2
Sparse	28.19%	9.23%	30%	1.25	1e-4
DSD	**27.19%**	**8.67%**	0%	18	1e-5
LLR	29.33%	10.00%	0%	20	1e-7
Improve (abs)	4.31%	2.65%	-	-	-
Improve (rel)	**13.7%**	**23.4%**	-	-	-

rates were reduced by 1.12% (Top-1) and 0.62% (Top-5) over the baseline.

We compared DSD training with conventional training under the *same number of training epochs*. On top of a pre-trained model, we lowered the learning rate upon convergence and continued to learn. The result is shown in row LLR (lower the learning rate). The training epochs for LLR is equal to that of DSD as a fair comparison. LLR cannot achieve the same accuracy as DSD training.

Table 5.4: DSD results on ResNet-18 and ResNet-50

	ResNet-18		ResNet-50				
	Top-1 Err	Top-5 Err	Top-1 Err	Top-5 Err	Sparsity	Epochs	LR
Baseline	30.43%	10.76%	24.01%	7.02%	0%	90	1e-1
Sparse	30.15%	10.56%	23.55%	6.88%	30%	45	1e-2
DSD	**29.17%**	**10.13%**	**22.89%**	**6.47%**	0%	45	1e-3
LLR	30.04%	10.49%	23.58%	6.84%	0%	90	1e-5
Improve (abs)	1.26%	0.63%	1.12%	0.55%	-	-	-
Improve (rel)	**4.14%**	**5.86%**	**4.66%**	**7.83%**	-	-	-

VGG-16. We explored DSD training on VGG-16 [3], which is widely used in classification, detection, and segmentation tasks. The baseline model is obtained from the Caffe Model Zoo [124]. Similar to GoogleNet, each layer of VGG-16 is pruned to 30% sparsity (70% non-zeros). DSD training greatly reduced the error by 4.31% (Top-1) and 2.65% (Top-5), detailed in Table 5.3. DSD also shows a large margin of improved accuracy over the LLR result.

ResNet. We applied DSD training to ResNet-18 and ResNet-50 [5]. The baseline ResNet-18 and ResNet-50 models are provided by Facebook [125]. We used 30% sparsity during DSD training. A single DSD training pass for these networks reduced the top-1 error by 1.26% (ResNet-18) and

✗ **Baseline**: a boy in a red shirt is climbing a rock wall.

✗ **Sparse**: a young girl is jumping off a tree.

✓ **DSD**: a young girl in a pink shirt is swinging on a swing.

○ **Baseline**: a basketball player in a red uniform is playing with a ball.

○ **Sparse**: a basketball player in a blue uniform is jumping over the goal.

✓ **DSD**: a basketball player in a white uniform is trying to make a shot.

✓ **Baseline**: two dogs are playing together in a field.

✓ **Sparse**: two dogs are playing in a field.

✓ **DSD**: two dogs are playing in the grass.

✗ **Baseline**: a man and a woman are sitting on a bench.

○ **Sparse**: a man is sitting on a bench with his hands in the air.

○ **DSD**: a man is sitting on a bench with his arms folded.

✗ **Baseline**: a person in a red jacket is riding a bike through the woods.

✓ **Sparse**: a car drives through a mud puddle.

✓ **DSD**: a car drives through a forest.

Figure 5.3: Visualization of DSD training improving the performance of image captioning.

Table 5.5: DSD results on NeuralTalk

NeuralTalk	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Sparsity	Epochs	LR
Baseline	57.2	38.6	25.4	16.8	0	19	1e-2
Sparse	58.4	39.7	26.3	17.5	80%	10	1e-3
DSD	**59.2**	**40.7**	**27.4**	**18.5**	0	6	1e-4
Improve(abs)	2.0	2.1	2.0	1.7	-	-	-
Improve(rel)	**3.5%**	**5.4%**	**7.9%**	**10.1%**	-	-	-

1.12% (ResNet-50), shown in Table 5.4. As a fair comparison, we continue training the original model by lowering the learning rate by another decade but cannot reach the same accuracy as DSD, as shown in the LLR row. Under the same training time, just lowering the learning rate of conventional training can not achieve the same accuracy as DSD training.

5.3.2 DSD for RNN

NeuralTalk. We evaluated DSD training on RNN and LSTM beyond CNN. We applied DSD to NeuralTalk [7], an LSTM for generating image descriptions. It uses a CNN as an image feature extractor and an LSTM to generate captions. To verify DSD training works on LSTMs, we fix the CNN weights and only train the LSTM weights. The baseline NeuralTalk model is downloaded from the NeuralTalk Model Zoo.

In the pruning step, we pruned all layers except W_s, the word embedding lookup table, to 80% sparse. We retrained the remaining sparse network using the same weight decay and batch size as the original paper. The learning rate is tuned based on the validation set, shown in Table 5.5. Retraining the sparse network improved the BLUE score by [1.2, 1.1, 0.9, 0.7]. After eliminating the sparsity constraint and restoring the pruned weights, the final results of DSD further improved the BLEU score by [2.0, 2.1, 2.0, 1.7] over baseline.

The BLEU score is not the sole quality metric of an image captioning system. We visualized the captions generated by DSD training in Figure 5.3. In the first image, the baseline model mistakes the girl for a boy and the girl's hair with a rock wall; the sparse model can tell that the image is a girl, and the DSD model can further identify the swing. In the second image, DSD training can accurately determine that the player in a white uniform is trying to make a shot, rather than the baseline model that says the player is in a red uniform and just playing with a ball. It is interesting to notice that the sparse model sometimes works better than the DSD model: in the last image, the sparse model correctly captured the mud puddle, while the DSD model only captured the forest from the background. The performance of DSD training generalizes beyond these examples; more image caption results generated by DSD training are provided in the Appendix at the end of this chapter.

DeepSpeech. We explore DSD training on speech recognition tasks using both Deep Speech 1 (DS1) and Deep Speech 2 (DS2) networks [8,9]. DSD training improves the relative accuracy of both DS1 and DS2 models on the Wall Street Journal (WSJ) test sets by 2.1%∼7.4%.

The DS1 model is a five-layer network with one Bidirectional Recurrent layer, as described in Table 5.6. This model was trained using the Wall Street Journal (WSJ) data set that contains 81 hours of speech. The validation set consists of 1 hour of speech. The test sets are from WSJ'92 and WSJ'93 and contain 1 hour of speech combined. The Word Error Rate (WER) reported on the test sets for the baseline models is different from [9] due to two factors: first, in DeepSpeech2, the models were trained using a much larger data sets containing approximately 12,000 hours of multi-speaker speech data. Second, WER was evaluated with beam search and a language model in DeepSpeech2; here the network output is obtained using only max decoding to show improvement in the neural network accuracy and filtering out the other parts.

The first dense phase was trained for 50 epochs. In the sparse phase, weights are pruned only in the fully connected layers and the bidirectional recurrent layer, which are the majority of the weights. Each layer is pruned to achieve the same 50% sparsity and trained for 50 epochs. In the final dense phase, the pruned weights are initialized to zero and trained for another 50 epochs. We keep the hyper parameters unchanged except for the learning rate. The learning rate is picked using our validation set that is separate from the test set.

The DSD training requires 150 epochs, 50 epochs for each D-S-D training step. We want to compare the DSD results with a baseline model trained for the *same* number of epochs. The first three rows of Table 5.7 shows the WER when the DSD model is trained for 50+50+50=150 epochs, and the 6th line shows the baseline model trained by 150 epochs (the same #epochs as DSD). DSD training improves WER by 0.13 (WSJ'92) and 1.35 (WSJ'93) given the same number of epochs as the conventional training.

Given a second DSD iteration (DSDSD), accuracy can be further improved. In the second DSD

Table 5.6: Deep Speech 1 Architecture

Layer ID	Type	#Params
layer 0	Convolution	1814528
layer 1	FullyConnected	1049600
layer 2	FullyConnected	1049600
layer 3	Bidirectional Recurrent	3146752
layer 4	FullyConnected	1049600
layer 5	CTCCost	29725

Table 5.7: DSD results on Deep Speech 1

DeepSpeech 1	WSJ '92	WSJ '93	Sparsity	Epochs	LR
Dense Iter 0	29.82	34.57	0%	50	8e-4
Sparse Iter 1	27.90	32.99	50%	50	5e-4
Dense Iter 1	27.90	32.20	0%	50	3e-4
Sparse Iter 2	27.45	32.99	25%	50	1e-4
Dense Iter 2	**27.45**	**31.59**	0%	50	3e-5
Baseline	28.03	33.55	0%	150	8e-4
Improve(abs)	0.58	1.96	-	-	-
Improve(rel)	**2.07%**	**5.84%**	-	-	-

iteration, we assume that the network is closer stable and pruned less parameters: each layer is pruned to 25% sparse. Similar to the first iteration, the sparse model and subsequent dense model are further retrained for 50 epochs. The learning rate is scaled down for each retraining step. The results are shown in Table 5.7. We can do more DSD iterations (DSDSD) to further improve the performance. The second DSD iteration improves WER by 0.58 (WSJ'92) and 1.96 (WSJ'93), a relative improvement of 2.07% (WSJ'92) and 5.84% (WSJ'93).

DeepSpeech 2. To show how DSD works on deeper networks, we evaluated DSD on the Deep Speech 2 (DS2) network, described in Table 5.8. This network has seven bidirectional recurrent layers with approximately 67 million parameters, around eight times larger than the DS1 model. A subset of the internal English training set is used. The training set consists of 2,100 hours of speech. The validation set consist of 3.46 hours of speech. The test sets are from WSJ'92 and WSJ'93, which contain 1 hour of speech combined. The DS2 model is trained using Nesterov SGD for 20 epochs for each training step. Similar to DS1 experiments, the learning rate is reduced by an order of magnitude with each retraining. The other hyper parameters remain unchanged.

Table 5.9 shows the results of the two iterations of DSD training. For the first sparse re-training, similar to DS1, 50% of the parameters from the bidirectional recurrent layers and fully connected layers are pruned to zero. The Baseline model is trained for 60 epochs to provide a fair comparison with DSD training. The baseline model shows no improvement after 40 epochs. With one iteration of DSD training, WER improves by 0.44 (WSJ'92) and 0.56 (WSJ'93) compared to the fully trained baseline.

Table 5.8: Deep Speech 2 Architecture

Layer ID	Type	#Params
layer 0	2D Convolution	19616
layer 1	2D Convolution	239168
layer 2	Bidirectional Recurrent	8507840
layer 3	Bidirectional Recurrent	9296320
layer 4	Bidirectional Recurrent	9296320
layer 5	Bidirectional Recurrent	9296320
layer 6	Bidirectional Recurrent	9296320
layer 7	Bidirectional Recurrent	9296320
layer 8	Bidirectional Recurrent	9296320
layer 9	FullyConnected	3101120
layer 10	CTCCost	95054

Table 5.9: DSD results on Deep Speech 2

DeepSpeech 2	WSJ '92	WSJ '93	Sparsity	Epochs	LR
Dense Iter 0	11.83	17.42	0%	20	3e-4
Sparse Iter 1	10.65	14.84	50%	20	3e-4
Dense Iter 1	9.11	13.96	0%	20	3e-5
Sparse Iter 2	**8.94**	14.02	25%	20	3e-5
Dense Iter 2	9.02	**13.44**	0%	20	6e-6
Baseline	9.55	14.52	0%	60	3e-4
Improve(abs)	0.53	1.08	-	-	-
Improve(rel)	**5.55%**	**7.44%**	-	-	-

Here we show again that DSD can be applied multiple times or iteratively for further performance gain. The second iteration of DSD training achieves better accuracy (Table 5.9). For the second sparse iteration, 25% of parameters in the fully connected layers and bidirectional recurrent layers are pruned. Overall, DSD training achieves relative improvement of 5.55% (WSJ'92) and 7.44% (WSJ'93) on the DS2 architecture. These results are in line with DSD experiments on the smaller DS1 network. We can conclude that DSD re-training continues to show improvement in accuracy with larger layers and deeper networks.

5.4 Significance of DSD Improvements

DSD training improves the baseline model performance by consecutively pruning and restoring the network weights. We conducted more intensive experiments to validate that the improvements are significant and not due to any randomness in the optimization process. To evaluate the significance, we repeated the baseline training, DSD training and conventional fine-tuning 16 times. The statistical significance of DSD improvements is quantified on the Cifar-10 dataset using ResNet-20.

Table 5.10: DSD results for ResNet-20 on Cifar-10. The experiment is repeated 16 times to get rid of noise.

ResNet-20	AVG Top-1 Err	STD Top-1 Err	Sparsity	Epochs	LR
Baseline	8.26%	-	0%	164	1e-1
Direct Finetune (First half)	8.16%	0.08%	0%	45	1e-3
Direct Finetune (Second half)	7.97%	0.04%	0%	45	1e-4
DSD (Fist half, Sparse)	8.12%	0.05%	50%	45	1e-3
DSD (Second half, Dense)	**7.89%**	**0.03%**	0%	45	1e-4
Improve from baseline(abs)	0.37%	-	-	-	-
Improve from baseline(rel)	**4.5%**	-	-	-	-

Training on Cifar-10 is fast enough that it is feasible to conduct intensive experiments within a reasonable time to evaluate DSD performance. The baseline models were trained with the standard 164 epochs and initial LR of 0.1 as recommended in the code released by Facebook [125]. After 164 epochs, we obtained a model with an 8.26% top-1 testing error that is consistent with the Facebook result. Initialized from this baseline model, we repeated DSD training 16 times. We also repeated the conventional fine-tuning 16 times. DSD training used a sparsity of 50% and 90 epochs (45 for sparse training and 45 for re-dense training). To provide a fair comparison, the conventional fine-tuning is also based on the ***same*** baseline model with the ***same*** hyper-parameters and settings.

Detailed results are listed below. On Cifar-10 using the ResNet-20 architecture, DSD training achieved on average a Top-1 error of 7.89%, which is a 0.37% absolute improvement (4.5% relative improvement) over the baseline model and relatively 1.1% better than the conventional fine-tuning. The experiment also shows that DSD training can reduce the variance of learning: the trained models after the sparse training and the final DSD training both have a lower standard deviation of errors compared with their counterparts using conventional fine-tuning. We did a T-test to compare the error rate from DSD and conventional training. The T-test result demonstrates that DSD training achieves significant improvements compared with the baseline model (with $p<0.001$) and conventional fine tuning (with $p<0.001$).

5.5 Reducing Training Time

DSD training involves pruning a fully pre-trained model. It is more timing-consuming than the common training method. Therefore, we want to see if the first dense training step can be eliminated, i.e., pruning the model early. We experimented with AlexNet and DeepSpeech and found that we can do early pruning before the model fully converges, but we can not eliminate the first dense training step.

The most aggressive approach is to eliminate the initial dense training step and train a sparse network from scratch. However, our experiment find that this method does not converge well: we

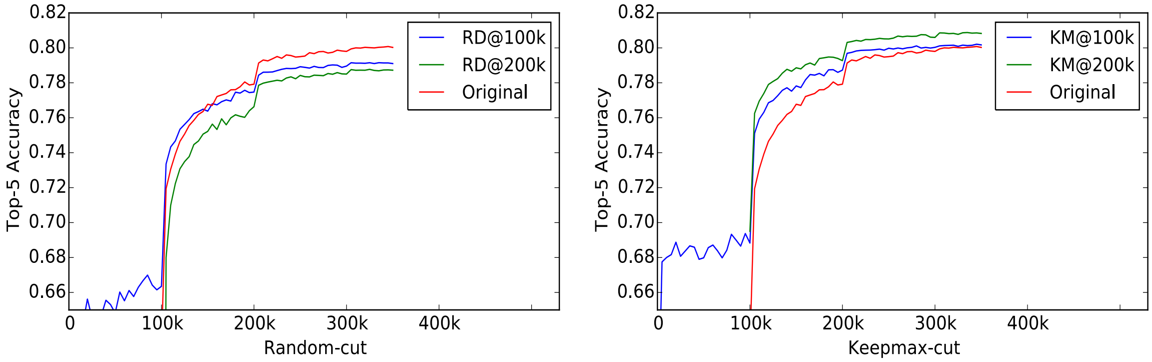


Figure 5.4: Learning curve of early pruning: Random-Cut (Left) and Keepmax-Cut (Right).

prune half of the parameters of the BVLC AlexNet without any training using the same sparsity pattern as the magnitude based pruning. In our two repeated experiments on training such a sparse network, one model diverged after 80k training iterations, and the other converged at a top-1 accuracy of only 33.0% and a top-5 accuracy of 41.6%. So the observation is that sparse training from scratch leads to lower accuracy.

A less aggressive approach is to train the network for a short period and then prune it. In this way, the network will go through a warm up training period to learn which connections are important. In the experiment, we pruned AlexNet after training for 100k and 200k iterations *before* it converged at 400k. (Figure 5.4), both methods converged at a better accuracy compared with the baseline, which indicates that we can reduce the training epochs by fusing the first dense and sparse training steps, i.e. early pruning doesn't hurt the accuracy but can save DSD training time.

The DeepSpeech experiments provide the same support for reducing the DSD training time. In the experiments, weights are pruned early after training the initial model prematurely for only 50 epochs (in contrast, a fully trained model requires 75 epochs). This still achieves better performance than the fully trained model and shows that we can reduce the number of training epochs by fusing the dense training with the sparse training step. Similar findings are observed in DeepSpeech-2.

Next, we compared two pruning heuristics: magnitude-based pruning (Keepmax-Cut), which we used throughout this paper, and random pruning (Random-Cut). The results are plotted in Figure 5.4. Keepmax-Cut@200K obtained 0.36% better top-5 accuracy. Random-Cut, in contrast, greatly lowered the accuracy and caused the model to converge with lower accuracy. Random-Cut@100k deteriorated the final accuracy by 1.06% and Random-Cut@200k by 1.51%. Thus, magnitude-based pruning is better than random pruning.

In sum, we have three observations: early pruning by fusing the first D and S step together can make DSD training converge faster and more accurately; magnitude-based pruning learns the correct sparsity pattern better than random pruning; sparsity from scratch leads to poor convergence.

5.6 Discussion

Dense-Sparse-Dense training changes the optimization process and improves accuracy by significant margins. During training, DSD re-structures the network by pruning and restoring the connections. We hypothesize that the following aspects contribute to the efficacy of DSD training.

Escape Saddle Point: Based on previous studies, one of the most profound difficulties of optimizing deep networks is the proliferation of saddle points [126]. The proposed DSD method helps to escape the saddle points by pruning and restoring the connections. Pruning the converged model perturbs the learning dynamics and allows the network to jump away from saddle points, which gives the network a chance to converge at a better local minimum. This idea is similar to simulated annealing [127]. Both simulated annealing and DSD training can escape sub-optimal solutions, and both can be applied iteratively to achieve further performance gains. The difference between simulated annealing and DSD training is that simulated annealing *randomly* jumps with decreasing probability on the search graph, DSD *deterministically* deviates from the converged solution achieved in the first dense training phase by removing the small weights and enforcing a sparsity support.

Significantly Better Minima: After escaping saddle points, DSD can achieve a better local minima. We measured both the training loss and validation loss, DSD training decreased the loss and error on both the training and the validation sets on ImageNet, indicating that it found a better local minima. We have also validated the significance of the improvements compared with conventional fine-tuning use a T-test (Section 5.4).

Regularization with Sparse Training: The sparsity regularization in the sparse training step moves the optimization to a lower-dimensional space where the loss surface is smoother and tends to be more robust to noise. Numerical experiments verified that sparse training reduced the variance of training error (Section 5.4).

Robust Re-initialization: Weight initialization plays a significant role in deep learning [128]. Conventional training gives each connection only one chance of initialization. DSD gives the connections additional opportunities to re-initialize during the restoration step, where those restored weights gets re-initialized from zero.

Break Symmetry: The permutation symmetry of the hidden units makes the weights symmetrical, which is prone to co-adaptation. In DSD, by pruning and restoring the weights, the training process breaks the symmetry of the hidden units associated with the weights, and the weights are no longer symmetrical in the final dense training phase.

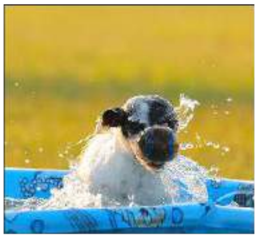
There is rich prior work on regularization techniques to prevent over-fitting, such as weight decay, dropout, and batch normalization. Our DSD experiments work on top of these techniques and give better accuracy. For example on VGG-16, DSD training was used together with weight decay and dropout while the baseline only used weight decay and dropout; on ResNet-50, DSD training was used together with weight decay and batch normalization while the baseline only used weight decay and batch normalization.

5.7 Conclusion

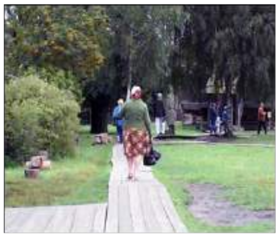
This chapter introduced a dense-sparse-dense training method (DSD) that regularizes neural networks by pruning and then restoring the connections. Our method learns which connections are important during initial dense training. DSD then regularizes the network by pruning the unimportant connections and retraining to a sparser and more robust solution with the same or better accuracy. Finally, the pruned connections are restored, and the entire network is retrained again. This increases the dimensionality of parameters and the model capacity.

DSD training achieves higher prediction accuracy. Our experiments using GoogleNet, VGGNet, and ResNet on ImageNet; NeuralTalk on Flickr-8K; DeepSpeech and DeepSpeech-2 on the WSJ dataset show that the accuracy of CNNs, RNNs, and LSTMs can significantly benefit from DSD training. We also did a T-test to verify that the DSD training improvements are statistically significant. The experimental results demonstrate the effectiveness of DSD training in improving the accuracy.

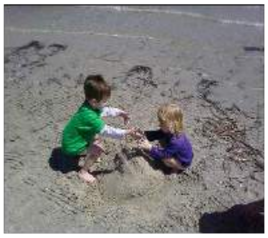
Appendix: Examples of DSD Improves Image Captioning



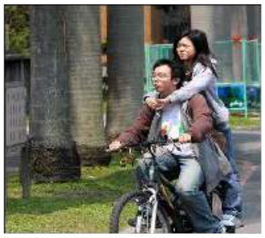
✗ **Baseline**: a boy is swimming in a pool.
○ **Sparse**: a small black dog is jumping into a pool.
✓ **DSD**: a black and white dog is swimming in a pool.



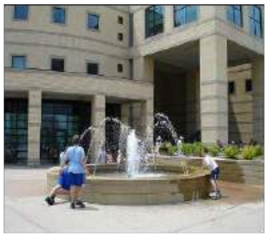
✗ **Baseline**: a group of people are standing in front of a building.
✗ **Sparse**: a group of people are standing in front of a building.
✓ **DSD**: a group of people are walking in a park.



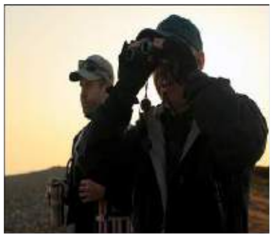
✗ **Baseline**: two girls in bathing suits are playing in the water.
✓ **Sparse**: two children are playing in the sand.
✓ **DSD**: two children are playing in the sand.



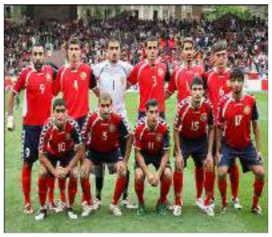
○ **Baseline**: a man in a red shirt and jeans is riding a bicycle down a street.
○ **Sparse**: a man in a red shirt and a woman in a wheelchair.
✓ **DSD**: a man and a woman are riding on a street.



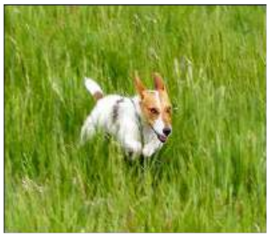
✗ **Baseline**: a group of people sit on a bench in front of a building.
○ **Sparse**: a group of people are standing in front of a building.
✓ **DSD**: a group of people are standing in a fountain.



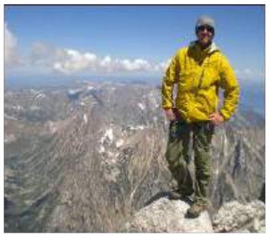
✗ **Baseline**: a man in a black jacket and a black jacket is smiling.
✗ **Sparse**: a man and a woman are standing in front of a mountain.
✓ **DSD**: a man in a black jacket is standing next to a man in a black shirt.



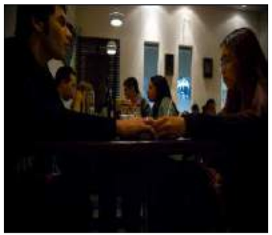
○ **Baseline**: a group of football players in red uniforms.
○ **Sparse**: a group of football players in a field.
✓ **DSD**: a group of football players in red and white uniforms.



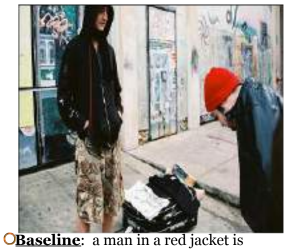
○ **Baseline**: a dog runs through the grass.
○ **Sparse**: a dog runs through the grass.
✓ **DSD**: a white and brown dog is running through the grass.



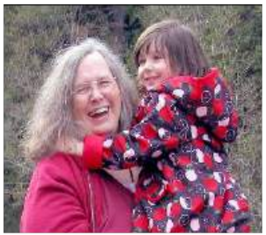
○ **Baseline**: a man in a red shirt is standing on a rock.
○ **Sparse**: a man in a red jacket is standing on a mountaintop.
✓ **DSD**: a man is standing on a rock overlooking the mountains.



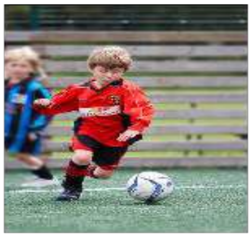
✗ **Baseline**: a group of people are sitting in a subway station.
✗ **Sparse**: a man and a woman are sitting on a couch.
✓ **DSD**: a group of people are sitting at a table in a room.



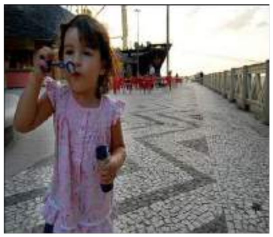
○ **Baseline**: a man in a red jacket is standing in front of a white building.
○ **Sparse**: a man in a black jacket is standing in front of a brick wall.
✓ **DSD**: a man in a black jacket is standing in front of a white building.



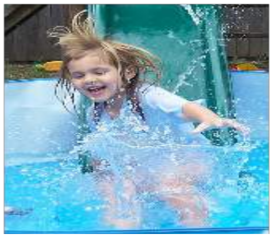
✗ **Baseline**:a young girl in a red dress is holding a camera.
✗ **Sparse**: a little girl in a pink dress is standing in front of a tree.
○ **DSD**: a little girl in a red dress is holding a red and white flowers.



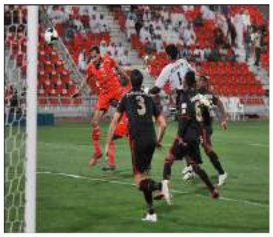
○ **Baseline**: a soccer player in a red and white uniform is playing with a soccer ball.
✓ **Sparse**: two boys playing soccer.
✓ **DSD**: two boys playing soccer.



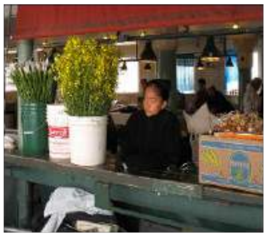
○ **Baseline**: a girl in a white dress is standing on a sidewalk.
○ **Sparse**: a girl in a pink shirt is standing in front of a white building.
✓ **DSD**: a girl in a pink dress is walking on a sidewalk.



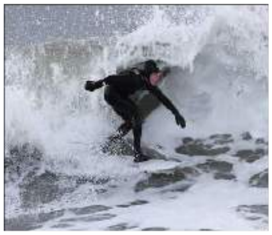
○ **Baseline**: a young girl in a swimming pool.
✗ **Sparse**: a young boy in a swimming pool.
○ **DSD**: a girl in a pink bathing suit jumps into a pool.



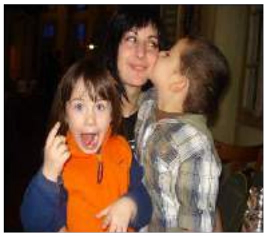
✗ **Baseline**: a soccer player in a red and white uniform is running on the field.
○ **Sparse**: a soccer player in a red uniform is tackling another player in a white uniform.
✓ **DSD**: a soccer player in a red uniform kicks a soccer ball.



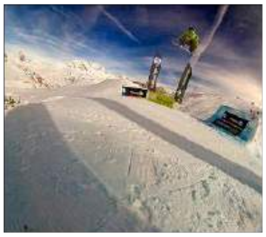
✗**Baseline**: a man in a red shirt is sitting in a subway station.
○**Sparse**: a woman in a blue shirt is standing in front of a store.
○**DSD**: a man in a black shirt is standing in front of a restaurant.



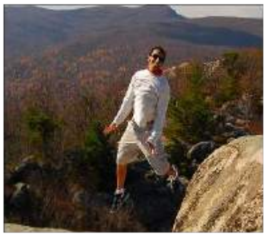
✓**Baseline**: a surfer is riding a wave.
✓**Sparse**: a man in a black wetsuit is surfing on a wave.
✓**DSD**: a man in a black wetsuit is surfing a wave.



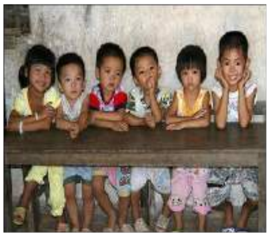
○**Baseline**: two young girls are posing for a picture.
✗**Sparse**: a young girl with a blue shirt is blowing bubbles.
✓**DSD**: a young boy and a woman smile for the camera.



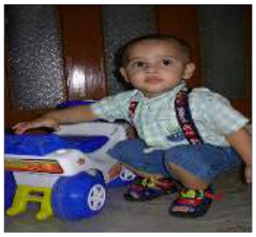
✗**Baseline**: a snowboarder flies through the air.
✓**Sparse**: a person is snowboarding down a snowy hill.
✓**DSD**: a person on a snowboard is jumping over a snowy hill.



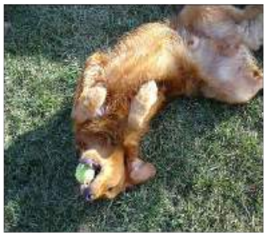
○**Baseline**: a man in a red shirt is standing on top of a rock.
○**Sparse**: a man in a red shirt is standing on a cliff overlooking the mountains.
✓**DSD**: a man is standing on a rock overlooking the mountains.



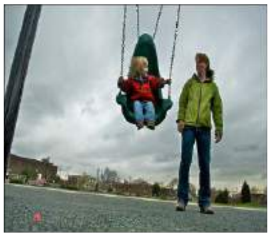
○**Baseline**: a group of people sit on a bench.
○**Sparse**: a group of people are sitting on a bench.
✓**DSD**: a group of children are sitting on a bench.



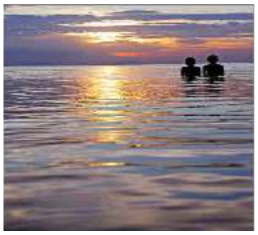
○**Baseline**: a little boy is playing with a toy.
✗**Sparse**: a little boy in a blue shirt is playing with bubbles.
✓**DSD**: a baby in a blue shirt is playing with a toy.



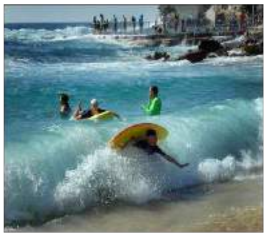
✗**Baseline**: a brown dog is running through the grassy.
✓**Sparse**: a brown dog is playing with a ball.
✓**DSD**: a brown dog is playing with a ball.



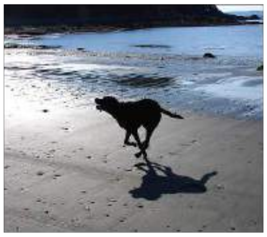
○**Baseline**: a boy in a red shirt is jumping on a trampoline.
○**Sparse**: a boy in a red shirt is jumping in the air.
✓**DSD**: a boy in a red shirt is jumping off a swing.



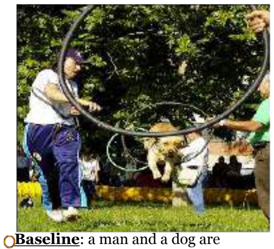
✗**Baseline**: a man is standing on the edge of a cliff.
○**Sparse**: a man is standing on the shore of a lake.
✓**DSD**: a man is standing on the shore of the ocean.



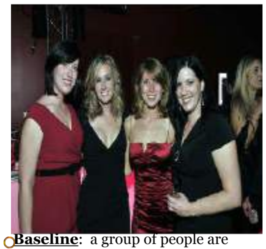
✗**Baseline**: two people are riding a boat on the beach.
○**Sparse**: two people are riding a wave on a beach.
✓**DSD**: a man in a yellow kayak is riding a wave.



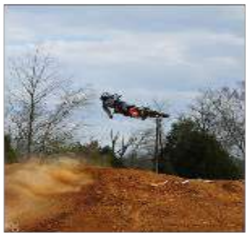
○**Baseline**: a black and white dog is running on the beach.
○**Sparse**: a black and white dog running on the beach.
✓**DSD**: a black dog is running on the beach.



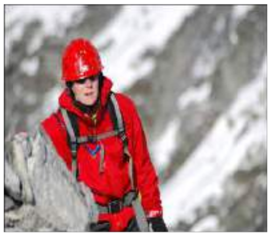
○**Baseline**: a man and a dog are playing with a ball.
✗**Sparse**: a man and a woman are playing tug of war.
✓**DSD**: a man and a woman are playing with a dog.



○**Baseline**: a group of people are standing in a room.
○**Sparse**: a group of people gather together.
✓**DSD**: a group of people are posing for a picture.



○**Baseline**: a man in a red jacket is riding a bike through the woods.
✗**Sparse**: a man in a red jacket is doing a jump on a snowboard.
✓**DSD**: a person on a dirt bike jumps over a hill.



✓**Baseline**: a man in a red jacket and a helmet is standing in the snow.
✓**Sparse**: a man in a red jacket and a helmet is standing in the snow.
✓**DSD**: a man in a red jacket is standing in front of a snowy mountain.

Chapter 6

EIE: Efficient Inference Engine for Sparse Neural Network

6.1 Introduction

Having described the efficient ***methods*** for deep learning in Chapters 3, 4 and 5, this chapter focuses on ***hardware*** to efficiently implement these methods, the "Efficient Inference Engine" (EIE) [28]. This machine can perform inference directly on the sparse, compressed model, which saves memory bandwidth and results in significant speedup and energy savings.

Deep Compression via pruning and trained quantization [25] [26] described in previous chapters significantly reduces the model size and memory bandwidth required for fetching parameters in deep neural networks. However, taking advantage of the compressed DNN model in hardware is a challenging task. Though compression reduces the total number of operations, the irregular pattern caused by compression hinders the effective acceleration. For example, the weight sparsity brought by pruning makes the parallelization more difficult and makes it impossible to use well-optimized dense linear algebra libraries. Moreover, the activation sparsity depends on the computed output of the prior layer, which is only known during the algorithm execution. Furthermore, trained quantization and weight sharing lead to another level of indirection to fetch the weight values. To solve these problems and efficiently operate on the sparse, compressed DNN models, we developed EIE, a specialized hardware accelerator that performs customized sparse matrix vector multiplication with weight sharing, which reduces the memory footprint and results in significant speedup and energy savings when performing inference.

EIE is a scalable array of processing elements (PEs). It distributes the sparse matrix and parallelizes the computation by interleaving matrix rows over the PEs. Every PE stores a partition of the network in SRAM and performs the computations associated with that sub-network. EIE

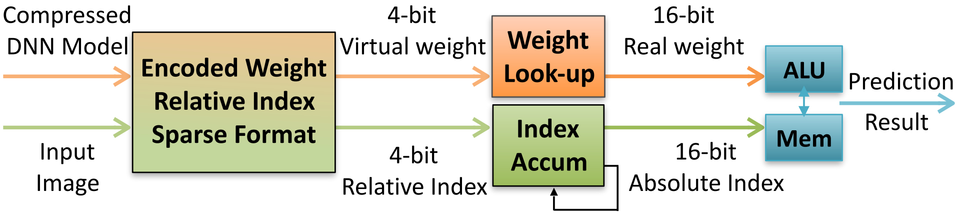


Figure 6.1: Efficient inference engine that works on the compressed deep neural network model for machine learning applications.

takes advantage of static weight sparsity, dynamic activation vector sparsity, relative indexing, weight sharing and extremely narrow weights (4-bits).

An overview of the EIE architecture is shown in Figure 6.1. EIE stores the encoded sparse weight matrix W in compressed sparse column (CSC) format with only the non-zero weights. EIE performs multiplications only when both the weight and the activation are non-zero. EIE stores the location index of each weight in run-length encoded format. After the trained quantization and weight sharing, each weight takes only 4-bits, which is decoded to 16-bits by accessing a lookup table that is implemented with 16 registers.

To evaluate the performance of EIE, we created a full behavior and RTL model of EIE. The RTL model was then synthesized and place-and-routed to extract accurate energy and clock frequency. Evaluated on nine DNN benchmarks, EIE is 189× and 13× faster when compared to CPU and GPU implementations of the same DNN without compression. EIE has a processing power of 102 GOPS/s working directly on a sparse network, corresponding to 3 TOPS/s on a dense network with the same accuracy, dissipating a power consumption of only 600mW. EIE is 24,000× and 3,400× more energy efficient than a CPU and GPU, respectively. The contributions of EIE include:

1. **Sparse Weight:** EIE is the first accelerator for sparse and compressed deep neural networks. Operating directly on the sparse, compressed model enables the weights of neural networks to fit in on-chip SRAM, which results in 120× better energy savings compared to accessing from external DRAM. By skipping the zero weights, EIE saves 10× the computation cycles.

2. **Sparse Activation:** EIE exploits the dynamic sparsity of activations to save computation and memory reference. EIE saves 65.16% energy by avoiding computation on the 70% of activations that are zero in typical deep learning applications.

3. **Coded Weights:** EIE is the first accelerator to exploit the non-uniformly quantized, extremely narrow weights (4-bits per weight) to perform inference with lookup tables. This saves 8× memory footprint to fetch the weights compared with 32 bit floating-point and 2× compared with int-8.

4. **Parallelization:** EIE introduces methods to distribute the storage and the computation across

multiple PEs to parallelize a sparsified layer. EIE also introduces architecture changes to achieve load balance and good scalability.

6.2 Parallelization on Sparse Neural Network

One simple way to parallelize the computation of sparse neural network is to recover the sparse matrix into dense format, and to use the dense linear algebra parallelization techniques to accelerate the computation. We can perform gating to save the computation energy by skipping zero operands. However, this method only saves energy but does not save the computation cycles. So instead of converting a sparse model to a dense model, we propose to perform the computation directly on the sparse model, which saves both computation cycles and energy.

6.2.1 Computation

One layer of a deep neural network performs the computation $b = f(Wa + v)$, where a is the input activation vector, b is the output activation vector, v is the bias, W is the weight matrix, and f is the non-linear function, typically the Rectified Linear Unit(ReLU) [129] in CNN and some RNN. The bias term v is combined with W by appending an additional one to vector a. Therefore, we neglect the bias term in the following equations. The output activations are computed as

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij} a_j\right). \tag{6.1}$$

Deep Compression [26] describes a method to compress DNNs without loss of accuracy through a combination of pruning and weight sharing. Pruning makes matrix W sparse with density D ranging from 4% to 25% for our benchmark layers. Weight sharing replaces each weight W_{ij} with a four-bit index I_{ij} into a shared table S of 16 possible weight values. With deep compression, the per-activation computation of Equation (2) becomes

$$b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}] a_j\right). \tag{6.2}$$

where X_i is the set of columns j for which $W_{ij} \neq 0$, Y is the set of indices j for which $a_j \neq 0$, I_{ij} is the index to the shared weight that replaces W_{ij}, and S is the table of shared weights. Here X_i represents the static sparsity of W, and Y represents the dynamic sparsity of a. The set X_i is fixed for a given model. The set Y varies from input to input.

Comparing Equation 6.1 and Equation 6.2, we highlight two differences. The W_{ij} term is replaced with $S[I_{ij}]$, which is the result from trained quantization and weight sharing. It is a lookup operation because weight sharing added a level of indirection. The other difference is that the "$j = 0$ to $n-1$" in the summation is replaced with "$j \in X_i \cap Y$", which is the result of sparsity. We perform the multiply-add only for those columns for which both W_{ij} and a_j are non-zero, so that both the sparsity of the matrix and the vector are exploited. The sparsity pattern of W is fixed during execution (static), while the sparsity pattern of a depends on the input (dynamic). Performing the indexing itself involves bit manipulations to extract four-bit I_{ij} and an extra lookup with a table of 16 entries.

6.2.2 Representation

To exploit the sparsity of activations, we store our sparse weight matrix W in a variation of compressed sparse column (CSC) format [130]. For each column W_j of matrix W we store a vector v that contains the non-zero weights, and a second, equal-length vector z that encodes the number of zeros before the corresponding entry in v. Each entry of v and z is represented by a four-bit value. If more than 15 zeros appear before a non-zero entry, we add a zero in vector v. For example, we encode the following column

$$[0,0,1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\mathbf{0},0,0,3]$$

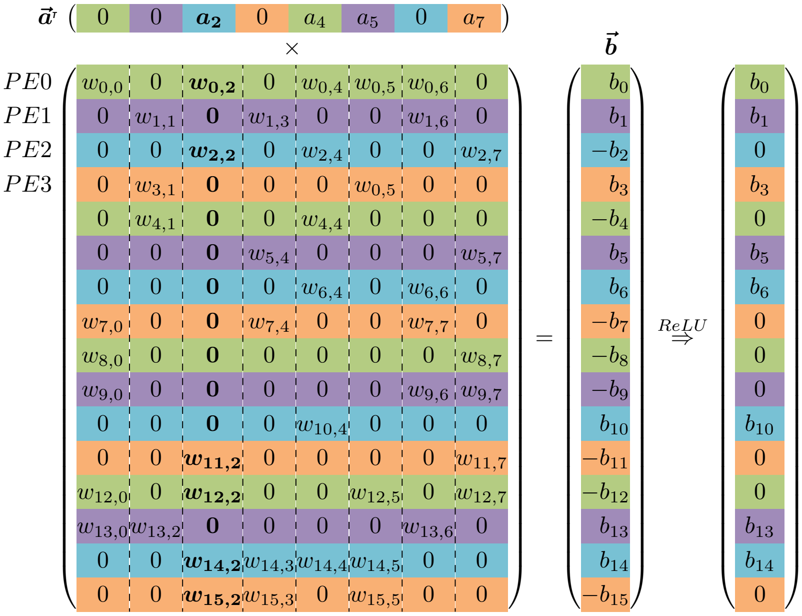
as $v = [1,2,\mathbf{0},3]$, $z = [2,0,\mathbf{15},2]$. v and z of all columns are stored in one large pair of arrays with a pointer vector p pointing to the beginning of the vector for each column. A final entry in p points one beyond the last vector element so that the number of non-zeros in column j (including padded zeros) is given by $p_{j+1} - p_j$.

Storing the sparse matrix by columns in CSC format makes it easy to exploit activation sparsity. We simply multiply each non-zero activation by all of the non-zero elements in its corresponding column.

6.2.3 Parallelization

We distribute the matrix and parallelize our matrix-vector computation by interleaving the rows of the matrix W over multiple processing elements (PEs). With N PEs, PE$_k$ holds all rows W_i, output activations b_i, and input activations a_i for which $i \pmod{N} = k$. The portion of column W_j in PE$_k$ is stored in the CSC format described in Section 6.2.2 but with the zero counts referring only to zeros in the subset of the column in this PE. Each PE has its own v, x, and p arrays that encode its fraction of the sparse matrix.

Figure 6.2 shows an example of multiplying an input activation vector a (of length 8) by a 16×8 weight matrix W, yielding an output activation vector b (of length 16) on $N = 4$ PEs. The elements of a, b, and W are color coded with their PE assignments. Each PE owns 4 rows of W, 2 elements of



$$\vec{a} \; \begin{pmatrix} 0 & 0 & \boldsymbol{a_2} & 0 & a_4 & a_5 & 0 & a_7 \end{pmatrix}$$

$$\times$$

$$\begin{array}{c} PE0 \\ PE1 \\ PE2 \\ PE3 \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{pmatrix}
w_{0,0} & 0 & \boldsymbol{w_{0,2}} & 0 & w_{0,4} & w_{0,5} & w_{0,6} & 0 \\
0 & w_{1,1} & \mathbf{0} & w_{1,3} & 0 & 0 & w_{1,6} & 0 \\
0 & 0 & \boldsymbol{w_{2,2}} & 0 & w_{2,4} & 0 & 0 & w_{2,7} \\
0 & w_{3,1} & \mathbf{0} & 0 & 0 & w_{0,5} & 0 & 0 \\
0 & w_{4,1} & \mathbf{0} & 0 & w_{4,4} & 0 & 0 & 0 \\
0 & 0 & \mathbf{0} & w_{5,4} & 0 & 0 & 0 & w_{5,7} \\
0 & 0 & \mathbf{0} & 0 & w_{6,4} & 0 & w_{6,6} & 0 \\
w_{7,0} & 0 & \mathbf{0} & w_{7,4} & 0 & 0 & w_{7,7} & 0 \\
w_{8,0} & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & w_{8,7} \\
w_{9,0} & 0 & \mathbf{0} & 0 & 0 & 0 & w_{9,6} & w_{9,7} \\
0 & 0 & \mathbf{0} & 0 & w_{10,4} & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{w_{11,2}} & 0 & 0 & 0 & 0 & w_{11,7} \\
w_{12,0} & 0 & \boldsymbol{w_{12,2}} & 0 & 0 & w_{12,5} & 0 & w_{12,7} \\
w_{13,0} & w_{13,2} & \mathbf{0} & 0 & 0 & 0 & w_{13,6} & 0 \\
0 & 0 & \boldsymbol{w_{14,2}} & w_{14,3} & w_{14,4} & w_{14,5} & 0 & 0 \\
0 & 0 & \boldsymbol{w_{15,2}} & w_{15,3} & 0 & w_{15,5} & 0 & 0
\end{pmatrix} = \overset{\vec{b}}{\begin{pmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7 \\ -b_8 \\ -b_9 \\ b_{10} \\ -b_{11} \\ -b_{12} \\ b_{13} \\ b_{14} \\ -b_{15} \end{pmatrix}} \overset{ReLU}{\Rightarrow} \begin{pmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \\ 0 \\ 0 \\ b_{10} \\ 0 \\ 0 \\ b_{13} \\ b_{14} \\ 0 \end{pmatrix}$$

Figure 6.2: Matrix W and vectors a and b are interleaved over 4 PEs. Elements of the same color are stored in the same PE.

Virtual Weight	$W_{0,0}$	$W_{8,0}$	$W_{12,0}$	$W_{4,1}$	$W_{0,2}$	$W_{12,2}$	$W_{0,4}$	$W_{4,4}$	$W_{0,5}$	$W_{12,5}$	$W_{0,6}$	$W_{8,7}$	$W_{12,7}$
Relative Row Index	0	1	0	1	0	2	0	0	0	2	0	2	0
Column Pointer	0	3	4	6	6	8	10	11	13				

Figure 6.3: Memory layout for the relative indexed, indirect weighted and interleaved CSC format, corresponding to PE_0 in Figure 6.2.

a, and 4 elements of b.

We perform the sparse matrix × sparse vector operation by scanning vector a to find its next non-zero value a_j and broadcasting a_j along with its index j to all PEs. Each PE then multiplies a_j by the non-zero elements in its portion of column W_j — accumulating the partial sums in accumulators, one for each element of the output activation vector b. In the CSC representation, these non-zeros weights are stored contiguously, so each PE simply walks through its v array from location p_j to $p_{j+1} - 1$ to load the weights. To address the output accumulators, the row number i corresponding to each weight W_{ij} is generated by keeping a running sum of the entries of the x array.

In the example in Figure 6.2, the first non-zero is a_2 on PE_2. The value a_2 and its column index 2 is broadcast to all PEs. Each PE then multiplies a_2 by every non-zero in its portion of column 2. PE_0 multiplies a_2 by $W_{0,2}$ and $W_{12,2}$; PE_1 has all zeros in column 2, and so it performs no multiplications; PE_2 multiplies a_2 by $W_{2,2}$ and $W_{14,2}$ and so on. The result of each product is

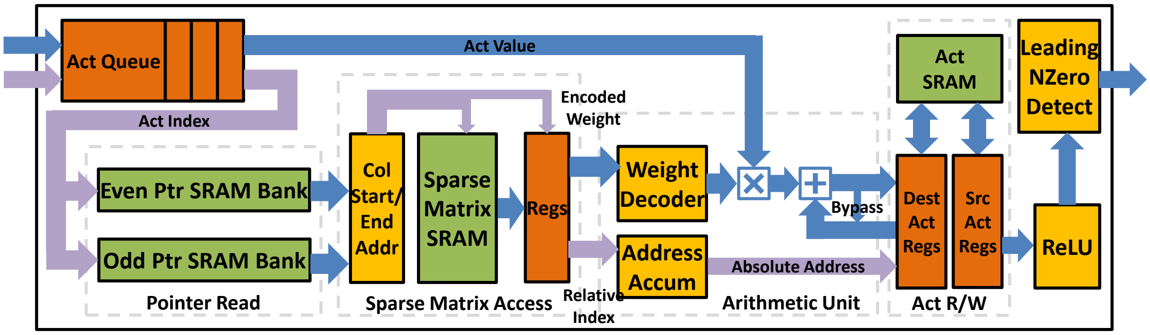


Figure 6.4: The architecture of the processing element of EIE.

summed into the corresponding row accumulator. For example, PE_0 computes $b_0 = b_0 + W_{0,2}a_2$ and $b_{12} = b_{12} + W_{12,2}a_2$. The accumulators are initialized to zero before each layer computation.

The interleaved CSC representation facilitates exploitation of both the dynamic sparsity of activation vector a and the static sparsity of the weight matrix W. We exploit activation sparsity by broadcasting only non-zero elements of input activation a. Columns corresponding to zeros in a are completely skipped. The interleaved CSC representation allows each PE to quickly find the non-zeros in each column to be multiplied by a_j. This organization also keeps all the computation local to a PE except for broadcasting the input activations. The interleaved CSC representation of the matrix in Figure 6.2 is shown in Figure 6.3.

Note that this process may suffer load imbalance because each PE may have a different number of non-zeros in a particular column. We have partially solved the load-balancing problem from the algorithm perspective in Chapter 3 by load-balance-aware pruning, which produces globally balanced workload for the matrix. For each column of the matrix, we will see in Section 6.3 how the effect of this load imbalance can be reduced using hardware queues.

6.3 Hardware Implementation

We present the architecture of EIE in Figure 6.4. Almost all the computation in EIE is local to the PEs except for the collection of non-zero input activations that are broadcast to all PEs. However, the computation of the activation collection and broadcast is not on the critical path because most PEs take many cycles to consume each input activation. The flow of data through the EIE, and the blocks that process this flow are described below.

Activation Queue and Load Balancing. A non-zero activation enters the PE and starts its operations on one column. The sparsity of the column data can cause load imbalance when the number of multiply accumulation operations performed by every PE is different: those PEs with more non-zero elements have to wait until the PE with the most computation tasks finishes.

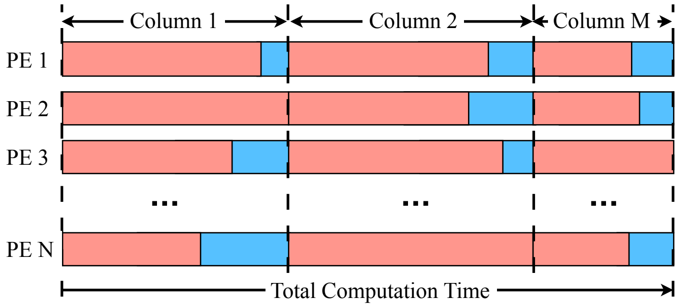


Figure 6.5: Without the activation queue, synchronization is needed after each column. There is load-balance problem within each column, leading to longer computation time.

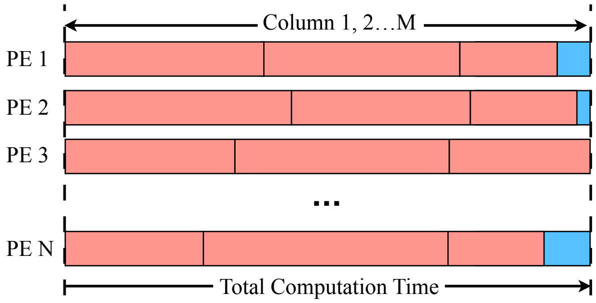


Figure 6.6: With the activation queue, no synchronization is needed after each column, leading to shorter computation time.

There are two kinds of load-imbalance: global and local. Global load imbalance refers to the situation when the non-zero element partition in the *whole matrix* is imbalanced. Local load imbalance refers to the situation when the non-zero element partition in *one single column* of the matrix is imbalanced. Global load-imbalance can be solved by load-balance-aware pruning, as discussed in Chapter 3. Local load-imbalance is generally solved using a queue to buffer up pending work and decouple the parallel execution units, and EIE uses this technique. Thus each PE uses an activation queue to build up a backlog of work to even out load imbalance that may arise because the number of non-zeros in a given column j may vary from PE to PE.

With an activation queue in each PE, the central control unit can broadcast a new activation as long as no activation queue is full, and each PE processes the activation at the head of its queue unless its queue is empty.

The benefit of the activation queue is illustrated in Figure 6.5 and Figure 6.6. Without the activation queue, all the PEs need to synchronize after finishing a column. Thus, fast PE has to wait for slow PE, leading to bubbles in the pipeline (shown in blue). With the activation queue, the PEs can directly consume the workload from the queue and won't be blocked by slow PEs. As a result, bubble cycles can only occur when the FIFO is either full or empty. Longer the activation queues reduce bubbles but require mode resources. In Section 6.5 we quantitatively measure this relationship between the performance and the depth of the activation queue.

Pointer Read Unit. Once an activation is read from the queue, the index j of the entry is used to look up the start and end pointers p_j and p_{j+1} for the v and x arrays for column j. To allow both pointers to be read in one cycle using single-ported SRAM arrays, we store pointers in two SRAM banks, one for even j, and one for odd j and use the LSB of the address to select between banks. p_j and p_{j+1} will always be in different banks. EIE pointers are 16-bits in length.

Sparse Matrix Read Unit. The sparse-matrix read unit uses pointers p_j and p_{j+1} to read the non-zero elements (if any) of this PE's slice of column I_j from the sparse-matrix SRAM. Each entry in the SRAM is 8-bits in length and contains one 4-bit element of v and one 4-bit element of x. For efficiency (see Section 6.5) the PE's slice of encoded sparse matrix I is stored in a 64-bit-wide SRAM. Thus eight entries are fetched on each SRAM read. The high 13 bits of the current pointer p selects a SRAM row, and the low 3-bits select one of the eight entries in that row. A single (v, x) entry is provided to the arithmetic unit each cycle.

Arithmetic Unit. The arithmetic unit receives a (v, x) entry from the sparse matrix read unit and performs the multiply-accumulate operation $b_x = b_x + v \times a_j$. Index x is used to index an accumulator array (the destination activation registers), while v is multiplied by the activation value at the head of the activation queue. Because v is stored in 4-bit encoded form, it is first expanded to a 16-bit fixed-point number via a table lookup. A bypass path is provided to route the output of the adder to its input if the same accumulator is selected on two adjacent cycles.

Activation Read/Write Unit. Once the computation of b_x is complete, it is passed to the Activation Read/Write Unit. This unit is double buffered — it contains two register files — so the unit can read from the results of the last layer, while it stores the current results. Like all double buffer systems, the source and destination register files exchange their role for next layer. Thus, no additional data transfer is needed to support multi-layer feed-forward computation.

The finite size of the register file limits the max size of the vector that can be produced: each activation register file holds 64 activations, which is sufficient to accommodate 4K activations across 64 PEs. When the activation vector has a length greater than 4K, the M×V is completed in batches of length 4K. Each PE has 2KB of SRAM sitting behind the register file as a second level buffer.

Distributed Leading Non-Zero Detection. To reduce data movement, the output activation are left distributed across the PEs, so when they become the input activation to the next layer, they are still distributed across the PEs. To take advantage of the input vector sparsity, we use leading non-zero detection logic to select the first non-zero result, and because the inputs are physically distributed, this detector must be distributed as well. Each group of 4 PEs does a local leading non-zero detection on their input activation. The result is then sent to a Leading Non-zero Detection Node (LNZD Node). Each LNZD node finds the next non-zero activation across its four children and sends this result up to the quadtree. The quadtree is arranged so that wire lengths only grows logarithmically as we add more PEs. At the root LNZD Node, the selected non-zero activation is broadcast back to all the PEs via a separate wire placed in an H-tree.

Central Control Unit. The Central Control Unit (CCU) is the root LNZD Node. It communicates with the master, for example a CPU, and monitors the state of every PE by setting the control registers. There are two modes in the Central Unit: I/O and Computing. In I/O mode, all of the PEs are idle during which the activations and weights in every PE can be accessed by a DMA connected with the Central Unit. This is a one-time cost. In Compute mode, the CCU repeatedly collects a non-zero value from the LNZD quadtree and broadcasts this value to all PEs. This process continues until the input length is exceeded. By setting the input length and starting address of pointer array, EIE is instructed to execute different layers.

6.4 Evaluation Methodology

Simulator, RTL and Layout. We implemented a custom cycle-accurate C++ simulator for the accelerator. Each hardware module is abstracted as an object that implements two abstract methods: propagate and update, corresponding to combinational logic and the flip-flop in RTL. The simulator is used for design space exploration. It also serves as the golden model for RTL verification.

To measure the area, power, and critical path delay, we implemented the RTL of EIE in Verilog. The RTL is verified against the cycle-accurate simulator. Then we synthesized EIE using the Synopsys Design Compiler (DC) under the 45nm GP standard VT library at the worst case PVT corner. We placed and routed the PE using the Synopsys IC compiler (ICC). We used Cacti [131] to get SRAM area and energy numbers. We annotated the toggle rate from the RTL simulation to the gate-level netlist, which was dumped to switching activity interchange format (SAIF), and estimated the power using Prime-Time PX.

Comparison Baseline. We compared EIE with three different off-the-shelf computing units: CPU, GPU, and mobile GPU.

1) CPU. We used Intel Core i-7 5930k CPU, a Haswell-E class processor, that has been used in NVIDIA Digits Deep Learning Dev Box as a CPU baseline. To run the benchmark on CPU, we used MKL CBLAS GEMV to implement the original dense model and MKL SPBLAS CSRMV for the compressed sparse model. CPU socket and DRAM power are as reported by the `pcm-power` utility provided by Intel.

2) GPU. We used NVIDIA Maxwell Titan X GPU as our baseline using `nvidia-smi` utility to report the power. To run the benchmark, we used cuBLAS GEMV to implement the original dense layer. For the compressed sparse layer, we stored the sparse matrix in CSR format and used cuSPARSE CSRMV kernel, which is optimized for sparse matrix-vector multiplication on GPUs.

3) Mobile GPU. We used NVIDIA Tegra K1 that has 192 CUDA cores as our mobile GPU

Table 6.1: Benchmark from state-of-the-art DNN models

Layer	Size	Weight%	Act%	FLOP%	Description
Alex-6	9216, 4096	9%	35.1%	3%	Compressed AlexNet [2] for large scale image classification
Alex-7	4096, 4096	9%	35.3%	3%	
Alex-8	4096, 1000	25%	37.5%	10%	
VGG-6	25088, 4096	4%	18.3%	1%	Compressed VGG-16 [3] for large scale image classification and object detection
VGG-7	4096, 4096	4%	37.5%	2%	
VGG-8	4096, 1000	23%	41.1%	9%	
NT-We	4096, 600	10%	100%	10%	Compressed NeuralTalk [7] with RNN and LSTM for automatic image captioning
NT-Wd	600, 8791	11%	100%	11%	
NTLSTM	1201, 2400	10%	100%	11%	

baseline. We used cuBLAS GEMV for the original dense model and cuSPARSE CSRMV for the compressed sparse model. Tegra K1 does not have software interface to report power consumption, so we measured the total power consumption with a power-meter, then assumed 15% AC to DC conversion loss, 85% regulator efficiency and 15% power consumed by peripheral components [114,115] to report the AP+DRAM power for Tegra K1.

Benchmarks. We compared the performance of EIE on two sets of models: the uncompressed DNN model and compressed DNN model. The uncompressed DNN model was obtained from Caffe model zoo [58] and NeuralTalk model zoo [7]; The compressed DNN model is produced as described in [25, 26]. The benchmark networks have nine layers in total and they are obtained from AlexNet, VGGNet, and NeuralTalk. We use the Image-Net dataset [54] and the Caffe [58] deep learning framework as the golden model to verify the correctness of the hardware design.

6.5 Experimental Results

Figure 6.7 shows the layout (after place-and-route) of an EIE processing element. The power/area breakdown is shown in Table 6.2. We brought the critical path delay down to 1.15ns by introducing four pipeline stages to update one activation: codebook lookup and address accumulation (in parallel), output activation read and input activation multiplication (in parallel), shift/add, and output activation write. Activation read and write access a local register and activation bypassing is employed to avoid a pipeline hazard. Using 64 PEs running at 800MHz yields a performance of

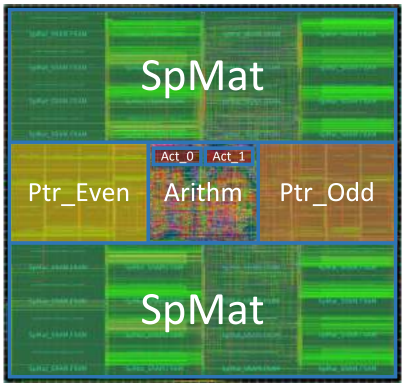


Figure 6.7: Layout of the processing element of EIE.

Table 6.2: The implementation results of one PE in EIE and the breakdown by component type and by module. The critical path of EIE is 1.15 ns.

	Power (mW)	(%)	**Area (μm^2)**	(%)
Total	9.157		638,024	
memory	5.416	(59.15%)	594,786	(93.22%)
clock network	1.874	(20.46%)	866	(0.14%)
register	1.026	(11.20%)	9,465	(1.48%)
combinational	0.841	(9.18%)	8,946	(1.40%)
filler cell			23,961	(3.76%)
Act_queue	0.112	(1.23%)	758	(0.12%)
PtrRead	1.807	(19.73%)	121,849	(19.10%)
SpmatRead	4.955	(54.11%)	469,412	(73.57%)
ArithmUnit	1.162	(12.68%)	3,110	(0.49%)
ActRW	1.122	(12.25%)	18,934	(2.97%)
filler cell			23,961	(3.76%)

102 GOP/s. Considering 10× weight sparsity and 3× activation sparsity, this requires a dense DNN accelerator at 3TOP/s to have equivalent application throughput.

The total SRAM capacity (Spmat+Ptr+Act) of each EIE PE is 162KB. The activation SRAM is 2KB. The Spmat SRAM is 128KB and stores the compressed weights and indices. Each weight is 4bits, and each index is 4bits. Weights and indices are grouped to 8bits and addressed together. The Spmat access width is optimized at 64 bits (discussed in Section 6.5.3). The Ptr SRAM is 32KB storing the pointers in the CSC format. In the steady state, both Spmat SRAM and Ptr SRAM are accessed every $64/8 = 8$ cycles. The area and power are dominated by SRAM; the ratio is 93% and 59% , respectively. Each PE is $0.638mm^2$ consuming $9.157mW$. Each group of 4 PEs needs an LNZD unit for nonzero detection. A total of 21 LNZD units are needed for 64 PEs $(16 + 4 + 1 = 21)$. The synthesized result shows that one LNZD unit takes only $0.023mW$ and an area of $189um^2$, less

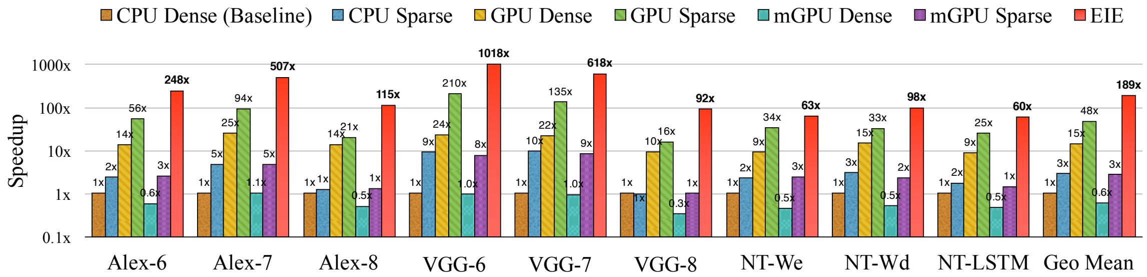


Figure 6.8: Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

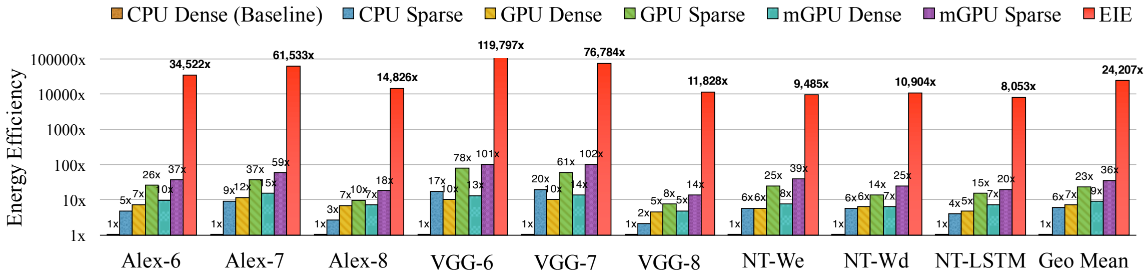


Figure 6.9: Energy efficiency of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

than 0.3% of a PE.

6.5.1 Performance

We compared EIE against CPU, desktop GPU, and the mobile GPU on nine benchmarks selected from AlexNet, VGG-16, and Neural Talk. The overall results are shown in Figure 6.8. There are seven columns for each benchmark, comparing the computation time of EIE on the compressed network over CPU / GPU / TK1 on the uncompressed / compressed network. Time is normalized to CPU. EIE significantly outperforms the general purpose hardware and is, on average, 189×, 13×, 307× faster than CPU, GPU, and mobile GPU, respectively.

EIE's theoretical computation time is calculated by dividing workload GOPs by peak throughput. The actual computation time is around 10% more than the theoretical computation time due to load imbalance. In Fig. 6.8, the comparison with CPU / GPU / TK1 is reported using actual computation time. The wall clock times of CPU / GPU / TK1/ EIE for all benchmarks are shown in Table 6.3.

EIE is targeting extremely latency-focused applications, which require real-time inference. Since assembling a batch adds significant amounts of latency, we consider the case when batch size = 1 when benchmarking the performance and energy efficiency with CPU and GPU (Figure 6.8). As a

Table 6.3: Wall clock time comparison between CPU, GPU, mobile GPU and EIE. The batch processing time has been divided by the batch size. Unit: μs

Platform	Batch Size	Matrix Type	AlexNet			VGG16			NT-LSTM		
			FC6	FC7	FC8	FC6	FC7	FC8	We	Wd	LSTM
CPU (Core-i7 5930k)	1	dense	7516.2	6187.1	1134.9	35022.8	5372.8	774.2	605.0	1361.4	470.5
		sparse	3066.5	1282.1	890.5	3774.3	545.1	777.3	261.2	437.4	260.0
	64	dense	318.4	188.9	45.8	1056.0	188.3	45.7	28.7	69.0	28.8
		sparse	1417.6	682.1	407.7	1780.3	274.9	363.1	117.7	176.4	107.4
GPU (Titan X)	1	dense	541.5	243.0	80.5	1467.8	243.0	80.5	65	90.1	51.9
		sparse	134.8	65.8	54.6	167.0	39.8	48.0	17.7	41.1	18.5
	64	dense	19.8	8.9	5.9	53.6	8.9	5.9	3.2	**2.3**	**2.5**
		sparse	94.6	51.5	23.2	121.5	24.4	22.0	10.9	11.0	9.0
mGPU (Tegra-K1)	1	dense	12437.2	5765.0	2252.1	35427.0	5544.3	2243.1	1316	2565.5	956.9
		sparse	2879.3	1256.5	837.0	4377.2	626.3	745.1	240.6	570.6	315
	64	dense	1663.6	2056.8	298.0	2001.4	2050.7	483.9	87.8	956.3	95.2
		sparse	4003.9	1372.8	576.7	8024.8	660.2	544.1	236.3	187.7	186.5
EIE	**Theor. Time**		**28.1**	**11.7**	**8.9**	**28.1**	**7.9**	**7.3**	**5.2**	13.0	6.5
	Actual Time		**30.3**	**12.2**	**9.9**	**34.4**	**8.7**	**8.4**	**8.0**	13.9	7.5

comparison, we also provided the result for batch size = 64 in Table 6.3. EIE can not handle batch size larger than one; it can handle one input at a time. We relaxed this constraint on the follow-up design, ESE [29], by supporting batching.

The GOP/s required for EIE to achieve the same application throughput (Frames/s) is much lower than competing approaches because EIE exploits sparsity to eliminate 97% of the GOP/s performed by dense approaches. 3 TOP/s on an uncompressed network requires only 100 GOP/s on a compressed network. EIE's throughput is scalable to over 256 PEs. Without EIE's dedicated logic, however, model compression by itself applied on a CPU/GPU yields only 3× speed up.

6.5.2 Energy

In Figure 6.9, we report the energy efficiency on different benchmarks. There are 7 columns for each benchmark, comparing the energy efficiency of EIE on a compressed network over CPU / GPU / TK1 on a uncompressed / compressed network. Energy is obtained by multiplying computation time and total measured power as described in section 6.4.

EIE consumes, on average, $24,000\times$, $3,400\times$ and $2,700\times$ less energy compared to CPU, GPU and the mobile GPU respectively. This three order of magnitude energy savings derives from three factors: first, the required energy per memory read is saved (SRAM over DRAM): using a compressed network model enables state-of-the-art neural networks to fit in on-chip SRAM, reducing energy consumption by 120× compared to fetching a dense uncompressed model from DRAM (Figure 6.9). Second, the number of required memory reads is reduced. The compressed DNN model has 10% of the weights where each weight is quantized to only 4 bits. Last, taking advantage of the activation sparsity saved 65.14% of redundant computation cycles. Multiplying those factors $120 \times 10 \times 8 \times 3$ gives a $28,800\times$ theoretical energy savings. Our actual savings are about 10× less than this number because of index overhead and because EIE is implemented in 45nm technology compared to the

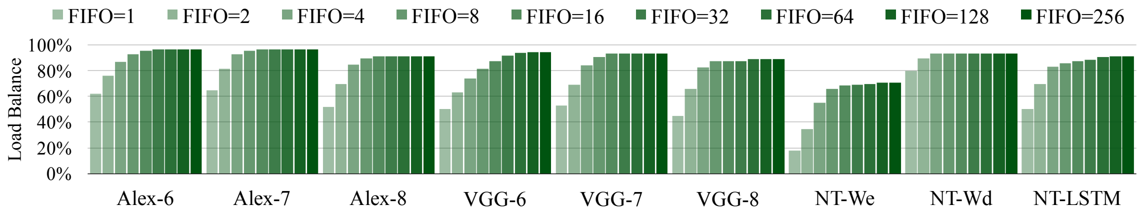


Figure 6.10: Load efficiency improves as FIFO size increases. When FIFO deepth>8, the marginal gain quickly diminishes. So we choose FIFO depth=8.

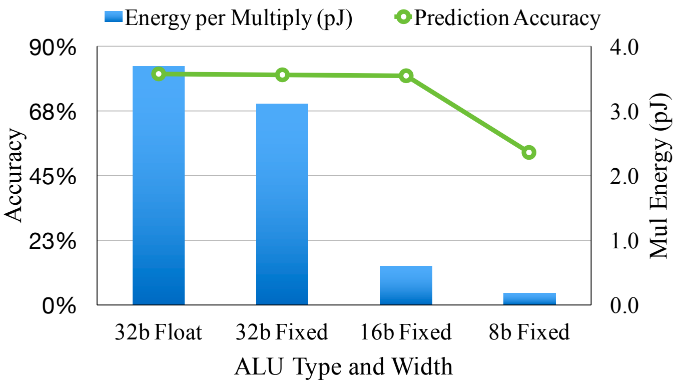


Figure 6.11: Prediction accuracy and multiplier energy with different arithmetic precision.

28nm technology used by the Titan-X GPU and the Tegra K1 mobile GPU.

6.5.3 Design Space Exploration

Queue Depth. The activation queue deals with load imbalance between the PEs. A deeper FIFO queue can better decouple the producer and the consumer, but with diminishing returns (Figure 6.10). We varied the FIFO queue depth from 1 to 256 in powers of 2 across nine benchmarks using 64 PEs and measured the load balance efficiency. This efficiency is defined as the number of bubble cycles (due to starvation) divided by total computation cycles. At FIFO size = 1, around half of the total cycles are idle, and the accelerator suffers from severe load imbalance. Load imbalance is reduced as FIFO depth is increased but with diminishing returns beyond a depth of 8. Thus, we choose eight as the optimal queue depth.

Notice the NT-We benchmark has poorer load balance efficiency compared with others. This is because it has only 600 rows. Divided by 64 PEs and considering the 11% sparsity, each PE on average gets a single entry, which is highly susceptible to variation among PEs, leading to load imbalance. Such small matrices are more efficiently executed on 32 or fewer PEs.

Arithmetic Precision. We use 16-bit fixed-point arithmetic, which consumes much less energy than 32-bit fixed-point 32-bit floating-point. At the same time, using 16-bit fixed-point arithmetic

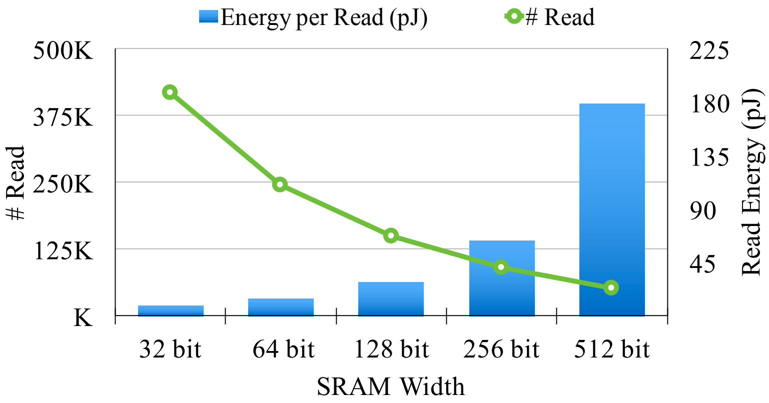


Figure 6.12: SRAM read energy and number of reads benchmarked on AlexNet.

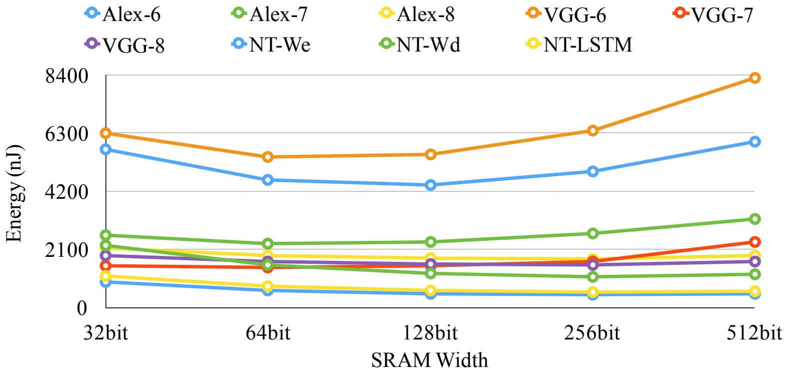


Figure 6.13: Total energy consumed by SRAM read at different bit width.

results in less than 0.5% loss of prediction accuracy: 79.8% compared with 80.3% using 32-bit floating point arithmetic. With 8-bit fixed-point, however, the accuracy dropped to only 53%. The accuracy is measured on ImageNet dataset [54] with AlexNet [2], and the energy is obtained from synthesized RTL under the 45nm process. The trade-off between energy and accuracy is shown in Figure6.11.

SRAM Width. We choose a SRAM with a 64-bit interface to store the sparse matrix (Spmat) since it minimized the total energy. Wider SRAM interfaces reduce the number of total SRAM accesses but increase the energy cost per SRAM read. The experimental trade-off is shown in Figure 6.12 and Figure 6.13. SRAM energy is modeled using Cacti [131] under the 45nm process. SRAM access times are measured by the cycle-accurate simulator on the AlexNet benchmark. As the total energy is shown in Figure 6.13, the minimum total access energy is achieved when SRAM width is 64 bits. Why is this the case? For SRAM width larger than 64 bits, some read data will be wasted: the typical number of activation elements of the FC layer is 4K [2,3]; thus, assuming 64 PEs and 10% density [25], each column in a PE will have 6.4 elements on average. This matches a

64-bit SRAM interface that provides 8 elements. If more elements are fetched and the next column corresponds to zero activation, those elements are wasted.

6.6 Discussion

In this section we discuss the alternative design choices of EIE's workload partitioning, and how we use EIE to handle larger problems (scalability) and a diverse categories of problems (flexibility). Fianlly, we compare EIE with other hardware platforms.

6.6.1 Partitioning

Sparse matrix-vector multiplication (SPMV) is they key component of EIE. We compare three different approaches to partition the workload for sparse matrix-vector multiplication and discuss their pros and cons.

Column Partition. The first approach is to distribute matrix ***columns*** to PEs. Each PE is responsible for a subset of columns and handles the multiplication between its columns of W and the corresponding element of a to get a partial sum of the output vector b.

The benefit of this solution is that each element of a is only associated with one PE — giving full locality for vector a. The drawback is that a reduction operation between PEs is required to obtain the final result, resulting in inter-PE communications. Moving data is expensive and we want to minimize such inter-PE communication. This approach also suffers from load imbalance problem given that vector a is also sparse. Each PE is responsible for a column. PE$_j$ will be completely idle if their corresponding element in the input vector a_j is zero, resulting in idle cycles and a discrepancy between the peak performance and the real performance.

Row Partition. The second approach (ours in EIE) is to distribute matrix ***rows*** to PEs, i.e. each PE is responsible for a subset of rows. A central unit broadcasts one vector element a_j to all PEs. Each PE computes a number of output activations b_i by performing inner products of the corresponding row of W, W_j that is stored in the PE with vector a. The benefit of this solutions is that each element of the output b is only associated with one PE — giving full locality for vector b. The drawback is that vector a needs to be broadcast to all PEs. However, the broadcast is not on the critical path and can be decoupled and overlapped with the computation by introducing the activation queue. The consumer of the activation queue takes much longer to process one entry than the producer, which is the broadcast operation.

Mixed Partition. The third approach combines the previous two approaches by distributing blocks of W to the PEs in a 2D fashion. This solution is more scalable for distributed systems where communication latency cost is significant [132]. In this approach both of the collective communication operations "Broadcast" and "Reduction" are exploited but on a smaller scale and hence this solution

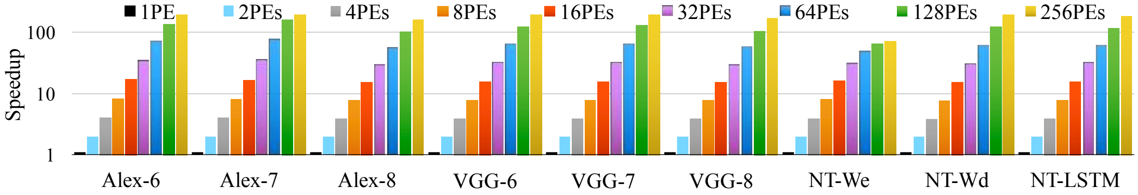


Figure 6.14: System scalability. It measures the speedups with different numbers of PEs. The speedup is near-linear.

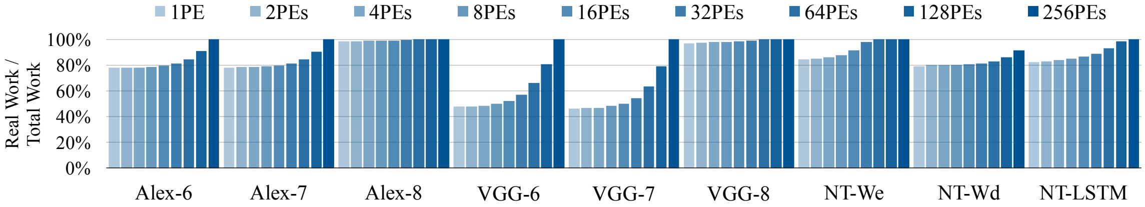


Figure 6.15: As the number of PEs goes up, the number of padding zeros decreases, leading to less padding zeros and less redundant work, thus better compute efficiency.

is more scalable. However, the hybrid solution will suffer from inherent complexity, and still possible load imbalance since multiple PEs sharing the same column might remain idle.

We build our solution based on the second distribution scheme because it can take advantage of the activation sparsity, which is is dynamic. Partitioning by rows allows one activation to be simultaneously processed by different PEs in different rows of the same column. To be specific, EIE performs computations by in-order look-up of non-zeros in a. Each PE gets all the non-zero elements of a in order and performs the inner products by looking-up the matching element that needs to be multiplied by a_j, W_j. This requires the matrix W being stored in CSC format (not CSR format) so the PE can multiply all the elements in the j-th column of W by a_j.

6.6.2 Scalability

As the matrix gets larger, the system can be scaled up by adding more PEs. Each PE has local SRAM storing distinct rows of the matrix without duplication, so the SRAM is efficiently utilized.

Wire delay increases with the square root of the number of PEs. However, this is not a problem in our architecture because EIE only requires one broadcast over the computation of the entire column, which takes many cycles. Consequently, the broadcast is not on the critical path and can be pipelined because the activation queue decouples the producer and the consumer.

Figure 6.14 shows EIE achieves good scalability all the way up to 256 PEs on all benchmarks except NT-We. NT-We is very small (4096 × 600). Dividing the columns of size 600 and sparsity 10% (average of 60 non-zeros per column) to 64 or more PEs causes serious load imbalance.

Figure 6.15 shows the number of padding zeros with different number PEs. A padding zero occurs

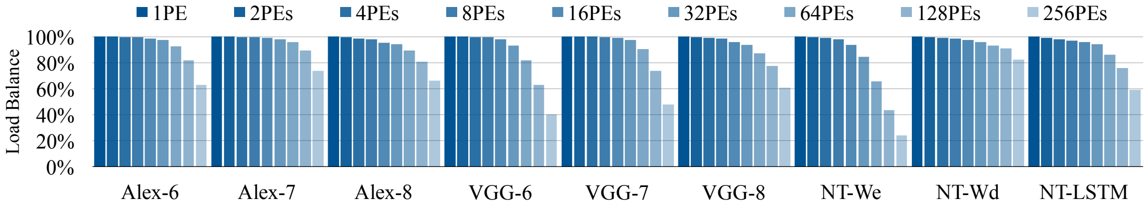


Figure 6.16: Load efficiency is measured by the ratio of stalled cycles over total cycles in ALU. More PEs lead to worse load balance, but less padding zeros and more useful computation.

when the distance between two consecutive non-zero elements in the sparse matrix is larger than 16, the largest number that 4 bits can encode. Padding zeros are considered non-zero and lead to wasted computation. Using more PEs reduces padding zeros because the distance between non-zero elements gets smaller due to matrix partitioning, and 4-bits encoding a max distance of 16 will more likely be enough.

Figure 6.16 shows the load balance with a different number of PEs, measured with FIFO depth equal to 8. With more PEs, load balance becomes worse, but padding zero overhead decreases, which yields constant efficiency for most benchmarks. The scalability result is plotted in Figure 6.14.

6.6.3 Flexibility

EIE is the basic building block for accelerating MLP, CNN, RNN and LSTM. EIE can directly accelerate MLP, since each layer of MLP is a fully-connected layer. EIE has the potential to support 1x1 convolution and 3x3 Winograd convolution by turning the channel-wise reductions into $M \times V$ operations. Winograd convolution saves 2.25× multiplications than direct convolution [133], and for each Winograd patch, the 16 $M \times V$ can be scheduled on an EIE. EIE can also accelerate sparse RNNs and LSTMs, which are widely used in sequence modeling tasks such as speech recognition and natural language processing. We used EIE as the basic building block and designed the efficient speech recognition engine (ESE) [29]. ESE addressed a more general problem of accelerating not only feedforward neural networks but also recurrent neural networks. Implemented on Xilinx XCKU060 FPGA, ESE running on the sparse LSTM achieved 6.2x speedup over the original dense model.

6.6.4 Comparison

Table 6.4 shows the comparison of performance, power, and area on different hardware platforms. The performance is evaluated on the FC7 layer of AlexNet. [1]. We compared six platforms for neural networks: Core-i7 (CPU), Titan X (GPU), Tegra K1 (mobile GPU), A-Eye (FPGA), DaDianNao (ASIC), TrueNorth (ASIC). All other four platforms suffer from low-efficiency during

[1] The FC7 result is not provided for TrueNorth, so we use the TIMIT LSTM result for comparison instead (they differ less than 2×)

Table 6.4: Comparison with existing hardware platforms for DNNs.

Platform	Core-i7 5930K	Maxwell Titan X	Tegra K1	Angel Eye [23]	DaDian Nao [20]	TrueNorth [134]	**EIE ours, 64PE [28]**	**EIE ours, 256PE [28]**
Year	2014	2015	2014	2015	2014	2014	2016	2016
Platform Type	CPU	GPU	mGPU	FPGA	ASIC	ASIC	ASIC	ASIC
Technology	22nm	28nm	28nm	28nm	28nm	28nm	45nm	28nm
Clock(*MHz*)	3500	1075	852	150	606	async	800	1200
Memory type	DRAM	DRAM	DRAM	DRAM	eDRAM	SRAM	SRAM	SRAM
Max model size(*#Params*)	<16G	<3G	<500M	<500M	18M	256M	84M	336M
Quantization Stategy	32-bit float	32-bit float	32-bit float	16-bit fixed	16-bit fixed	1-bit fixed	4-bit fixed	4-bit fixed
Area(mm^2)	356	601	-	-	67.7	430	40.8	63.8
Power(*W*)	73	159	5.1	9.63	15.97	0.18	0.59	2.36
M×V Throughput(*Frames/s*)	162	4,115	173	33	147,938	1,989	81,967	426,230
Area Efficiency(*Frames/s/*mm^2)	0.46	6.85	-	-	2,185	4.63	2,009	6,681
Energy Efficiency(*Frames/J*)	2.22	25.9	33.9	3.43	9,263	10,839	138,927	180,606

matrix-vector multiplication. A-Eye is optimized for CONV layers and all of the parameters are fetched from the external DDR3 memory, making it sensitive to the bandwidth problem. DaDianNao distributes weights on 16 tiles, each tile with 4 eDRAM banks, thus has a peak memory bandwidth of $16 \times 4 \times (1024bit/8) \times 606MHz = 4964GB/s$. Its performance on $M \times V$ is estimated based on the peak memory bandwidth because $M \times V$ is completely memory bound. In contrast, EIE maintains a high throughput for $M \times V$ because, after compression, all weights fit in on-chip SRAM, even for very large networks. With 256 PEs, EIE has 3.25× more throughput than 64 PEs and can hold 336 million parameters, even larger than VGGnet. The right column projected EIE to the same technology (28nm) as the other platforms, with 256 PEs, EIE has 2.9× throughput, 3× area efficiency and 19× power efficiency than DaDianNao.

In previous neural network accelerators, the weights are uncompressed and stored in dense format, making the accelerator constrained by memory size. For example, ShiDianNao [21], which contains 128KB on-chip RAM, can only handle very small DNN models up to 64K parameters, which is three orders of magnitude smaller than the 60 million parameters AlexNet. Such large networks are impossible to fit on chip on ShiDianNao without compression. DaDianNao stores the uncompressed model in eDRAM taking 6.12W memory power and 15.97W total power. EIE stores the compressed model in SRAM taking only 0.35W memory power and only 0.59W total power; DaDianNao cannot exploit the sparsity from weights and activations and they must expand the network to dense form before any operation. It cannot exploit weight sharing either. Using just the compression (and decompressing the data before computation) would reduce DaDianNao total power to around 10W in 28nm, compared to EIE's power of 0.58W in 45nm. For compressed deep networks, previously proposed SPMV accelerators can only exploit the static weight sparsity. They are unable to exploit dynamic activation sparsity (3×), and they are unable to exploit weight sharing (8×); altogether, a 24× energy saving is lost.

Since EIE was published in ISCA'2016, sparse neural network accelerator has been the topic of multiple academic papers and EIE has impacted the industry. These related works are discussed in Chapter 2.

6.7 Conclusion

This chapter has presented EIE, an energy-efficient engine optimized to operate on sparse, compressed deep neural networks. By leveraging sparsity in both the activations and the weights, and taking advantage of weight sharing with extremely narrow weights, EIE reduces the energy needed to compute a typical FC layer by 3,400× compared with GPU. This energy saving comes from four main factors: first, the number of parameters is pruned by 10×; second, weight-sharing reduces the weights to only 4 bits; third, the smaller model can be fetched from SRAM and not DRAM, giving a 120× energy advantage; fourth, since the activation vector is also sparse, only 30% of the matrix columns need to be fetched for a final 3× savings. These savings enable an EIE PE to do 1.6 GOPS in an area of 0.64mm^2 and dissipate only 9mW. 64 PEs can process FC layers of AlexNet at 1.88×10^4 frames/sec. The architecture is scalable from one PE to over 256 PEs with nearly linear scaling of energy and performance. On 9 fully-connected layer benchmarks, EIE outperforms CPU, GPU and mobile GPU by factors of 189×, 13× and 307×, and consumes $24,000\times$, $3,400\times$ and $2,700\times$ less energy than CPU, GPU and mobile GPU, respectively.

Chapter 7

Conclusion

Deep neural networks have revolutionized a wide range of AI applications and are changing our lives. However, deep neural networks are both computationally and memory intensive. Thus they are difficult to deploy on embedded systems with limited computation resources and power budgets. To address this problem, we presented methods and hardware for improving the efficiency of deep learning.

This dissertation focuses on improving the efficiency of deep learning from three aspects: smaller model size by Deep Compression, higher prediction accuracy by DSD regularization, and fast and energy efficient inference hardware by EIE acceleration (Figure 7.1). All these aspects share a common principle: utilizing the sparsity in neural networks for compression, regularization, and acceleration.

Compression. To achieve smaller models, we proposed Deep Compression, which can significantly reduce the storage and energy required to run inference on large neural networks. With proper retraining techniques, model compression doesn't hurt prediction accuracy. Deep Compression is a three-stage pipeline. Chapter 3 details the first step of Deep Compression, pruning the model to remove redundant connections. We also described the retraining method to fully recover the accuracy with the remaining connections. Model pruning can remove >90% of parameters in fully-connected layers and 70% of parameters in convolutional layers without loss of accuracy on ImageNet. This technique was verified on modern neural networks including AlexNet, VGG-16, GoogleNet, SqueezeNet, ResNet-50, and NeuralTalk. The pruning process learns not only the weights but also the network connectivity, much as in the development of the human brain [109] [110].

We presented quantization and weight sharing technique to further compresses the DNN model in Chapter 4. Trained quantization reduces the bit width per parameter. The number of effective weights is limited by having multiple connections share the same weight, and then fine-tune those shared weights. As a result, neural network weights can be represented with only $2 \sim 4$ bits, saving $8\times \sim 16\times$ storage compared to a 32-bit floating point, $2\times \sim 4\times$ compared to a 8-bit integer. Finally,

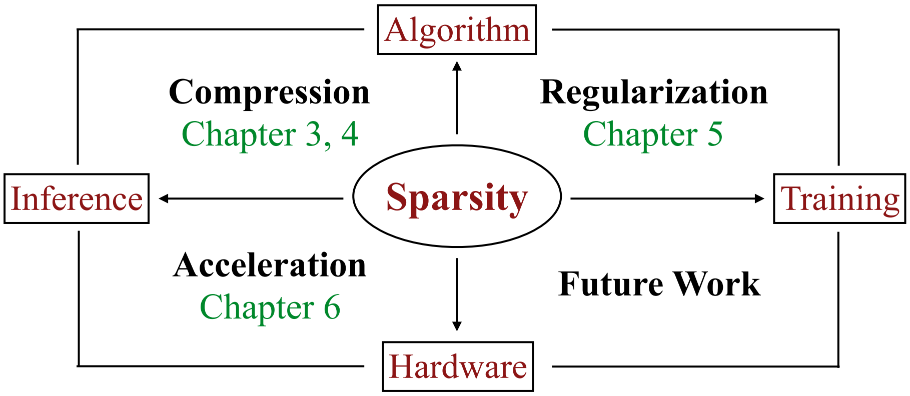


Figure 7.1: Summary of the thesis.

we applied variable-length coding to further compress the model. Combining the above steps, Deep Compression reduces the model sizes by 35×, 49×, 22× and 17× for AlexNet, VGG16, Inception-V3 and ResNet-50, respectively, without loss of accuracy on ImageNet.

The insight of Deep Compression is that neural networks are highly tolerant to approximation. The noisy nature of training data blurs away the need for high precision computation. In fact, redundant parameters and redundant bits are prone to capturing the noise rather than the intended pattern. Hardware designers should be aware of this particular characteristic of deep learning workloads to avoid wasting computation cycles and memory bandwidth. Deep Compression laid the algorithm foundation for the EIE hardware we designed later.

Regularization. Since the compressed model can achieve the same accuracy as the uncompressed model, the original sized model should have the capacity to achieve higher accuracy if optimized properly. In Chapter 5 we proposed the dense-sparse-dense (DSD) training, a new training strategy that can regularize DNN training and improve the accuracy.

DSD starts by training a dense model, then regularizes this model with sparsity-constrained optimization, finally increases the model capacity by restoring and retraining the weights. The final neural network model produced by DSD still has the same architecture and dimensions as the original dense model, and DSD training doesn't incur any inference overhead. DSD training changes the optimization dynamics and improves the accuracy with significant margins. We tested DSD training to seven mainstream CNNs/RNNs/LSTMs and found consistent performance improvements for image classification, image captioning and speech recognition tasks.

Acceleration. Deep Compression significantly reduced the memory bandwidth required for fetching parameters in deep neural networks. However, taking advantage of the compressed DNN model is a non-trivial task. The compressed model brings an irregular computation pattern that entails a variety of challenges such as: (i) how to parallelize the computation and do runtime decompression, rather than having to decompress the model before performing inference; (ii) how to deal with sparsity and avoid pointer-chasing; (iii) how to achieve good load-balance given that different processors have different amounts of non-zero workloads.

In Chapter 6, we systematically addressed these challenges by designing a hardware accelerator called "Efficient Inference Engine" (EIE) to directly execute the compressed DNN. EIE distributes the sparse matrix and parallelizes the computation by interleaving matrix rows over multiple processing elements (PEs). EIE stores the encoded sparse weight matrix W densely in compressed sparse column (CSC) format and uses a leading non-zero detection circuit to perform only non-zero multiplications. The codebook from weight sharing is implemented as a lookup table with registers. A FIFO queue allows each PE to build up a backlog of work, thus evening out load imbalance. EIE fully exploits the sparse, compressed model and yields significant speedup and energy savings, being 13x faster and 3,400x more energy-efficient than a GPU for non-batched work.

Future Work. Deep Compression and EIE Accelerator offer useful insights into the special computational characteristics of deep learning inference, such as strong tolerance for low precision and the need to support sparse computation. It remains future work to investigate the limit of model compression: what is the minimum model capacity we need given a task and data set, what is the minimum amount of computation to achieve targeted accuracy. For a target model size, the product of the number of parameters and bit-width per parameter is constant, so it will be interesting to work out the trade-off between the number of parameters and bit-width per parameter that optimizes accuracy. Similar to the success of SqueezeNet + Deep Compression that produced a network 500× smaller than AlexNet but with the same accuracy [6], figuring out how to systematically design efficient neural network architectures before model compression remains a key challenge.

There is also opportunity to generalize these insights of compression beyond inference to training, and to build efficient hardware for training. The computation requirement for training neural networks is becoming more demanding with the explosion of big-data. Deep learning researchers take days or even weeks to train large-scale neural network models, which bottlenecks their productivity. There are opportunities to improve the training efficiency. For example, generalizing the sparsity and model compression technique from inference to training, which has the potential to reduce the communication bandwidth when exchanging the gradients during distributed training. In the future, we envision that compact and sparse neural networks will be automatically produced at training time. Sparsity support will exist in both training and inference hardware.

Once we have efficient hardware primitives, we hope to support not only inference but also training on edge devices. Future AI applications will be customized to each end user. For example, different users will have a different tone for speech recognition, different sets of friends to do face clustering, etc. Learning on edge devices will meet such customization demand and grantee privacy at the same time. However, mobile devices may not be able to hold large training dataset, nor able to train large models. How to partition the computation between the edge device and the cloud in an efficient way remains an interesting question.

Since Moore's law is slowing down, we are in the post-Moore's Law world where programmers **no longer get more computation** at a constant dollar and power cost every few years. Yet at the same time, the last ImageNet challenge has ended; we are now in the post-ImageNet era when researchers in computer vision and artificial intelligence are now solving more complicated AI problems which **require more computation**. The clash in supply and demand for computation highlights the need for algorithm and hardware co-design. Only by tuning the hardware to the application, and mapping the application to efficient hardware operations will the application performance continue to scale. More efficient hardware will make AI cheaper and more accessible, not only in the labs but also in everyone's lives. We hope the efficient methods and hardware we described in this thesis will open more space and help democratize AI in the future. Further, we hope our examples of algorithm and hardware co-design will be useful not only for deep learning but also for other applications.

Bibliography

[1] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[6] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 1MB model size. *arXiv:1602.07360*, 2016.

[7] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *arXiv:1412.2306*, 2014.

[8] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Ng. Deep speech: Scaling up end-to-end speech recognition. *arXiv, preprint arXiv:1412.5567*, 2014.

[9] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

[10] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[12] Build more intelligent apps with machine learning, https://developer.apple.com/machine-learning.

[13] Guidance: A revolutionary visual sensing system for aerial platforms, https://www.dji.com/guidance.

[14] Giving cars the power to see, think, and learn, http://www.nvidia.com/object/drive-automotive-technology.html.

[15] Mark Horowitz. Energy table for 45nm process, Stanford VLSI wiki.

[16] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. CNP: An fpga-based processor for convolutional networks. In *FPL*, 2009.

[17] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE, 2011.

[18] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 24–35. ACM, 2013.

[19] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM, 2014.

[20] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, December 2014.

[21] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, pages 92–104. ACM, 2015.

[22] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 2016.

[23] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, 2016.

[24] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM, 2010.

[25] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[26] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations*, 2016.

[27] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, et al. DSD: Dense-sparse-dense training for deep neural networks. *International Conference on Learning Representations*, 2017.

[28] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[29] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[30] Ling Zhuo and Viktor K. Prasanna. Sparse Matrix-Vector Multiplication on FPGAs. In *FPGA*, 2005.

[31] J. Fowers and K. Ovtcharov and K. Strauss and E.S. Chung and G. Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *FCCM*, 2014.

[32] Richard Dorrance and Fengbo Ren and Dejan Marković. A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-blas on FPGAs. In *FPGA*, 2014.

[33] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[34] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[35] Jian Zhou and Olga G Troyanskaya. Predicting effects of noncoding variants with deep learning–based sequence model. *Nature methods*, 12(10):931, 2015.

[36] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11), 2017.

[37] Ashish Bora, Ajil Jalal, Eric Price, and Alexandros G Dimakis. Compressed sensing using generative models. *arXiv preprint arXiv:1703.03208*, 2017.

[38] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint arXiv:1609.04802*, 2016.

[39] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

[40] Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, and Alexei A Efros. Generative visual manipulation on the natural image manifold. In *European Conference on Computer Vision*, pages 597–613. Springer, 2016.

[41] Qifeng Chen and Vladlen Koltun. Photographic image synthesis with cascaded refinement networks. *arXiv preprint arXiv:1707.09405*, 2017.

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

[43] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *JMLR*, 12:2493–2537, 2011.

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al.

Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[45] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 3357–3364. IEEE, 2017.

[46] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

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

[48] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, page 0278364917710318, 2016.

[49] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

[50] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[52] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. *AT&T Labs. Available: http://yann.lecun.com/exdb/mnist*, 2010.

[53] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[54] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition. 2009.*

[55] Micah Hodosh, Peter Young, and Julia Hockenmaier. Framing image description as a ranking task: Data, models and evaluation metrics. *Journal of Artificial Intelligence Research*, 47:853–899, 2013.

[56] wikipedia. Bleu score, https://en.wikipedia.org/wiki/bleu.

[57] John S Garofolo, Lori F Lamel, William M Fisher, Jonathon G Fiscus, and David S Pallett. Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon technical report n*, 93, 1993.

[58] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[59] Tensors and dynamic neural networks in python with strong gpu acceleration, http://www.pytorch.org.

[60] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[61] NVIDIA DIGITS DevBox, https://developer.nvidia.com/devbox.

[62] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013.

[63] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.

[64] Babak Hassibi, David G Stork, et al. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, pages 164–164, 1993.

[65] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, pages 1269–1277, 2014.

[66] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pages 442–450, 2015.

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

[68] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[69] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1131–1135. IEEE, 2015.

[70] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE, 2014.

[71] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[72] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788*, 2015.

[73] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *International Conference on Learning Representations*, 2017.

[74] Sajid Anwar and Wonyong Sung. Compact deep convolutional neural networks with coarse pruning. *arXiv preprint arXiv:1610.09639*, 2016.

[75] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv preprint arXiv:1611.05128*, 2016.

[76] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *arXiv preprint arXiv:1707.06168*, 2017.

[77] Sharan Narang, Gregory Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.

[78] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient DNNs. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.

[79] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

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

[81] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *arXiv preprint arXiv:1708.06519*, 2017.

[82] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342*, 2017.

[83] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *J. Emerg. Technol. Comput. Syst.*, 13(3):32:1–32:18, February 2017.

[84] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[85] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 2861–2865. IEEE, 2017.

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

[87] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3105–3113, 2015.

[88] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016.

[89] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

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

[91] Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[92] Facebook Developer Conference 2017. Delivering Real-Time AI In the Palm of Your Hand. https://developers.facebook.com/videos/f8-2017/delivering-real-time-ai-in-the-palm-of-your-hand at 11'30".

[93] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[94] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 15:1929–1958, 2014.

[95] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *ICML*, pages 1058–1066, 2013.

[96] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[97] Xiaojie Jin, Xiaotong Yuan, Jiashi Feng, and Shuicheng Yan. Training skinny deep neural networks with iterative hard thresholding methods. *arXiv preprint arXiv:1607.05423*, 2016.

[98] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 27–39. IEEE Press, 2016.

[99] Ali Shafiee and et al. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ISCA*, 2016.

[100] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 255–266. IEEE Press, 2016.

[101] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 1–13. IEEE, 2016.

[102] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose MigueÍ Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. *ISCA*, 2016.

[103] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[104] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.

[105] Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing. *arXiv preprint arXiv:1705.00125*, 2017.

[106] Sicheng Li, Wei Wen, Yu Wang, Song Han, Yiran Chen, and Hai Li. An fpga design framework for cnn sparsification and acceleration. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 28–28. IEEE, 2017.

[107] NVIDIA. NVIDIA GPU Technology Conference (GTC) 2017 Keynote, https://youtu.be/hpxtsvu1huq at 2:00:40.

[108] NEC. NEC accelerates machine learning for vector computers, http://www.nec.com/en/press/201707/images/0302-01-01.pdf.

[109] JP Rauschecker. Neuronal mechanisms of developmental plasticity in the cat's visual system. *Human neurobiology*, 3(2):109–114, 1983.

[110] Christopher A Walsh. Peter huttenlocher (1931-2013). *Nature*, 502(7470):172–172, 2013.

[111] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, pages 3320–3328, 2014.

[112] Yangqing Jia, et al. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[113] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[114] NVIDIA. Whitepaper: GPU-based deep learning inference: A performance and power analysis.

[115] NVIDIA. Technical brief: NVIDIA jetson TK1 development kit bringing GPU-accelerated computing to embedded systems.

[116] Jan Van Leeuwen. On the construction of huffman trees. In *ICALP*, pages 382–410, 1976.

[117] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[118] Glen G. Langdon. Arithmetic coding. *IBM J. Res. Develop*, 23:149–162, 1979.

[119] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[120] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[121] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

[122] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. *arXiv preprint arXiv:1412.7149*, 2014.

[123] Maxwell D Collins and Pushmeet Kohli. Memory bounded deep convolutional networks. *arXiv preprint arXiv:1412.1442*, 2014.

[124] Yangqing Jia. BVLC caffe model zoo. http://caffe.berkeleyvision.org/model_zoo.

[125] Facebook. Facebook.ResNet.Torch. https://github.com/facebook/fb.resnet.torch, 2016.

[126] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

[127] Chii-Ruey Hwang. Simulated annealing: theory and applications. *Acta Applicandae Mathematicae*, 12(1):108–111, 1988.

[128] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[129] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[130] Richard Wilson Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, UC Berkeley, 2003.

[131] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[132] Victor Eijkhout. *LAPACK working note 50: Distributed sparse data structures for linear algebra operations*. 1992.

[133] Andrew Lavin. Fast algorithms for convolutional neural networks. *arXiv:1509.09308*, 2015.

[134] Steven K Esser and et al. Convolutional networks for fast, energy-efficient neuromorphic computing. *arXiv:1603.08270*, 2016.