Energy Efficiency Enhancement for CNN-based Deep Mobile Sensing

Ruitao Xie, Xiaohua Jia, Lu Wang, and Kaishun Wu

Abstract

Recently, deep learning has been used to tackle mobile sensing problems, and the inference phase of deep learning is preferred to be run on mobile devices for speedy responses. However, mobile devices are resource-constrained platforms for both computation and power. Moreover, an inference task with deep learning involves tens of billions of mathematical operations and tens of millions of parameter reads. Thus, it is a critical issue to reduce the energy consumption of deep learning inference algorithms. In this article, we survey various energy reduction approaches, and classify them into three categories: the compressing neural network model, minimizing the data transfer required in computation, and offloading workloads. Moreover, we simulate and compare three techniques of model compression, by applying them to an object recognition problem.

Introduction

Modern mobile devices are equipped with various sensors such as GPS, camera, accelerator, gyroscope, and so on. Based on these sensors, many mobile sensing applications are developed, such as tracking, locating, object recognition, and so on. Nowadays, deep learning is used to tackle these mobile sensing problems [1–6], since it can achieve higher accuracy than traditional methods. The big success can be attributed to three main reasons. First, neural networks containing tens of millions of parameters have powerful generalization capability. Second, a large volume of sensing data enables the training of neural networks to converge. Third, automatic feature extraction from raw data avoids manual feature designing.

The computation of a deep learning problem consists of training and inference. For training, the parameters of a neural network are adjusted such that the prediction error is minimized. This process is not only data-intensive but also compute-intensive, and thus has to run on a server or a cluster of servers. For inference, the well trained model receives new data, and yields a prediction result.

When using a neural network for an inference task, we desire that an accurate result is produced with low latency and with little energy consumed. Accuracy is already determined by model design and training before starting inference. Latency and energy consumption are a pair of opposing objectives, and highly depend on where the inference task runs. On the one hand, if it runs on the server side, then it has to be called remotely from the mobile device; as a result, latency would be high but energy consumption would be low. Here, we consider only the energy consumed by the mobile device rather than by the server, since the former is energy-constrained while the latter is not. On the other hand, if the inference task runs on the mobile device side, then latency would be low but energy consumption would be huge for energy-constrained mobile devices.

The reason for huge energy consumption is that each inference, a forward pass in a neural network, involves tens of billions of arithmetic operations and tens of millions of parameter reads [7]. Moreover, this inference may be called frequently by mobile sensing applications, draining energy very quickly.

To obtain speedy responses, it is preferred to place inference on the mobile device side. As a result, the most challenging problem is *how to reduce the energy consumption involved in neural network inference*. In this article, we survey various approaches to improve energy efficiency, and classify them into three categories: the compressing neural network model, minimizing the data transfer required in computation and offloading workloads. The first two approaches aim to reduce the energy cost by memory access. The compressing model decreases the amount of model parameters; minimizing data transfer reduces the amount of data reads. The offloading approach decreases energy by utilizing both local low-power processors and remote energy-abundant devices.

We make two contributions in this article:

- Our work is the first to survey all three kinds of approaches, which are highly relevant to energy-critical applications.
- We simulate and compare the techniques of model compression, by applying them to an object recognition problem.

Note that we only consider inference tasks using a convolutional neural network (CNN). Those tasks using a recurrent neural network (RNN), which involve recurrent feedback connections, are beyond the scope of this article. They are more challenging, because it is unknown in advance how many feedback cycles will take place. It is however useful in some RNN-CNN hybrid networks where the parameter count of the CNN dominates (e.g., networks for Visual Question Answering).

Ruitao Xie, Lu Wang and Kaishun Wu (corresponding author) are with Shenzhen University; Kaishun Wu is also with Peng Cheng Laboratory; Xiaohua Jia is with City University of Hong Kong.

Digital Object Identifier: 10.1109/MWC.2019.1800321

1536-1284/19/$25.00 © 2019 IEEE

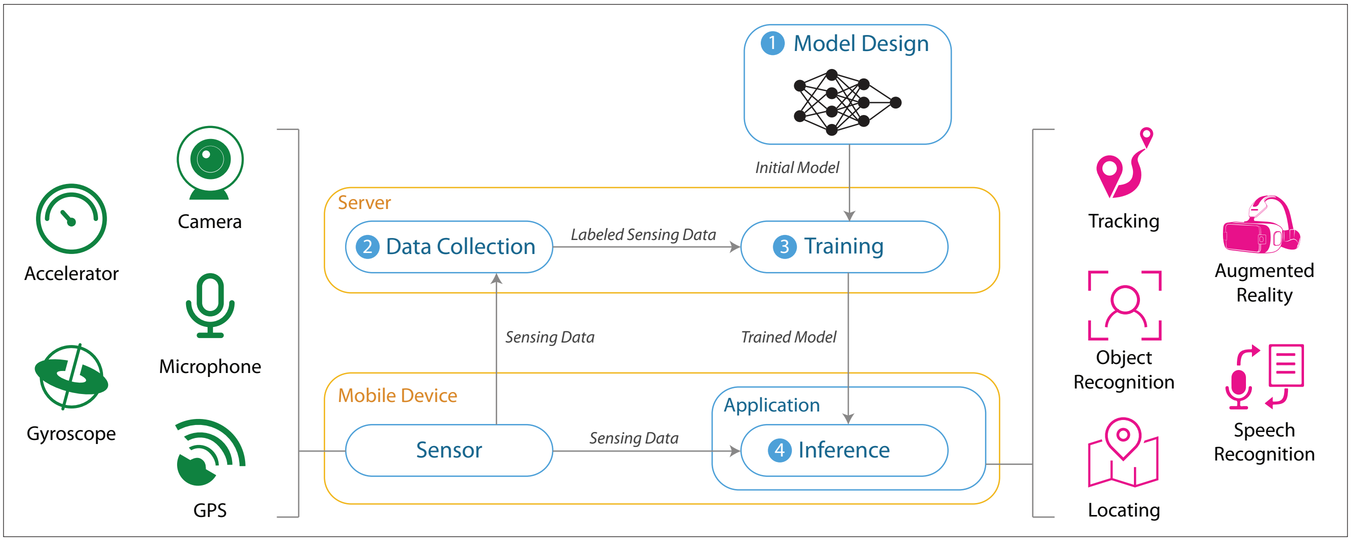

FIGURE 1. The framework of a deep mobile sensing application.

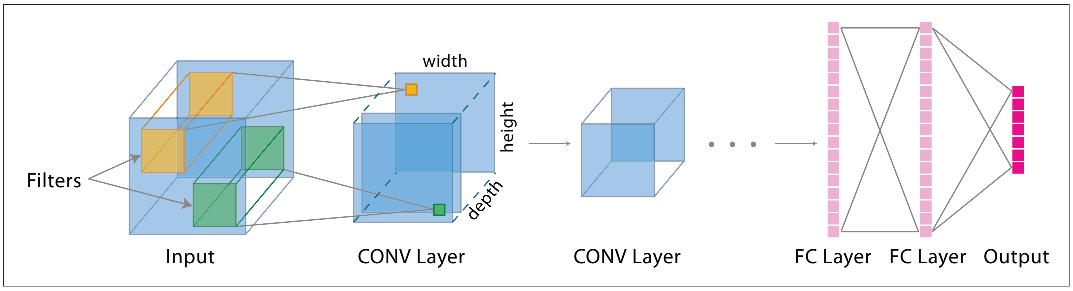

FIGURE 2. An illustration of a neural network structure. An input is transformed to an output through a series of convolutional (CONV) layers and fully connected (FC) layers.

This article is organized as follows. First, we present the framework of using deep learning for mobile sensing applications. Second, we introduce three categories of energy reduction approaches. We discuss core ideas, challenging issues and solutions, together with trade-offs for each of them. We simulate and compare the techniques of model compression, present open research issues. Finally, we conclude this article.

Deep Mobile Sensing: An Overview

In this section, we first introduce the basic framework by which the deep learning approach is used to solve a mobile sensing problem. Then, we give a brief introduction to the structure of neural networks.

Framework

For an application using the deep learning approach, there are four important phases, as shown in Fig. 1. First, a neural network has to be *designed*. Second, abundant sensing data have to be *collected and labeled*, which is to identify the true prediction result for each sensing measurement. Third is training. A collection of data are fed into the designed model, and then its parameters are adjusted iteratively until the prediction error becomes sufficiently low. Finally, a neural network model is obtained for *inference*. New data is given as an input of the model, and then after a forward pass through the model an inference result is produced.

As illustrated in Fig. 1, the training phase is placed on the server side, while the inference is placed on the side of the mobile device. This is because training is not only data-intensive but also compute-intensive. A neural network normally has tens of millions of parameters; it needs sufficient data and computation to make the learning algorithm converge. Thus, the training process can only be executed on a server or a cluster of servers. Compared to training, inference is less compute-intensive, since only one forward pass through the model is required for each task. It thus may be able to run on mobile devices.

Neural Networks

A neural network is the core of deep learning. As shown in Fig. 2, a model receives an input at one end, transforms it through a series of hidden layers and produces a result at the other end. The output of a hidden layer is called activation. The *activation* of a layer is the input of the following layer. There are several types of hidden layers. Among them, the fully connected layer and convolutional layer are the most important. We introduce each of them below.

As shown in Fig. 2, a fully connected layer is always illustrated as a column of neurons. In each layer, every neuron is connected to all the neurons of the previous layer. Each connection has a weight and each neuron has a bias, both of which are learnable parameters. For each neuron, the output value is computed as $f(\mathbf{w}^T\mathbf{x} + b)$, where f is a nonlinear function, for example, ReLU $f(y) = \max(0, y)$, $\mathbf{w}$ is a vector of weights corresponding to the full connections toward this neuron, $\mathbf{x}$ is the output of the previous layer, and b is a bias corresponding to this neuron.

A convolutional layer is always illustrated as a block of neurons in three dimensions, i.e., width, height, and depth, as shown in Fig. 2. For each neuron, instead of connecting to all the neurons in the previous layer, it connects only to a local region, which is small along width and height but extends through the full depth of the input volume. As illustrated in Fig. 2, the green neuron of the second layer connects to the input-layer neurons covered by the green block. This block of connection weights is called a *filter* or *kernel*. By sliding a filter across the width and the height of the input layer and computing dot products between the filter and the input region, a 2-dimensional activation map is produced. The bias parameters and the nonlinear function similar as above are also applied here. When several filters are adopted, several activation maps are produced, each of which corresponds to a depth slice. As shown in Fig. 2, the yellow filter produces the top activation map, while the green filter produces the bottom one.

Model design is to decide the structure of a neural network for an application. Once the structure is determined, an optimization problem is formulated, that is to find values for the parameters (weights and biases) such that a prediction error is minimized. The problem is solved using a gradient descent algorithm. Finally, a neural network with parameters known is ready for inference tasks.

Approaches for Energy Efficiency

In the execution of inference tasks, energy consumption is dominated by memory access. As shown in Fig. 3, memory (DRAM) access is two orders of magnitude more energy expensive than a float multiplication operation [8]. Thus, a straightforward idea for energy reduction is to decrease the amount of memory access. Since the majority of memory accesses are reading model parameters into processors, a possible energy reduction approach is to decrease the amount of model parameters, i.e., the compressing model. The smaller a model becomes, the less memory energy is consumed. When the model is sufficiently small, it can even be stored in a SRAM cache, which is very limited in capacity but as energy efficient as an arithmetic operation, as shown in Fig. 3.

In computing an inference, each time a batch of operations execute, data (input activation and weights) have to be fetched from memory. Redundant data reading may happen. For example, in a convolutional layer the kernel weights are shared across an input activation. Thus, data reading can be optimized to avoid repeated reading and improve energy efficiency, i.e., minimizing data transfer.

For an inference, besides memory access, the other part of energy is mainly consumed by the computations in the CPU and GPU, which consists of several billion or even tens of billions of operations. Fortunately, besides the energy-hungry processors, modern mobile devices are usually equipped with low-power processors such as an LPU and DSP. This provides an opportunity to reduce energy by offloading some computation on them. This idea can be extended to remote energy-abundant devices such as cloud servers.

We summarize these approaches and their features in Table 1. None of them is free of sacrifice: the compressing model may trade off accuracy; minimizing data transfer needs designing specialized hardware; offloading may trade off latency and network usage. When choosing an approach to use, different trade-offs have to be made. In the following, we introduce each of the approaches.

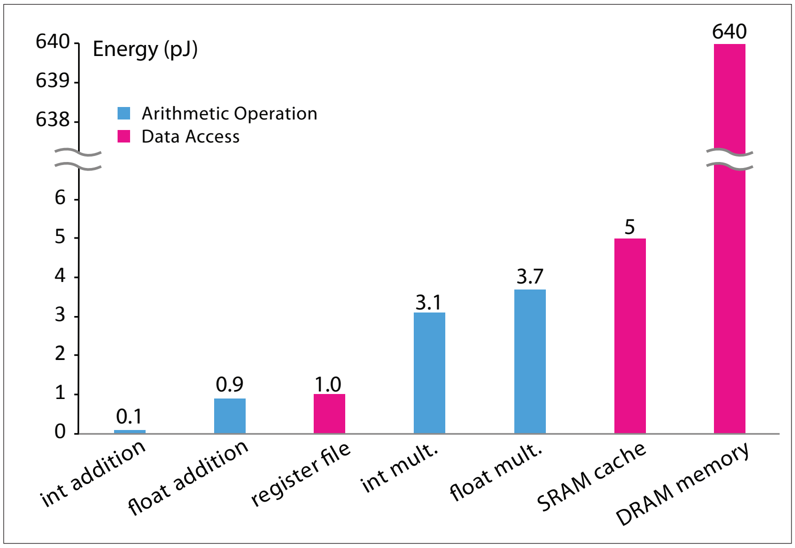

FIGURE 3. The energy consumption of 32-bit operations on a 45nm CMOS process [8].

Compressing Model

The compressing model aims to reduce the amount of model parameters. Neural networks are usually extremely over parameterized, because designing a complex model to achieve high accuracy is generally easier than designing a simple model that performs equally well, as the latter requires skills and time that are not always available. Since they contain large redundancy, they are capable of compression. However, this may result in the loss of accuracy. Thus, a challenging issue is to maximally simplify the model while mitigating resulting accuracy loss. Several approaches have been proposed to solve this problem, such as pruning unimportant parameters, replacing some layers with low-rank approximations and quantizing weights to enforce the representation with lower bits. We introduce each of them below, followed by a simulation.

Network Pruning: With network pruning, all the connections whose weights are below a threshold are removed. The value of the threshold affects the trade-off between network sparsity and the loss of accuracy. After pruning, the resulting sparse neural network is retrained in order to mitigate the accuracy loss. This process of pruning following retraining may be repeated several times to further improve sparsity. This approach is used in [9], and can reduce model size and computation by nine times with negligible accuracy loss.

Layer Decomposition: Another approach to reduce model redundancy is layer decomposition. It exploits some techniques of matrix approximation to decrease parameter redundancy, while mitigating accuracy degradation. Singular value decomposition (SVD), a common matrix approximation, can be used to compress the convolutional layers, by representing a large number of filters as the linear combinations of a much smaller set of filters [10]. This can effectively reduce the num-

Categories	Approaches	Ref.	Trade-offs	Additional benefits
Compressing model	Network pruning	[9]	Accuracy	Speedup
	Layer decomposition	[10]		
	Quantization	[9, 11–13]		
Minimizing data transfer	Data reuse	[14]	Specialized hardware	Speedup
	Data sparsity	[8, 15]		
Offloading	Local offloading	[3]	Latency, network usage	–
	Remote offloading	[4]		

TABLE 1. Three categories of energy reduction approaches.

ber of parameters by a factor of two to three with negligible accuracy loss, and also accelerate the computation by a factor of four or more.

Quantization: Another powerful approach to reduce the model size is quantizing model weights to enforce that many connections share the same value. In this way, a neural network is represented by several full-precision weights and codewords with fewer bits, and thus becomes significantly smaller. A quantization algorithm normally has a pipeline as follows. First, it generates a code book. Second, it quantizes weights according to the code book and produces quantization indices. Finally, it encodes these indices into codewords. A model after quantization is always retrained to fine-tune the quantized values and mitigate accuracy loss.

Next, we introduce the strategies used in each step. First, before the generation of a code book, the number of quantized values, denoted by N, should be determined first. In [9], all convolutional layers use 256 quantized values (8 bits) and all fully-connected layers use 32 quantized values (5 bits). It is also possible to use different N for each layer. For instance, in [11] the authors proposed an optimization approach to identify the optimal N for each layer, by minimizing the quantization error across all layers. With N determined, the code book can be generated in several ways. One way is to use k-means clustering to group weights into N clusters, and take the centroids of clusters as a code book, as in [9]. Another way is to partition the value range of the weights into equal intervals as in [11]. Second, the quantization step is straightforward. Each weight is transformed to be an index by identifying a cluster or an interval where the weight belongs. Finally, for coding, either fixed-length coding or variable-length coding can be used. Huffman coding, a famous variable-length coding algorithm, is used in [9], which can further reduce the model size compared to fixed-length coding.

Among quantization approaches, there are two extreme cases worth discussing: binary and ternary quantization, where only binary (1 bit) or ternary values (2 bits) and one or two shared weights are needed to represent a neural model. For example, in a ternary approach TTQ [12], the parameters are quantized to $\{-w_n, 0, +w_p\}$, where coefficients w_n and w_p are shared by each layer and learned from training. The quantization signs are determined according to some thresholds, which are selected by minimizing the L2 distance between ternary and full-precision weights. TTQ compresses model AlexNet by 16 times. In addition, BNN [13] is a binary quantization, where all parameters are quantized to $\{-1, +1\}$. The quantization signs are determined by a probabilistic method.

In conclusion, a compressed model is usually several or even 20 times smaller than the original one. Model compression improves energy efficiency due to three reasons. First, a sufficient small model is possible to be stored in SRAM, which is significantly more energy efficient than memory access. Second, the amount of storage access, from either DRAM or SRAM, is reduced drastically. Third, the energy consumed by computation also decreases due to fewer operations after compression. This kind of approach can also reduce latency due to reduced operations. However, it may result in a bit of accuracy loss, but usually can be mitigated by retraining.

Simulation and Comparison: We simulate the above techniques with TensorFlow and compare their performance. Our result is consistent with existing works and demonstrates that these techniques are effective in energy reduction.

We use a VGG-16 neural network to perform a 100-class object recognition on a CIFAR-100 dataset. We start from fine-tuning a well-trained VGG-16 model on an ImageNet dataset, and obtain a model with the top-five accuracy of 75.29 percent after training for 50 epochs. This model is 513 MB and contains 30.9 billion FLOP. Then, we compress this model. We summarize the result as follows:

- After pruning 70 percent weights, we reduce the model to 154 MB and the amount of FLOP to 9.7 billion. After fine-tuning for one epoch, the accuracy becomes 75.02 percent, which is very similar to the unpruned one.
- 90 percent pruning reduces the model to 51 MB and the amount of FLOP to 3.2 billion. After training for 32 epochs, the accuracy reaches 73.27 percent, which is slightly lower than the unpruned case.
- We use SVD-based layer decomposition to all layers. This reduces the model to 89 MB and the amount of FLOP to 8.8 billion. After fine-tuning, the accuracy reaches 73.55 percent.
- With 8-bit fixed point quantization, we reduce the model to 128 MB. The accuracy is 75.33 percent after fine-tuning, which is similar to the original one.

We illustrate the result in Fig. 4. The energy is reduced by a factor of 2~5, which is significant. Energy is computed by considering both data access and arithmetic operations. The former is computed by multiplying the model size and the energy consumed by accessing DRAM memory in Fig. 3, while the later is computed by multiplying the FLOP count and the energy consumption of a single FLOP. For 32-bit FLOP, we assume its energy consumption is 2.3 pJ (averaging multiplication and addition operations as in Fig. 3), while for 8-bit quantized FLOP, we conservatively approximate that as 1.6 pJ (averaging integer multiplication and addition operations).

Minimizing Data Transfer

The approach of minimizing data transfer also aims to reduce the energy cost of memory access.

Data Reuse: Instead of naively reading input data for all operations, the data for an operation may be reused by some other operations. This data reuse is possible due to the structure of neural networks, as illustrated in Fig. 2. For any convolutional layer, there are several kinds of data reuse:

- Filter reuse: each filter is reused in computing all the neurons of the same output activation map.
- Convolutional reuse: a filter is moved with a small stride, which is less than the filter width. As such, some pixels of an input activation are reused in computing neighboring output neurons.
- Input activation reuse: each input activation is reused by all filters to generate multiple output activation maps.

The first two patterns are suitable for convolutional layers rather than fully connected layers. In contrast, the last one is proper for both.

Many research works have been proposed to leverage data reuse to minimize data transfer and thus improve energy efficiency. Among recent works, Eyeriss [14] is a notable one, which uses all kinds of data reuse. It is shown that for convolutional layers their approach minimizes memory accesses by maintaining the data transfer in local registers. Each register is a tiny storage of 512 bytes, located in each processing element and 200 times more energy efficient than memory access.

Data Sparsity: Besides the data reuse, another characteristic of CNN computation is that a substantial amount of activations are zero, 50-70 percent for typical datasets [15]. This is because negative activations are transformed to zero by the non-linear ReLU function. Since the zero values in activations contribute nothing to computation, they can be eliminated from memory reading and computing for energy efficiency.

The similar sparsity also appears in the weights of either pruned networks or ternary networks, as discussed in model compression. Those zero weights can also be avoided in data delivery and computation.

These sparsity patterns are exploited in several works, among which SCNN [15] and EIE [8] are notable. SCNN maintains the sparse activations and weights in a compressed form throughout the entire computation and is 2.3 times more energy efficient than the dense computing approach. SCNN is designed for convolutional layers, while EIE is designed for fully connected layers. In EIE, it is reported that leveraging weight sparsity reduces energy consumption to 1/10, and leveraging activation sparsity reduces to 1/3.

In conclusion, the approaches exploiting data reuse and sparsity can improve energy efficiency, and they can shorten latency due to local data access and a reduced amount of computations. They do not affect the accuracy of inference, because no approximation is involved. However, implementing these approaches always requires the design of specialized hardware.

Offloading

The third type of approach to save energy is to utilize heterogeneous resources either locally or remotely.

Local Offloading: Low-power processing units such as LPUs and DSPs provide an opportunity to reduce computational energy by offloading some workloads. However, naive offloading may result in unwanted long latency, because low-power processing units usually take a longer time to compute. Thus, smart offloading which can make a trade-off between energy efficiency and latency is required. It can be formulated as a scheduling problem, that is to partition an inference task (a forward pass in a neural network) into several subtasks and then allocate each of them to one of the processors, such that the energy consumption is minimized while the latency requirement is satisfied.

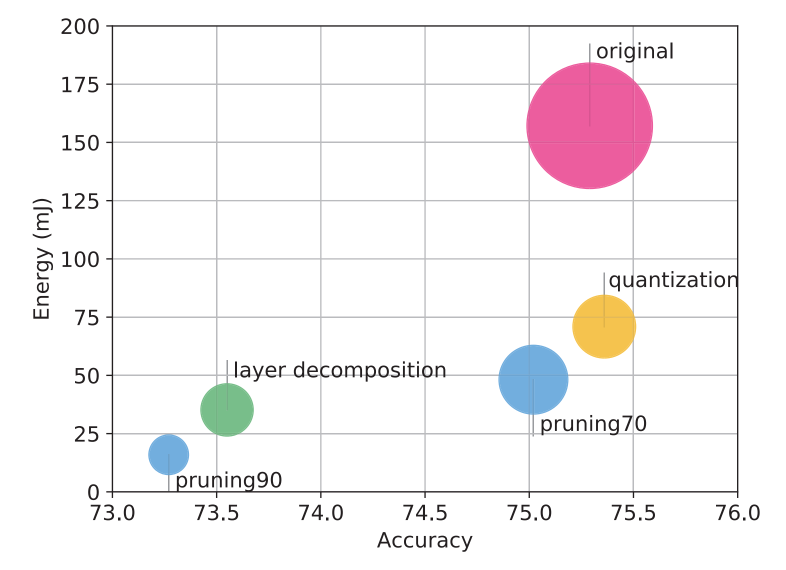

FIGURE 4. Energy vs accuracy, where the circle size represents model size.

For *partitioning* an inference task, each layer may be partitioned into several parts (subtasks), each computing for a group of neurons. In order to formulate the offloading problem, the energy consumption and the latency taken by each subtask running on different processors have to be known. This can be obtained by an *offline profiling*. However, there may be other applications running on the same device and contending for these processors, so that the latency may be affected. Thus, a better approach is to do profiling under various levels of processor utilization, and select a profiled record according to real-time utilization. These real-time system conditions are obtained by a *resource monitor*.

So far, the offloading problem can be formulated as an integer linear programming (ILP). We give a simple example here. Suppose we are dealing with a layer of a neural network, which is composed of *m* computational units. Here, a computational unit is considered as computing the output value for a single neuron. In addition, suppose there are *n* processors available. From offline profiling and resource monitoring, we can learn the energy and the time per unit of computation consumed by each processor. Then, the problem is to determine the amount of computations assigned on each processor. It is formulated as an ILP. The objective is to minimize the total amount of energy, while the constraint is to make the latency requirement satisfied for every processor.

Next, the problem is solved by a *solver*, such as a heuristic algorithm. Finally, a *scheduler* assigns each subtask on its allocated processor. The pipeline of the offloading approach is illustrated in Fig. 5.

DeepX [3] is an interesting work using this approach. In this work, the offloading problem is

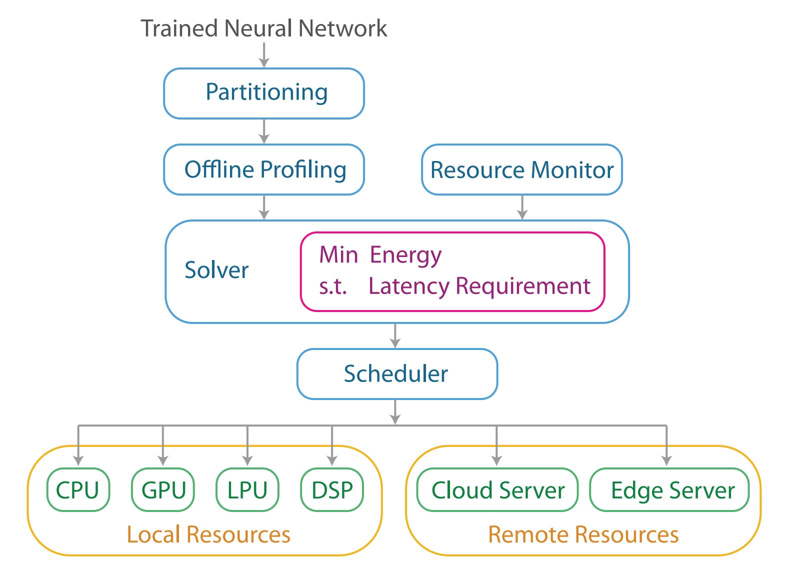

FIGURE 5. The pipeline of an offloading approach. The inference task can be offloaded to either local low-power processors or remote servers.

formulated as minimizing a weighted sum of energy consumption and execution latency, and solved by using a standard ILP solver.

Remote Offloading: Besides local computational resources, we may utilize remote resources through the network, such as a cloud server or an edge server.

Naively offloading the entire workload to remote servers may be unsatisfied due to several reasons. First, data transmission incurs delay. In a complex network environment, the delay may be unpredictable, and a lengthy delay may be unacceptable for some applications. Second, data transmission itself consumes energy, sometimes more than the energy required for doing all the computation locally [3]. Third, depending on the nature of the application, partitioning the workload could achieve a good balance between transmission delay and energy consumption. For example, some neural networks transform high dimensional inputs to low dimensional features in the first few layers, and then spend most of the time working on those features. In this scenario, it is quite reasonable to process high dimensional inputs locally and send the low dimensional features to a remote server for further processing.

Partitioning the workload leads to an optimization problem. In optimization formulation, the energy consumed by remote resources needs no consideration and the computational latency on remote resources is negligible sometimes. However, the energy and latency consumed by network transfer is significant and have to be considered.

LEO [4] is a scheduler that offloads an inference task to both local processors and remote resources. In this work, a heuristic algorithm is proposed to solve the formulated ILP. As a result, the runtime of scheduling is short and the energy cost is low, which enables frequent re-scheduling.

In conclusion, the offloading approach utilizes low-power processors and remote energy-abundant devices to reduce energy consumption at a tolerable cost of latency.

Open Issues

There are still many challenges and open issues with energy-efficient deep mobile sensing.

Dynamically Selecting Model and Input

As we mentioned in the discussion of the compressing model, simplifying neural networks trades off accuracy for low complexity. With several different accuracy requirements, a catalog of neural models with different levels of complexity and energy consumption are obtained. Then, the most energy-efficient model can be selected from the catalog depending on the dynamic accuracy requirements and energy constraints. The model catalog concept appears in [6], but no lightweight approaches are proposed. In addition, the sensing data with various resolutions should be considered as well, for example, the images with different resolutions. Dynamically selecting both input and model may further reduce energy.

Offline Scheduling and Caching

For offloading approaches, each time an inference task arrives, its scheduling model is updated with the latest system conditions (such as processor utilization, network speed, and so on) and solved again. Frequently calling the solver is inefficient, since it consumes both computation and energy resources and causes latency. For this issue, a promising method is using offline scheduling and caching rather than online scheduling. In this way, an optimization model is formulated for each possible system condition and solved in advance. Those solutions are cached and can be reused in future scheduling. However, the amount of system conditions possibly happening is vast. Due to the limited cache space, a small group of system conditions can be selected and serve as the approximations for others.

Conclusions

For deep mobile sensing, it is a challenging issue to reduce the energy consumption of the inference job executed on mobile devices. In this article, we surveyed various of energy reduction approaches and classified them into three categories: compressing model, minimizing data transfer and offloading. We discussed motivations, challenging issues and solutions, as well as performance trade-offs for each of them. Finally, several open research issues are discussed.

Acknowledgments

This work was supported by the Young Scientists Fund of China NSF (Grant No. 61802263); the Faculty Research Fund of Shenzhen University (2019050); Tencent Rhino-Bird Young Faculty Research Fund; and Peacock Plan Research Fund. It was partially supported by China NSF (61872248, 61872246, U1736207); Guangdong NSF (2017A030312008); Shenzhen Science and Technology Foundation (JCYJ20170302140946299, JCYJ20170412110753954, JCYJ20170817095418831); Fok Ying-Tong Education Foundation for Young Teachers in the Higher Education Institutions of China (161064), GDUPS (2015). It was also partially supported by the Tianjin Key Laboratory of Advanced Networking (TANK); the project "PCL Future Regional

Network Facilities for Large-scale Experiments and Applications (PCL2018KP001)"; and the Guangdong Special Support Program.

References

[1] S. Yao *et al.*, "Deepsense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing," *Proc. 26th Int'l. Conf. World Wide Web*, 2017, pp. 351–60.

[2] P. Georgiev *et al.*, "Accelerating Mobile Audio Sensing Algorithms through On-Chip GPU Offloading," *Proc. 15th Annual Int'l. Conf. Mobile Systems, Applications, and Services*, 2017, pp. 306–18.

[3] N. D. Lane *et al.*, "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices," *Proc. 2016 15th ACM/IEEE Int'l. Conf. Information Processing in Sensor Networks (IPSN)*, April 2016, pp. 1–12.

[4] P. Georgiev *et al.*, "LEO: Scheduling Sensor Inference Algorithms Across Heterogeneous Mobile Processors and Network Resources," *Proc. 22nd Annual Int'l. Conf. Mobile Computing and Networking*, 2016, pp. 320–33.

[5] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile GPU-Based Deep Learning Framework for Continuous Vision Applications," *Proc. 15th Annual Int'l. Conf. Mobile Systems, Applications, and Services*, 2017, pp. 82–95.

[6] S. Han *et al.*, "MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing under Resource Constraints," *Proc. 14th Annual Int'l. Conf. Mobile Systems, Applications, and Services*, 2016, pp. 123–36.

[7] A. Canziani, E. Culurciello, and A. Paszke, An Analysis of Deep Neural Network Models for Practical Applications; available: https://arxiv.org/pdf/1605.07678.pdf

[8] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *Proc. 2016 ACM/IEEE 43rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2016, pp. 243–54.

[9] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *Proc. 4th Int'l. Conf. Learning Representations*, May 2016.

[10] X. Zhang *et al.*, "Accelerating Very Deep Convolutional Networks for Classification and Detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 38, no. 10, Oct. 2016, pp. 1943–55.

[11] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," *Proc. 33rd Int'l. Conf. Machine Learning*, vol. 48, 2016, pp. 2849–58.

[12] C. Zhu *et al.*, "Trained Ternary Quantization," *Proc. 5th Int'l. Conf. Learning Representations*, Apr. 2017.

[13] Z. Lin *et al.*, "Neural Networks with Few Multiplications," *Proc. 4th Int'l. Conf. Learning Representations*, May 2016.

[14] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *Proc. 43rd Int'l. Symposium on Computer Architecture*, 2016, pp. 367–79.

[15] A. Parashar *et al.*, "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks," *Proc. 44th Annual Int'l. Symposium on Computer Architecture*, 2017, pp. 27–40.

Biographies

RUITAO XIE received her Ph.D. degree in computer science from City University of Hong Kong in 2014, and the B.Eng. degree from Beijing University of Posts and Telecommunications in 2008. She is currently an assistant professor at the College of Computer Science and Software Engineering, Shenzhen University. Her research interests include mobile computing, distributed systems and cloud computing.

XIAOHUA JIA received his B.Sc. (1984) and M.Eng. (1987) from the University of Science and Technology of China, and a D.Sc. (1991) in information science from the University of Tokyo. He is currently Chair Professor with the Dept. of Computer Science at the City University of Hong Kong. His research interests include cloud computing and distributed systems, data security and privacy, computer networks and mobile computing. He is an editor of *IEEE Internet of Things*, *IEEE Transactions on Parallel and Distributed Systems* (2006-2009), *Wireless Networks*, *Journal of World Wide Web*, and the *Journal of Combinatorial Optimization*, among others. He is the General Chair of ACM MobiHoc 2008, TPC Co-Chair of IEEE GlobeCom 2010-Ad Hoc and Sensor Networking Symp, and Area-Chair of IEEE INFOCOM 2010, 2015-2017. He is a Fellow of IEEE.

LU WANG is currently an assistant professor at the College of Computer Science and Software Engineering, Shenzhen University. She received the B.S. degree in communication engineering from Nankai University in 2009, and the Ph.D. degree in computer science and engineering from Hong Kong University of Science and Technology in 2013. Her research interests focus on wireless communications and mobile computing.

KAISHUN WU received his Ph.D. degree in computer science and engineering from HKUST in 2011. After that, he worked as a research assistant professor at HKUST. In 2013, he joined SZU as a distinguished professor. He has co-authored two books and published over 100 high quality research papers in international leading journals and prime conferences, such as IEEE TMC, IEEE TPDS, ACM MobiCom, and IEEE INFOCOM. He is the inventor of six U.S. and over 90 Chinese pending patents. He received the 2012 Hong Kong Young Scientist Award, the 2014 Hong Kong ICT Awards: Best Innovation and 2014 IEEE ComSoc Asia-Pacific Outstanding Young Researcher Award. He is an IET Fellow.

Partitioning the workload leads to an optimization problem. In optimization formulation, the energy consumed by remote resources needs no consideration and the computational latency on remote resources is negligible sometimes. However, the energy and latency consumed by network transfer is significant and have to be considered.