INVITED PAPER

Deep Learning With Edge Computing: A Review

This article provides an overview of applications where deep learning is used at the network edge. Computer vision, natural language processing, network functions, and virtual and augmented reality are discussed as example application drivers.

By Jiasi Chen and Xukan Ran

ABSTRACT | Deep learning is currently widely used in a variety of applications, including computer vision and natural language processing. End devices, such as smartphones and Internet-of-Things sensors, are generating data that need to be analyzed in real time using deep learning or used to train deep learning models. However, deep learning inference and training require substantial computation resources to run quickly. Edge computing, where a fine mesh of compute nodes are placed close to end devices, is a viable way to meet the high computation and low-latency requirements of deep learning on edge devices and also provides additional benefits in terms of privacy, bandwidth efficiency, and scalability. This paper aims to provide a comprehensive review of the current state of the art at the intersection of deep learning and edge computing. Specifically, it will provide an overview of applications where deep learning is used at the network edge, discuss various approaches for quickly executing deep learning inference across a combination of end devices, edge servers, and the cloud, and describe the methods for training deep learning models across multiple edge devices. It will also discuss open challenges in terms of systems performance, network technologies and management, benchmarks, and privacy. The reader will take away the following concepts from this paper: understanding scenarios where deep learning at the network edge can be useful, understanding common techniques for speeding up deep learning inference and performing distributed training on edge devices, and understanding recent trends and opportunities.

KEYWORDS | Artificial intelligence; edge computing; machine learning; mobile computing; neural networks.

Manuscript received February 7, 2019; revised April 30, 2019; accepted May 29, 2019. Date of publication July 15, 2019; date of current version August 5, 2019. This work was supported in part by NSF CNS-1817216. *(Corresponding author: Jiasi Chen.)*

The authors are with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA USA (e-mail: jiasi@cs.ucr.edu).

Digital Object Identifier 10.1109/JPROC.2019.2921977

I. INTRODUCTION

Deep learning has recently been highly successful in machine learning across a variety of application domains, including computer vision, natural language processing, and big data analysis, among others. For example, deep learning methods have consistently outperformed traditional methods for object recognition and detection in the ISLVRC Computer Vision Competition since 2012 [1]. However, deep learning's high accuracy comes at the expense of high computational and memory requirements for both the training and inference phases of deep learning. Training a deep learning model is space and computationally expensive due to millions of parameters that need to be iteratively refined over multiple time periods. Inference is computationally expensive due to the potentially high dimensionality of the input data (e.g., a high-resolution image) and millions of computations that need to be performed on the input data. High accuracy and high resource consumption are defining characteristics of deep learning.

To meet the computational requirements of deep learning, a common approach is to leverage cloud computing. To use cloud resources, data must be moved from the data source location on the network edge [e.g., from smartphones and Internet-of-Things (IoT) sensors] to a centralized location in the cloud. This potential solution of moving the data from the source to the cloud introduces several challenges.

1) *Latency:* Real-time inference is critical to many applications. For example, camera frames from an autonomous vehicle need to be quickly processed to detect and avoid obstacles or a voice-based-assistive application needs to quickly parse and understand the user's query and return a response. However, sending data to the cloud for inference or training may incur additional queuing and propagation delays from the network and cannot satisfy

strict end-to-end low-latency requirements needed for real time, interactive applications; for example, real experiments have shown that offloading a camera frame to an Amazon Web Services server and executing a computer vision task take more than 200-ms end-to-end [2].

2) *Scalability:* Sending data from the sources to the cloud introduces scalability issues, as network access to the cloud can become a bottleneck as the number of connected devices increases. Uploading all data to the cloud is also inefficient in terms of network resource utilization, particularly if not all data from all sources are needed by the deep learning. Bandwidth-intensive data sources, such as video streams, are particularly a concern.
3) *Privacy:* Sending data to the cloud risks privacy concerns from the users who own the data or whose behaviors are captured in the data. Users may be wary of uploading their sensitive information to the cloud (e.g., faces or speech) and of how the cloud or application will use these data. For example, the recent deployment of cameras and other sensors in a smart city environment in New York City incurred serious concerns from privacy watchdogs [3].

Edge computing is a viable solution to meet the latency, scalability, and privacy challenges described earlier in this section. In edge computing, a fine mesh of compute resources provides computational abilities close to the end devices [4]. For example, an edge compute node could be co-located with a cellular base station and an IoT gateway or on a campus network. Edge computing is already being deployed by industry; for example, a major cellular Internet service provider in the United States and a national fast-food chain have both deployed edge compute services [5], [6]. To address latency challenges, edge computing's proximity to data sources on the end devices decreases end-to-end latency and thus enables real-time services. To address scalability challenges, edge computing enables a hierarchical architecture of end devices, edge compute nodes, and cloud data centers that can provide computing resources and scale with the number of clients, avoiding network bottlenecks at a central location. To address privacy challenges, edge computing enables data to be analyzed close to the source, perhaps by a local trusted edge server, thus avoiding traversal of the public Internet and reducing exposure to privacy and security attacks.

While edge computing can provide the latency, scalability, and privacy benefits discussed earlier in this section, several major challenges remain to realize deep learning at the edge. One major challenge is accommodating the high resource requirements of deep learning on less powerful edge compute resources. Deep learning needs to execute on a variety of edge devices, ranging from reasonably provisioned edge servers equipped with a GPU, to smartphones with mobile processors, to barebones Raspberry Pi devices. A second challenge is understanding how the edge devices should coordinate with other edge devices and with the cloud, under heterogeneous processing capabilities and dynamic network conditions, to ensure a good end-to-end application-level performance. Finally, privacy remains a challenge, even as edge computing naturally improves privacy by keeping data local to the network edge, as some data often still need to be exchanged between edge devices and possibly the cloud. Researchers have proposed various approaches from diverse angles to tackle these challenges, ranging from hardware design to system architecture to theoretical modeling and analysis. The purpose of this paper is to survey works at the confluence of the two major trends of deep learning and edge computing, in particular focusing on the software aspects and their unique challenges therein. While excellent surveys exist on deep learning [7] as well as edge computing [8], [9] individually, this paper focuses on works at their intersection.

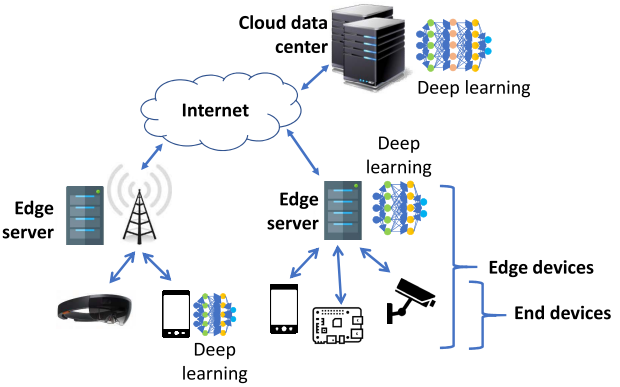


Fig. 1. *Deep learning can execute on edge devices (i.e., end devices and edge servers) and on cloud data centers.*

Deep learning on edge devices has similarities to, but also differences from, other well-studied areas in the literature. Compared to cloud computing that can help run computationally expensive machine learning (e.g., machine learning as a service), edge computing has several advantages, such as lower latency and greater geospatial specificity that have been leveraged by researchers [10]. Several works have combined edge computing with cloud computing, resulting in hybrid edge-cloud architectures [11]. Compared to traditional machine learning methods (outside of deep learning), deep learning's computational demands are particularly a challenge, but deep learning's specific internal structure can be exploited to address this challenge (see [12]–[14]). Compared to the growing body of work on deep learning for resource-constrained devices, edge computing has additional challenges relating to shared communication and computation resources across multiple edge devices.

In the rest of this paper, we define the edge devices that include both end devices (e.g., smartphones or IoT sensors), as well as edge compute nodes or servers, as shown in Fig. 1. This paper is organized as follows.

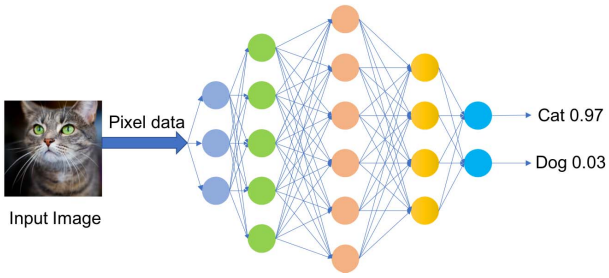


Fig. 2. *DNN example of image classification.*

We first provide a brief background on deep learning (see Section II). We then describe several application domains where deep learning on the network edge can be useful (see Section III). In Section IV, we discuss different architectures and methods to speed up deep learning inference, focusing on device-only execution, always computing on the edge server, and intermediate alternatives, such as offloading, hybrid edge-cloud, and distributed computing approaches. We then discuss training deep learning models on edge devices, with an emphasis on distributed training across devices and privacy (see Section V). Finally, we finish with open research challenges (see Section VI) and conclusions (see Section VII).

II. BACKGROUND, MEASUREMENTS, AND FRAMEWORKS

A. Background on Deep Learning

Since some of the techniques discussed in this paper rely on the specific internals of deep learning, therefore, we first provide a brief background on deep learning. Further details can be found in reference texts (see [7]).

A deep learning prediction algorithm, also known as a model, consists of a number of layers, as shown in Fig. 2. In deep learning inference, the input data pass through the layers in sequence, and each layer performs matrix multiplications on the data. The output of a layer is usually the input to the subsequent layer. After data are processed by the final layer, the output is either a feature or a classification output. When the model contains many layers in sequence, the neural network is known as a deep neural network (DNN). A special case of DNNs is when the matrix multiplications include convolutional filter operations, which is common in DNNs that are designed for image and video analysis. Such models are known as convolutional neural networks (CNNs). There are also DNNs designed especially for time series prediction; these are called recurrent neural networks (RNNs) [7], which have loops in their layer connections to keep state and enable predictions on sequential inputs.

In deep learning training, the computation proceeds in reverse order. Given the ground-truth training labels, multiple passes are made over the layers to optimize the parameters of each layer of matrix multiplications, starting from the final layer and ending with the first layer. The algorithm used is typically stochastic gradient descent.

Table 1 Common Performance Metrics

Performance metrics
Latency (s)
Energy (mW, J)
Concurrent requests served (#)
Network bandwidth (Mbps)
Accuracy (application-specific)

In each pass, a randomly selected "mini-batch" of samples is selected and used to update the gradients in the direction that minimizes the training loss (where the training loss is defined as the difference between the predictions and the ground truth). One pass through the entire training data set is called a training epoch [15].

A key takeaway for the purposes of this work is that there are a large number of parameters in the matrix multiplications, resulting in many computations being performed and thus the latency issues that we see on end devices. A second takeaway is that there are many choices (hyperparameters) on how to design the DNN models (e.g., the number of parameters per layer, and the number of layers), which makes the model design more of an art than a science. Different DNN design decisions result in tradeoffs between system metrics; for example, a DNN with higher accuracy likely requires more memory to store all the model parameters and will have higher latency because of all the matrix multiplications being performed. On the other hand, a DNN model with fewer parameters will likely execute more quickly and use less computational resources and energy, but it may not have sufficient accuracy to meet the application's requirements. Several works exploit these tradeoffs, which will be discussed in Sections IV-B and IV-C.

B. Measurements of Deep Learning Performance

Deep learning can be used to perform both supervised learning and unsupervised learning. The metrics of success depend on the particular application domain where deep learning is being applied. For example, in object detection, the accuracy may be measured by the mean average precision (mAP) [1], which measures how well the predicted object location overlaps with the ground-truth location, averaged across multiple categories of objects. In machine translation, the accuracy can be measured by the bilingual evaluation understudy score metric [16], which compares a candidate translation with several ground-truth reference translations. Other general system performance metrics not specific to the application include throughput, latency, and energy. These metrics are summarized in Table 1.

Designing a good DNN model or selecting the right DNN model for a given application is challenging due to the large number of hyperparameter decisions. A good understanding of the tradeoffs between the speed, accuracy, memory, energy, and other system resources can be helpful for the DNN model designer or the application developer. These comparative measurements are typically presented

in research papers proposing new models or standalone measurement papers [17]. An especially important consideration in the context of edge computing is the testbed that the measurements are conducted on. Machine learning research typically focuses on accuracy metrics, and their system performance results are often reported from powerful server testbeds equipped with GPUs. For example, Huang *et al.* [17] compared the speed and accuracy tradeoffs when running on a high-end gaming GPU (Nvidia Titan X). The YOLO DNN model [18], which is designed for real-time performance, provides timing measurements on the same server GPU.

Specifically targeting mobile devices, Lu *et al*. [19] provided the measurements for a number of popular DNN models on mobile CPUs and GPUs (Nvidia TK1 and TX1). Ran *et al*. [20] further explored the accuracy–latency tradeoffs on mobile devices by measuring how reducing the dimensionality of the input size reduces the overall accuracy and latency. DNN models designed specifically for mobile devices, such as MobileNets [21], report system performance in terms of a number of multiply–add operations, which could be used to estimate latency characteristics and other metrics on different mobile hardware, based on the processing capabilities of the hardware.

Once the system performance is understood, the application developer can choose the right model. There has also been much recent interest in automated machine learning, which uses artificial intelligence to choose which DNN model to run and tune the hyperparameters. For example, Tan *et al*. [22] and Taylor *et al*. [23] proposed using reinforcement learning and traditional machine learning, respectively, to choose the right hyperparameters for mobile devices, which is useful in edge scenarios.

C. Frameworks Available for DNN Inference and Training

To experiment with deep learning models, researchers commonly turn to open-source software libraries and hardware development kits. Several open-source software libraries are publicly available for deep learning inference and training on end devices and edge servers. Google's TensorFlow [24], released in 2015, is an interface for expressing machine learning algorithms and an implementation for executing such algorithms on heterogeneous distributed systems. Tensorflow's computation workflow is modeled as a directed graph and utilizes a placement algorithm to distribute computation tasks based on the estimated or measured execution time and communication time [25]. The placement algorithm uses a greedy approach that places a computation task on the node that is expected to complete the computation the soonest. Tensorflow can run on edge devices, such as Raspberry Pi and smartphones. TensorFlow Lite was proposed in the late 2017 [26], which is an optimized version of Tensorflow for mobile and embedded devices, with mobile GPU support added in early 2019. Tensorflow Lite only provides on-device inference abilities, not training, and achieves low latency by compressing a pre-trained DNN model.

Caffe [27]–[29] is another deep learning framework, originally developed by Jia, with the current version, Caffe2, maintained by Facebook. It seeks to provide an easy and straightforward way for deep learning with a focus on mobile devices, including smartphones and Raspberry Pis. PyTorch [30] is another deep learning platform developed by Facebook, with its main goal differing from Caffe2 in which it focuses on the integration of research prototypes to production development. Facebook has recently announced that Caffe2 and PyTorch will be merging.

GPUs are an important factor in efficient DNN inference and training. Nvidia provides GPU software libraries to make use of Nvidia GPUs, such as CUDA [31] for general GPU processing and cuDNN [32] which is targeted toward deep learning. While such libraries are useful for training DNN models on a desktop server, cuDNN and CUDA are not widely available on current mobile devices such as smartphones. To utilize smartphone GPUs, Android developers can currently make use of Tensorflow Lite, which provides experimental GPU capabilities. To experiment with edge devices other than smartphones, researchers can turn to edge-specific development kits, such as the Nvidia Jetson TX2 development kit for experimenting with edge computing (e.g., as used in [33]), with Nvidia-provided SDKs used to program the devices. The Intel Edison kit is another popular platform for experimentation, which is designed for IoT experiments (e.g., as used in [34]). Additional hardware-based platforms will be discussed in Section IV-A3.

III. APPLICATIONS OF DEEP LEARNING AT THE EDGE

We now describe several example applications where deep learning on edge devices is useful, and what "real time" means for each of these applications. Other applications of deep learning exist alongside the ones described in the following; here, for brevity, we highlight several applications that are relevant in the edge computing context. The common theme across these applications is that they are complex machine learning tasks where deep learning has been shown to provide good performance and they need to run in real time and/or have privacy concerns, hence necessitating inference and/or training on the edge.

A. Computer Vision

Since the success of deep learning in the ISLVRC Computer Vision Competition from 2012 onward [1], deep learning has been recognized as the state of the art for image classification and object detection. Image classification and object detection are fundamental computer vision tasks that are needed in a number of specific domains, such as video surveillance, object counting, and vehicle detection. Such data naturally originate from cameras located at the network edge, and there have even been

commercial cameras released with built-in deep learning capabilities [35]. Real-time inference in computer vision is typically measured in terms of frame rate [36], which could be up to the frame rate of the camera, typically 30–60 frames/s. Uploading camera data to the cloud also has privacy concerns, especially if the camera frames contain sensitive information, such as people's faces or private documents, further motivating computation at the edge. Scalability is a third reason why edge computing is useful for computer vision tasks, as the uplink bandwidth to a cloud server may become a bottleneck if there are a large number of cameras uploading large video streams.

Vigil [37] is one example of an edge-based computer vision system. Vigil consists of network of wireless cameras that perform processing at edge compute nodes to intelligently select frames for analysis (object detection or counting), for example, to search for missing people in surveillance cameras or analyze customer queues in retail environments. The motivation for edge computing in Vigil is twofold: to reduce the bandwidth consumption compared to a naive approach of uploading all frames to the cloud for analysis and for scalability as the number of cameras increases.

VideoEdge [38] similarly motivates the edge-based video analysis from a scalability standpoint. They use a hierarchical architecture of edge and cloud compute nodes to help with load balancing while maintaining high prediction accuracy (further details are provided in Section IV). Commercial devices, such as Amazon DeepLens [35], also follow an edge-based approach, where image detection is performed locally in order to reduce latency, and scenes of interest are only uploaded to the cloud for remote viewing if an interesting object is detected, in order to save bandwidth.

B. Natural Language Processing

Deep learning has also become popular for natural language processing tasks [39], including for speech synthesis [40], named entity recognition [41] (understanding different parts of a sentence), and machine translation [42] (translating from one language to another). For conversational artificial intelligence, latency on the order of hundreds of milliseconds has been achieved in recent systems [43]. At the intersection of natural language processing and computer vision, there are also visual question-and-answer systems [44], where the goal is to pose questions about an image (e.g., "how many zebras are in this image?") and receive natural language answers. Latency requirements differ based on how information is presented; for example, conversational replies are preferably returned within 10 ms, while a response to a written Web query can tolerate around 200 ms [45].

An example of natural language processing on the edge is voice assistants, such as Amazon Alexa or Apple Siri. While voice assistants perform some of their processing in the cloud, they typically use on-device processing to detect wakewords (e.g., "Alexa" or "Hey Siri"). Only if the wakeword is detected, then the voice recording is sent to the cloud for further parsing, interpretation, and query response. In the case of Apple Siri, the wakeword processing uses two on-device DNNs to classify speech into one of 20 classes (including general speech, silence, and wakeword) [46]. The first DNN is smaller (5 layers with 32 units) and runs on a low-power always-ON processor. If the first DNN's output is above a threshold, it triggers a second, more powerful DNN (5 layers with 192 units) on the main processor.

Wakeword detection methods need to be further modified to run on even more computationally constrained devices, such as a smartwatch or an Arduino. On the Apple Watch, a single DNN is used, with a hybrid structure borrowing from the aforementioned two-pass approach. For speech processing on an Arduino, researchers from Microsoft optimized an RNN-based wakeword ("Hey Cortana") detection module to fit in 1 kB of memory [47]. Overall, while edge computing is currently used for wakeword detection on edge devices, latency remains a significant issue for more complex natural language tasks (e.g., a professional translator can translate 5× faster than Google Translate with the Pixel Buds earbuds [48]), as well as the need for constant cloud connectivity.

C. Network Functions

Using deep learning for network functions, such as for intrusion detection [49], [50] and wireless scheduling [51], has been proposed. Such systems, by definition, live on the network edge and need to operate with stringent latency requirements. For example, an intrusion detection system that actively responds to a detected attack by blocking malicious packets needs to perform detection at a line rate to avoid creating a bottleneck, e.g., 40 μs [52]. If the intrusion detection system operates in the passive mode, however, its latency requirements are less strict. A wireless scheduler also needs to operate at a line rate in order to make real-time decisions on which packets should be delivered where.

In-network caching is another example of a network function that can use deep learning at the network edge. In an edge computing scenario, different end devices in the same geographical region may request the same content many times from a remote server. Caching such contents at an edge server can significantly reduce the perceived response time and network traffic. There are generally two approaches to apply deep learning in a caching system: use deep learning for content popularity prediction or use deep reinforcement learning to decide a caching policy [53]. Saputra *et al.* [54], for example, used deep learning to predict content popularity. To train the deep learning model, the cloud collects the content popularity information from all of the edge caches. Deep reinforcement learning for caching, on the other hand, avoids popularity prediction

and is solely based on reward signals from its actions. Chen *et al*. [55], for example, trained deep reinforcement learning for caching using the cache hit rate as the reward.

D. Internet of Things

Automatic understanding of IoT sensor data is desired in several verticals, such as wearables for healthcare, smart city, and smart grid. The type of analysis that is performed on these data depends on the specific IoT domain, but deep learning has been shown to be successful in several of them. Examples include human activity recognition from wearable sensors [56], pedestrian traffic in a smart city [57], and electrical load prediction in a smart grid [58]. One difference in the IoT context is that there may be multiple streams of data that need to be fused and processed together, and these data streams typically have space and time correlation that should be leveraged by the machine learning. DeepSense [56] is one framework geared toward IoT data fusion leveraging spatiotemporal relationships. It proposes a general deep learning framework that incorporates a hierarchy of CNNs (to capture multiple sensor modalities) and RNNs (to capture temporal correlations) and demonstrates how this general framework can be applied to different tasks with multiple sensor inputs: car tracking, human activity recognition, and biometric identification using inertial sensors (gyroscope, accelerometer, and magnetometer).

Another line of work in the context of IoT deep learning focuses on compressing the deep learning models to fit onto computationally weak end devices, such as Arduino or Raspberry Pi, which typically have only kilobytes of memory and low-power processors. Bonsai [59] does experiments with Arduino Uno, DeepThings [60] experiments with Raspberry Pi 3, and DeepIoT [34] works with Intel's IoT platform, the Edison board. More details on how they shrink the deep learning model to fit in memory and run on these lightweight devices are discussed in Section IV. Other examples of applying deep learning on IoT scenarios, including agriculture, industry, and smart home, can be found in the excellent survey by Mohammadi *et al*. [61].

Another motivation for edge computing with IoT devices is that the significant privacy concerns when IoT sensors are placed in public locations; for example, the Hudson Yards smart city development in New York City seeks to use air quality, noise, and temperature sensors, along with cameras, to provide advertisers with estimates of how many and how long people looked at advertisements, as well as their sentiment based on facial expressions. However, this has raised significant warnings from privacy watchdogs [3]. Thus, while analyzing IoT sensor data in real time is not always a requirement, and communication bandwidth requirements from sensors are typically small (unless cameras are involved), privacy is a major concern that motivates IoT processing on the edge.

E. Virtual Reality and Augmented Reality

In 360° virtual reality (VR), deep learning has been proposed to predict the field of view of the user [62]–[64]. These predictions are used to determine which spatial regions of the 360° video to fetch from the content provider and must be computed in real time to minimize stalls and maximize the quality-of-experience of the user. In augmented reality (AR), deep learning can be used to detect objects of interest in the user's field of view and apply virtual overlays on top [33], [65].

Latency in AR/VR is often measured in terms of the "motion-to-photons" delay. This is defined as the end-to-end delay starting from when the user moves her headset to when the display is updated in response to her movement. Motion-to-photons' latency is typically required to be on the order of tens to hundreds of milliseconds [66]. Since deep learning is only one possible part of the AR/VR pipeline (retrieving virtual objects from memory and rendering them can also consume significant latency), the motion-to-photons' latency requirement is an upper bound on the latency requirement of deep learning. The motion-to-photons' latency requirement also depends on the specific application and the type of user interaction in that application; Chen *et al*. provided the latency requirements for different cognitive assistance AR applications [67]. Since offloading AR computation to the cloud can incur latencies on the order of hundreds of milliseconds, edge computing is needed to provide satisfactory performance, as it is done in Gabriel, a cognitive assistance framework using Google Glass [68].

IV. METHODS FOR FAST INFERENCE

To enable the above-mentioned applications to meet their latency requirements, different architectures for quickly performing DNN inference have been proposed. In this section, we discuss research centered around three major architectures: 1) on-device computation, where DNNs are executed on the end device; 2) edge server-based architectures, where data from the end devices are sent to one or more edge servers for computation; and 3) joint computation among end devices, edge servers, and the cloud. We also discuss privacy-preserving techniques when data are communicated between edge devices and with the cloud, as in scenarios 1 and 2. Fig. 3 shows the taxonomy of these methods, and Fig. 5 shows the examples of different scenarios, which will be discussed in further detail in the following. Tables 2 and 3 provide a summary of the discussed works.

A. On-Device Computation

Many research efforts have focused on ways to reduce the latency of deep learning when it is executed on a resource-constrained device [see Fig. 5(a)]. Such efforts can have benefits throughout the edge ecosystem, by reducing the latency of the DNN while running on

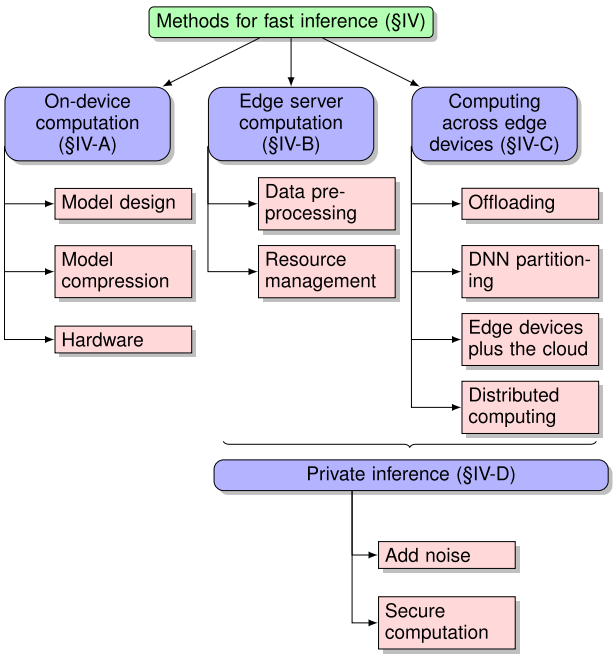


Fig. 3. *Taxonomy of DNN inference speedup methods on the edge.*

the end devices or edge servers. Here, we describe major efforts in efficient hardware and DNN model design.

1) Model Design: When designing DNN models for resource-constrained devices, machine learning researchers often focus on designing models with a reduced number of parameters in the DNN model, thus reducing memory and execution latency, while aiming to preserve high accuracy. There are many techniques for doing so, and we briefly mention several popular deep learning models for resource-constrained devices drawn from computer vision. These models include MobileNets [21], solid-state drive (SSD) [69], YOLO [18], and SqueezeNet [70], with the state of the art that is evolving rapidly. MobileNets decomposes the convolution filters into two simpler operations, reducing the number of computations needed. SqueezeNet downsamples the data using special 1×1 convolution filters. YOLO and SSD are both single shot detectors that jointly predict the location and class of the object at the same time, which is much faster than performing these steps sequentially. Many of these models, with pre-trained weights, are available for download on open-source machine learning platforms such as Tensorflow [24] and Caffe [28] for fast bootstrapping.

2) Model Compression: Compressing the DNN model is another way to enable DNNs on edge devices. Such methods usually seek to compress the existing DNN models with minimal accuracy loss compared with the original model. There are several popular model compression methods: parameter quantization, parameter pruning, and knowledge distillation. We briefly outline these approaches in the following.

Parameter quantization takes an existing DNN and compresses its parameters by changing from floating-point numbers to low-bit width numbers, thus avoiding costly floating-point multiplications. Pruning involves removing the least important parameters (e.g., those that are close to 0), as shown in Fig. 4. Quantization and pruning approaches have been considered individually as well as jointly [71]. Specifically for edge and mobile devices, DeepIoT [34] presents a pruning method for commonly used deep learning structures in IoT devices, and the pruned DNN can be immediately deployed on edge devices without modification. Lai and Suda [72] provided CMSIS-NN, a library for ARM Cortex-M processors that maximize DNN performance through quantization. It also optimizes data reuse in matrix multiplication to speed up DNN execution. Han *et al.* [73] proposed pruning and quantization for an RNN model, with $10\times$ speedup resulting from pruning and $2\times$ from quantization. Bhattacharya and Lane [74] compressed the neural network by sparsifying the fully connected layers and decomposing the convolutional filters on wearable devices.

Knowledge distillation involves creating a smaller DNN that imitates the behavior of a larger, more powerful DNN [75]. This is done by training the smaller DNN using the output predictions produced from the larger DNN. Essentially, the smaller DNN approximates the function learned by the larger DNN. Fast exiting [76] is another technique where not all layers are computed; only the result from computing the initial layers is used to provide approximate classification results.

Several works have explored the combinations of these model compression techniques. Adadeep [77] automatically chooses between different compression techniques, including pruning and the special filter structures borrowed from MobileNet and SqueezeNet, to meet application requirements and satisfy mobile resource constraints. DeepMon [78] combines quantization with caching of results from intermediate layers on GPUs. The caching leverages the insight that an input video does not change much between the subsequent frames, so some computation results from a previous frame can be reused in the current frame, reducing redundant computations and speeding up execution.

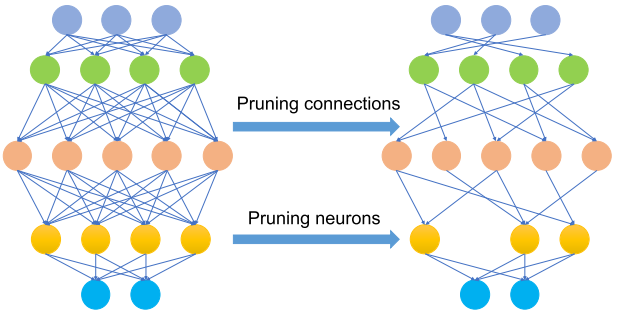


Fig. 4. *Pruning a neural network.*

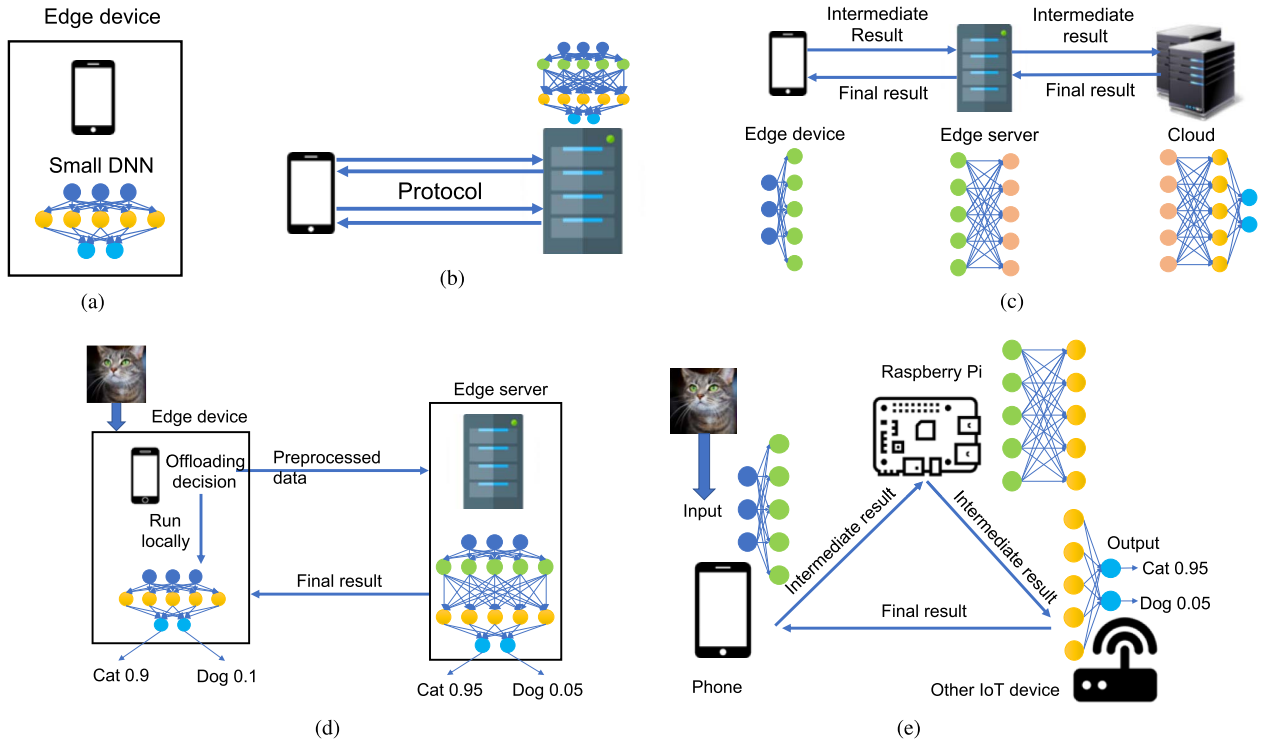


Fig. 5. ***Architectures for deep learning inference with edge computing. (a) On-device computation. (b) Secure two-party communication. (c) Computing across edge devices with DNN model partitioning. (d) Offloading with model selection. (e) Distributed computing with DNN model partitioning.***

3) Hardware: To speed up inference of deep learning, hardware manufacturers are leveraging existing hardware such as CPUs and GPUs, as well as producing custom application-specific integrated circuits (ASICs) for deep learning, such as Google's tensor processing unit (TPU) [79]. ShiDianNao [80] is another recently proposed custom ASIC, which focuses on efficient memory accesses in order to reduce latency and energy consumption. It is part of the DianNao [81] family of DNN accelerators, but it is geared toward embedded devices, which is useful in the edge computing context. field-programmable gate array (FPGA)-based DNN accelerators are another promising approach, as FPGA can provide fast computation while maintaining re-configurability [82]. These custom ASICs and FPGA designs are generally more energy efficient than the traditional CPUs and GPUs, which are designed for flexible support of various workloads at the expense of higher energy consumption.

Vendors also provide software tools for application developers to leverage the accelerations provided by the hardware. Chip manufacturers have developed software tools to optimize deep learning on the existing chips, such as Intel's OpenVINO Toolkit to leverage Intel chips, including Intel's CPUs, GPUs, FPGAs, and vision processing unit [83], [84]. Nvidia's EGX platform [85] is another recent entrant into this space, with support for Nvidia hardware ranging from lightweight Jetson Nanos to powerful T4 servers. Qualcomm's Neural Processing software development kit (SDK) is designed to utilize its Snapdragon chips [86]. There are also general libraries developed for mobile devices not tied to specific hardware, such as RSTensorFlow [87], which uses the GPU to speed up matrix multiplication in deep learning. Software approaches have also been developed to efficiently utilize hardware, e.g., Lane *et al.* [88] decomposed DNNs and assigning them to heterogeneous local processors (e.g., CPU and GPU) to accelerate execution. More detail on hardware-accelerated deep learning can be found in the excellent survey by Sze *et al.* [89]. Since Sze's survey has covered hardware-based DNN accelerations in great depth, the remainder of this paper mainly focuses on software-based approaches.

B. Edge Server Computation

While the above-mentioned hardware speedup and compression techniques can help DNNs run on end devices, deploying large, powerful DNNs with real-time execution requirement on edge devices is still challenging because of resource limitations (e.g., power, computation, and memory). Thus, it is natural to consider offloading DNN computations from end devices to more powerful entities, such as edge servers or the cloud. However, the cloud is not suitable for edge applications that require short response times [8]. Since the edge server is close to users and can respond quickly to users' request, it becomes the first-choice helper.

Table 2 Summary of the Selected Works on Fast Deep Learning Inference With Edge Computing

Architecture	Work	DNN Model	Application	End Devices	Speedup Method	Key Metrics
On-Device Computation (§IV-A)	Taylor *et al.* [23]	MobileNet [21], others	image classification	NVIDIA Jetson TX2	model selection	latency, accuracy
	DeepIoT [34]	LeNet5 [96], VGGNet, BiLSTM [97], others	text, image, speech recognition	Intel Edison computing platform	model pruning	latency, energy, memory
	Lai *et al.* [72]	7-layer CNN	image classification	Arm Cortex-M	model quantization	memory, number of operations
	ESE [73]	LSTM [98]	speech recognition	XCKU060 FPGA	model pruning, quantization	number of operations, memory
	Bhattacharya *et al.* [74]	AlexNet [99], VGGNet [100]	speech, image recognition	Qualcomm Snapdragon 400Nvidia Tegra K1ARM Cortex M0 and M3	model sparsification	memory
	Adadeep [77]	LeNet, AlexNet, and VGGNet	image, audio, activity classification	smartphones, wearable devices, development boards, smart home devices	model selection	latency, memory, energy
	DeepMon [78]	Yolo [18], MatConvNet [101]	object detection	Samsung Galaxy S7	GPU	latency, caching hit rates
	RSTensorFlow [87]	24-layer CNN, LSTM	image and hand gestures classification	Nexus 6 and Nexus 5X	GPU	latency
	DeepX [88]	AlexNet, others	speech, image recognition	Qualcomm Snapdragon 800Nvidia Tegra K1	heterogeneous processors	energy, memory
Edge Server Computation (§IV-B)	Precog [10]	general image classifiers	image classification	Nexus 7	cached specialized models	latency, accuracy, energy
	Glimpse [91]	GoogleNet [102]	feature extraction	Samsung Galaxy Nexus, Google Glass	preprocessing, offloading	accuracy, latency
	Liu *et al.* [92]	AlexNet with one more NN layer	image classification	Xiaomi Note	preprocessing, offloading	accuracy, latency, energy
	VideoStorm [93]	Caffe models	image classification	n/a (video dataset)	parameter tuning	quality of queries, latency
	Chameleon [94]	Faster RCNN [36], Yolo	object detection	n/a (video dataset)	parameter tuning	accuracy, GPU resource usage
	VideoEdge [38]	AlexNet, others	image classification	NVIDIA Tegra K1	offloading, parameter tuning	accuracy, latency
	Mainstream [95]	MobileNets, others	image classification and event detection	n/a (video dataset)	transfer learning, offloading	number of concurrent apps, accuracy
	Liu *et al.* [33]	Faster RCNN, ResNet-50 [103]	object detection	Nvidia Jetson TX2	selective offloading	accuracy, latency
Computing Across Edge Devices (§IV-C)	DeepDecision [20]	Yolo	object detection	Samsung Galaxy S7	offloading, parameter tuning	accuracy, latency, energy, network bandwidth
	MCDNN [104]	AlexNet, VGGNet, DeepFace [105]	image classification	NVIDIA Jetson board TK1	offloading, application scheduling	memory, energy, latency
	Li *et al.* [106]	AlexNet	image classification	IoT device and gateways	DNN partitioning	number of deployed tasks
	DeepThings [60]	Yolo	object detection	Raspberry Pi 3	DNN partitioning, distributed computing	memory, latency
	MoDNN [107]	MXNet [108]	image classification	LG Nexus 5	DNN partitioning, distributed computing	latency
	DDNN [109]	GoogleNet, BranchyNet [110]	image classification	n/a (simulation)	DNN partitioning	accuracy, communication cost, number of end devices

The most straightforward method to utilize the edge server is to offload all the computation from end devices to the edge server. In such scenarios, the end devices will send its data to a nearby edge server and receive the corresponding results after server processing. Wang *et al.* [90], for example, always offloaded DNNs to the edge server (an IoT gateway) to analyze wireless signals.

1) Data Preprocessing: When sending data to an edge server, data preprocessing is useful to reduce data redundancy and thus decrease communication time. Glimpse [91] offloads all DNN computation to a nearby edge server, but it uses change detection to filter which camera frames are offloaded. If no changes are detected, Glimpse will perform frame tracking locally on the end device. This preprocessing improves system processing

Table 3 Summary of the Selected Works on Privacy-Preserving Inference

Category	Work	DNN Model	Main Ideas	Key Metrics
Private Inference (§IV-D)	Wang *et al.* [115]	MobileNets, GoogLeNet, others	add noise to offloaded data, train DNN on noisy data	accuracy, energy, memory
	CryptoNets [116]	5- and 9- layer neural network	homomorphic encryption	latency, communication size
	MiniONN [117]	CNN	secure homomorphic encryption, two-party computation	accuracy, latency, communication size
	DeepSecure [118]	custom DNN and CNN, LeNet, others	secure two-party computation	latency
	Chameleon [94]	5-layer CNN	homomorphic encryption, secure two-party computation with trusted third-party	latency, communication size
	GAZELLE [119]	custom CNN	homomorphic encryption, secure two-party computation, and their tradeoffs	latency, communication size

ability and makes real-time object recognition on mobile devices possible. Along similar lines, Liu *et al*. [92] built a food recognition system with two preprocessing steps: first, they discard blurry images, and second, they crop the image so that it only contains the objects of interest. Both preprocessing steps are lightweight and can reduce the amount of offloaded data. We note that while feature extraction is a common preprocessing step in computer vision, it does not apply in the context of deep learning, because the DNNs themselves serve as the feature extractors.

2) Edge Resource Management: When DNN computations run on edge servers, DNN tasks from multiple end devices need to run and be efficiently managed on shared compute resources. Several works have explored this problem space, focusing on the tradeoffs between accuracy, latency, and other performance metrics, such as a number of requests served. VideoStorm [93] was one of the first works in this space, and profiles these tradeoffs to choose the right DNN configuration for each request, to meet the accuracy and latency goals. The configuration can also be updated online during the streaming video input, as done in Chameleon [94]. VideoEdge [38] additionally considers computation that is distributed across a hierarchy of edge and cloud servers and how to jointly tune all the DNN hyperparameters. Mainstream [95] considers a similar problem setup of accuracy versus latency tradeoffs on edge servers, but their solution uses transfer learning to reduce the computational resources consumed by each request. Transfer learning enables multiple applications to share the common lower layers of the DNN model and computes higher layers unique to the specific application, thus reducing the overall amount of computation.

C. Computing Across Edge Devices

Although the edge server can accelerate DNN processing, it is not always necessary to have the edge devices executing DNNs on the edge servers—intelligent offloading can be used instead. We next discuss four offloading scenarios: 1) binary offloading of DNN computation, where the decision is whether to offload the entire DNN or not; 2) partial offloading of partitioned DNNs, where the decision is what fraction of the DNN computations should be offloaded; 3) hierarchical architectures where offloading is performed across a combination of edge devices, edge servers, and cloud; and 4) distributed computing approaches where the DNN computation is distributed across multiple peer devices.

1) Offloading: Recent approaches, such as DeepDecision [20], [111] and MCDNN [104], take an optimization-based offloading approach with constraints such as network latency and bandwidth, device energy, and monetary cost. These decisions are based on the empirical measurements of the tradeoffs between these parameters, such as energy, accuracy, latency, and input size for the different DNN models. The catalog of different DNN models can be chosen from the existing popular models (e.g., those discussed in Section IV-A2) or new model variants can be constructed through knowledge distillation or by "mix-and-matching" DNN layers from multiple models [104]. An example of offloading, combined with model selection where a powerful DNN is available on the edge server and a weaker DNN is available on the end device, is shown in Fig. 5(d).

We note that while offloading has long been studied in the networking literature [112], even in the context of edge computing [113], DNN offloading can consider the additional degree of freedom of not only where to run, but which DNN model or which portion of the model to run. The decision of whether to offload or not thus depends on the size of the data, the hardware capabilities, the DNN model to be executed, and the network quality, among other factors.

2) DNN Model Partitioning: A fractional offloading approach can also be considered, which leverages the unique structure of DNNs, specifically its layers. In such model partitioning approaches, some layers are computed on the device, and some layers are computed by the edge server or the cloud, as shown in Fig. 5(c). This is known as DNN model partitioning. These approaches can potentially offer latency reductions by leveraging the compute cycles of other edge devices; however, care must also be taken that the latency of communicating the intermediate results at the DNN partition point still leads to overall net benefits. The intuition behind model partitioning is that after the first few layers of the DNN model have been computed, the size of the intermediate results is relatively small, making them faster to send over the network to an edge server than the original raw data [60]. This motivates the approaches that partition after the initial layers. Neurosurgeon [13] is one work that intelligently decides where to partition the DNN, layer-wise, while accounting for network conditions.

In addition to partitioning the DNN by layers, the DNN can also be partitioned along the input dimension (e.g., select rows of the input image). Such input-wise partitioning allows fine-grained partitioning, because the input and output data size and the memory footprint of each partition can be arbitrarily chosen, instead of the minimum partition size being defined by the discrete DNN layer sizes. This is especially important for extremely lightweight devices, such as IoT sensors, which may not have the necessary memory to hold an entire DNN layer. However, input-wise partitioning can result in increased data dependence, as computing subsequent DNN layers requires data results from adjacent partitions. Two examples of input-wise partitioning as MoDNN [107] and DeepThings [60].

Overall, these partial offloading approaches through DNN partitioning are similar in spirit to past, non-DNN offloading approaches such as MAUI [112] and

Odessa [114], which divide an application into its constituent subtasks, and decide which subtasks to execute where based on energy and/or latency considerations. However, a new decision in the deep learning scenario is how to decide the constituent subtasks, as the DNN can be divided layer-wise, input-wise, or possibly in other ways yet to be explored.

3) Edge Devices Plus the Cloud: Deep learning computation can be performed not only on edge devices but also on the cloud, as shown in Fig. 5(c). While solely offloading to the cloud can violate the real-time requirements of the deep learning applications under consideration, judicious use of the powerful compute resources in the cloud can potentially decrease the total processing time. Different from a binary decision of whether to perform computation on the edge server or cloud, approaches in this space often consider DNN partitioning, where some layers can execute in the cloud, edge server, and/or end device.

Li *et al*. [106] divided the DNN model into two parts—the edge server computes the initial layers of the DNN model, and the cloud computes the higher layers of the DNN. The edge server receives the input data, performs lower layer DNN processing, and then sends the intermediate results to the cloud. The cloud, after computing the higher layers, sends back the final results to the end devices. Such designs utilize both the edge server and the cloud, where the cloud can help with computationally heavy requests and increase the edge server's request processing rate while reducing the network traffic between the edge server and the cloud. DDNN [109] also distributes computation across a hierarchy of cloud, edge servers, and end devices, and additionally combines this with the fast exiting idea (discussed in Section IV-A2), so that the computation requests do not always reach the cloud.

A unique characteristic of edge computing is that the edge server typically serves users within a limited geographical area, suggesting that their input data and, thus, their DNN outputs may be similar. Precog [10] leverages this insight in the case of image recognition and places smaller, specialized image classification models on the end devices, based on what has recently been observed by other devices served by the same edge server. If on-device classification fails, the query is sent to the edge server that stores all the classification models. Although their evaluation does not use DNNs, they discuss how their classification model placement decisions would apply to DNNs. This approach has similarities to knowledge distillation for compressed models (see Section IV-A2), in which it uses a combination of weaker and stronger classification models, but it provides a more careful look at what specialized models are needed on the end devices in edge scenarios.

4) Distributed Computation: The above-mentioned approaches mainly consider offloading computation from end devices to other more powerful devices (e.g., edge servers or the cloud). Another line of work considers the problem from a distributed computing perspective, where the DNN computations can be distributed across multiple helper edge devices, as shown in Fig. 5(e). For example, MoDNN [107] and DeepThings [60] distribute DNN executions using fine-grained partitioning on lightweight end devices such as Raspberry Pis and Android smartphones. The DNN partition decision is made based on the computation capabilities and/or memory of the end devices. At runtime, the input data are distributed to helpers according to the load-balancing principles, with MoDNN using a MapReduce-like model and DeepThings designing a load-balancing heuristic. The assignment of data to the helper devices can be adjusted online to account for dynamic changes in compute resource availability or network conditions. More formal mechanisms from distributed systems could also be applied in these scenarios to provide provable performance guarantees.

D. Private Inference

When data from the end devices traverse the edge network (e.g., from end devices to edge servers, as discussed in Section IV-B), it may contain sensitive information (e.g., GPS coordinates, camera images, and microphone audio), leading to privacy concerns. This is especially important in edge computing, where the data are typically sourced from a limited set of users within a limited geographical region, making privacy breaches more concerning. Although edge computing naturally improves privacy by reducing data transfers through the public Internet to the cloud, additional techniques can further enhance privacy between end devices and edge servers and protect from eavesdroppers. In this section, we discuss two methods of privacy-preserving inference: adding noise to obfuscate the data uploaded by end devices to edge servers and secure computation using cryptographic techniques.

1) Add Noise to Data: Several works have considered how to obfuscate, or add noise, to the inference samples uploaded by end devices to a central machine (e.g., an edge server) performing inference. Wang *et al*. [115] deployed a smaller DNN locally on the device to extract features, add noise to the features, and then upload the features to the cloud for further inference processing by a more powerful DNN. The DNN on the cloud is pre-trained with noisy samples so that the noisy inference samples uploaded from the end devices can still be classified with high accuracy at test time. The formal notion of privacy used in this paper is differential privacy which, at a high level, guarantees that a machine learning model does not remember details about any specific device's input data.

2) Secure Computation: Cryptographic techniques can be used to compute the DNN prediction. The goal of secure computation in this setup is to ensure that the end device receives an inference result without learning anything about the DNN model, and the edge server processes the data without learning anything about the device's data.

In other words, an end device and an edge server want to compute the DNN prediction $f(a, b)$, where a is an input sample (e.g., a camera frame) known only to the end device and b are the DNN parameters known only to the edge server. Secure computation enables both the device and the server to compute $f(a, b)$ without knowing the other party's data.

One method of secure computation is homomorphic encryption, in which the communicated data are encrypted and computation can be performed on the encrypted data, as done in CryptoNets [116]. The idea is to approximate common computations used in DNNs, such as weighted sum, max pooling, mean pooling, sigmoid function, and rectified linear unit (RELU), by low-degree polynomials, which are amenable to homomorphic encryption. However, a bottleneck of homomorphic encryption tends to be their compute times, which means that offline preprocessing is needed. CryptoNets also requires re-training of the DNN because of the approximations used.

Multiparty computation is another technique for secure computation [see Fig. 5(b)]. In secure multiparty computation, multiple machines work together and communicate in multiple rounds to jointly compute a result (e.g., a DNN prediction in our scenario). Different from differential privacy, secure multiparty computation focuses on the privacy of the intermediate steps in the computation, while differential privacy focuses on the privacy guarantees of the overall constructed model. MiniONN [117] and DeepSecure [118] employ secure two-party computation and homographic encryption and work with the existing pre-trained DNN models without needing to change the DNN training or structure. However, a bottleneck of secure multiparty computation techniques tends to be their communication complexity.

Chameleon [120] and Gazelle [119] are two works that try to choose between the above-mentioned cryptographic techniques (homomorphic encryption and secure multiparty computation) based on their computation and communication tradeoffs. Specifically, Gazelle studies the tradeoffs between homomorphic encryption (high computation and low communication) and two-party secure computation (low computation and high communication) and chooses the right techniques for the scenario. It further accelerates the training process with efficient implementation of the cryptographic primitives. Their evaluation compared to CryptoNets, MiniONN, DeepSecure, and Chameleon using standard image classification data sets suggests low runtime latency and communication cost.

V. TRAINING IN PLACE ON EDGE DEVICES

Thus far, edge computing and deep learning have mostly been discussed for inference, with goals including low latency, privacy, and bandwidth savings. These methods assume that a deep learning model has already been trained offline on a centralized, existing data set. In this section, we discuss the methods of training deep learning models with edge computing, primarily with a focus on communication efficiency and privacy.

Traditionally, training data produced by end devices would be sent to the cloud, which would then perform the training with its large computational resources and finally distribute the trained model back to the edge devices as needed. However, sending the data to the cloud can consume large amounts of bandwidth and also has privacy concerns. Leaving data *in situ* on the end devices is useful when privacy is desired and also helps reduce the network bandwidth requirements. For example, a deep learning-based typing prediction model for smartphones may benefit from training data from multiple users, but individual users may not wish to upload their raw keystroke data to the cloud; similarly, in an image classification service, uploading all camera frames from end devices to the cloud would consume large amounts of bandwidth and risk uploading sensitive information.

Edge-based training borrows from distributed DNN training in data centers. In data centers, training is performed across multiple workers, with each worker holding either a partition of the data set (known as data parallelism) or a partition of the model (known as model parallelism). While both system designs have been explored, data parallelism is widely used in practical systems [121] and is the focus of the remainder of this section. In data parallelism, each worker computes the gradients of its local partition of the data set, which are then collected by a central parameter server, some aggregate computation performed, and the updates sent back to the workers [see Fig. 7(a)].

Training on edge devices borrows from the data center setup, where the workers are end devices instead of powerful servers in a data center, and the central parameter server is an edge compute node or server. For example, DeepCham [122] consists of a master edge server that trains domain-aware object recognition on end devices, leveraging the insight that users connected to the same edge server may have similar domains (e.g., time of day and physical environment). In an edge scenario, communication latency, network bandwidth, and the compute capabilities of the end device are key considerations of training performance.

Training deep learning on edge devices typically involves distributed deep learning training techniques. This section discusses about the techniques to perform distributed training on edge devices from the following perspectives: the frequency and size of training updates, which both contributes to communication cost (see Sections V-A and V-B, respectively); decentralized information sharing (see Section V-C); and finally, privacy-preserving DNN training (see Section V-D). A taxonomy of these techniques is shown in Fig. 6, and a summary of the works discussed in the following is shown in Table 4.

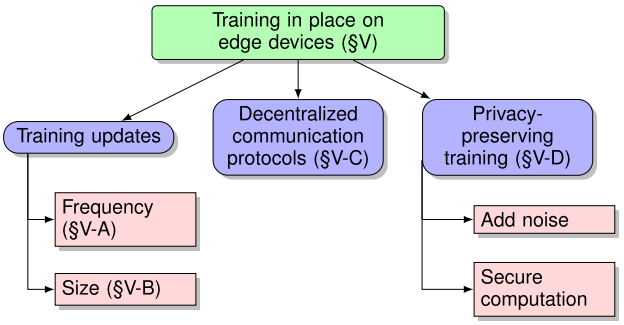


Fig. 6. *Taxonomy of DNN training in place on edge devices.*

A. Frequency of Training Updates

Communication costs are a major concern for edge devices. Reducing the frequency of communications and the size of each communication is a key method to reduce communication costs. In this section, we discuss the distributed training methods that focus on communication timing and frequency, while in Section V-B, we discuss the size of the communicated data. There are two general methods for synchronizing updates to a central edge server: synchronous and asynchronous stochastic gradient descent [123]. In synchronous stochastic gradient descent (SGD), individual devices update their parameters in lockstep when all the devices have finished computing the gradients on their current batch of training data. In asynchronous SGD, the devices update their parameters independently to the central server. Both synchronous SGD and asynchronous SGD have their own pros and cons. Although synchronous SGD typically converges to better solutions, it is often slower in practice because of the need to wait for straggler devices in each iteration. Asynchronous SGD, on the other hand, tends to converge faster than synchronous SGD, but it may update parameters using stale information from devices and can suffer from convergence to poor solutions.

Distributed training algorithms usually focus on how to make synchronous SGD faster or how to make asynchronous SGD converge to better solutions. In a distributed setting, communication frequency and data volume are also important. Elastic averaging [124] reduces the communication costs of synchronous and asynchronous SGD training methods, by allowing each device to perform more local training computations and deviate/explore further from the globally shared solution before synchronizing its updates. This reduces the amount of communication between the local devices and the edge server. Federated learning [125] is similar in spirit, but it considers non-ideal scenarios, such as non-independent and identically distributed (i.i.d) data distributions (e.g., one device has more data samples of a given class than another device). Computing more local gradient updates without uploading the raw training data to the server trades off accuracy for communication cost: doing more computation locally lowers the prediction accuracy (due to overfitting to local data sets), but it can also save communication cost, and vice versa. Wang *et al*. [126] further explored this issue by considering some practical concerns with implementation on a real testbed. They proposed a control policy for deciding how much computation should be performed locally in between global gradient updates and performed experiments with Raspberry Pis and laptops.

Tiered architectures and their communication costs have also been considered. Gaia [127] studies synchronous SGD in the scenario where devices are geographically distributed across a large area. In their test setup, the clients are servers inside a data center and across data centers. Because bandwidth constraints are tighter across geo-distributed data centers than within a single data center, gradient updates need to be carefully coordinated between the workers. Gaia proposes a policy where updates are synchronized across different data centers only

Table 4 Summary of the Selected Works on Distributed Training

Category	Work	DNN Model	Main Ideas	Key Metrics
Communication frequency (§V-A)	EASGD [124]	7-layer CNN	allow local parameters to deviate from central parameters	training loss, test loss
	Federated Learning [125]	CNN and LSTM	trade off local computation for communication rounds	training loss, test loss, # of communication rounds
	Gaia [127]	Caffe model	hierarchical communication of geo-distributed nodes	training time
	Codistillation [128]	LSTM	teacher-student models	accuracy
	Dean *et al*. [121]	fully-connected DNN with 42 million parameters	partition DNN across different machines.	# of training nodes, training time
Communication size (§V-B)	Lin *et al*. [129]	AlexNet, ResNet-5, 2- or 5-layer LSTM	send sparsified gradient	gradient size
	Hardy *et al*. [130]	CNN with 2 convolution layer and 2 fully connected layer	send sparsified and non-stale gradients	accuracy
Gossip-based communication (§V-C)	Blot *et al*. [131]	7-layer CNN	gossip with random neighbors	training loss
	Jin *et al*. [132]	ResNet	scalability of synchronous and asynchronous SGD	test error, training time/epochs
	INCEPTIONN [133]	ResNet-50, AlexNet, VGG-16	lossy gradient compression and hierarchical gossip	gradient size, # of training nodes, gradient exchange time
Private training (§V-D)	Shokri *et al*. [14]	CNN	modify gradient updates for per-parameter privacy loss	accuracy, differential privacy
	Abadi *et al*. [134]	CNN	modify gradient updates for per-model privacy loss	accuracy, differential privacy
	Zhang *et al*. [135]	34-layer residual network, LeNet, others	add noise to training data	accuracy with privacy leaking defense
	SecureML [136]	two hidden layers with 128 neurons in each layer	secure two-party computation	accuracy with secure computation
	Mao *et al*. [137]	VGG-16	add noise to offloaded data, DNN partition	accuracy

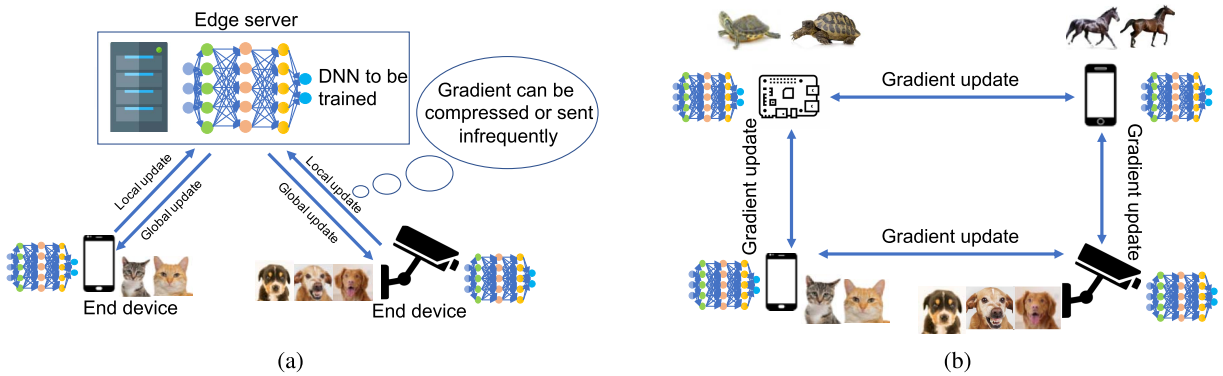


Fig. 7. *Architectures for deep learning training on the edge. (a) Centralized training. (b) Decentralized training.*

when the aggregated updates are higher than a given threshold.

Along with synchronous and asynchronous updates, distillation is another method that has been applied to reduce communication frequency. Distillation, as discussed in Section IV-A in the context of inference, uses the prediction outputs of one model to help train another model. Anil *et al*. [128] proposed incorporating distillation into distributed training of DNNs. In their method, each device trains on a subset of the data and updates its gradients based on its computed training loss as usual, but it also uses the prediction outputs from other devices that are also simultaneously training to improve training efficacy. Since they find that the training is robust to stale prediction results from other devices, information needs to be exchanged with other devices less frequently (compared to the gradient sharing methods described earlier in this section). In this way, frequent communication of gradients is avoided or reduced. Furthermore, distillation can be combined with distributed SGD and can improve training efficacy even when distributed SGD is not possible due to network constraints.

Finally, if some devices have poor connectivity and are subjected to atypically long latencies, they can hold up distributed training. Chen *et al*. [123] proposed improvements to synchronous SGD to mitigate such straggler effects. Their main idea is to have backup devices that are "on call" to compute the gradient updates of any straggling regular devices. Once the server receives the gradient updates from a sufficient of devices, the training process will update the global parameters and move on to next iteration, without waiting for the straggler devices, thereby reducing training latency.

B. Size of Training Updates

Along with the frequency of training updates, the size of training updates also contributed to bandwidth usage. With model sizes on the order of hundreds of megabytes, and multiple rounds of communication needed, the bandwidth demands can be considerable. Bandwidth concerns are crucial in the edge scenario, where last-mile bandwidth (e.g., wireless and access networks) can be quite constrained. In this section, we review gradient compression techniques, which can reduce the size of the updates communicated to a central server.

There are two general approaches to gradient compression: gradient quantization and gradient sparsification [129]. Gradient quantization approximates the floating-point gradients using low-bit width numbers. For example, a 32-bit floating-point numbers can be approximated by an 8-bit number, reducing the size by a factor of 4. Note that gradient quantization is similar to parameter quantization (see Section IV-A), with the difference being whether the quantization is applied to the model gradients or the model parameters. Gradient sparsification discards unimportant gradient updates and only communicates updates that exceed a certain threshold. Gradient quantization and sparsification can work together. For example, Lin *et al*. [129] did gradient sparsification combined with other training tricks such as momentum correction [138] and warm-up training [139] techniques to speed up training convergence. Hardy *et al*. [140] performed gradient sparsification and also chose which gradients to communicate based on their staleness.

C. Decentralized Communication Protocols

Thus far, we have considered centralized training architectures where multiple end devices communicate with an edge server. Having a central edge compute or server node helps ensure that all devices converge to the same model parameters. However, communication throughput of a centralized architecture is limited by the bandwidth of the central node. To overcome this, a gossip-type algorithm has been proposed as a method to exchange training information in a decentralized fashion. In gossip-type algorithms, each device computes its own gradient updates based on its training data and then communicates its updates to some of the other devices [see Fig. 7(b)]. The overall goal is to design a gossiping algorithm that allows the devices to reach a consensus on a good DNN model. Gossiping can be considered as a decentralized version of elastic averaging,

where clients are allowed to deviate more significantly from each other.

Blot *et al*. [131] proposed an asynchronous algorithm for gossip-based training of deep learning. Their experiments show faster convergence than the elastic averaging. Jin *et al*. [132] proposed gossiping SGD based on their study of convergence rates for synchronous and asynchronous SGD. Their primary concern was scalability, i.e., which SGD methods would be appropriate for a different number of clients. They found that asynchronous methods, such as gossiping and elastic averaging, converged more quickly with a small number of workers (32 workers in their simulations), whereas synchronous SGD scaled up better and had higher accuracy when there were more workers (100 workers in their simulations). Li *et al*. [133] developed a distributed system called INCEPTIONN, which combines gradient compression and gossiping. Their gossiping method involves dividing devices into different groups, and within each group, each device shares some of its gradients with the next device. The algorithm guarantees that all parts of the DNN across all devices will be updated after several iterations. Within each group, the parameters can be shared either in the traditional centralized way or through gossiping.

D. Private Training

We now shift gears and return to the baseline SGD algorithms (e.g., synchronous SGD) but consider the privacy implications of communicating gradient information. Such techniques can be useful whenever the training data collected by end devices are shared with other edge devices. While *in situ* training naturally improves privacy by eliminating direct sharing of end devices' collected data, gradient information communicated between the edge devices can still indirectly leak information about the private data [14]. Hence, further privacy-enhancing techniques are needed. In this section, we will consider two main classes of privacy-enhancing techniques: adding noise to gradient or data transmissions as part of training and secure computation for training a DNN.

1) Add Noise to Data or Training Updates: In the following works, the threat model consists of a passive adversary, such as an end device, which follows the prescribed training protocol and is not actively malicious, but it may attempt to learn about the model or data from observing others' communicated data. Shokri and Shmatikov [14] considered the privacy aspects of training a DNN in such a scenario, specifically with respect to differential privacy, and they modified the typical policy of devices uploading all the gradients to a central server by: 1) selecting only some gradients above a threshold to be transmitted and 2) adding noise to each uploaded gradient. This enables the model to be trained reasonably accurately while reducing information leakages from the training updates (intuitively, since fewer of the updates are sent). Abadi *et al*. [134] studied a similar problem where the privacy loss over the overall model was bounded, rather than per parameter as in Shokri, and their method involves modifying the gradient by clipping, averaging, and adding noise before communicating it to the parameter server. Mao *et al*. [137] combined differential privacy with model partitioning, where the initial layers of the DNN were computed on the device, mixed with noise, and uploaded to the edge server, in order to obfuscate the uploaded training data and preserve privacy.

Along with modifying the gradients, adding noise to the training data has also been considered. Zhang [135] considered different types of noise that can be added to the input data before training. Rather than using formal notions of differential privacy, they empirically guard against the adversary discovering statistical properties of individual training data samples or aggregate statistics about groups of training samples. This is essentially a preprocessing step for the training data, which can provide protection even if the adversary has taken over the parameter server and has access to the model parameters or post-processed training data.

2) Secure Computation: SecureML [136] proposes a two-server model where end devices communicate their data to two servers, which then train a neural network based on the combined data from the end devices without learning anything beyond the DNN parameters. Their scheme is based on secure two-party linear and logistic regression that are fundamental computations in DNN training. A modified softmax function and RELU function are also proposed in order to improve efficiency. Unlike the multiparty computation schemes discussed earlier (see Section IV-D), SecureML focuses on DNN training, rather than inference.

VI. OPEN CHALLENGES

Many challenges remain in deploying deep learning on the edge, not only on end devices but also on the edge servers and on a combination of end devices, edge servers, and the cloud. We next discuss some of the open challenges.

A. Systems Challenges

1) Latency: While several works described in the earlier sections have focused on reducing inference latency, the current state of the art still results in quite high latency, particularly when operating on high-dimensional input data, such as images, and on mobile devices. For example, even DNN models designed for mobile devices execute at 1–2 frames/s on modern smartphones [20], [78]. There is still much work remaining on DNN model compression to enable deep learning to run on edge devices, particularly on IoT devices that tend to have the most severe resource constraints. Furthermore, while the offloading approaches described earlier (see Sections IV-B and IV-C) propose innovative approaches to minimize latency, machine learning

experts are also constantly innovating, leading to new DNN models with ever more parameters and new layer designs. For example, the DNN partitioning approach may work well for standard sequential DNNs, but not as well for other deep learning methods such as RNNs, which have loops in their layer structure. Keeping up with new deep learning designs will continue to be a major systems' challenge.

2) Energy: Minimizing the energy consumption of deep learning is very important for battery-powered edge devices, such as smartphones. While reducing the amount of computation implicitly reduces energy consumption, understanding the interactions of the deep learning computations with other battery management mechanisms, such as CPU throttling or sensor hardware optimizations [141], is an important avenue for investigation. Performing change detection on the input data, either in software or hardware [142], can help reduce the frequency of deep learning executions and the overall energy consumption. Reducing energy consumption of the specific hardware chips (e.g., GPUs and TPUs) is already a key priority for hardware designers, but understanding their interaction with the rest of the system (e.g., battery management mechanisms and tradeoffs with edge server compute resources) is needed to reduce overall energy consumption.

3) Migration: Migrating edge computing applications between different edge servers can be useful for load balancing or to accommodate user movement, with the goal of minimizing the end-to-end latency of the user. While edge migration has been studied in the general case, for example, using VM migration techniques [143] or Docker containers [144] or using multipath TCP to speed up the migration [145], understanding how deep learning applications should be migrated is still an area of consideration. DNN models can be fairly large; for example, a pre-trained YOLO model [18] is approximately 200 MB, and loading a DNN model can take several seconds, in our experience. What parts of the DNN model should be migrated and what parts should be included in the standalone virtual image? Can the program state be migrated in the midst of a DNN execution, similar to the DNN partitioning approaches for offloading (see Section IV-C2)? Addressing these challenges requires system measurements and experiments to gain an empirical understanding of the migration challenges.

B. Relationship to SDN and NFV Technologies

Recently, network abstractions, such as software-defined networking (SDN), to abstract the data plane from the control plane, and network function virtualization (NFV), to abstract the network functions from the hardware, are gaining importance and are being adopted by the telecommunications industry. If deep learning grows in popularity and these flows containing deep learning data appear on the edge network, this leads to questions of how SDN and NFV should manage these types of flows and what types of QoS guarantees the flows require. How can deep learning flows be identified, even under encryption? Given a set of network functions that need to operate on deep learning flows, how to design an SDN controller to best manage these flows (e.g., by carving out network slices for deep learning traffic)? How should network resources be shared between competing deep learning flows or with other non-deep learning traffic, such as Web or video?

Another direction is using deep learning itself as a network function, such as the network intrusion detection and caching applications described in Section III. If deep learning is adopted for various network tasks, NFV platforms need to account for the resource requirements of deep learning in order, for the network functions, to run in real time. While fast instantiation and performance of NFVs has been investigated [146], deep learning inference can be greatly accelerated with GPU access on the edge server, necessitating GPU support in NFV platforms.

C. Management and Scheduling of Edge Compute Resources

Deep learning is often treated as a black box by application developers and network administrators. However, deep learning models have many tradeoffs between latency, accuracy, battery, and so on. While several works described earlier (in Section IV-C) have discussed how to tune such control knobs to achieve overall good system performance [20], [93], exposing these control knobs in a consistent and unified manner to the application developer and/or server administrator through a standard specification could be valuable. This would enable developers and server administrators without in-depth machine learning knowledge to understand the available knobs and tune them to achieve good system performance, especially on edge compute nodes with limited resources. Specifying the application's needs and the tradeoffs of the DNN model being run can allow the edge server to effectively schedule the end device requests. Not doing this carefully (e.g., incurring long latency on a video frame analysis request from an AR headset) would negate the latency benefits of edge computing.

A natural question is then how to schedule such requests, given knowledge of the tradeoffs and control knobs. The question is complicated by time dependence between sequential inputs from an end device (e.g., multiple frames from a camera), which could introduce priority into the scheduling policy and thus influence the decisions made by the edge server of which requests to serve when. For example, should a new camera frame inference request from device A receive higher priority than the hundredth frame from device B? Incorporating freshness metrics, such as the age of information [147], could allow for more intelligent scheduling decisions by the edge server. While this problem has some overlap with task scheduling in

cloud data centers, edge computing brings new challenges in which the number and variety of requests are likely less on an edge server serving geo-located end devices, so statistical multiplexing cannot necessarily be relied on. New analysis of load balancing and request scheduling mechanisms is needed. Furthermore, the compute resource allocations may be coupled with the traffic steering from the end devices to the edge server. Existing work on mainly considers proximity as the primary factor behind traffic steering decisions [148].

D. Deep Learning Benchmarks on Edge Devices

The state of the art of deep learning is evolving rapidly. For researchers and developers wishing to deploy deep learning on edge devices, choosing the right DNN model is difficult due to lack of apples-to-apples comparison on the target hardware. Even though new machine learning papers contain comparative evaluation with prior existing models, the subset of models compared is chosen at the discretion of the researchers and may not include the desired comparisons or hardware platforms. Furthermore, standalone measurement papers can quickly become outdated as new DNN models emerge. A public repository containing apples-to-apples containing benchmark comparisons between the models on different hardwares could be of great benefit to the community. This task is made slightly easier by the existence of standard data sets in certain application domains, such as image classification and natural language processing, as well as standard machine learning platforms such as TensorFlow, Caffe, and PyTorch. Especially important to edge computing is the comparison on a variety of edge device hardware, including the simple devices (e.g., Raspberry Pi), smartphones, home gateways, and edge servers. Much of the current work has focused on either on powerful servers or on smartphones, but as deep learning and edge computing become prevalent, a comparative understanding of deep learning performance on heterogeneous hardware is needed.

E. Privacy

While privacy has been studied generally in the context of distributed deep learning, there are several implications for edge computing, which merit further investigation. One possible concern is membership attacks. A membership attack seeks to determine whether a particular item was part of the training set used to generate the deep learning model [149]. This attack gains significance in edge computing, as a successful attack on an edge server's DNN training process means that the data item can be more easily pinpointed as belonging to a small subset of users who accessed that edge server. Another concern is data obfuscation. While data obfuscation techniques have been studied in cases where there are a large number of users, such as in the cloud, whether such obfuscation can still be successful in an edge computing scenario, where more specialized deep learning models are being used [10], [122], or smaller training sets are available due to fewer end devices connected to each edge server is unclear. Finally, the definition of differential privacy [150] means that as there are fewer devices, more noise must be added. This is exactly the scenario of edge computing, where a smaller set of geo-located end devices communicate with an edge server. How much noise must be added to compensate for fewer end devices? Overall, the privacy problems described earlier (see Sections IV-D and V-D) have been studied mainly in the context of general distributed machine learning, but their study with regard to edge computing, which has a smaller set of users and more specialized deep learning models, could be valuable.

VII. CONCLUSION

This paper reviewed the current state of the art for deep learning operating on the network edge. Computer vision, natural language processing, network functions, and VR and AR were discussed as example application drivers, with the commonality being the need for real-time processing of data produced by end devices. Methods for accelerating deep learning inference across end devices, edge servers, and the cloud were described, which leverage the unique structure of DNN models as well as the geospatial locality of user requests in edge computing. The tradeoffs between accuracy, latency, and other performance metrics were found to be important factors in several works discussed. Training of deep learning models, where multiple end devices collaboratively train a DNN model (possibly with the help of an edge server and/or the cloud) was also discussed, including techniques for further enhancing privacy.

Many open challenges remain, both in terms of further performance improvements, as well as privacy, resource management, benchmarking, and integration with other networking technologies such as SDN and NFV. These challenges can be addressed through technological innovations in algorithms, system design, and hardware accelerations. As the pace of deep learning innovation remains high in the near term, new technical challenges in edge computing may emerge in the future, alongside the existing opportunities for innovation. ■

REFERENCES

[1] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

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

[3] D. Jeans. (Mar. 2019). *Related's Hudson Yards: Smart City or Surveillance City?* [Online]. Available: https://therealdeal.com/2019/03/15/hudson-yards-smart-city-or-surveillance-city/

[4] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009.

[5] *AT&T Multi-Access Edge Computing*. [Online]. Available: https://www.business.att.com/products/multi-

access-edge-computing.html
[6] C.-F.-A. T. Blog. *Edge Computing at Chick-Fil-A*. [Online]. Available: https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2
[7] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*, vol. 1. Cambridge, MA, USA: MIT Press, 2016.
[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
[9] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 4th Quart., 2017.
[10] U. Drolia, K. Guo, and P. Narasimhan, "Precog: Prefetching for image recognition applications at the edge," in *Proc. ACM/IEEE Symp. Edge Comput.*, 2017, pp. 1–17.
[11] H. Li, K. Ota, and M. Dong, "Learning IoT in edge: Deep learning for the Internet of Things with edge computing," *IEEE Netw.*, vol. 32, no. 1, pp. 96–101, Jan./Feb. 2018.
[12] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, 2017, pp. 82–95.
[13] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 615–629, 2017.
[14] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1310–1321.
[15] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, pp. 1–14, Sep. 2016.
[16] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.
[17] J. Huang *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors," in *Proc. IEEE CVPR*, vol. 4, Jul. 2017, pp. 7310–7311.
[18] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE CVPR*, Jul. 2017, pp. 7263–7271.
[19] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices," in *Proc. ACM Multimedia*, 2017, pp. 1663–1671.
[20] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "DeepDecision: A mobile deep learning framework for edge video analytics," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1421–1429.
[21] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: https://arxiv.org/abs/1704.04861
[22] M. Tan *et al.*, "MnasNet: Platform-aware neural architecture search for mobile," 2018, *arXiv:1807.11626*. [Online]. Available: https://arxiv.org/abs/1807.11626
[23] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive deep learning model selection on embedded systems," in *Proc. LCTES*, 2018, pp. 31–43.
[24] *Tensorflow*. [Online]. Available: https://www.tensorflow.org/
[25] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, pp. 1–19, Mar. 2016.
[26] *Tensorflowlite*. [Online]. Available: https://www.tensorflow.org/lite
[27] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[28] *Caffe2*. [Online]. Available: https://caffe2.ai/
[29] *Caffe*. [Online]. Available: https://caffe.berkeleyvision.org/
[30] *Pytorch*. [Online]. Available: https://pytorch.org
[31] NVIDIA. *Cuda*. [Online]. Available: https://developer.nvidia.com/cuda-zone
[32] NVIDIA. *Cudnn*. [Online]. Available: https://developer.nvidia.com/cudnn
[33] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *Proc. ACM MobiCom*, 2019, pp. 1–16.
[34] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "DeepIoT: Compressing deep neural network structures for sensing systems with a compressor-critic framework," in *Proc. SenSys*, 2017, pp. 1–4.
[35] Amazon. *AWS Deeplens*. [Online]. Available: https://aws.amazon.com/deeplens/
[36] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-Cnn: Towards real-time object detection with region proposal networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[37] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proc. 21st Annu. Int. Conf. Mobile Comput. Netw.*, 2015, pp. 426–438.
[38] C.-C. Hung *et al.*, "VideoEdge: Processing camera streams using hierarchical clusters," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 115–131.
[39] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, Aug. 2018.
[40] Apple. *Deep Learning for Siri's Voice: On-Device Deep Mixture Density Networks for Hybrid Unit Selection Synthesis*. [Online]. Available: https://machinelearning.apple.com/2017/08/06/siri-voices.html
[41] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," 2016, *arXiv:1603.01360*. [Online]. Available: https://arxiv.org/abs/1603.01360
[42] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*. [Online]. Available: https://arxiv.org/abs/1609.08144
[43] Google AI Blog. *Google Duplex: An AI System for Accomplishing Real-World Tasks Over the Phone*. [Online]. Available: https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html
[44] S. Antol *et al.*, "VQA: Visual question answering," in *Proc. IEEE Int. Conf. Comput. Vis.*, Dec. 2015, pp. 2425–2433.
[45] *Pagespeed Insights: Improve Server Response Time*, Google.
[46] Apple. (2017). *Hey Siri: An On-Device DNN-Powered Voice Trigger for Apple's Personal Assistant*. [Online]. Available: https://machinelearning.apple.com/2017/10/01/hey-siri.html
[47] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 9017–9028.
[48] Raymond Wong. *Google's Pixel Buds are no Match for Professional Interpreters*. [Online]. Available: https://mashable.com/2017/12/05/google-pixel-buds-real-time-translations-vs-un-interpreter/?europe=true
[49] J. Ryan, M.-J. Lin, and R. Miikkulainen, "Intrusion detection with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 1998, pp. 943–949.
[50] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," 2018, *arXiv:1802.09089*. [Online]. Available: https://arxiv.org/abs/1802.09089
[51] S. Chinchali *et al.*, "Cellular network traffic scheduling with deep reinforcement learning," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 1–9.
[52] N. Tsikoudis, A. Papadogiannakis, and E. P. Markatos, "LEoNIDS: A low-latency and energy-efficient network-level intrusion detection system," *IEEE Trans. Emerg. Topics Comput.*, vol. 4, no. 1, pp. 142–155, Jan. 2016.
[53] H. Zhu, Y. Cao, W. Wang, T. Jiang, and S. Jin, "Deep reinforcement learning for mobile edge caching: Review, new features, and open issues," *IEEE Netw.*, vol. 32, no. 6, pp. 50–57, Nov. 2018.
[54] Y. M. Saputra, D. T. Hoang, D. N. Nguyen, E. Dutkiewicz, D. Niyato, and D. I. Kim, "Distributed deep learning at the edge: A novel proactive and cooperative caching framework for mobile edge networks," *CoRR*, vol. abs/1812.05374, pp. 1–4, Dec. 2018.
[55] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," *CoRR*, vol. abs/1712.08132, pp. 1–6, Dec. 2017.
[56] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, "DeepSense: A unified deep learning framework for time-series mobile sensing data processing," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 351–360.
[57] W. Ouyang and X. Wang, "Joint deep learning for pedestrian detection," in *Proc. IEEE Int. Conf. Comput. Vis.*, Dec. 2013, pp. 2056–2063.
[58] L. Li, K. Ota, and M. Dong, "When weather matters: IoT-based electrical load forecasting for smart grid," *IEEE Commun. Mag.*, vol. 55, no. 10, pp. 46–51, Oct. 2017.
[59] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the Internet of Things," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, 2017, pp. 1935–1944.
[60] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.
[61] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for IoT big data and streaming analytics: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2923–2960, 4th Quart., 2018.
[62] X. Hou, S. Dey, J. Zhang, and M. Budagavi, "Predictive view generation to enable mobile 360-degree and VR experiences," in *Proc. Morning Workshop Virtual Reality Augmented Reality Netw.*, 2018, pp. 20–26.
[63] Y. Xu *et al.*, "Gaze prediction in dynamic 360 immersive videos," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 5333–5342.
[64] S. Afzal, J. Chen, and K. K. Ramakrishnan, "Characterization of 360-degree videos," in *Proc. ACM SIGCOMM Workshop Virtual Reality Augmented Reality Netw.*, 2017, pp. 1–6.
[65] A. Jindal *et al.* (Jan. 2018). *Enabling Full Body AR With Mask R-CNN2GO*. [Online]. Available: https://research.fb.com/enabling-full-body-ar-with-mask-r-cnn2go/
[66] S. LaValle, *Virtual Reality*. Cambridge, U.K.: Cambridge Univ. Press, 2016.
[67] Z. Chen *et al.*, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, Oct. 2017, p. 14.
[68] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. ACM MobiSys*, 2014, pp. 68–81.
[69] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.* Springer, 2016, pp. 21–37.
[70] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*. [Online]. Available: https://arxiv.org/abs/1602.07360
[71] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: https://arxiv.org/abs/1510.00149
[72] L. Lai and N. Suda, "Enabling deep learning at the IoT edge," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, 2018, p. 135.
[73] S. Han *et al.*, "ESE: Efficient speech recognition

engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 75–84.
[74] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proc. 14th ACM Conf. Embedded Netw. Sensor Syst. CD-ROM (SenSys)*, 2016, pp. 176–189.
[75] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*. [Online]. Available: https://arxiv.org/abs/1503.02531
[76] S. Teerapittayanon, B. McDanel, and H. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. Int. Conf. Pattern Recognit.*, Dec. 2016, pp. 2464–2469.
[77] S. Yao, Y. Zhao, Z. Aston, L. Su, and T. Abdelzaher, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proc. MobiSys*, 2018, pp. 389–400.
[78] N. Loc Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. ACM MobiSys*, 2017, pp. 82–95.
[79] *Edge TPU*. [Online]. Available: https://cloud.google.com/edge-tpu/
[80] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 92–104, 2015.
[81] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "Diannao family: Energy-efficient hardware accelerators for machine learning," *Commun. ACM*, vol. 59, no. 11, pp. 105–112, 2016.
[82] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/CNN20Whitepaper.pdf
[83] *VPU*. [Online]. Available: https://www.movidius.com/solutions/vision-processing-unit
[84] S. Rivas-Gomez, A. J. Pena, D. Moloney, E. Laure, and S. Markidis, "Exploring the vision processing unit as co-processor for inference," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 589–598.
[85] Nvidia. *NVIDIA EGX Edge Computing Platform*. [Online]. Available: https://www.nvidia.com/en-us/data-center/products/egx-edge-computing/
[86] Qualcomm. *Qualcomm Neural Processing SDK for AI*. [Online]. Available: https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk
[87] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "RSTensorFlow: GPU enabled tensorflow for deep learning on commodity android devices," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl. (EMDL)*, 2017, pp. 7–12.
[88] N. D. Lane *et al.*, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, 2016, p. 23.
[89] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[90] X. Wang, X. Wang, and S. Mao, "RF sensing in the Internet of Things: A general deep learning framework," *IEEE Commun. Mag.*, vol. 56, no. 9, pp. 62–67, Sep. 2018.
[91] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proc. ACM SenSys*, 2015, pp. 155–168.
[92] C. Liu *et al.*, "A new deep learning-based food recognition system for dietary assessment on an edge computing service infrastructure," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 249–261, Jan. 2018.
[93] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proc. USENIX NSDI*, 2017, pp. 377–392.
[94] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 253–266.
[95] A. H. Jiang *et al.*, "Mainstream: Dynamic stem-sharing for multi-tenant video processing," in *Proc. USENIX Annu. Tech. Conf. (USENIXATC)*, 2018, pp. 29–42.
[96] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[97] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF models for sequence tagging," *CoRR*, vol. abs/1508.01991, pp. 1–10, Aug. 2015.
[98] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[99] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[100] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, pp. 1–14, Sep. 2014.
[101] A. Vedaldi and K. Lenc, "MatConvNet: Convolutional neural networks for MATLAB," in *Proc. 23rd ACM Int. Conf. Multimedia (MM)*, 2015, pp. 689–692.
[102] C. Szegedy *et al.*, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, pp. 1–12, Sep. 2014.
[103] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, pp. 1–12, Dec. 2015.
[104] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An approximation-based execution framework for deep stream processing under resource constraints," in *Proc. ACM Mobisys*, 2016, pp. 123–136.
[105] Y. Taigman, M. Yang, M. A. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 1701–1708.
[106] H. Li, K. Ota, and M. Dong, "Learning IoT in edge: Deep learning for the Internet of Things with edge computing," *IEEE Netw.*, vol. 32, no. 1, pp. 96–101, Jan. 2018.
[107] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2017, pp. 1396–1401.
[108] *Arcore Overview*. [Online]. Available: https://mxnet.apache.org/
[109] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 328–339.
[110] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," *CoRR*, vol. abs/1709.01686, pp. 1–7, Sep. 2017.
[111] X. Ran, H. Chen, Z. Liu, and J. Chen, "Delivering deep learning to mobile devices via offloading," in *Proc. ACM SIGCOMM Workshop Virtual Reality Augmented Reality Netw.*, 2017, pp. 42–47.
[112] E. Cuervo *et al.*, "MAUI: Making smartphones last longer with code offload," *ACM MobiSys*, 2010, pp. 49–62.
[113] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proc. ACM/IEEE Symp. Edge Comput.*, 2017, p. 15.
[114] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," *ACM MobiSys*, 2011, pp. 43–56.
[115] J. Wang, J. Zhang, W. Bao, X. Zhu, B. Cao, and P. S. Yu, "Not just privacy: Improving performance of private deep learning in mobile cloud," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2018, pp. 2407–2416.
[116] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 201–210.
[117] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 619–631.
[118] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable provably-secure deep learning," in *Proc. 55th Annu. Design Autom. Conf.*, 2018, p. 2.
[119] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *Proc. 27th USENIX Secur. Symp. (USENIX Security)*, 2018, pp. 1651–1669.
[120] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *Proc. Asia Conf. Comput. Commun. Secur.*, 2018, pp. 707–721.
[121] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.
[122] D. Li, T. Salonidis, N. V. Desai, and M. C. Chuah, "DeepCham: Collaborative edge-mediated adaptive deep learning for mobile object recognition," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, 2016, pp. 64–76.
[123] J. Chen, R. Monga, S. Bengio, and R. Józefowicz, "Revisiting distributed synchronous SGD," *CoRR*, vol. abs/1604.00981, pp. 1–10, Apr. 2016.
[124] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD," in *Advances in Neural Information Processing Systems*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2015, pp. 685–693.
[125] H. B. McMahan *et al.*, "Communication-efficient learning of deep networks from decentralized data," 2016, *arXiv:1602.05629*. [Online]. Available: https://arxiv.org/abs/1602.05629
[126] S. Wang *et al.*, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," *CoRR*, vol. abs/1804.05271, pp. 1–20, Feb. 2018.
[127] K. Hsieh *et al.*, "Gaia: Geo-distributed machine learning approaching LAN speeds," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 629–647.
[128] R. Anil *et al.*, "Large scale distributed neural network training through online distillation," *CoRR*, vol. abs/1804.03235, pp. 1–12, Apr. 2018.
[129] Y. Lin, S. Han, H. Mao, Y. Wang, and J. William Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *CoRR*, vol. abs/1712.01887, pp. 1–13, Feb. 2017.
[130] C. Hardy, E. L. Merrer, and B. Sericola, "Distributed deep learning on edge-devices: Feasibility via adaptive compression," *CoRR*, vol. abs/1702.04683, pp. 1–8, Nov. 2017.
[131] M. Blot, D. Picard, M. Cord, and N. Thome, "Gossip training for deep learning," Nov. 2016, *arXiv:1611.09726*. [Online]. Available: https://arxiv.org/abs/1611.09726
[132] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer, "How to scale distributed deep learning?" 2016, *arXiv:1611.04581*. [Online]. Available: https://arxiv.org/abs/1611.04581
[133] Y. Li *et al.*, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 175–188.
[134] M. Abadi *et al.*, "Deep learning with differential privacy," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 308–318.

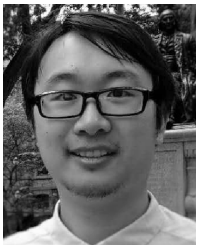
[135] T. Zhang, Z. He, and R. B. Lee, "Privacy-preserving machine learning through data obfuscation," Jul. 2018, *arXiv:1807.01860*. [Online]. Available: https://arxiv.org/abs/1807.01860
[136] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 19–38.
[137] Y. Mao, S. Yi, Q. Li, J. Feng, F. Xu, and S. Zhong, "Learning from differentially private neural activations with edge computing," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 90–102.
[138] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Netw.*, vol. 12, no. 1, pp. 145–151, 1999.
[139] P. Goyal *et al.*, "Accurate, large minibatch SGD: Training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, pp. 1–12, Jun. 2017.
[140] C. Hardy, E. L. Merrer, and B. Sericola, "Distributed deep learning on edge-devices: Feasibility via adaptive compression," in *Proc. IEEE 16th Int. Symp. Netw. Comput. Appl. (NCA)*, Oct. 2017, pp. 1–8.
[141] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *Proc. 11th Annu. Int. Conf. Mobile Syst., Appl., Services*, 2013, pp. 69–82.
[142] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan, "Glimpse: A programmable early-discard camera architecture for continuous mobile vision," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, 2017, pp. 292–305.
[143] K. Ha *et al.*, "You can teach elephants to dance: Agile VM handoff for edge computing," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, p. 12.
[144] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, p. 11.
[145] L. Chaufournier, P. Sharma, F. Le, E. Nahum, P. Shenoy, and D. Towsley, "Fast transparent virtual machine migration in distributed edge clouds," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, p. 10.
[146] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, 2016, pp. 3–17.
[147] S. Kaul, R. Yates, and M. Gruteser, "Real-time status: How often should one update?" in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2731–2735.
[148] J. Cho, K. Sundaresan, R. Mahindra, J. van der Merwe, and S. Rangarajan, "ACACIA: Context-aware edge computing for continuous interactive applications over mobile networks," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, 2016, pp. 375–389.
[149] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 3–18.
[150] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proc. Theory Cryptogr. Conf.* Springer, 2006, pp. 265–284.

ABOUT THE AUTHORS

Jiasi Chen received the B.S. degree from Columbia University, New York, NY, USA, with internships at AT&T Labs Research, Florham Park, NJ, USA, and NEC Labs America, Princeton, NJ, USA, and the Ph.D. degree from Princeton University, Princeton, NJ, USA.

She is currently an Assistant Professor with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA, USA. Her current research interests include edge computing, wireless and mobile systems, and multimedia networking, with a recent focus on machine learning at the network edge to aid augmented reality (AR)/virtual reality (VR) applications.

Dr. Chen was a recipient of the Hellman Fellowship and the UCR Regents Faculty Fellowship.



Xukan Ran received the B.S. degree in network engineering from Xidian University, Xi'an, China. He is currently working toward the Ph.D. degree in computer science at the University of California at Riverside, Riverside, CA, USA. He also studied computer science at Zhejiang University, Hangzhou, China.

His current research interests include edge computing, deep learning, and simultaneous localization and mapping on mobile devices.

Mr. Ran received the Best-in-Session Presentation Award at the IEEE INFOCOM in 2018.