Distributing Deep Neural Networks with Containerized Partitions at the Edge

Li Zhou[1], Hao Wen[2], Radu Teodorescu[1], and David H.C. Du[2]
[1]The Ohio State University, [2]University of Minnesota, Twin Cities

Abstract

Deploying machine learning on edge devices is becoming increasingly important, driven by new applications such as smart homes, smart cities, and autonomous vehicles. Unfortunately, it is challenging to deploy deep neural networks (DNNs) on resource-constrained devices. These workloads are computationally intensive and often require cloud-like resources. Prior solutions attempted to address these challenges by either sacrificing accuracy or by relying on cloud resources for assistance.

In this paper, we propose a containerized partition-based runtime adaptive convolutional neural network (CNN) acceleration framework for Internet of Things (IoT) environments. The framework leverages spatial partitioning techniques through convolution layer fusion to dynamically select the optimal partition according to the availability of computational resources and network conditions. By containerizing each partition, we simplify the model update and deployment with Docker and Kubernetes to efficiently handle runtime resource management and scheduling of containers.

1 Introduction

Edge devices are becoming increasingly important components of the machine learning (ML) revolution. Various emerging applications are driving the need for deploying machine learning algorithms to edge devices in smart homes, smart cities, autonomous vehicles, and healthcare [4, 18, 21]. However, in most cases, the bulk of the computation, even for inference problems, is performed entirely within the cloud or through a hybrid combination of edge and cloud computing.

Cloud-based processing of user-generated data faces several challenges and limitations. For instance, uploading user data to the cloud raises privacy concerns. Consumers are becoming increasingly aware of the privacy implications associated with online services. Furthermore, many IoT applications require frequent decision making that render cloud-based computing impractical due to the communication latency it brings.

In response to the aforementioned concerns, efforts have been made to push machine learning inference from the cloud to the edge, with the benefits of keeping data closer to its source to provide real-time responses while protecting the privacy of the end-user. Unfortunately, most machine learning algorithms are computationally demanding, making edge devices inadequate for handling such workloads due to their constrained performance, energy, and memory capacities. To this end, a significant amount of research has investigated efficient approaches for deploying DNNs to the edge. This includes collaborative computation between edge devices and the cloud [9, 16, 26], model compression and parameters pruning [5, 10, 27, 29], or customized mobile implementations [11, 12, 22, 31]. Despite all these efforts, having the ability to scale existing DNNs without sacrificing the model accuracy and processing the collected data streams in real time present ongoing challenges to the deployment of machine learning across edge devices.

In this work, we explore parallel execution of DNN inference across multiple heterogeneous devices that are energy-constrained. A possible application that we envision for our work is local smart home processing. Instead of relaying collected data that includes voice commands, sensor readings, and video streams from a camera to the cloud, our solution leverages other smart home devices, such as speakers, light switches, and hubs to resolve the request. This approach creates a number of research challenges that need to be addressed: (1) how to partition the workload efficiently across devices; (2) how to optimize execution across heterogeneous devices possessing different compute capabilities, and account for the higher communication latency in wirelessly connected devices; (3) how to efficiently deploy and schedule the workloads adapting the change of resources in IoT environment.

To summarize, (1) we design a dynamic programming-based search algorithm to decide the optimal partition and parallelization for a DNN model. (2) we present a collaborative CNN acceleration framework that adapts to the computational resources and network condition in the presence of heterogeneity. (3) we improve the framework flexibility and reliability by utilizing Docker[1] and Kubernetes[2] to dynamically assign containers composing a DNN job onto a single or multiple devices based on the status of the available devices.

2 Background and Related Work

CNN layers. A convolutional neural network consists of an input, an output layer, and multiple hidden layers. The hidden layers typically consist of a stack of convolution, nor-

[1]https://www.docker.com/
[2]https://kubernetes.io/

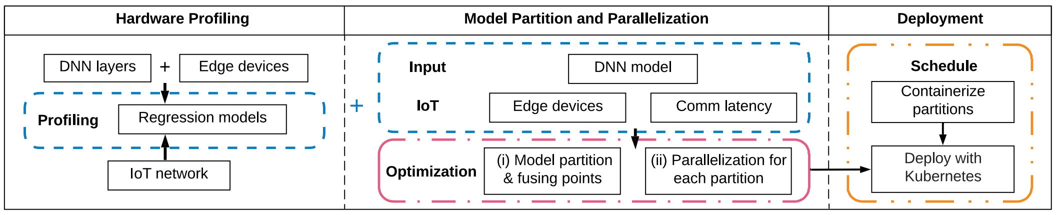

Figure 1: Design overview.

malization, ReLU (i.e., activation functions), pooling, and fully-connected layers. The **convolution** (*conv*) **layer** is the core building block, and takes most of the computation time in a CNN model (e.g., 73.8% of VGG-16 [25] and 99.93% of YOLOv2 [23]). The normalization/ReLU/pooling layers are less compute intensive and often optimized to compute with a previous operator. As such, we group them with the corresponding previous layer that produces their input. The high-level reasoning in the neural network is done via a dense or **fully-connected** (fc) **layer**. The value of each output is calculated from the weighted sum of all inputs. Efficiently parallelizing a CNN model is equivalent to parallelizing *conv* and *fc* layers, which are among the most compute and data intensive layers of a CNN model.

Existing approaches. Current techniques enabling DNN-based intelligent applications fall into two categories: *cloud-based* and *edge-only* approaches. Cloud-based approaches [3, 8, 14, 16, 20, 26, 30] fully (i.e., cloud-only) or partially (i.e., edge-cloud collaboration) offload the computation to the cloud. Neurosurgeon [16] proposes to distribute a DNN model between edge devices and the cloud by deciding a single partition point. IONN [14] designs incremental model uploading with multiple partition points to overlap the local client and server executions.

Edge-only approaches [6, 9, 10, 12, 19, 22, 28, 32] execute the DNNs on a single edge device with specialized hardware or a small IoT cluster. Collaborative computing enables a small IoT cluster to run larger models or speedup inference by employing the available idle devices. MoDNN [19] uses layer-wise parallelization, however the MapReduce-like execution results in a large amount of intermediate data to be transferred among devices. Collaborative perception [9] pipelines the computation by partitioning a DNN model and distributing the partitioned blocks to multiple edge devices. DeepThings [32] reduces the communication cost by fusing the early convolution layers and parallelizing these layers in multiple devices.

Deep learning in containers. The cloud infrastructure like Amazon Web Services[3], Microsoft Azure[4], and Google Cloud Platform[5] is built on the idea of virtual machine (VM), Compared with VMs, containers are more lightweight and can run multiple instances of an application on top of one operating system. Without the need for a unique operating system, startup and shutdown of a container are much faster than a VM [24], while an equivalent or better performance is provided [7]. Not surprisingly, cloud providers are deploying ML services with containers [1, 13], using Docker for creating and running containers and Kubernetes for orchestrating and managing containers.

[3]https://aws.amazon.com/
[4]https://azure.microsoft.com/
[5]https://cloud.google.com/

3 Distributing DNNs Inference at the Edge

Figure 1 provides an overview of our approach. First, a simple model of the available devices and their compute capabilities is generated. Next, our framework uses parameterized performance prediction models for multiple DNN layer types. At runtime, the framework predicts the execution time for each layer based on available devices and communication latency, and selects the best partition points and parallelization strategies for each partition. To accommodate the heterogeneity of the compute environment, partition sizes are chosen to match device capabilities. Then, the partitions are containerized, distributed and executed across multiple devices.

3.1 Parallelizing DNN Layers

We first explore the dimensions that parallelize layers in a DNN model. In our framework, we use *channel partitioning* and *spatial partitioning* to parallelize a 2-dimensional convolution layer. As shown in Figure 2a, in a *conv* layer each filter generates a feature map (i.e, a channel of the output feature maps). The output feature maps can be partitioned along the *channel* dimension such that each device computes a subset of the output feature maps. This requires mapping the corresponding set of filters to each device. The input maps have to be replicated across all the devices. The fully-connected layer can also be parallelized using channel partitioning.

An alternative approach is to partition the output feature maps spatially (i.e., by height and width) and assign them to multiple devices. Each device keeps a copy of the network and computes a subset of the output feature maps. As Figure 2b shows, compared with channel partitioning where the entire input has to be transferred to all devices, in spatial partitioning each device only requires a subset of the input. This greatly reduces communication costs in edge networks that rely on wireless communication.

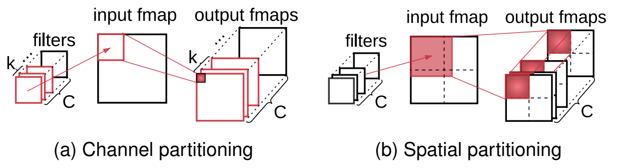

Figure 2: Parallelizing a 2D convolution layer with channel (a) and spatial (b) partitioning.

We now introduce two parallelization strategies used in our framework to parallelize a DNN model.

Layer-wise parallelization. Prior work [17] has shown that the best parallelization strategy depends on the characteristics of the DNN (i.e., layer type, shape of feature maps and size of filters). As a result, layer-wise parallelization [15] has been proposed to allow each layer to be parallelized independently using the appropriate technique to obtain the best performance. In our evaluation, we parallelize each layer using either channel or spatial partitioning. However, in layer-wise parallelization, each device computes part of the output of the current layer and all the subsets of the output need to be merged and re-partitioned before executing the next layer. This requires output to be gathered by a host node, partitioned and re-sent to all client devices. In a wireless network this results in substantial communication overhead. For our system, the benefits of layer-wise parallelization are generally defeated by the communication costs.

Fused-layer parallelization. To reduce the data movement between layers, we propose using fused-layer parallelization. The concept of layer fusion was first proposed in [2] as a method to reduce off-chip data movement in a CNN accelerator. The idea is to send the output of one layer directly to the input of the next layer without going through memory.

We propose extending this concept combined with spatial partitioning by parallelizing multiple fused layers as a single fused-layer block instead of single layers individually. Partitioning is performed layer-by-layer starting from the last layer in the fused block. Each layer's input is the output of the previous layer. The required input elements for each partition are calculated based on its output elements. For *conv* layer, we also need to extend each partition's input by $\lfloor f_i/2 \rfloor$ (f_i is the size of filters of layer i) on height and width for overlapping elements. The process is applied recursively up to the first layer in a fused block.

The partitions of fused-layer blocks are next distributed for computation. All convolution layers in the fused block will now be computed locally and only the output of the last layer will be merged. This reduces communication costs because only the input of the first layer and the output of the last layer need to be communicated between devices. The more layers are fused, the more communication costs are reduced. However, fused-layer partitioning introduces additional costs compared to layer-wise partitioning. The overlap in input partitions adds to the communication cost of distributing those partitions, and adds redundant computation to each node. This cost increases with the number of fused layers. As a result, finding the optimal number of layers to be included in a fused block requires carefully balancing the costs and benefits of layer fusion.

Deploying fused-layer parallelism in our environment in an optimal way requires answering the following questions: (1) which layers should be fused, (2) how many layers to include in each fused block, (3) for each fused and unfused block, how many partitions/devices offer the optimal performance and (4) how to match the partition size to the capability of the device in a heterogeneous environment?

3.2 Partitioning a DNN Model

Answering the aforementioned questions requires solving a multivariate optimization problem. We present a dynamic programming based search algorithm to find the parameter values that are projected to achieve the lowest execution time under our cost model.

Problem definition. Given a CNN model G with n layers, where $l_i \in G$ is a layer in the model and edge (l_i, l_j) is a tensor that is an output of layer l_i and an input of layer l_j. The model runs on a list of devices D, and we assume the communication bandwidth of each connection between two devices (d_i, d_j) is known. Our goal is to find a hybrid parallelization strategy $S = \mathcal{S}^{lw} \bigcup \mathcal{S}^{fl}$ (lw: layer-wise, fl: fused-layer) such that the execution time $\mathcal{T}(G, D, S)$ is minimized.

Cost model. We develop a performance model to guide the optimization search. To construct the model for each layer type, we vary the configuration parameters of the layer and measure the latency for each configuration. Using the profiles, we build a regression model for each layer type to predict execution latency. To predict the communication cost, we use a similar approach by varying the data transfer size and measuring the latency.

The cost functions for each layer $l_i \in \mathcal{G}^{lw}$ (i.e., $t_l(i)$) is defined as the total time of the input/output tensors transfer and layer computation under layer-wise parallelization strategy $\mathcal{S}^{lw}(i)$. Next, assuming several consecutive convolution layers are grouped (i.e., fusing j consecutive layers from layer i) under fused-layer parallelization strategy $\mathcal{S}^{fl}$. The cost function for a fused block $l_{(i,j)} \in \mathcal{G}^{fl}$ (i.e., $t_f(i,j)$) is then defined as the total data transfer time of the first layer's input tensor, the last layer's output tensor, and the sum of the computation time of all grouped layers. Next we define:

$$\mathcal{T}(G, D, S) = \sum_{l_i \in \mathcal{G}^{lw}} t_l(i) + \sum_{l_{(i,j)} \in \mathcal{G}^{fl}} t_f(i,j) \qquad (1)$$

where $\mathcal{T}(G, D, S)$ estimates the total execution time of a single inference for a model $G = \mathcal{G}^{lw} \bigcup \mathcal{G}^{fl}$ on a list of devices D under a hybrid layer-wise/fused-layer parallelization strategy $S = \mathcal{S}^{lw} \bigcup \mathcal{S}^{fl}$.

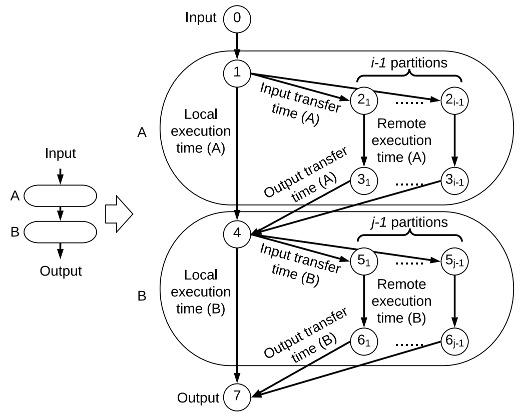

Figure 3: Example of model execution graph.

Dynamic programming-based optimization. Given the relatively small optimization space, we use dynamic programming-based search through the space of possible solutions. We start by determining the optimal parallelization configuration for each layer, and each fused block by finding the optimal $\mathcal{S}^{lw}$ and $\mathcal{S}^{fl}$ with two tables of cost $t_l(.)$ and $t_f(.)$. Then we use the dynamic programming based search algorithm to find the optimal placement of fused blocks in a model. We define the *fused-layer partitioning problem* as follows. Given a CNN model G with n layers and tables of cost $t_l(.)$ and $t_f(.)$, determine the minimum cost $t_o(i,j)$ (equivalent to $\mathcal{T}_o(G,D,S)$ when $i=0, j=n$) obtainable by partitioning the model that can be achieved through layer fusion.

We give the general equation for dynamic programming as below:

$$t_o(i,j) = \begin{cases} 0 & j=0, \\ t_l(i) & j=1, \\ \min\limits_{1\le k\le j} (t_f(i,k)+t_o(i+k,j-k)) & otherwise. \end{cases} \tag{2}$$

3.3 Deploying a DNN Model with Containerized Partitions

Figure 3 shows an example of model execution graph after partitioning. Assuming a CNN model is divided into two blocks, and each block includes i and j partitions respectively. Each block may include one or more fused layers. The local device that captures the input (e.g., a video camera or smart speaker) is responsible for partitioning the input tensor, and distributing the partitioned input among devices. The remote devices start to execute once a task is received and send the results back once the job is completed. All results will be merged at the local device and re-partitioned for the next block.

We containerize each partition and deploy the model by utilizing Kubernetes to launch pods on the edge devices. Every pod runs a same Docker image, but may run different partitions and take different data as input. Note that it is possible that the optimal configuration for some blocks is to run on a single device, in which case they will be deployed using a single pod. Otherwise, multiple pods will be launched and scheduled across edge devices according to the computation capabilities of each device and the resources required by each partition. Partitions belonging to different blocks form a pipeline, while partitions belonging to the same block run in parallel.

The performance of edge devices in IoT network often fluctuates. We may need to recalculate the optimal partitions based on the current status of each device after a period of time. Note that the interval of recalculating the optimal partitions should be carefully selected to avoid DNN performance degradation and high overhead caused by rescheduling pods frequently. In our system, it takes around 2000 ms to find the optimal partition points, create and launch a pod before it starts to run. The system periodically recalculating the optimal partition points, once there is a change in the model execution graph, we will adjust the configurations of pods and reschedule them.

In addition, devices may fail in IoT network. In order to recover from device failures quickly without affecting the pods running in normal, a static virtual IP for each pod is allocated by Kubernetes. Each pod talks to its upstream and downstream pod via virtual IPs. The association of a virtual IP with a pod is based on where the pod locates in the model execution graph. If a device fails, we can easily launch a new pod on another device and associate the new pod with the virtual IP. In this way, pods running normally are not affected.

4 Evaluation

Experimental setup. We create multiple virtual machines (VMs) with limited capabilities to emulate IoT devices. All VMs are running in a cluster, where each physical server has two six-core Intel Xeon 2.40 GHz E5-2620 v3 CPUs, 64 GB of memory, and is connected to an HP ProCurve 5406zl switch through a 1Gb/s Broadcom NetXtreme BCM5720 NIC port. The inbound and outbound network bandwidth of each VM is configured through Linux Traffic Control (TC) infrastructure. All physical servers and VMs run Ubuntu 18.04. We deploy a Kubernetes (Release-1.7) cluster among these VMs. In the following experiments, we set up 5 VMs to run as 5 IoT devices.

4.1 Latency Improvement

We explore three different parallelism schemes in our preliminary experiments and show how they react to different computation and network conditions in Figure 4 by running VGG-16 inference using TensorFlow.

In *All-In-One* case, the whole VGG-16 model runs in a single device. Based on Sec. 3.2, we partition the VGG-16 model into five blocks. The first three blocks are fused layer blocks containing 7, 3 and 3 convolution layers respectively,

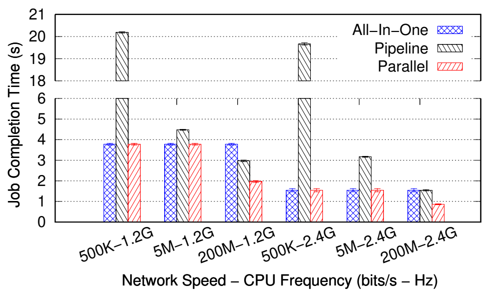

Figure 4: Latency under different computation and network conditions with 95% confidence interval.

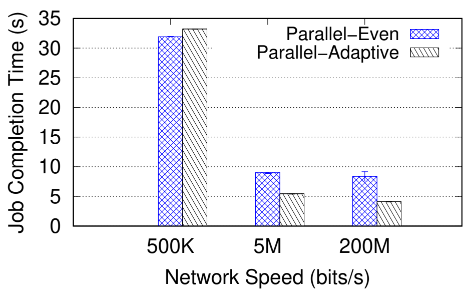

Figure 5: Latency with 95% confidence interval under different network conditions with heterogeneous devices.

while the last two blocks include 1 and 2 fully-connected layers respectively. In the *Pipeline* case, each device holds one block of the VGG-16 model, and each block is running inside a Kubernetes pod. Computations on these five devices form a pipeline.

In the *Parallel* case, we use four devices to compute and one device to act as a hub. For each block, we divide the input of the first layer into four partitions. Each partition runs in a Kubernetes pod. Unlike in the *Pipeline* case, each block in the *Parallel* case has four pods running at the same time, working on different partitions of the input. This way four devices run in parallel to contribute to the computation of a single block to speedup execution. After finishing the computation, each pod will send its result to the hub. After receiving the results from all four pods belonging to the same block, the hub will merge and re-partition the results and send new input partitions to the same four pods again to compute the next block.

For each case, we test the latency under three different CPU and network conditions. For network, we configure *slow*, *median* and *fast*, with bandwidth among devices limited to 500 Kbps, 5 Mbps and 200 Mbps respectively. Under each network configuration, we allocate either one CPU core or 500 millicores to each pod to emulate a CPU at 2.4 GHz or 1.2 GHz. We ensure the memory is enough for each pod so that there will be no disk swap involved.

We test each case 100 times with randomly generated input tensors to rule out the possible influence of the cache, and present the results with 95% confidence intervals in Figure 4. When the network is *slow*, we observe that *Parallel* can be 10x or 20x slower than *All-In-One*. However, parallelizing a DNN model starts to improve performance as network speed increases. When network is *fast*, *Parallel* achieves 47.94% (1.2 GHz) and 44.15% (2.4 GHz) reduction in execution time compared with *All-In-One*. This is because *Parallel* has the highest parallelism among these three cases. More speedup is expected with even faster networks, e.g. with the arrival of the 5G standard. For a lower computation capability, the improvement is more obvious. Since our framework is able to evaluate the cost of different partition and parallelization schemes, it opts to run VGG-16 in a single device, and thus has the same performance as *All-In-One* when network is *slow*. As *Pipeline* does not increase parallelism, it does not bring obvious latency reduction when the network is fast. What's worse, it suffers the same problem as running in parallel when the network is slow.

4.2 Heterogeneous Devices

To adapt to the heterogeneity of an IoT environment, the framework adjusts the task size for each device based on the current availability of computational resources. We define a device as busy when its resources are almost occupied, for example, running an another job at the same time. We test the latency of running a VGG-16 reference in the presence of heterogeneity. In the experiment, we reduce the available CPU to 125 millicores to represent a busy device, while all other normal devices allocate 500 millicores. To accommodate this change, the input tensor size of assigned partition will be adjusted accordingly. In Figure 5, we show the effectiveness of adapting to the heterogeneous devices. In *Parallel-Adaptive*, the size of input we allocate to the busy device is 1/4 of that allocated to a normal device. In *Parallel-Even*, we distribute the input tensor evenly to four devices. In *fast* and *median* network, we achieve a speedup of 50.69% and 39.45% respectively. That's because the busy device has a smaller size of task to compute and finish almost at the same time with normal ones for each block. However, when network is *slow*, it increases the amount of data to be transmitted on network for a normal device. The large latency in network transfer cancels out the speedup in computation. Nevertheless, our framework will opt to run the DNN job in a single normal device in this case.

5 Conclusion

In this paper, we propose containerized partition-based CNN inference acceleration framework at the edge. The framework dynamically partitions a DNN model that adapts to the changes of computational resources and network condition. We containerize and deploy each partition on a small cluster of IoT devices using Kubernetes for better resource management and scheduling.

6 Discussion Topics

We envision the deployment of our system in an environment such as a smart homes, in which devices of different types and compute capabilities collaborate to solve a joint task. We would like to open up discussions around this work and gather feedback for the following aspects:

Granularity for DNN containerizations. Containers are widely used in the cloud for big data analytics and ML applications. The whole model is often containerized as a service. In this work, we explore the finer granularity of containerizing a DNN model at the edge, and discuss the opportunity of utilizing multiple containers to run a single DNN model across multiple edge devices.

Impact of 5G for ML at the edge. Our work is based on the observations of limited computing ability of edge devices and the upcoming 5G network. In the 5G era, much faster communication speed will be available, which can potentially change today's computation and communication ratio at the edge. It would be interesting to discuss what changes it will bring in the cases of smart homes, cities, self-driving cars and so on, where a small local cluster is available while internet connection is not present. This paper shows the benefits of distributing a DNN job onto multiple edge devices when the network is fast. The benefits will be further increased by the 5G network. However, if network speed is slow, it may be more suitable to run DNN on a single device, which has been discussed in Sec. 4. We plan to fully explore the trade-offs between computation and communication when running DNN jobs, and study more complicated cases such as the presence of network heterogeneity.

Fault-tolerance support. IoT devices may go offline from the network at any time for any reason. A discussion on fault-tolerance design at the edge is important. In future work, we plan to focus on improving fault-tolerance of containerized model partitions. We will look into the implementation of efficient container migration when a device runs into unexpected situations, such as receiving higher priority tasks. The containerized partition in execution should be quickly transferred to another available device and resume the job. We also plan to validate the framework on IoT hardware like Raspberry Pi.

Sensitivity to status change. In this paper, we state that we will adjust the configurations of pods and reschedule them, when there is a change in the model execution graph caused by status changes of edge devices. How sensitive the system should adapt to status change is an important issue. A high sensitivity will lead to frequent pods rescheduling which may block DNN job execution. A low performance will result in performance degradation due to overload on busy devices. We leave setting an appropriate sensitivity as a future study.

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback. We also extend special thanks to Irfan Ahmad for suggestions on the camera ready. This work was supported in part by NSF Award 1439622, 1812537 and NSF XPS Award 60053525.

References

[1] Containers overview. https://cloud.google.com/ml-engine/docs/containers-overview.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 22. IEEE Press, 2016.

[3] Amazon. Machine learning on AWS. https://aws.amazon.com/machine-learning/.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[6] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th ACM International Conference on Mobile Computing and Networking*, pages 115–127, 2018.

[7] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[8] Google. Cloud machine learning engine. https://cloud.google.com/ml-engine/.

[9] Ramyad Hadidi, Jiashen Cao, Matthew Woodward, Michael S Ryoo, and Hyesoon Kim. Distributed perception by collaborative robots. *IEEE Robotics and Automation Letters*, 3(4):3709–3716, 2018.

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

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

[12] Intel. Movidius Neural Compute Stick. https://software.intel.com/en-us/movidius-ncs.

[13] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262. IEEE, 2018.

[14] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. IONN: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 401–411, 2018.

[15] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018.

[16] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[17] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[18] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[19] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and

Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1396–1401. IEEE, 2017.

[20] Microsoft. Azure machine learning service. https://azure.microsoft.com/en-us/services/machine-learning-service/.

[21] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018.

[22] Nvidia. Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[23] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint*, 2017.

[24] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111):2, 2014.

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

[26] Surat Teerapittayanon, Bradley McDanel, and HT Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339, 2017.

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

[28] Zirui Xu, Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. Direct: Resource-aware dynamic model reconfiguration for convolutional neural network in mobile systems. In *Proceedings of the ACM International Symposium on Low Power Electronics and Design*, page 37, 2018.

[29] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[30] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[31] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[32] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.