Adaptive Distributed Convolutional Neural Network Inference at the Network Edge with ADCNN

Sai Qian Zhang
Harvard University
zhangs@g.harvard.edu

Jieyu Lin*
University of Toronto
jieyu.lin@mail.utoronto.ca

Qi Zhang*
Microsoft
q8zhang@gmail.com

ABSTRACT

The emergence of the Internet of Things (IoT) has led to a remarkable increase in the volume of data generated at the network edge. In order to support real-time smart IoT applications, massive amounts of data generated from edge devices need to be processed using methods such as deep neural networks (DNNs) with low latency. To improve application performance and minimize resource cost, enterprises have begun to adopt Edge computing, a computation paradigm that advocates processing input data locally at the network edge. However, as edge nodes are often resource-constrained, running data-intensive DNN inference tasks on each individual edge node often incurs high latency, which seriously limits the practicality and effectiveness of this model.

In this paper, we study the problem of distributed execution of inference tasks on edge clusters for Convolutional Neural Networks (CNNs), one of the most prominent models of DNN. Unlike previous work, we present *Fully Decomposable Spatial Partition* (FDSP), which naturally supports resource heterogeneity and dynamicity in edge computing environments. We then present a compression technique that further reduces network communication overhead. Our system, called *ADCNN*, provides up to 2.8× speed up compared to state-of-the-art approaches, while achieving a competitive inference accuracy.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms.*

KEYWORDS

Edge computing, Convolutional neural networks, Distributed inference

ACM Reference Format:
Sai Qian Zhang, Jieyu Lin*, and Qi Zhang*. 2020. Adaptive Distributed Convolutional Neural Network Inference at the Network Edge with ADCNN. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404473

*Equal contribution, names are ranked alphabetically.

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404473

1 INTRODUCTION

The proliferation of the Internet of Things (IoT) has promised a future where billions of edge devices will be collecting, processing and sharing information over the Internet. As enterprise organizations begin to embrace IoT, tremendous volumes of data generated from edge devices need to be processed quickly, reliably and cost-effectively. This technological trend also coincides with the rapid advances in machine learning. In particular, deep learning models such as Convolutional Neural Networks (CNNs) have achieved great success in areas such as object detection [26], image classification [30], and speech recognition [37], all of which are important applications of IoT. As a result, many device manufactures are beginning to offer machine learning capabilities in their products. These smart, machine learning-capable edge devices can perform complex sensing and recognition tasks to support a wide variety of IoT scenarios, and are expected to dominate the IoT market in the near future [14].

Despite recent developments, however, executing CNN inference tasks on IoT devices still poses many challenges. Given the limited computing resources on IoT devices, processing delay-sensitive tasks (e.g., object detection, facial recognition) entirely on a single device may consume significant amounts of resources and incur high processing delays, resulting in poor user experience. A natural alternative approach is to send these tasks to cloud data centers. However, doing so introduces additional latency and bandwidth costs, and may not be practical in situations where latency and bandwidth constraints are stringent (e.g., remote drone mission control). To address this challenge, enterprises have begun to adopt *Edge computing*, which leverages a cluster of *edge nodes* (compute nodes in edge networks) to perform substantial amounts of computation, storage and communication. As edge nodes are often located close to the data source, processing CNN inference tasks on edge nodes can reduce bandwidth cost and reduce processing latency. However, the downside of approach is that edge nodes are often resource-constrained, heterogeneous and unreliable [18], making them inadequate for handling heavy workloads such as CNN inference in a reliable and timely manner [14].

To address this limitation, in this paper, we present a novel approach for running CNN inference tasks in Edge computing environments. Specifically, we study the distributed CNN inferencing problem in dynamic edge computing environments. We present *ADCNN*, an efficient partitioning framework for fast CNN inference. ADCNN partitions the convolutional layers into many small independent computational tasks. To compensate for the accuracy degradation rooted from the partitioning, ADCNN retrains the original CNN model at a low cost (Figure 1(a)). The partitioned tasks are then assigned to edge nodes dynamically based on their current operational condition (Figure 1(b)) for parallel processing.

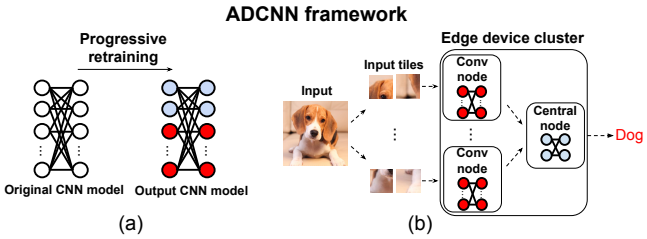


Figure 1: (a) CNN model is first retrained to compensate the accuracy degradation caused by partition. (b) During operation, the separable layer blocks (shown in red) and stored in the Conv nodes, and the rest layers (shown in blue) are saved in the Central node. Input image are partitioned into tiles with FDSP and sent to node cluster for fast inference.

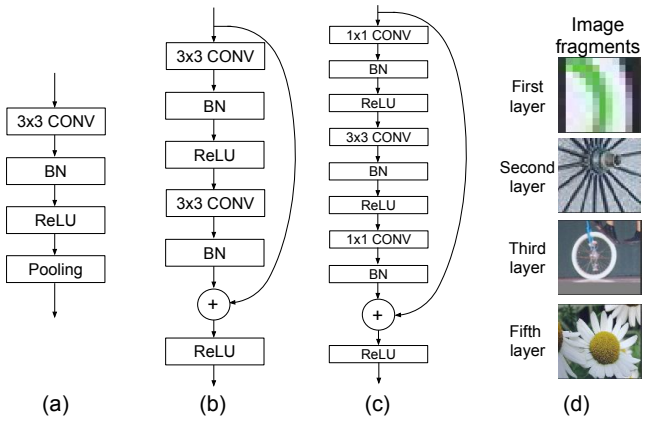


Figure 2: Common layer block architectures. (a) shows a basic layer block. (b) and (c) show layer blocks in ResNet. (d) Visualization of detected features at 1st, 2nd, 3rd and 5th layers of AlexNet [21]. For these four layers, the detection focuses on edges, textures, shapes, and objects, respectively.

This framework offers numerous benefits, including (1) providing resilience on system performance that keeps the impact of fluctuation on edge node performance at minimum, (2) fine grained load balancing across edge nodes, depending on their runtime processing capability, and (3) enabling the overlapping of computation and network transfer to improve resource utilization. Secondly, we present a compression scheme that further minimizes the communication cost across layers. Third, we show that these two techniques can be easily incorporated into any existing CNN model with low retraining cost and achieve nearly no accuracy degradation.

Unlike the previous approaches [19, 23, 24, 38, 39], which rely on static partitioning and task assignments and not adapt to dynamic environments where the system variability is present, ADCNN has the following advantages. First, ADCNN can quickly adapt to the performance variation and heterogeneity in the edge environment, which is a well-known feature in edge computing [25, 29]. Second, ADCNN offers a novel full-stack optimization framework to joint optimize and CNN architecture and computing system to assure that overall performance is optimal in terms of both processing latency (system side) and prediction accuracy (machine learning side).

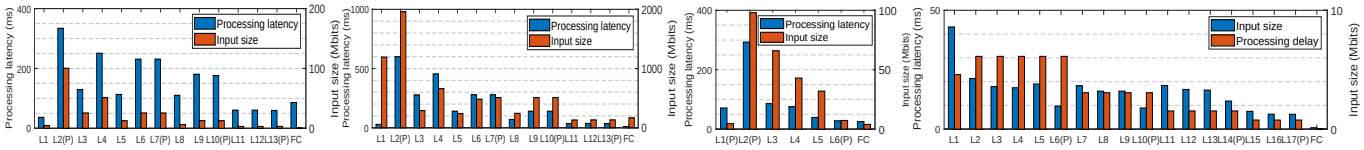
We have implemented ADCNN and evaluated its performance using various CNN models for image classification (e.g., VGG16 [30], ResNet34 [17]), object detection (YOLO [26]), semantic segmentation (FCN [22]) and text classification (CharCNN [37]). We also deployed ADCNN on a real testbed. Both simulation results and testbed evaluations show that ADCNN can greatly accelerate CNN inference compared to other approaches with nearly no accuracy degradation. Finally we compare ADCNN with Neurosurgeon [19] and AOFL [39], two state-of-the-art CNN partitioning frameworks, and find that ADCNN outperforms them by 2.8× and 1.6× respectively in terms of the processing latency. Additionally, ADCNN can naturally adapt to the variation on the system conditions while maintaining a decent performance.

The rest of the paper is organized as follows: Section 2 summarizes the background and presents our motivation. We then discuss different CNN partitioning options in Section 3. We present the training strategy in Section 4 and progressive retraining method in Section 5. The implementation of ADCNN is described in Section 6. Our evaluation results are presented in Section 7. We survey related work in Section 8, and conclude the paper in Section 9.

2 BACKGROUND AND MOTIVATION

2.1 CNN Architecture

A modern CNN usually consists of five types of layers: convolutional layer, activation layer, batch normalization layer, pooling layer and fully connected layer. The *convolutional layer* (CONV) is the most computational intensive layer that performs high-dimensional convolution operations. The input activations are structured as a group of input feature maps (ifmaps), each of which is called *channel*. The parameters of this layer are a set of weight filters. Each filter has the same number of channels as the ifmaps. During inference, the weight filters slide across the ifmaps. At each position, the dot product between the entries of each ifmap and weight filter are calculated. These dot products are aggregated into a 2D output feature map (ofmap), and therefore the total amount of multiplication and addition operations is proportional to the spatial size of ifmap. The CONV layer usually contains multiple filters. Each filter is responsible for extracting a single type of feature from the input. For most of the CNNs, the filter size is usually small (e.g., 3×3). This is because convolution with large filters is equivalent to using more CONV layers with small filters. The *batch normalization layer* performs batch normalization (BN) [28], a technique that accelerates speed, stability and performance of CNN training by shaping the distribution of the input activations. During the forward propagation of the training phase, the ifmap x is first normalized using the batch mean μ and standard deviation σ (i.e., $x' = \frac{x-\mu}{\sigma}$). After that, the normalized maps are further re-scaled using a scale parameter γ and a bias parameter β (i.e., $y = \gamma x' + \beta$), where both γ and β are learnable. During inference, all the parameters μ, σ, γ and β are fixed. This is equivalent to multiplying the ifmaps by a scalar $a = \frac{\gamma}{\sigma}$ and then added with an offset $b = \beta - \frac{\mu\gamma}{\sigma}$.



(a) VGG16 (b) FCN (c) CharCNN (d) Resnet18

Figure 3: Execution time and ifmap size of each layer block for different types of CNNs on Raspberry Pi device. In the diagrams, "Lx" denotes xth layer block, Lx(P) denotes the layer block with pooling at the end, and FC denotes the fully connected layer. For dataset, we use ImageNet [5] for VGG16 and ResNet18, VOC [8] for FCN and CamVid [3] for CharCNN.

The BN layer is usually followed by the *activation layer*, where a nonlinear activation function (e.g. ReLU, tanh) is applied elementwisely to the output of the BN layer. The activation layer does not change the dimension of the ofmaps.

At the *pooling layer*, the ifmaps are partitioned into a set of non-overlapping rectangle regions called receptive fields. Depending on the design, either the maximum or the average of each receptive field is then computed. The pooling layer is used to shrink spatial dimensions of the CNN, making it robust to small positional changes in the input.

At the *fully connected layer* (FC), each output neuron is connected to all the neurons in the previous layer. The FC layers are usually at the end of the CNN, as they are responsible for producing the final results.

Most of the CNNs (e.g. VGG [30], AlexNet [21]) consist of a stack of repeated *layer blocks*. A layer block is a concatenation of a CONV layer, a BN layer, an activation layer and an optional pooling layer (shown in Figure 2(a)). One or more FC layers are attached to the end of the layer block stack. Each layer produces a successively higher level abstraction from the previous layers, and transfers the output to the deeper layer for further processing. As the layer goes deeper, the spatial size of the feature maps decreases and the number of channels increases. More recently, ResNet [17] introduces shortcut connects (shown in Figure 2(b,c)) which is also adopted by YOLO [26]. It is an identity connection where the ifmaps is added elementwisely to the outputs of BN layer.

2.2 CNN Workload Characteristics

Designing fast and efficient CNN inference frameworks requires a careful understanding of CNN layer performance characteristics. We have conducted CNN inference experiments and measured the average execution time and ifmap size for each layer. We use a Raspberry Pi 3 Model B+ [11] as a resource-constrained edge device. It contains a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM, 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN and no GPU. We further use Pytorch [9], a popular deep learning library for deep neural network training and inference.

Specifically, we evaluate four types of CNNs from three different applications, including (1) VGG16 [30] and ResNet18 [17] for image classification. (2) Fully Convolutional Network (FCN) [22] for semantic segmentation, and (3) Character-level Convolutional Networks (CharCNN) [37] for text classification. Figure 3 shows the latency and ifmap size distributions for each layer block. For all four models, both execution time and ifmap size grow tremendously after the first layer block, and subsequently decrease after the second layer block. Furthermore, the earlier layer blocks take much longer to process than the later layer blocks. For example, the first four layer blocks of VGG16 and FCN account for 41.4% and 57% of the total processing latency, respectively. In contrast, the later layers contribute to a small amount of computations relatively. For example, in VGG16, FC layer only accounts for less than 2% of the total computations.

The execution time and ifmap size observations are consistent across the vast majority of the CNN models, and can be interpreted as follows. Given an input image, the CNN first applies numerous weight filters on the input image to extract different types of information, which increases data size tremendously. Over the next few layer blocks, even though the number of channels continues to increase, the data size at each layer block actually decreases due to the subsampling operations performed by the pooling layers. These observations suggest that early CNN layers usually contribute to most of the computation workload of the CNN, and thereby should be the primary target for performance optimization.

2.3 Interpretation of Convolutional Features

In CNN, each layer progressively extracts higher level features of the input image, until the last layer which aggregates all the high-level abstraction and makes a final decision. It has been shown in the previous literature [35, 36] that early CNN layers tend to focus on detecting the local features (e.g., edge or corner in the image), whereas later layers usually look for the high-level abstractions (e.g., shapes of the object in the image) [35]. To validate this, we use the testing approach suggested by [35]. Specifically, we apply the deconvolutional operations on the ofmaps of AlexNet [20], and search for the image fragments in the training input images which yield the largest response for a given weight filter (*i.e.*,magnitude of the corresponding ofmap is the largest). A larger filter response indicates that the weight filter is sensitive to the features contained in this image fragment, and vice versa. Figure 2(d) illustrates the extracted features at different layers of AlexNet [21]. We observe that the input fragments which produce the largest filter response usually involve local features such as edge, corner and textures for layer 1 and layer 2 of AlexNet, while the filters in layer 4 and 5 are sensitive to the high-level features such as shapes or object.

Since the early layers only extract local features, the execution of these layers can be made parallelizable, namely, we can partition the image into tiles and process them independently to extract local features for each tile. This is our primary motivation for leveraging model partitioning to speed up inference processing. We will elaborate the partitioning strategies in the next section.

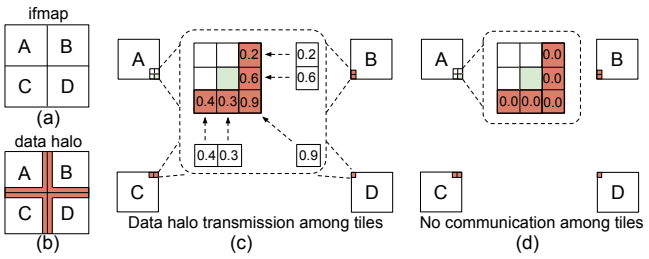


Figure 4: (a) A ifmap is partitioned spatially into 2×2 pieces. (b) Data halo (highlighted in red) of the ifmap shown in (a). (c) Transmission of the neurons in data halo for accurate convolution. (d) Zero padding for convolution at edge pixel.

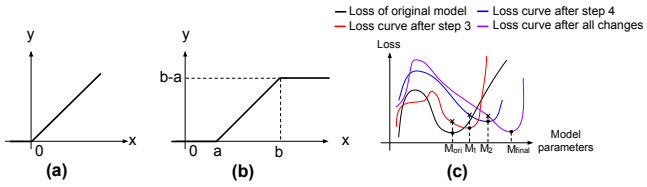


Figure 5: (a) Plot for regular ReLU function. (b) Plot for clipped ReLU with lower bound a and upper bound b. (c) Training loss curves for the CNN models with progressive modifications, the plot is only for illustration.

3 CNN PARTITIONING STRATEGIES

In this section, we examine different model partitioning strategies in order to minimize total inference latency.

3.1 Traditional Partitioning Strategies

The simplest partitioning scheme is *batch partitioning*, which assigns input images in batches to each edge node. Although this scheme could benefit throughput, it does not mitigate resource bottlenecks on edge nodes and hence does not minimize latency. In *channel partitioning*, feature maps are split along the channels and distributed across different nodes. Nonetheless, for the processing of subsequent layers, each node needs to exchange their partially accumulated ofmaps to produce final ofmaps, which may lead to a significant communication overhead. Suppose we apply channel partitioning to VGG16 with two edge devices. The ofmap size of the first layer block is $224 \times 224 \times 64$. To process the subsequent layers, suppose each pixel is a 32-bits floating-point number, the communication overhead between a pair of devices can be estimated as $224 \times 224 \times 64/2 \times 32 = 51.38$Mbits, which is $11\times$ larger than the input image ($224 \times 224 \times 3$). Worse still, this estimated transmission overhead is only for a single CONV layer, and the problems on communication overhead will only be aggravated as the model becomes deeper. Therefore, we conclude that channel partitioning is not a good option.

In *spatial partitioning*, each feature map is split spatially into smaller tiles (shown in Figure 4(a)), and each tile is processed by one edge node. The naïve spatial partitioning scheme still involves cross-device communication during each convolution operation, because the sliding-window nature of the convolution operation generates cross-tile dependencies for the neurons at edges of each tile, which are called *data halos* [12]. Figure 4(a) shows an example of 2×2 partition on a ifmap and Figure 4(b) shows the corresponding halo. To reproduce the same result as the original convolution operation, the neurons in the halo must be exchanged between adjacent devices as shown in Figure 4(c). Suppose the weight filters have a size of 3×3, to compute the convolution at the corner of map A (shown in green in Figure 4(c)), map B,C,D need to send the rest of the neurons within the 3×3 region to map A. In contrast to channel partitioning scheme, where the whole feature map needs to be transferred between devices, spatial partition incurs much lower communication overhead because only the neurons in the halos are transmitted. However, the tiles can not be processed independently due to the cross-tile dependency, which seriously limits the performance of the this partitioning scheme under dynamic conditions.

3.2 Fully Decomposable Spatial Partition

To further reduce communication and overhead and eliminate cross-tile dependencies, we leverage the fact that early CNN layers mainly focus on local feature extraction, as mentioned in section 2.3. This means that no cross-tile information transfers are actually needed for the early CNN layers. Specifically, we propose *Fully Decomposable Spatial Partition (FDSP)*, a new spatial partition strategy which allows the parallel processing of the tiles by padding the cross-tile edge pixels with zeros (Figure 4(d)). FDSP essentially eliminates cross-tile information exchange, enabling each tile to be processed independently. Additionally, the tiles can be made very small to achieve fine grained load balancing, as we shall discuss in Section 6. On the downside, FDSP may incur accuracy loss due to the zero-padding introduced. However, this drawback can be effectively mitigated using the retraining technique described in Section 5. Besides the convolution layers, the activation layer and BN layer involve only element-wise operations, therefore they can be executed independently on each device without any communication. The pooling layer may introduce cross-tile dependency, but the receptive field of the pooling can be made to remain totally within one tile, and therefore no transmission is required between nodes.

Even though FDSP works well for early CNN layers, it is not suitable for later layers of CNN. This is because later layers require global knowledge exchange between the tiles, applying FDSP on the later layers will block the global knowledge exchange between the tiles and harms the prediction accuracy. Fortunately, later layers involve substantially less computation workload and contribute less to the overall processing latency, as indicated in Figure 3. Hence, it is reasonable to run these layers centrally in a single node instead of distributing them across nodes. For illustration simplicity, we say a CNN layer block is *separable* if FDSP can be applied to its convolutional layers while maintaining a good accuracy.

In summary, with FDSP, we can remove cross-tile dependencies hence eliminate the need for inter-tile communication. As FDSP is most suitable for the earlier CNN layers, the best strategy to execute a CNN model is to distribute the tiles across edge nodes for independent and parallel computation of the separable layer blocks. We call these edge node *Conv nodes*. Once the execution is complete, a *Central node* is responsible for collecting the outputs of Conv

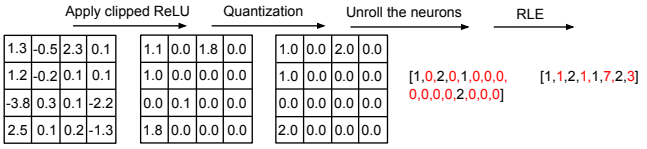


Figure 6: ReLU$_{(0.2,2)}$ is applied to a 4×4 ofmap (Section 4.1), then the neurons are quantized (Section 4.2) and encoded with run-length encoding (Section 4.3).

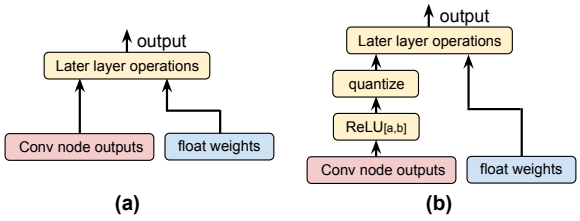


Figure 7: The original and modified versions of the training graphs for connecting the layer blocks and FC layer.

nodes and processing the later CNN layers to generate the final output. This model is depicted in Figure 1(b). Since most of the CNN computations are concentrated in the first few layers, our execution model can greatly reduce the inference latency. Moreover, as each Conv node only operates on parts of ifmaps, its storage complexity and power consumption will also be reduced accordingly.

4 MODIFYING CNN STRUCTURE FOR COMMUNICATION REDUCTION

As mentioned in the previous section, we can improve CNN inference latency by processing the early layers in parallel with Conv nodes, and executing the later layers on the Central node. Unfortunately, the Conv nodes need to transmit the intermediate results to the Central node, which may still cause a significant communication overhead. For instance, in the FCN model, assume the input size is $3 \times 224 \times 224$, and the first seven layers can be partitioned into 4×4 pieces without accuracy degradation. The $28 \times 28 \times 512$ ofmap of layer 7 needs to be sent from the Conv nodes to the Central node for later layer processing. Assume each data in the ofmap is represented with a 32-bits floating number, this will result in a transmission overhead of $28 \times 28 \times 512 \times 32 = 25.7 Mbits$, which is 2.7× larger than the input image ($3 \times 224 \times 224 \times 32$). A similar communication overhead also holds for other CNNs like YOLO, VGG16 and ResNet. Therefore next, we present our modifications on the CNN model for reducing this communication overhead.

4.1 Adjustable ReLU Threshold

Almost all the CNN models nowadays use the rectified linear unit (ReLU) as the activation function. The biggest advantage of ReLU is the non-saturation property of its gradient, which greatly accelerates the convergence of stochastic gradient descent compared to the other activation functions (e.g., sigmoid and tanh). Specifically, $ReLU(x) = x$ if $x \geq 0$ and $ReLU(x) = 0$ otherwise for all $x \in \mathbb{R}$.

Algorithm 1: Progressive retraining algorithm

1. **Input**: CNN model M_{ori} which is trained under the original configuration.
2. **Output**: New CNN model M_{final} with all the modifications applied.
3. Use M_{ori} as the initialization, for separable layer blocks, apply FDSP to the convolutional layers. Retrain the CNN for several epochs until the prediction accuracy is recovered, denote the resulting model as M_1.
4. Insert the clipped ReLU on separable layer block outputs as shown in Figure 7 (b), retrain the CNN until the prediction accuracy is recovered, denote the resulting model as M_2.
5. Apply quantization on the output of clipped ReLU. Retrain the CNN for several epochs until the prediction accuracy is restored, denote the resulting CNN model as M_{final}.
6. **Return** M_{final}

To reduce the communication cost, we apply a clipped version of ReLU with adjustable lower and upper bound for the output of the Conv nodes. The clipped ReLU function $ReLU_{[a,b]}(.)$ with lower bound a and upper bound b is defined as:

$$ReLU_{[a,b]}(x) = \begin{cases} b - a & \text{if } x > b \\ x - a & \text{if } a \leq x \leq b \\ 0 & \text{if } x < a \end{cases}$$

Figure 5(a) and (b) show the plots for both regular ReLU and clipped ReLU. With the adjustable upper and lower bound, we can control the degree of sparsity on the Conv node outputs. Figure 6 shows an example of applying the clipped ReLU functions on an ofmap tile. By utilizing a clipped ReLU with both the upper and lower bound, the ofmap ends up with a high sparsity. In practice, both the upper bound and lower bound are treated as hyperparameters during the training phase.

4.2 Quantize the Conv Nodes Output

By default, each non-zero value in Conv node outputs is represented by a 32-bit floating point number. To further reduce the communication cost, we employ a low-precision quantization on the non-zero outputs of the clipped ReLU, which quantizes underlying full-precision numbers to their nearest quantized values. An example is given in Figure 6. We quantize all non-zero Conv node output values to 4 bits, which further reduces the communication overhead by 8×.

4.3 Run Length Encoding

Run-length encoding (RLE) [27] is a lossless data compression scheme in which consecutive zeros are stored as a single counter in the output. We encode sparse Conv node outputs with RLE to remove lengthy zero-runs, as shown in Figure 6.

4.4 Modification on Training Graphs

Since the architectures of the CNN model have been modified due to FDSP and the compression scheme presented in section 4.1-4.3, the weights of the original CNN may no longer produce the same

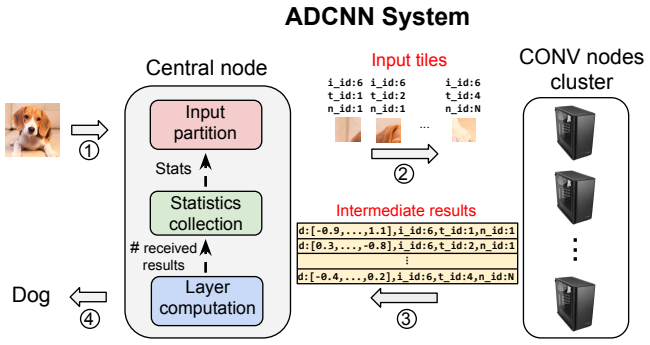


Figure 8: ADCNN system architecture, the step numbers are shown in circles.

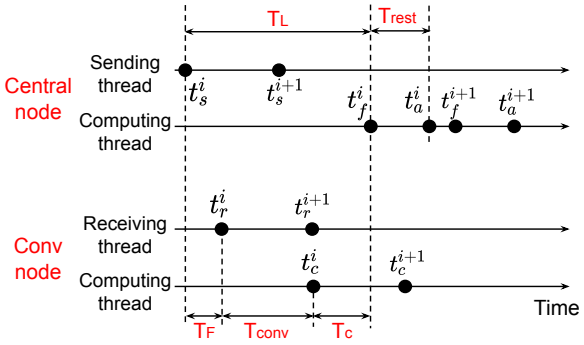


Figure 9: Timeline of ADCNN system operations. T_{Conv} and T_{rest} is the time taken for separable layer block computation and rest layer computation. T_F and T_C are transmission time for the input tiles and the intermediate results. T_L is the time limit for Central node to wait for Conv nodes outputs.

accuracy in the modified model. Therefore we need to retrain the CNN to prevent accuracy loss. We use the filter weights in the original CNN as the initial value, and retrain the CNN with a few more iteration under the modified CNN architecture. Specifically, we apply FDSP to the CONV layers in the separable layer blocks and the compression scheme presented in Section 4.1-4.3 to the separable layer block outputs. Figure 7 (a) and (b) show the original and modified versions of the training graphs for the connection between the separable layer blocks and later layers. The clipped ReLU and quantization are inserted in the training graph. During the forward propagation, the outputs from the separable layer blocks are first pruned with a clipped ReLU, and then quantized before being processed by the later layers. For back propagation, full-precision gradients are used to update the weights.

5 PROGRESSIVE RETRAINING OF CNN

Although the retraining can improve the accuracy of the modified CNN, unfortunately, in practice we find that even after retraining, the accuracy of the modified CNN is still $4-5\%$ lower than the original CNN. The root cause is the discrepancy between the forward and backward propagation paths due to the compression scheme, which causes a highly unstable convergence behaviour for stochastic gradient descent. To mitigate this issue, we propose a progressive retraining strategy which iteratively introduces small modifications to the training graph and retrains the new graph to recover the accuracy. The algorithm is shown in Algorithm 1. By gradually modifying the training graph, the disparity between the forward and backward propagation is minimized. Furthermore, the CNN trained under the previous training graph is likely to be a good starting point for the next training graph, which leads to a faster convergence rate and a better local optimum. This effect is illustrated in Figure 5(c). The local minimum of the original model, M_{ori}, serves as the starting point of our progressive retraining algorithm. The red curve shows the loss function for the new model after applying the modification in step 3 of Algorithm 1. The two curves are likely to be similar to each other because of the small incremental modifications applied. Therefore, the new local minimum M_1 tends to be close to original local optimum M_{ori}. By retraining with the starting point M_{ori}, the training process will converge to M_1 quickly with almost no accuracy loss. The resulting model M_1 will be used again as the starting point for training the next model, until all the modifications on the training graph have been adopted.

6 ADCNN SYSTEM DESIGN

We have implemented our modified CNN model in a system called ADCNN. ADCNN takes advantage of the fine-grained, fully independent tiles generated by FDSP and adapt it to dynamic conditions, allowing it to achieve fine-grained load balancing across heterogeneous edge nodes, as well as tolerating run-time node failure. By dividing image into fine-grained tiles, ADCNN also enables the overlapping of communication and network transfer on Conv nodes, which further speeds up the inference process.

As shown in Figure 8, the ADCNN system consists of one Central node and multiple Conv nodes, where the Central node contains three major building blocks: *Input partition block*, *Statistic collection block* and *Layer computation block*. Meanwhile, the Central node also serves as the entrance to the system, which processes input requests according to their arrival order.

6.1 Inference Workflow

The ADCNN inference workflow is depicted in Figure 8. Before the operation starts, the original CNN model is first retrained by using the progressive retraining technique described in Section 5. The filter weights for the separable layer blocks and remaining layers are stored in the Conv nodes and Central node, respectively. The raw input image is then streamed to the Central node (step 1). The Input partition block takes the input and partitions input into multiple tiles with FDSP. The tiles are then allocated by the Input partition block to each Conv node based on node performance statistics, so that the faster node will be assigned with more tiles, and vice versa (step 2). During the process, the nodes execute independently, and the intermediate results are sent from each Conv node to the Central node (step 3). The Central node will take these intermediate results and compute the final CNN output. (step 4). Figure 9 depicts the timeline for processing an input image with ADCNN. For simplicity, we only show the timing of the Central node and one of the Conv nodes, and we only show the timing for data transmission and layer

Algorithm 2: Statistics collection at the Central node

Input: Time limit T_L. Number of intermediate results n_k^i of input image i received by the Central node from node k within T_L. Total number of Conv nodes K. Decay parameter γ.
Set $s_k = 0$ for each Conv node k.
for *each input image i* **do**
  for $1 \le k \le K$ **do**
    Count the number of partial results n_k^i received from k within the time limit T_L.
    Set $s_k = (1-\gamma)s_k + \gamma n_k^i$

computation at the nodes, even though the Central node also needs to perform input partition and statistics collection concurrently, which will be described later in detail.

As shown in Figure 9, the Central node first partitions the input image i into tiles and transmits one (or more) tiles to a Conv node at t_s^i. After T_F, the Conv node receives the input tiles at t_r^i and performs the convolutional operation, the intermediate results are generated at t_c^i and sent back to the Central node. The Central node receives the intermediate results at t_f^i and performs the computation for the later layers. The final output for input i is produced at t_a^i. To maximize the resource utilization, the Central node will transmit the input tiles for the next image $i+1$ at $t_s^{i+1} < t_c^i$, so that the Conv node can immediately starts the next computation without extra delay. To prevent resource starvation, the Central node will start a timer after transmitting all the tiles of a input image, if the intermediate results from a Conv node is not received within time limit T_L, the Central node will start executing the later layers by setting the missing input to zero.

Each tile sent by the Input partition block is assigned with an image ID *i_id* and a tile ID *t_id*, as indicated in Figure 8. The image index can be used to uniquely identify the input image that is currently under processing, and the tile index indicates which tile of the input image is being processed by the corresponding Conv node. Once the processing is completed at Conv node, the Conv node sends the intermediate results together with the corresponding image ID and tile ID back to the Central node, as shown in Figure 8.

6.2 Statistics Collection Process

To gather the statistics on the node performance, at run time, the Central node runs a statistics collection algorithm (Algorithm 2) in the background. Basically, the Central node counts the number of intermediate results received from node k within the time limit T_L for each input image. It then updates the running mean s_k and reports s_k to the Input partition block periodically.

6.3 Input Partition Process

The running statistics s_k received from the Statistic collection block can be used to estimate the processing speed of node i. The problem of minimizing the task completion time of Conv cluster for an input

Algorithm 3: Input tile allocation algorithm

Set $x_k = 0$ for each Conv node k.
for *each input tile* **do**
  for *nodes which has enough storage capacity* **do**
    Find the node k which produces the least increase on $\max_{1\le k\le K} \frac{x_k}{s_k}$, if multiple nodes satisfy, randomly pick one.
    Set $x_k = x_k + 1$

image i can be formulated as:

$$\begin{aligned} &\min_{x_k} \max_{1\le k\le K} \frac{x_k}{s_k} \\ \text{s.t.} \quad &\sum_k x_k = D, x_k \in \mathbb{Z}^* \\ &Mx_k \le H_k, \forall k \end{aligned} \tag{1}$$

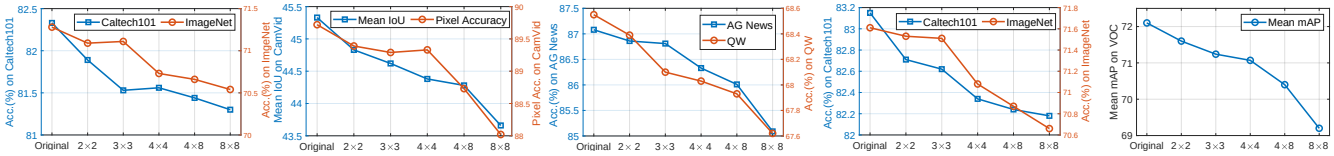
where D is the total number of tiles in input image i, x_k is the number of tiles allocated to node k, M is the size (in bits) of each tile, and H_k is the storage capacity in bits for Conv node k. This problem is a variant of classical parallel machine job scheduling problem and can be solved with a greedy approach [16] as described in Algorithm 3, which iteratively searches for the best node to allocate the new input tile. Although this assignment scheme is simple, in practice, we notice that it handles the performance variation extremely well, and achieves a nearly perfect utilization for the Conv node cluster. Moreover, this scheme naturally handles the Conv node failure, as the s_k changes based on the runtime performance of node k. If node k fails, s_k will become zero and no tiles will be assigned to it.

7 EVALUATIONS

In this section, we first present the performance of the retrained CNNs with different CNN models and datasets, then we describe our testbed implementation and comparison with the other state-of-the-art approaches.

7.1 Accuracy Evaluation

We choose four different CNN models for accuracy evaluation, including: 1. VGG16 [31] and ResNet34 [17] on Caltech101 [2] and ImageNet [5] for image classification, 2. YOLO [26] on PASCAL VOC [8] dataset for object detection task, 3. FCN [22] on CamVid [3] for semantic segmentation, 4. CharCNN [37] on AG_news [1] and Query_Well_formedness [10] for text classification. We apply FDSP to partition the early layer blocks of these CNN architectures, and adopt the adaptive ReLU and quantization to the output of the separable layer blocks for communication reduction. To evaluate FDSP, we first train the CNN with their original topology until model has converged. After that, the resulting CNN is used as the input for progressive retraining algorithm described in Section 5. The model is retrained for some epochs after each modification is applied, until the accuracy is recovered. For the lower and upper bounds of the clipped ReLU, we first search for a coarse parameter range based on separable layer block output statistics, and then perform grid search to produce expected output sparsity. For the other hyperparameters, (e.g., learning rate, weight decay), we use



(a) VGG16 (b) FCN (c) CharCNN (d) ResNet34 (e) YOLO

Figure 10: Performance comparison between original CNN and modified version of CNNs after retraining. For VGG16, FCN, CharCNN, ResNet34 and YOLO, the number of partitioned layer blocks are: seven, seven, four, twelve and twelve, respectively.

Table 1: Number of epochs needed for each modification during progressive retraining for 8×8 partition.

Models	FDSP	Clipped ReLU	Quantization	Total
VGG16	5	3	2	10
ResNet34	5	3	3	11
YOLO	7	4	2	13
CharCNN	2	2	1	5

Table 2: Conv nodes output size before and after pruning for 8×8 partition.

	VGG16	ResNet34	FCN	YOLO	CharCNN
Original	1×	1×	1×	1×	1×
After pruning	0.032×	0.043×	0.011×	0.020×	0.056×

the default setting in PyTorch github repository [6]. In addition, we apply 4-bit quantization on the outputs from the clipped ReLU, which are then encoded with RLE for transmission. We evaluate five spatial partitioning options, 2×2, 3×3, 4×4, 4×8 and 8×8 for each CNN.

Figure 10 shows the accuracies of the original CNNs as well as the corresponding retrained CNNs. Compared with the original CNN, the accuracy degradation for the retrained VGG16, ResNet34 and CharCNN are all less than 1% for all the partitions on both Caltech101 and ImageNet. For FCN, both mean Intersection over Unions (IoU) and pixel accuracies are within 1.3% of the original CNN model for all the partitions. Finally, we evaluate the performance of YOLO in terms of mean average precision (mAP), and we notice that the partitioned YOLO models have an average reduction of 1.2% on mAP across all the partition options compared with the original YOLO, which is relatively small. Therefore, we conclude that the partitioned CNN can achieve a very competitive accuracy via the progressive retraining technique.

Moreover, to evaluate the cost of the progressive retraining, we also record the number of additional epochs required for the progressive retraining process. Table 1 shows the number of additional epochs applied for each modification on the CNN architecture during the retraining process shown in Algorithm 1. Due to the space limit, we only show the results for 8×8 partition, and all the other partitions have similar results. Compared with the original CNN training, which usually requires hundreds of epochs, the number of epochs for progressive retraining is very small for all the CNN models. We conclude that, given the original CNN model, we can easily

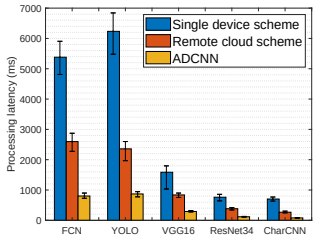


Figure 11: Latency comparison with 95% confidence interval between ADCNN and baselines.

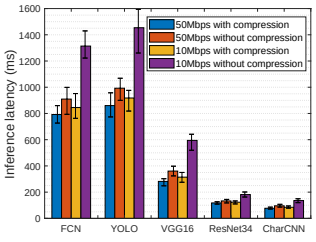


Figure 12: Effect of pruning under different transmission rates with 95% confidence interval.

and quickly produce the partitioned CNN model by retraining it with a few more epochs.

Finally, Table 2 shows normalized ofmap size of the last separable layer block after applying the pruning strategies described in Section 4. We notice that the ofmap size is reduced significantly (33× on average) after pruning, which greatly lowers the communication overhead between the Conv nodes and the Central node.

7.2 ADCNN System Evaluation with Stable Environment

To measure the system performance of ADCNN in a real world environment, we have built a testbed with a cluster of Raspberry-Pi3 B+ devices, each of which is the same as those used for latency measurement in Section 2.2. The ADCNN is evaluated by using the five CNNs shown in Figure 10. We allow a maximum accuracy degradation of 1% for all the retrained CNNs. According to the results shown in Figure 10, we apply 8×8 partitions for VGG16, ResNet34 and CharCNN, and 4×8 partition and 4×4 partition for FCN and YOLO, respectively. We implement ADCNN system with nine identical Raspberry Pi devices which simulate the edge devices. Among these nine devices, eight are used as Conv nodes, and the rest one is used as the Central node. The Conv nodes and the Central node are connected with a WiFi network with a measured bandwidth around 87.72 Mbps. Since all the eight Conv nodes are identical in terms of computing power, each of them should be assigned with the same number of input tiles under the normal condition. The time limits T_L in Algorithm 2 for intermediate results transmission is set to $30ms$. The parameter γ in algorithm 2 is set to 0.9. Each CNN is partitioned and retrained with Algorithm 1. The new weights of the separable layer blocks are saved inside each Conv node, and the weights of the rest layers are stored in the

Table 3: Latency breakdown of three schemes.

Scheme	Input/output transmission	Computation
ADCNN	37.14ms	202.88ms
Single-device	0ms	1586.53ms
Remote cloud	502.21ms	98.94ms

Central node for later layer processing. For evaluation, we compare ADCNN with two baseline schemes. The first one is the single device scheme, where only one Raspberry Pi device is used for local CNN inference. The second one is the remote cloud scheme, where the input is uploaded from the one Raspberry device to a remote cloud with a measured bandwidth around 61.30 Mbps for centralized processing, and the result is reported back. A single-core Amazon EC2 *p3.2xlarge* instance with one Tesla V100 GPU, eight vCPUs[1] and one 16GB RAM is deployed to simulate a cloud server.

We record the average end-to-end inference latencies of 100 input samples for all the three schemes. For ADCNN, the end-to-end inference delay is defined as the time period between the Central node starts partitioning the input image and its inference result is generated. For remote cloud scheme, the end-to-end inference latency is the time period between the edge device starts transmitting the input image and the result is received by the edge device.

Figure 11 illustrates the latency measurements. ADCNN significantly reduces the processing latency for all the five CNNs. Compared with the single device scheme and remote cloud scheme, ADCNN decreases the average processing time by 6.68× and 4.42×, respectively. To better understand the performance gains, we further measure the processing time breakdown of all the three schemes on VGG16 (Table 3). We notice that the performance of the single device scheme is hindered by the computing power of the edge device, and the effectiveness of remote cloud scheme is constrained by the transmission time between the edge device and the remote server. By comparison, ADCNN achieves a much lower transmission time between the Conv node cluster and the Central node, and obtains a superior inference speed.

7.2.1 Effect of Pruning on Conv Nodes Outputs. Next, we present the effect of pruning on the Conv node output by demonstrating its contribution on inference latency reduction under different transmission rates. Figure 12 shows the difference on inference latency with and without pruning under two transmission rates between the Conv nodes and the Central node. We notice that pruning the Conv node outputs leads to a reduction on the inference latency by 10.73% and 31.2% respectively under 87.72Mbps and 12.66 Mbps for all the five CNNs. Henceforth, we conclude that the compressing the Conv nodes outputs can effectively reduce the communication overhead and lower the processing latency.

7.2.2 Scalability of ADCNN System. We then demonstrate the scalability of ADCNN by measuring the inference latency with different number of Conv nodes. The transmission rate between the Conv nodes and Central node is 87.72Mbps. As described in the left plot of Figure 13, the speedup relative to a single device scheme grows from 1.8× to 6.2× as the number of Conv nodes increases from

[1]One vCPU is a thread of either an Intel Xeon core or AMD EPYC core.

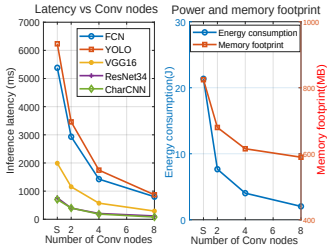


Figure 13: Latencies, energy consumption and mem. usage with different number of Conv nodes. 'S' in the x-axis means single device scheme.

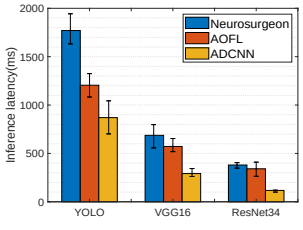


Figure 14: Latency measurement with 95% confidence interval of ADCNN, Neurosurgeon and AOFL on YOLO, VGG16 and ResNet34.

2 to 8. Moreover, we also notice that the growth rate of speedup decreases as more Conv node is introduced. This is because as more Conv node is used, the overall latency is restrained by the communication overhead between the Conv nodes and the Central node, as well as the processing speed of Central node.

ADCNN can greatly accelerate CNN inference by partitioning the ifmaps into multiple tiles with FDSP and processing them in parallel, and the latency decreases as more tiles and mode Conv nodes are used. Unfortunately, a growing number of input partitions will further lower the accuracy of the retrained model, which results in a tradeoff between the processing latency and the prediction accuracy of ADCNN. The evaluation results in Section 7.1 show that most of the CNNs can apply 8×8 partition without noticeable accuracy degradation. This is acceptable for the delay-sensitive applications which can tolerate minor performance degradation (*e.g.*,real-time object detection). In practice, network operator can decide the partition size based on their accuracy requirement.

7.2.3 Energy Consumption and Storage Complexity. Next we measure the average energy consumption and memory consumption of a single Conv node for processing ImageNet samples with VGG16. We use *MakerHawk USB Power Meter* [7] to measure the energy consumption of a Raspberry Pi device (Conv node). More specifically, we vary the number of Conv nodes and measure the energy consumption and memory footprint of a single Conv node. Since all the Conv nodes are identical in terms of processing power, they are allocated with the same number of input tiles from the Central node. For comparison, we also measure the corresponding energy consumption and memory footprint for single-device scheme. As shown in the right plot in Figure 13, compared with the single-device scheme, both energy consumption and memory footprint decrease as the number of Conv nodes increases. This is because each Conv node only needs to store and process its assigned tiles, which gives a lower computation workload and hence lower energy consumption and storage cost.

7.3 ADCNN System Evaluation with Heterogeneous Edge Environment

In this section, we show the resilience of ADCNN system against the variations on node performance, and demonstrate its performance with heterogeneous edge devices. To simulate the heterogeneity

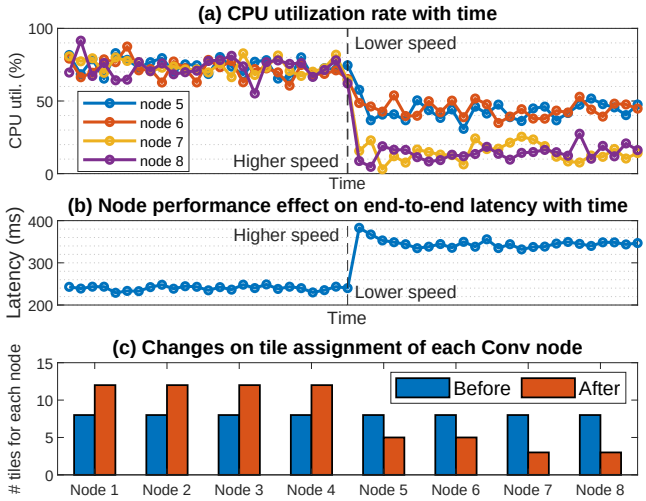


Figure 15: Impact of variation of node performance on latency. The black dotted bar in (a) and (b) show when node performance degrades, and (c) shows the changes on tile assignments before and after the node performance degrades.

between the Conv nodes, we adjust the CPU processing speed on four of the Conv nodes (node 5,6,7,8) in the middle of the processing 50 input images for VGG16 with 8×8 partitioning scheme, and detect its impact on tile assignment and overall inference latency. To degrade the node performance, we adopt *CPUlimit* [4] to restrict the its CPU usage on processing the input tile. More specifically, we reduce the CPU power by around 55% for node 5,6 and and 76% for node 7,8, respectively. The temporal variations on CPU utilization for these four nodes are depicted in Figure 15(a). The corresponding changes on overall inference latency and tile assignment are shown in Figure 15(b) and (c), respectively.

As shown in Figure 15 (c), the tiles are evenly distributed to each node in the beginning, and each node is allocated with 8 tiles. The performance degradation on node 5-8 triggers the Central node to move partial workload of node 5-8 to the rest Conv nodes, so that node 1-4 are each assigned with 12 tiles, and node 5-8 are assigned with 5,5,3,3 tiles, respectively. For latency, the performance degradation on node 5-8 immediately results in a growth on processing latency from 241ms to 392ms. Then ADCNN system collects the running statistics s_k from the slow nodes with Algorithm 2 and adjust the tile allocation with algorithm 3, generating a final processing latency of 351ms. The results indicate that ADCNN can be used in heterogeneous edge environment and adapt to dynamic variations on node performance at runtime.

7.4 Comparison with Other Designs

In this section, we compare ADCNN with two state-of-the-art CNN partitioning approaches: Neurosurgeon [19] and Adaptive Optimal Fused-layer (AOFL) [39]. Neurosurgeon adopts layerwise partition to split CNN into two parts, where the first part runs on the edge device and the second partition runs on the cloud. Neurosurgeon searches for the optimal partition position which generates the smallest inference latency. AOFL is similar to ADCNN in that it also applies spatial partition on the input images to accelerate the inference over the edge devices. Unlike ADCNN which retrains the CNN to eliminate the cross-tile communication, AOFL extends the spatial size of each tile such that the data halos between the tiles are covered, and therefore no cross tile communication is required during the inference. However, this method introduces additional computation overhead, as each device needs to perform the convolutional operation over a larger ifmap, and this overhead increases exponentially as the number of fused layer increases.

For implementation of Neurosurgeon, we utilize the aforementioned Raspberry Pi device to simulate the edge device as before, and deploy the same EC2 *p3.2xlarge* instance as used in the remote cloud scheme for cloud device. The transmission rate between the edge device and cloud is 61.30Mbps on average. We try every possible layerwise partition position for VGG16, ResNet34 and YOLO, and select the partition position with the minimum latency. For AOFL implementation, we partition the input image spatially into eight pieces, where each piece is processed by the same Raspberry Pi device as used in ADCNN. We measure the average processing speed and transmission latency for each CNN layer, and search exhaustively for the optimal fuse layer block selection.

We measure the average end-to-end processing latencies for all the three schemes over 200 input images. As shown in Figure 14, ADCNN outperforms Neurosurgeon and AOFL by 2.8× and 1.6× on average in terms of processing latency for all the three CNN models. To reduce the computation overhead on the edge device, Neurosurgeon partitions the CNN at early layers for all the three models. However, this produces a large communication overhead between the edge device and the cloud (as the ofmaps for the early layers are usually quite large), which corresponds to 67% of the overall processing latencies for all the three CNNs on average. By comparison, ADCNN allows the CNN to be processed completely at the network edge, which naturally reduces the communication overhead. Furthermore, the Conv node cluster also achieves a comparable processing speed as the centralized cloud server. For AOFL, most of the fused layers are consists of the early CNN layers. For example, for YOLO, VGG16 and ResNet34, the first 14,13 and 16 layers are fused, respectively. This is because early layers have a larger ifmap and ofmap than the later layers, which makes the extra computation overhead caused by the data halos relatively lower, as the data halos are smaller compared with the feature map size for the early layers. In contrast, ADCNN completely eliminates this computation overhead, thereby achieving a lower inference latency.

8 RELATED WORK

8.1 Edge computing and IoT

The application of edge computing under the IoT paradigm has been investigated by many of previous works [13, 15]. As IoT becomes increasingly popular, there is a growing trend towards pushing computation tasks to network edge. In [34], the authors deploy a face recognition application at the edge and show that the latency is reduced by 7× when the image is processed by the smartphone as opposed to the cloud. Wang *et al.* [33] push deep learning applications from cloud towards mobile devices. Two applications are implemented at the mobile device including mood disturbance

inference and user identification, both applications use data collected locally at mobile device. Both work consider a simple analytic model which can operate at the user end with low latency, while ADCNN targets to implement a large-scale deep neural network by splitting computation loads across edge devices.

8.2 Distributed Machine Learning

Recently, there have been several studies on distributed CNN inference using multiple edge devices. The authors of [38] and [39] apply spatial partition on CNN, and propose distributed CNN inference frameworks on edge devices. The authors have also noticed the significant communication overhead during inference at runtime, and proposed a multi-round workload scheduling scheme to maximize data halo reuse. In contrast, ADCNN completes the CNN inference in a single round, while incurring much lower communication overhead. Furthermore, these studies have not considered the impact of dynamic variation in the system, making them less reliable.

Kang *et al.* [19] and Teerapittayanon *et al.* [32] both propose a layerwise partition scheme for DNN. In [32], the authors describe a system named DDNN, which allows the fast inference of deep neural network by mapping the shallow portion of the neural network to the end devices. For the easy task, DDNN takes the output from the end devices without sending the results to the cloud for further processing. In [19], the authors introduced a system call Neurosurgeon, which can dynamic search for the optimal position for DNN layerwise partition with the lowest communication overhead and computation delay. However, both system introduces a large communication overhead between the end devices and cloud. In contrast, ADCNN allows inference task to be processed completely at the network edge, which greatly accelerates the inference.

9 CONCLUSION

With the growing popularity of the IoT, there is an increasing demand to run CNN inference locally in edge networks. However, due to the limited computing resources on edge nodes, running a CNN on a single edge node often leads to poor service quality in terms of latency. In this work, we introduce ADCNN, a distributed inference framework which jointly optimize CNN architecture and computing system for better performance in dynamic network environments. ADCNN applies FDSP to partition the compute-intensive convolutional layers into many small independent computational tasks which can be executed in parallel on separate edge devices. To mitigate the accuracy loss generated by the FDSP, ADCNN retrains the original CNN model at a low cost. The partitioned tasks are then allocated to edge devices according to their current operational condition. Compared to existing distributed CNN inference approaches, ADCNN provides up to 2.8× lower latency, while achieving a competitive inference accuracy. Additionally, ADCNN can quickly adapt to the variations on edge device performance.

REFERENCES

[1] [n.d.]. AG news. https://rdrr.io/cran/textdata/man/dataset_ag_news.html.
[2] [n.d.]. Caltech101. www.vision.caltech.edu/Image_Datasets/Caltech101/.
[3] [n.d.]. camvid. http://mi.eng.cam.ac.uk/research/projects/VideoRec/CamVid.
[4] [n.d.]. Cpulimit. http://cpulimit.sourceforge.net.
[5] [n.d.]. Imagenet dataset:. http://www.image-net.org/.
[6] [n.d.]. ImageNet training in PyTorch. ([n. d.]). .
[7] [n.d.]. Makerhawk USB power meter. https://www.makerhawk.com/.
[8] [n.d.]. PASCAL VOC dataset. http://host.robots.ox.ac.uk/pascal/VOC/.
[9] [n.d.]. Pytorch official website. https://pytorch.org/.
[10] [n.d.]. Query-wellformedness. https://github.com/google-research-datasets/query-wellformedness.
[11] [n.d.]. Raspberry Pi website:. https://www.raspberrypi.org.
[12] Parashar Angshuman et al. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture*.
[13] Flavio Bonomi et al. 2012. Fog Computing and Its Role in the Internet of Things. In *ACM MCC workshop on Mobile cloud computing*.
[14] Seraphin B Calo et al. 2017. Edge computing architecture for applying AI to IoT. In *2017 IEEE International Conference on Big Data*. IEEE.
[15] Dastjerdi et al. 2016. Fog computing: Helping the Internet of Things realize its potential. In *Computer 49, no. 8: 112-116*.
[16] Mohamed I Dessouky, Ben J Lageweg, Jan Karel Lenstra, and Steef L van de Velde. 1990. Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica* 44, 3 (1990), 115–123.
[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[18] Zhiming Hu et al. 2019. DeepHome: Distributed Inference with Heterogeneous Devices in the Edge. In *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications*. ACM, 13–18.
[19] Yiping Kang et al. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 615–629.
[20] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset.
[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[22] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3431–3440.
[23] Jiachen Mao et al. 2017. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *Proceedings of the 36th International Conference on Computer-Aided Design*.
[24] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1396–1401.
[25] Jianli Pan and James McElhannon. 2017. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal* 5, 1 (2017), 439–449.
[26] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *IEEE conference on computer vision and pattern recognition*.
[27] A. H. Robinson et al. 1967. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE 55.3: 356-364.* (1967).
[28] Ioffe Sergey et al. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *arXiv preprint arXiv:1502.03167*.
[29] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.
[30] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. In *arXiv preprint arXiv:1409.1556*.
[31] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[32] Surat Teerapittayanon et al. 2017. Distributed Deep Neural Networks over the Cloud, the Edge and End Devices. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*.
[33] Ji Wang et al. 2018. Deep Learning Towards Mobile Applications. In *IEEE 38th International Conference on Distributed Computing Systems*.
[34] Shanhe Yi et al. 2015. Fog Computing: Platform and Applications. In *IEEE Workshop on Hot Topics in Web Systems and Technologies*.
[35] Jason Yosinski et al. 2015. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579* (2015).
[36] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Advances in neural information processing systems*. 3320–3328.
[37] Xiang Zhang et al. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*.
[38] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. 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 (2018), 2348–2359.
[39] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. 2019. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 195–208.