Edge^{n}AI: Distributed Inference with Local Edge Devices and Minimal Latency

Maedeh Hemmat, Azadeh Davoodi, and Yu Hen Hu
Department of Electrical and Computer Engineering
University of Wisconsin - Madison, Madison, WI, USA
{hemmat2, adavoodi, yhhu}@wisc.edu

***Abstract*—We propose Edge^{n}AI, a framework to decompose a complex deep neural networks (DNN) over n available local edge devices with minimal communication overhead and overall latency. Our framework creates small DNNs (SNNs) from an original DNN by partitioning its classes across the edge devices, while taking into account their available resources. Class-aware pruning is applied to aggressively reduce the size of the SNN on each edge device. The SNNs perform inference in parallel, and are configured to generate a 'Don't Know' response when an unassigned class is identified. Our experiments show up to 17X inference speedup compared to a recent work, on devices of at most 150 MB memory when distributing a variant of VGG-16 over 20 parallel edge devices.**

I. Introduction

In recent years, there has been tremendous development in using deep neural network (DNN) machine learning algorithms to address computation-intensive, challenging pattern/object recognition problems in cutting edge information technology applications such as object recognition, speech recognition, smart homes, health care, autonomous vehicle navigation, etc. [2], [10], [7]. In order to achieve state-of-the-art performance in such real-life applications, DNNs have become more complex [14], [16], [18]. These translate to higher memory and computation requirements as in cloud computing.

On the other hand, DNN-based Artificial Intelligence heavily relies on devices such as sensors to gather application data (which are projected to grow into billions in near future [1]). A common solution today is to transfer such raw data via network to the cloud infrastructure for DNN processing and then transfer the results back to the edge devices. However, such an approach requires high communication bandwidth to transfer raw data to the cloud and will not provide an acceptable latency for real-time user experience.

Many recent approaches have developed strategies for distributed implementation of DNNs across edge devices to alleviate or completely eliminate reliance on the cloud. Saguil et al. study strategies to optimally partition a complex DNN between one edge device and cloud to minimize the overall latency [12]. Communication overhead to the cloud is reduced by sending extracted features from edge-processed layers, as opposed to raw data. Teerapittayanon et al. [17] propose a DNN partitioning scheme where at the front end, each device implements a feature extractor. This approach incorporates a local early exit strategy to avoid cloud for some inferences. But it still requires multiple levels of wireless communication across edge devices and it does not completely eliminate reliance on the cloud for all the inferences. Mao et al. [9] propose a distributed scheme to completely eliminate reliance on cloud. The approach partitions an input image and assigns them to distributed, local, worker edge devices. Wireless communication overhead is reduced using a map-reduce strategy. The overall latency, however, remains high when multiple levels of sequential, wireless communication is required.

The above approaches are effective schemes to avoid sending large volume data to the cloud and utilize local edge devices. However, they leave significant room for improvement of the overall latency because sequential stages of wireless communication should be performed for a single inference. Moreover, they are not designed to utilize available resource-constrained edge/IoT devices, as many, as projected to become available in near future [1].

Motivation: This work is inspired by two key factors. First, our goal is minimize the overall latency of inference for a distributed implementation of a complex DNN. This goal is naturally coupled with reducing the communication overhead across devices. Second, with the rising availability of edge devices, we fundamentally rethink how an inference may be performed for a distributed implementation. Instead of focusing on a single or few edge devices, we are motivated to utilize the available, local edge devices as many as possible to reduce latency of an inference and hence improve the user experience.

Contributions: We propose Edge^{n}AI, a framework to utilize many parallel, independent-running edge devices which communicate only once to a single 'back-end' device (also an edge device) to aggregate their predictions and produce the result of the inference. To achieve this distributed implementation, Edge^{n}AI first partitions the classes of the complex DNN into subsets to be assigned across the available edge devices while considering the computational resources of each device. The DNN is then aggressively pruned for each device for its set of assigned classes. Each smaller DNN (SNN) is further configured to return a 'Don't Know' when it identifies an input from a class that is not assigned to it. Each SNN is generated by Edge^{n}AI from the complex DNN at the beginning and then loaded onto its corresponding edge device, without the need for retraining. Each SNN will perform an inference based on its received input (which could be a common input broadcasted to all, or an input taken from the view of that edge device as in a context-aware distributed setup [3]). The results of the SNNs are sent to the back-end device which is trained by Edge^{n}AI as a simple classifier and hence can also fit on an edge device.

For an effective implementation of Edge^{n}AI, we make the following contributions:

- We utilize a normalized entropy metric to compare the probabilities generated for different classes across the SNNs in a meaningful manner. The normalized entropy is also used to generate a 'Don't Know' response for each SNN if it identifies an input that does not belong to any of its assigned classes.
- To decide the subset of classes that should be assigned to each SNN, we propose an effective procedure to evaluate many class partitioning candidates. This is the first step prior to generation of the SNNs. At the heart of our procedure is an efficient method to estimate the normalized entropy of an unknown SNN, solely based on the classes that are considered to be assigned to an edge device.

In our experiments we show how Edge^{n}AI can achieve an effective distributed implementation for a variety of hardware platforms with different number of devices. For example, we show up to 17X speedup in inference latency using Edge^{n}AI with negligible loss in the overall classification accuracy, compared to a recent work on distributed implementation [15]. This is when using 21 edge devices

2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC) | 978-1-6654-2135-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASP-DAC52403.2022.9712496

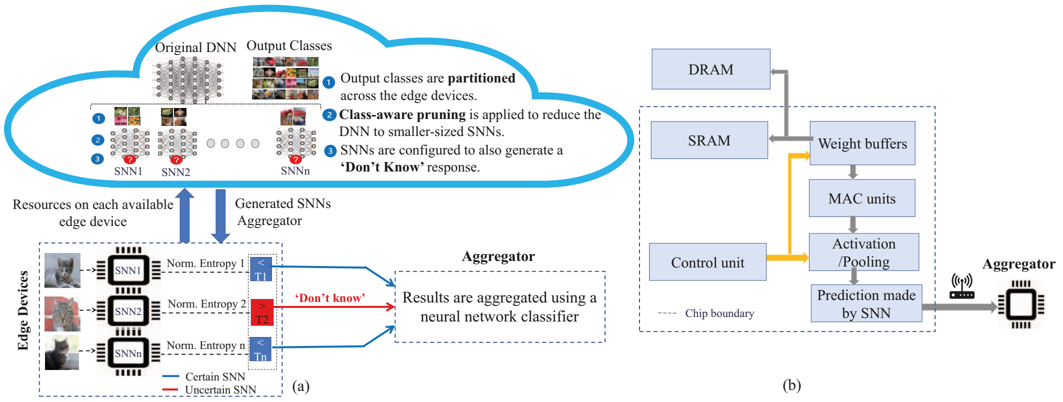

Original DNN
Output Classes
1 Output classes are partitioned across the edge devices.
2 Class-aware pruning is applied to reduce the DNN to smaller-sized SNNs.
3 SNNs are configured to also generate a 'Don't Know' response.
SNN1
SNN2
SNNn
Resources on each available edge device
Generated SNNs Aggregator
Edge Devices
Norm. Entropy 1
Norm. Entropy 2
Norm. Entropy n
< T1
> T2
< Tn
'Don't know'
Aggregator
Results are aggregated using a neural network classifier
Certain SNN
Uncertain SNN
(a)
DRAM
SRAM
Weight buffers
MAC units
Activation /Pooling
Control unit
Prediction made by SNN
Chip boundary
Aggregator
(b)

Fig. 1: (a) Overview of Edge^{n}AI; (b) Abstract hardware model of an edge device. Computation of latency incorporates all these components.

(20 for the SNNs and one for the aggregator in Edge^{n}AI), each with at most 150 MB on-chip memory (with no off-chip memory).

In the remainder of the paper we give an overview of Edge^{n}AI and its details in Sections II and III. Simulation results are presented in Section IV followed by conclusions.

II. Overview of Edge^{n}AI

Figure 1(a) gives an overview of Edge^{n}AI which resides in the cloud. Initially, Edge^{n}AI receives information about the available edge devices, and their memory and computational resources. Based on this information, the complex DNN is estimated as n smaller DNNs (SNNs) and an aggregator. (The number of available edge devices is $n + 1$). Figure 1(b) shows the abstract hardware model of each edge device which includes on-chip buffers and memory (SRAM) and possibly off-chip memory (DRAM), MAC units to perform the bulk of the computations, and wireless link to send the results to the aggregator[1]. Generation of the SNNs by Edge^{n}AI is performed only once as a pre-processing step.

To perform a distributed inference, each SNN will receive an input. This input could be the same image that is broadcasted to all, or it could be different views of the same object taken by each device, such as in a multi-view setup as in [3]. Each SNN generates probabilities associated with its assigned classes. In order for the aggregator to compare these probabilities across all SNNs in a meaningful way, first a normalized entropy is calculated for each SNN. This normalized entropy is also evaluated against a threshold (configured initially by Edge^{n}AI) to generate a 'Don't Know' response [8], [4] when an SNN identifies its input not to be from any of the classes that are assigned to it. The aggregator then combines the results of the SNNs (by looking at the normalized entropies and the generated Don't Know responses) and produces the final inference result. The aggregator is implemented as a neural network classifier which as we show can be implemented with a simple structure and small size. Augmenting the network architecture to add one more output class to generate a Don't Know response is also possible but inefficient since it requires retraining each SNN.

To generate the SNNs, first, a subset of the classes of the original DNN is assigned by Edge^{n}AI to each device through a class partitioning step. Next, class-aware pruning is applied to prune the DNN (for each case) and generate a smaller DNN (i.e., an SNN) for each device based on its subset of assigned classes. This step utilizes a recent class-aware pruning scheme proposed in [6] which is able to generate each SNN without need for retraining and with only a negligible loss in classification accuracy for its assigned classes.

[1]All of these components are accounted to estimate the inference latency.

Particular challenges for an effective implementation of Edge^{n}AI include how to fairly compare the per-class probability generated by individual SNNs against each other, and how to efficiently perform class partitioning across the available devices so that class-aware pruning (which can be time-consuming) is only performed a limited number of times. In the next section, we discuss these details. Specifically, in Section III-A we discuss calculation of the normalized entropy and how it helps with fair comparison of results of individual SNNs, and how it may be configured to generate a Don't Know response. We also discuss how the aggregator is designed to generate the final inference result using this information in Section III-B. In Section III-C we discuss an efficient procedure to evaluate class-aware partitioning combinations and identify the most promising candidates to enhance the overall classification accuracy.

Overall, Edge^{n}AI generates a distributed implementation suitable to fit in resource-constrained edge devices with minimum latency because it reduces the most time-consuming step (i.e., wireless communication) to a single step when sending the outputs from the SNNs to the aggregator. The transferred data are normalized entropies so the required bandwidth is minimal.

III. Details of Edge^{n}AI

Here we discuss three important design aspects of Edge^{n}AI. First, for a set of already-generated SNNs, we discuss how a normalized entropy is defined to combine their individual inferences in a meaningful manner. We also discuss how normalized entropy is used to define a Don't Know signal. Second, we discuss how the aggregator is designed to combine the results of the SNNs. Third, we discuss how SNNs are generated in an efficient and effective manner.

A. Normalized Entropy and Generation of A Don't Know Response

Recall, each SNN is a reduced version of the original DNN when pruning the DNN for a subset of its classes. These subsets are also assigned in a way that the SNNs together cover all the classes in the original DNN. During distributed inference, it is not meaningful to directly compare the probabilities generated for different classes across the SNNs with each other. For example, suppose one SNN infers an input image to be a cat with probability 0.6 because its output with highest probability corresponds to the class of cats. A second SNN does not even include cats in its output classes and infers the same image to be a dog with probability 0.7 because its highest probability output corresponds to the class of dogs. Simply combining the results of these two SNNs by taking the maximum probability is not meaningful and may result in the wrong classification outcome.

To address this issue we first compute a normalized entropy metric for each input and each SNN, and then use this metric to define a Don't Know response for each SNN to quantify the cases when an SNN is not certain about its prediction. Given an input, the normalized entropy (NE) of an SNN is defined as follows [17]:

$$\mathcal{E}(SNN) = -\sum_{c=1}^{|\mathcal{C}|} \frac{x_c \log(x_c)}{\log |\mathcal{C}|} \tag{1}$$

where x_c is the generated probability by SNN for class c for that input. Note $c \in \mathcal{S} \subset \mathcal{C}$ where $\mathcal{S}$ is the subset of classes assigned to the SNN and $\mathcal{C}$ is the set of all the classes (in the DNN), some of which may not have been assigned to the SNN. Lower values of NE correlates to higher certainty in the inference.

Calculating A Don't Know Response: The normalized entropy metric can further be used to define a Don't Know response. More specifically, we configure each SNN to predict if each of its received inputs is *relevant* or *irrelevant*. A relevant input is one predicted to belong to the set of classes covered by the SNN. An irrelevant input is predicted to belong to the remaining classes (not covered by SNN).

Our key observation is that, for each SNN, normalized entropy (NE) is a smaller quantity for relevant inputs compared to the irrelevant ones. This observation can be used to define a 'Don't Know' response for each SNN by calculating one threshold (per SNN) for its NEs. An NE below the threshold means the SNN outputs an inference result for an input because it identifies it as relevant. In this case, the NE is directly sent to the aggregator (together with the index of class with the highest output probability). Otherwise, if the NE is higher than the threshold, the input is rejected by the SNN for inference. In this case, a Don't Know response is encoded by setting the NE to 1 before sending it to the aggregator to indicate the SNN is uncertain and rejects the inference.

Algorithm 1 shows how the threshold is calculated by Edge^{n}AI as a pre-processing step for an SNN. (This algorithm is called independently for each SNN to find its corresponding threshold.) The inputs are the vectors NE_R and NE_I. (Parameters T_{start}, T_{end} and $step$ are also specified as inputs which we set in our experiments.) The output is the threshold Th for the SNN.

The algorithm starts with a low threshold ($T = T_{start}$) and counts the number of NEs in NE_R which are lower than T, as well as the number of NEs in NE_I which are higher than T. (See lines 3-13.) Next, $acceptRecall$ is calculated as the rate of relevant inputs which are correctly identified as relevant for that T. These inputs are correctly accepted by the SNN to make an inference (line 14). Similarly, a $rejectRecall$ is calculated as the rate of irrelevant inputs which are correctly identified to be irrelevant for that T (line 15). These irrelevant inputs are correctly rejected by the SNN to make an inference. Next in line 16, performance P is calculated as multiplication of these two quantities. Higher performance takes into account that we desire relevant inputs to be identified as relevant (higher accept recall), as well as irrelevant inputs to be identified as irrelevant (higher reject recall). Next in lines 17-20, threshold Th is updated only if P is the largest performance so far. Next, T is increased by $step$. The above steps are repeated until T_{end} is reached.

The above algorithm, in effect searches for the optimal value of threshold for an SNN within a range (T_{start} to T_{end} with granularity $step$) with the objective to maximize the performance. This is because the overall classification accuracy of the distributed inference depends on how accurate each individual SNN is at (1) accepting to make an inference on its relevant inputs; (2) rejecting its irrelevant inputs and generating 'Don't Know' response.

Figure 2 shows an example of per-device performance and probability density function (PDF) of normalized entropies of relevant and irrelevant inputs for a pruned variant of VGG-16 which is decomposed by Edge^{n}AI into 3 SNNs. Each row in the figure corresponds to one device (i.e., one SNN). Each plot in the first column shows 3 curves corresponding to the accept recall, reject recall, and device performance as a function of the entropy thresholds.

Algorithm 1 Find normalized entropy threshold for a given SNN.

Inputs: NE_R: Vector of normalized entropies calculated for relevant inputs (dimension $1 \times |R|$); NE_I: Vector of normalized entropies calculated for irrelevant inputs (dimension $1 \times |I|$); T_{start}; T_{end}; $step$.
Outputs: Entropy threshold Th for the SNN.

```
 1: $T = T_{start}$, $P_{max} = 0$
 2: while $T \le T_{end}$ do
 3:     relevantCount = irrelevantCount = 0
 4:     for each $i = 1$ to $|R|$ do
 5:         if $NE_R(i) \le T$ then
 6:             relevantCount++
 7:         end if
 8:     end for
 9:     for each $i = 1$ to $|I|$ do
10:         if $NE_I(i) \ge T$ then
11:             irrelevantCount++
12:         end if
13:     end for
14:     acceptRecall = relevantCount / $|R|$
15:     rejectRecall = irrelevantCount / $|I|$
16:     $P$ = acceptRecall $\times$ rejectRecall
17:     if $P > P_{max}$ then
18:         $Th = T$                        ▷ Update the entropy threshold.
19:         $P_{max} = P$
20:     end if
21:     $T = T + step$
22: end while
```

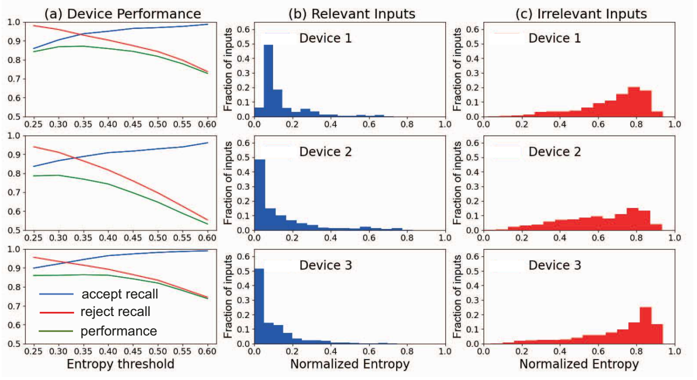


Fig. 2: Per-device performance and PDF of normalized entropy for relevant and irrelevant inputs using 3 SNNs.

As expected, for all three SNNs, relevant inputs have significantly smaller normalized entropy compared to irrelevant ones. It can hence be concluded that each SNN is more likely to correctly accept an inference when seeing a relevant input, as well as to correctly reject an irrelevant input and generate a 'Don't Know' response. Furthermore, as the plots in the first column show, there is a trade off between accept and reject recalls when the threshold is varied. Specifically, with increase in the threshold, the accept recall improves while the reject recall is degraded. Also, different SNNs may have different entropy thresholds. In this example, the entropy thresholds (i.e., points with highest performance) are 0.30, 0.25, and 0.25 for the first, second, and third device, respectively.

B. Design of the Aggregator

The aggregator receives the following inputs from each SNN. First, it receives a normalized entropy which also encodes a Don't Know signal. (Recall if the normalized entropy is found to be higher than the SNN's threshold, then the Don't Know signal is encoded by setting the normalized entropy to its highest value =1). It also receives the index of the class with the highest output probability.

A naive approach is to first eliminate the normalized entropies of all uncertain devices (all NEs=1), and then pick the inference result of the device with the lowest NE. However, such an approach cannot maintain accuracy, as verified, especially as number of SNNs grows.

To tackle this issue, the aggregator is implemented as a lightweight neural network, featuring only 3 layers with at most 60 neurons per layer. As a pre-processing step, the aggregator is trained using stochastic gradient descent algorithm, for at most 10 epochs. To obtain the training set, we randomly selected an equal number of samples (50 in our experiments) from each class to ensure that all classes are covered during training and they all contribute equally to train the network. We note this training set is different than the one used to calculate the threshold of normalized entropies in order to reduce overfitting. To train the aggregator, first the SNNs are generated by so the actual inputs to the aggregators (generated by SNNs) may be determined for training purposes.

C. Generation of the SNNs

As shown in Figure 1, Edge^{n}AI is responsible to generate the SNNs by decomposing a complex DNN as a pre-processing step, and based on received information about the available edge devices.

This is done as a two step approach. In the first step, Edge^{n}AI partitions the classes of the complex DNN across the available edge devices[2]. For a complex DNN with $|\mathcal{C}|$ classes and $n+1$ edge devices (for n SNNs and 1 aggregator), each SNN is assigned a subset of the classes $\mathcal{S}$ where $|\mathcal{S}|= \frac{|\mathcal{C}|}{n}$. Each class $c \in \mathcal{C}$ is assigned to only one SNN such that the assigned classes to the SNN cover all the classes in the complex DNN. Once the classes are assigned to the SNNs, in the second step, each SNN actually gets generated by applying 'class-aware' pruning to the original DNN. For class-aware pruning we use the approach given in the recent work [6]. This algorithm prunes an already-trained DNN to keep only a subset of the output nodes, without the need for retraining. It reports negligible loss in accuracy and promising model size reduction due to pruning. Class-aware pruning in [6] takes advantage of the correlation between neurons and output classes which is obtained by measuring class-specific firing rates of neurons. The details can be found in [6] and won't be discussed here due to lack of space. Instead, we focus on how class partitioning (step 1) is done.

First we motivate the need for an efficient procedure for class-aware partitioning by explaining how an optimal approach based on exhaustive search will be impractical.

Consider an *ideal* scenario where each of the $\binom{|\mathcal{C}|}{|\mathcal{S}|}$ possible partitions is evaluated and the best one is selected. To evaluate a partitioning candidate in this ideal case, first the corresponding distributed implementation must be generated. This means the SNNs must first be generated by class-aware pruning, and then Algorithm 1 must be called for each SNN to find the threshold for its normalized entropy. The aggregator then must be trained. Finally, classification accuracy corresponding to that partitioning candidate must be measured by performing inference for that implementation over a testing set. This ideal approach is time-consuming which makes it impractical. In fact, the most time-consuming step and runtime bottleneck in the above exhaustive search is the class-aware pruning step.

[2]In this work we assume the edge devices have the same amount of computation and memory resources. We note though that it is possible to expand Edge^{n}AI to handle non-uniform edge devices.

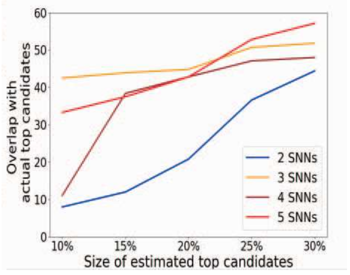


Fig. 3: Percentage overlap among actual top partitioning candidates and the ones estimated to be top candidates.

To efficiently evaluate a partitioning candidate, we propose a scheme to efficiently *estimate* the normalized entropy (NE) of each SNN. This estimation is *only* based on the assigned classes corresponding to a candidate. Once NE is estimated for an SNN, the threshold is found using Algorithm 1, and the corresponding classification accuracy is measured for that candidate, as explained in the ideal case. These steps together only require a limited number of inferences and are performed efficiently. After estimation of the NEs, we identify the top X% candidates with highest estimated classification accuracy where X is a small value (5% in our experiment). These top, few candidates are then accurately evaluated (by measuring the classifying accuracy with the actual not-estimated NEs), and the one with highest classification accuracy is selected.

To *estimate* the NE for a partitioning candidate, we use the following procedure. The partitioning candidate is specified by a subset of classes $\mathcal{S} \in \mathcal{C}$ as input to the procedure. First, we generate $|\mathcal{C}|$ pruned variations of the DNN. Each variation is optimized to accurately classify one of the output classes of the DNN. This step is done using class-aware pruning of [6] and is efficient because only $|\mathcal{C}|$ number of pruning calls are made to the DNN, as opposed to $\binom{|\mathcal{C}|}{|\mathcal{S}|}$. these single-class *pruned* variations are referred to as *PNN*s.

Next, for a given input, the normalized entropy is estimated by looking at those PNNs which are optimized for the output classes that belong to $\mathcal{S}$. These are the PNNs which are pruned to each classify only one of the classes in $\mathcal{S}$ accurately. Next, we apply the input to each PNN independently and measure their corresponding output probabilities. The normalized entropy is then found using Equation (1) where class c is only changed within the classes of $\mathcal{S}$ and the probability x_c is the output probability generated by each PNN.

The above procedure estimates an unknown SNN using only its known set of classes $\mathcal{S}$ and as union of $|\mathcal{S}|$ independent PNNs. It estimates the NE using the probabilities generated by these PNNs.

Figure 3 shows the effectiveness of estimating the NEs in identifying the top candidates. The Y-axis shows the percentage overlap between the actual top candidates and the ones found using the proposed estimation scheme. The X-axis is the number of top candidates to be identified (i.e., X%). The results are shown when implementing a variant of VGG-16 using different number of SNNs, ranging from 2 to 5. We observe a significant overlap between the two sets. For example for X=30%, the overlap is 50% for 3 SNNs.

IV. Results and Discussion

We experimented with different variants of VGG-16 and ResNet-152 networks to evaluate its distributed implementation using Edge^{n}AI. VGG-16 consists of 13 convolutional layers and 3 fully connected layers, with each layer being followed by a ReLU function.

TABLE I: Average model size of the base models after class-aware pruning when varying the number of output classes.

Number of Classes	VGG-16 model size (MB)	ResNet-152 model size (MB)
10 classes	82	62
20 classes	98	80
50 classes	131	86
70 classes	138	98
100 classes	164	104

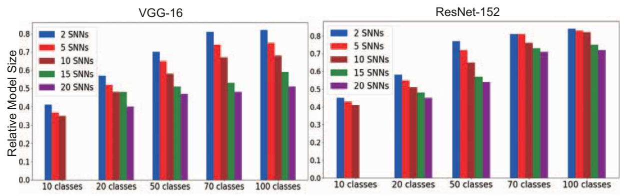


Fig. 4: Model size of the largest SNN relative to different base models (shown in the X axis) when varying the number of generated SNNs.

ResNet-152 includes 151 convolutional layers followed by a single fully connected layer. The networks are implemented in TensorFlow and trained and tested on the ImageNet (2012) dataset.

Base Model: We generated different variants of the networks and used them as different base models to implement using Edge^{n}AI. We experimented when the number of output classes are varied to $|\mathcal{C}| = 10, 20, 50, 70, 100$. For each number we randomly selected a subset of output classes from the original networks (and repeated this process 10 times). For each subset of selected output classes we then used the class-aware pruning scheme in [6] and pruned the network to obtain a base model. Overall, we experimented with 50 base models. Table I summarizes the relative model size of the pruned base models when varying the number of output classes for these networks.

A. Comparison of Classification Accuracy and Model Size

We evaluated the effectiveness of Edge^{n}AI in terms of per-device model size reduction and classification accuracy. We used the base models described before ($|\mathcal{C}| = 10, 20, 50, 70, 100$) and implemented them with Edge^{n}AI when the number of SNNs are $n = 2, 5, 10, 15, 20$. We set the following parameters in Algorithm 1: T_{start} = 0.2, T_{end} = 0.6, and $step$ = 0.05.

For each variation of base model and number of edge devices, we report the size of the largest SNN relative to the size of the base model, along with the top-1 classification accuracy.

Model Size. Figure 4 shows the relative model size of SNNs for different base models across $n+1$ devices for these two networks. Model size reflects the largest SNN among all generated SNNs and is measured by the number of its unique parameters including number of weights and biases. It is then normalized to the number of parameters in each base model. Finally, for each value of n and $|\mathcal{C}|$, the reported model size is averaged across the 10 base models with $|\mathcal{C}|$ classes.

As the results show, Edge^{n}AI can significantly reduce the size of each SNN, for all combinations of number of devices and classes. Furthermore, for a fixed value of $|\mathcal{C}|$, the size of the largest SNN reduces as we increase the number of devices. This is because with a higher number of devices, fewer classes will be assigned to each SNN, leading to more aggressive pruning. For example, in VGG-16, for a base model with 20 classes implemented on SNNs ranging from 2 to 20, the relative model size is reduced in the range of 0.57 to 0.40, respectively. These numbers change to 0.82 and 0.48 of the base model respectively, if the number of classes is increased to 100.

Classification Accuracy. Next, Table II reports the top-1 accuracy for different base models and their distributed implementation with Edge^{n}AI across different SNNs. As the results show, top-1 accuracy is maintained for all combinations of number of SNNs and base models. As an example, for VGG-16, for a base model with 100 output classes, accuracy degradation remains less than 0.5% (0.1%) once the network is partitioned into 2 (20) SNNs. For ResNet-152 with 100 output classes, accuracy degradation ranges from 0.2% to 1.1% for 2 to 20 SNNs. Also, as can be seen, changing the number of SNNs affects the top-1 accuracy of the networks given that both the aggregator design and number of classes assigned per SSN vary.

TABLE II: Comparison of top-1 accuracy of different base models implemented by Edge^{n}AI as the number of SNNs is varied.

Base Model	Base Accuracy	2 SNNs	5 SNNs	10 SNNs	15 SNNs	20 SNNs
VGG-16						
10 classes	88.3	87.8	86.7	86.2	–	–
20 classes	87.3	86.7	87.1	86.3	86.1	85.8
50 classes	85.1	84.9	84.7	83.1	83.3	83.5
70 classes	84.1	84.3	83.9	83.3	83.1	83.2
100 classes	82.4	81.9	81.6	82.1	82.2	82.3
ResNet-152						
10 classes	88.6	87.5	87.1	86.7	–	–
20 classes	87.9	87.2	86.8	86.4	86.1	86.3
50 classes	87.1	87.3	86.5	86.3	86.8	86.1
70 classes	85.6	85.1	85.3	84.8	84.3	84.6
100 classes	84.3	84.1	83.8	83.5	83.4	83.2

B. Comparison of Latency on Different Hardware Platforms

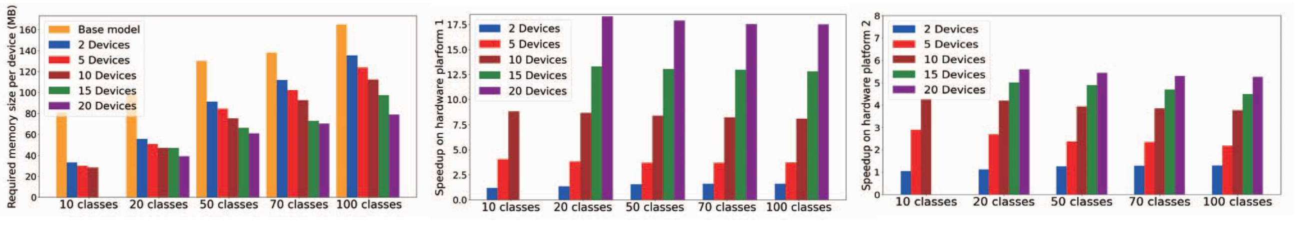
Here we evaluate the latency to perform one distributed inference as generated by Edge^{n}AI and compare it against a recent work [15]. The work [15] distributes a complex DNN by partitioning the output neurons of each layer. The complex network is partitioned such that each device calculates only a subset of output neurons of a layer and then broadcasts them to all remaining devices.

Estimation of Latency: The latency to perform a distributed inference is calculated as sum of of 3 main components: (1) latency of the slowest SNN, (2) latency of the wireless communication network, and (3) latency of the aggregator.

To measure the latency of each SNN, we use the hardware model shown in Figure 1(b). Delays of computational units including MAC, Pooling, ReLU are taken from [11], [13]. We also used NVSIM [5] to measure the latency of on-chip and off-chip memory accesses.

We then constructed an analytical model based on the architecture shown in Figure 1(b) and the sizes of memories, and number of computation units. Specifically, we estimated the number of on-chip and off-chip memory accesses and number of MAC operations for a hardware platform. To execute a convolutional layer, we assume the input feature maps are processed in parallel and the output feature maps in serial, similar to [19]. For the communication network, we measure the latency by setting the communication bandwidth to 100 Megabits per second (Mbps) as in [15]. Lastly, we use the same analytical model to measure the latency of SNNs and the aggregator since they are both neural networks.

Using the above model, we report the latency to perform a distributed inference using Edge^{n}AI under two different hardware platforms (both of which have the same architecture shown in Figure 1(b)): (1) edge devices with at most 150 MB on-chip storage per device and no off-chip memory, and (2) microcontrollers with at most 500 KB on-chip storage and a shared off-chip storage of 1.6 GB. Furthermore, in both platforms, each device has at most 4608 MAC units (two-input multipliers followed by two-input adders). For each case we experimented when the number of edge devices to



(a) (b) (c)

Fig. 5: (a) Per-device required memory size for different variants of the base model (X axis); (b) Speedup to perform one distributed inference compared to [15] on hardware platform 1, and (c) Speedup on hardware platform 2.

implement the SNNs is varied from 2 to 20. (The total number of devices including the aggregator is 3 to 21.)

First, Figure 5(a) shows the per-device required memory size (MB) to run on either platforms. The figure is essentially memory requirement corresponding to Figure 4. As Figure 5(a) shows, the per-device required memory can be significantly reduced using Edge^{n}AI. For example, for a base model (variant of VGG-16 with 100 classes), the required storage is reduced from 160 MB to around 80 MB when implementing the base model across 20 parallel edge devices (SNNs).

The actual latencies for each hardware platform are reported as speedups compared to [15] in Figures 5(b) and (c). Each platform will result in different number of on-chip and off-chip memory accesses and hence will result in different latencies. These are shown in for different base models (X axis) and different number of SNNs.

As can be seen in Figure 5(b) and 5(c), Edge^{n}AI can achieve a distributed inference with significant speedup for both hardware platforms. The speedup will increase as we increase the number of devices. This is because in [15] the communication overhead grows exponentially with the increase in number of devices (since each device should broadcast its computed neurons to the remaining $n-1$ devices). Lastly, Edge^{n}AI achieves a greater speedup on platform 1 compared to platform 2 because edge devices with at most 150 MB eliminate the need for off-chip memory accesses. Overall, Edge^{n}AI achieves speedups up to 17X (for platform 1) and 5.5X (for platform 2) for distributed inference of the base model with 100 classes, using 20 edge devices to implement the SNNs.

Edge^{n}AI also improves latency in the case of ResNet-152. The results are not shown here due to lack of space. For this network, distributed inference achieves 9X speedup (for platform 1) and 7.5X speedup (for platform 2) with 100 classes on 20 devices.

Overheads. In terms of the overhead to initially generate the SNNs, the run-time of Edge^{n}AI is greatly reduced using our technique to estimate the normalized entropy. Specifically, estimation of normalized entropy reduces the run-time of one inference—a step called many times during SNN generation—to 0.23 of when there is no estimation. Moreover, generation of SNNs does not incur a significant overhead since class-aware pruning does not involve any retraining step and relies on a set of efficient heuristics. Lastly, the overhead of training the aggregator is minimal (less than 10 epochs in our experiments), owing to the simple architecture of the network.

V. Conclusions

In this work, we propose Edge^{n}AI, a novel framework that enables partitioning and implementing a complex DNN across multiple edge devices with minimal latency overhead. We showed Edge^{n}AI achieves up to 17X speed up when distributing a variant of VGG-16 over 20 edge devices, without much loss in accuracy.

VI. Acknowledgments

This research was supported by Award No. 2006394 from National Science Foundation.

References

[1] “Deep learning market report, http://www.grandviewresearch.com/industry-analysis/deeplearning-market.”

[2] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, “Deepdriving: Learning affordance for direct perception in autonomous driving,” in *ICCV*, 2015, pp. 2722–2730.

[3] J. Chio, Z. Hakimi, P. Shin, W. Sampson, and V. Narayanan, “Context-aware convolutional neural network over distributed system in collaborative computing,” in *DAC*, 2019, pp. 1–6.

[4] C. Chow, “On optimum recognition error and reject tradeoff,” *IEEE Trans. on information theory*, vol. 16, no. 1, pp. 41–46, 1970.

[5] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, “NVSIM: A circuit-level performance, energy, and area model for emerging nonvolatile memory,” *IEEE TCAD*, vol. 31, no. 7, pp. 994–1007, 2012.

[6] M. Hemmat, J. San Miguel, and A. Davoodi, “CAP’NN: Class-aware personalized neural network inference,” in *DAC*, 2020, pp. 1–6.

[7] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[8] T. C. Landgrebe, D. M. Tax, P. Paclík, and R. P. Duin, “The interaction between classification and reject performance for distance-based reject-option classifiers,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 908–917, 2006.

[9] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, “MeDNN: A distributed mobile system with enhanced partition and deployment for large-scale DNNs,” in *ICCAD*, 2017, pp. 751–756.

[10] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, “Deep learning for healthcare: review, opportunities and challenges,” *Briefings in Bioinformatics*, vol. 19, no. 9, pp. 1236–1246, 2018.

[11] M. Nazemi, G. Pasandi, and M. Pedram, “Energy-efficient, low-latency realization of neural networks through Boolean logic minimization,” in *ASPDAC*, 2019, pp. 274 – 279.

[12] D. Saguil and A. Azim, “A layer-partitioning approach for faster execution of neural network-based embedded applications in edge networks,” *IEEE Access*, vol. 8, pp. 59 456–59 469, 2020.

[13] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, “ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *ISCA*, 2016, pp. 14–26.

[14] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[15] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, “Fully distributed deep learning inference on resource-constrained edge devices,” in *International Conference on Embedded Computer Systems*, 2019, pp. 77–90.

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *CVPR*, 2015, pp. 1–9.

[17] S. Teerapittayanon, B. McDanel, and H. T. Kung, “Distributed deep neural networks over the cloud, the edge and end devices,” in *ICDCS*, 2017, pp. 328–339.

[18] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer, “SqueezeDet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving,” in *CVPR*, 2017, pp. 446–454.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *FPGA*, 2015, pp. 161–170.