Split DNN Inference for Exploiting Near-Edge Accelerators

Hao Liu[1], Mohammed E. Fouda[2], Ahmed M. Eltawil[1] and Suhaib A. Fahmy[1]
[1] CEMSE Division, King Abdullah University of Science and Technology, Thuwal 23955, Saudi Arabia
[2]Rain Neuromorphics, Inc., San Francisco, CA, 94110, USA
Email: {hao.liu, ahmed.eltawil, suhaib.fahmy}@kaust.edu.sa, foudam@uci.edu

***Abstract*—The deployment of increasingly complex deep learning models for inference in real world settings requires dealing with the constrained computational capabilities of edge devices. Splitting inference between edge and cloud has been proposed to overcome these limitations, but entails significant communication latency. Newer edge accelerator devices can be distributed throughout layers of the network, supporting fine-grained offload. We propose a method for splitting a deep neural network (DNN) across the edge, near-edge accelerator, and cloud to exploit the combined computing capabilities of such devices while minimizing transmission bandwidth and, hence, energy. We formulate an approach to find near-optimal two-split configurations to optimize inference energy and latency. We thoroughly evaluate our approach on the VGG16 and ResNet50 models using the CIFAR-100 and ImageNet datasets to demonstrate that our method can navigate the trade-off space effectively.**

***Index Terms*—Deep learning inference, hardware acceleration, edge computing.**

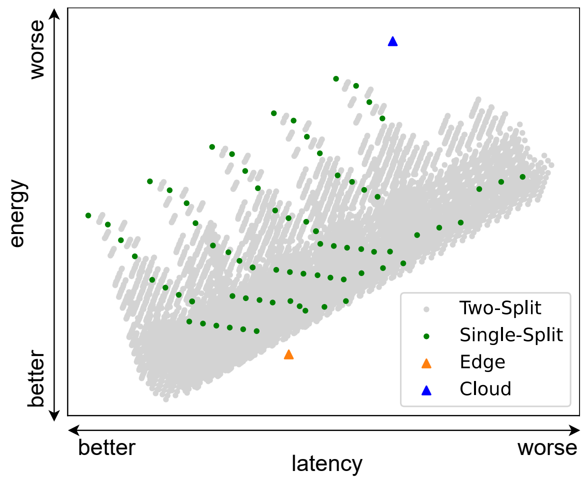


Fig. 1: Comparison of single-split execution of ResNet50 on the ImageNet dataset (green) and two-split configurations (grey).

I. Introduction

The Internet of Things (IoT) sees the wide proliferation of connected devices producing streams of data that can be used to improve human experience in many areas [1], [2]. A key enabler of these applications is machine learning (ML) using deep neural networks (DNNs) to process data and extract meaningful information. With the move to highly connected "cognitive cities", this information will be processed to determine real-time responses [3]. A key computational challenge in this regard is balancing the constrained computation capability of edge devices with these complex workloads, while considering the communication and latency cost of offloading to compute infrastructure such as the cloud. Offloading computation from the edge to the cloud can incur significant communication latency despite the improvement in computation latency. Research has been conducted on split computing for DNN inference, where DNN inference is split across the edge and the cloud. This allows inference of complex models to be completed with lower latency. Simple DNNs can optimized to be computed locally at the edge within the constraints of such hardware platforms.

With the advent of new generation edge hardware, such as Jetson Orin Nano and Jetson AGX Orin, computing capacity for DNN inference is significantly improved, and consequently energy efficiency. These devices can be deployed at the edge or near the edge to support offload from edge devices. They differ from traditional "cloudlet" ideas in that they are not full-featured servers further up the network, but rather can be deployed within a few hops of the edge, potentially integrated into infrastructure such as mobile base-stations [4]. In such a scenario, the traditional edge-cloud split can be extended to multiple splits involving the edge, near-edge accelerator(s), and the cloud. This can offer improved latency and energy consumption, but the varying compute capabilities of these devices must be suitably exploited [5]. The design space for splitting DNNs across multiple resources in this manner is large, and optimizing latency and energy is challenging.

In Figure 1 we show a comparison of energy and latency (based on the detailed profiling used in this paper) between edge execution (orange), cloud execution (blue), split execution across the edge and cloud (green), and the full design space for two splits (grey) across the edge, near-edge accelerator, and cloud for ResNet50 inference on the ImageNet dataset. These points do not consider accuracy degradation resulting from compression of the communicated data. Both latency and energy here include the communication and computation. Hence, while the cloud computation latency may be significantly faster than edge computation, the offload latency makes the overall execution latency longer. We see that a single split with various configurations has the potential to effectively reduce latency compared to individual edge and cloud execution. However, the addition of a second split and additional execution on the near-edge accelerator offers further advantages in latency and energy while also resulting in a much larger search space to explore. It is clear that a suitable search strategy is required to find profitable DNN splits for this scenario.

Naively splitting a DNN, without any architectural modification, can lead to high communication latency, since the intermediate activations can be large [6]–[9]. This can be addressed by inserting autoencoders at each split to finely control the data size (and hence communication latency) without significant accuracy loss [10]–[12]. Existing work has only explored single splits for edge-cloud execution with traditional under-powered edge hardware. We explore two split strategies exploiting modern hardware in a near-edge deployment, showing how the significantly larger configuration space can be tamed, while also running a large number of experiments to verify our approach. Our framework identifies near-optimal split strategies near the Pareto front under a given accuracy loss threshold without requiring the training of DNN models for all potential splits and autoencoder configurations. Our contributions are:

- General formulation of split DNN execution considering autoencoder overhead as well as latency and energy of communication and computation.
- Proposal of a two-split partitioning strategy that identifies splits near the Pareto front that satisfy an accuracy loss threshold.
- Experimental validation against the full search space for VGG16 and ResNet50 with the CIFAR-100 dataset and demonstration of a Pareto front trade-off for ImageNet on the same DNN models.

II. Related Work

A variety of techniques have been proposed for splitting DNN inference across devices. The first category of work splits the DNN without modifying its architecture, thereby preserving its accuracy. However, the cost of communication can be high since intermediate layers in a DNN can have large feature vectors. Kang *et al.* [6] establish a performance prediction model based on offline profiling data for different DNN layer types on the edge and cloud. At runtime, it is used to predict the latency and energy of partitions through exhaustive search. Li *et al.* [13] quantize data at the split between edge and cloud, selecting suitable candidate split positions through exhaustive search, considering the results of offline performance profiling. Luger *et al.* [9] propose a cost-aware splitting strategy which co-optimizes latency (using frequency as a proxy) and financial cost. These approaches only consider a single split.

Eshratifar *et al.* [7] formulate the execution of split DNNs into a Directed acyclic graph (DAG), with vertices representing the execution of layers on the edge or cloud and the length of edges representing execution cost. The optimal split position is obtained by finding the shortest path using integer linear programming. Wang *et al.* [8] formulate the multi-split problem as a DAG across the edge and cloudlets, with the optimal split positions decided by min-cost graph search. All the above approaches, without architecture modification, severely limit feasible splits due to the high bandwidth requirements at each split, increasing communication latency.

The second category of work relies on modifications to the DNN architecture to support enhanced splitting. In one class of work, the DNN is split into a *head* and a *tail*. The *head* is replaced by a smaller DNN with smaller output feature map resulting in lower computation and communication requirements when executed on the edge. Matsubara *et al.* [14] identify natural split positions, where the output is much smaller than the input. The *head* is used as a teacher model to train a small student model by knowledge distillation (KD). An artificial bottleneck is inserted in an early layer in the student model to further decrease communication cost. In their follow up work [15], they select the split position manually and design an autoencoder. However, the results are highly model-dependent and this approach only works for a single split.

Inserting an autoencoder at a split compresses intermediate feature maps sent between devices. The encoder-decoder pair can dramatically reduce the amount of data to be transmitted between devices at the split. At runtime, the encoder layer is executed on the device before the split and the the decoder is executed on the device after the split. By retraining the network with these autoencoders, it is possible to reduce communication overhead with minimal impact on network accuracy. Eshratifar *et al.* [12] place different configurations of autoencoders at all possible split positions and profile the latency and energy consumption of each combination, selecting the optimal split through exhaustive search. Shao *et al.* [11] insert autoencoders after each convolutional or pooling layer and select the split position that achieves lower communication cost than the input images. Jankowski *et al.* [16] insert autoencoders after pooling layers and prune the head model to reduce computing redundancy. Yao *et al.* [17] propose using asymmetric autoencoders to reduce communication cost. These works demonstrate the feasibility of modified DNN approaches to split inference, but do not scale to multiple splits due to their exhaustive search approaches.

Another possible approach to determine split positions is neural architecture search (NAS). A supernetwork including the possible split encoders is trained and at runtime, the sub-networks are evaluated and the optimal solution chosen for deployment. Dong *et al.* [10] jointly search the optimal DNN architectures and single optimal split position with an autoencoder. Tian *et al.* [18] propose a NAS-based multi-split method to co-optimize the structure of a deployed model and offloading strategy. However, NAS cannot be applied to a pre-trained DNNs and is highly time-consuming.

Our work extends previous work by applying modified autoencoder compression with fine-tuning to a more general two-split formulation. We demonstrate that this offers enhanced latency and energy compared to a single split for modern edge devices.

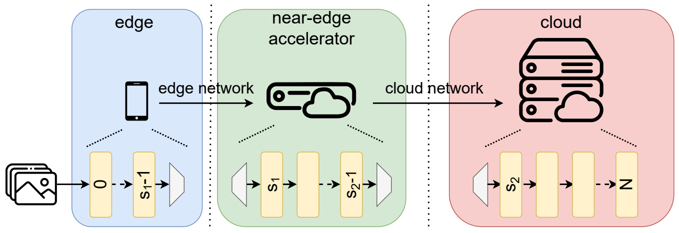


Fig. 2: System architecture comprising edge device, near-edge accelerator, and cloud. DNN layers are split across these devices with autoencoders inserted at each split to compress communication.

III. Proposed Method

A. Overview

The system configuration we consider in our work comprises an edge device, a near-edge accelerator device, and the cloud, as shown in Fig. 2. The near-edge accelerator is deployed some level into the network, on the path to the datacenter. For a wireless setup, this is assumed to be within the wireless basestation, while for a wired setup, this could be one or two hops into the wired network. We use the former setup in this paper. The model can extend to a general number of near-edge accelerator layers, but we present it here for a single accelerator. Shared inference requires a DNN to be split in two positions for this configuration, resulting in three portions to be executed on the three levels of the network. Input data is assumed to be generated at the edge device, e.g. from an attached camera. The first portion of the DNN is executed on the edge device with intermediate feature maps sent over a wireless network to the near-edge accelerator where the second portion of DNN inference is executed. The feature maps output by this second portion are transmitted over a high bandwidth network to the cloud where the remaining portion of the DNN is executed for the final inference result.

Each split requires the addition of an autoencoder into the DNN to control the size of the transmitted feature maps. The encoder and decoder layers are executed on the source and sink of each connection, respectively. Selection of autoencoder parameters and positions affects DNN inference accuracy. The aim of our work is to determine the optimal split positions and autoencoder configurations for a given accuracy loss threshold to provide a trade-off between latency and energy for multiple splits.

B. Problem Formulation

We consider a system comprising an edge device connected to a near-edge accelerator device, which is then connected to the cloud. We wish to map a DNN comprising $N+1$ layers, and therefore N potential split positions. A split $s_i \in \{1..N\}$ can be implemented using an autoencoder $k_i \in K$, where K is the set of possible autoencoder configurations. In this setup $i \in \{1,2\}$ since there are only two splits. The latency model considers both computing latency and transmission latency.

1) Computation Latency Model: We profile the execution of each of the $N+1$ DNN layers (or portions where splits are feasible), as well as different autoencoders, K, on each of the different hardware devices: edge (E), near-edge accelerator(N), and cloud (C). From profiling, we calculate the execution latency of the nth layer or portion on the edge, near-edge, and cloud as $Texec_{n,E}$, $Texec_{n,N}$, and $Texec_{n,C}$, respectively.

The autoencoder at each split is comprised of an encoder and a decoder, executed on the source and sink device, respectively. For a given autoencoder configuration k, we profile the execution time of the encoders on the edge and accelerator as $Texec_{enc_k,E}$ and $Texec_{enc_k,N}$ respectively (since the cloud is never a source).

We also profile the execution time of the decoders on the accelerator and cloud as $Texec_{dec_k,N}$ and $Texec_{dec_k,C}$, respectively (since the edge is never a sink).

Therefore, the total execution latency of the first portion of DNN layers and the first split encoder on the edge is:

$$Texec_E = \sum_{i=0}^{s_1-1} Texec_{i,E} + Texec_{enc_{k1},E} \tag{1}$$

where s_1 is the chosen first split position and k_1 is the chosen autoencoder configuration for the first split.

The execution latency of the first split decoder, second portion of DNN layers, and the second split encoder on the near-edge accelerator is then:

$$Texec_N = Texec_{dec_{k1},N} + \sum_{i=s_1}^{s_2-1} Texec_{i,N} + Texec_{enc_{k2},N} \tag{2}$$

where s_2 is the chosen second split position and k_2 is the chosen autoencoder configuration for the second split.

Finally, the execution latency of the second split decoder and remaining DNN layers on the cloud is:

$$Texec_C = Texec_{dec_{k2},C} + \sum_{i=s_2}^{N} Texec_{i,C} \tag{3}$$

Hence, the total execution latency is:

$$Texec = Texec_E + Texec_N + Texec_C \tag{4}$$

2) Communication Latency Model: The output data size of autoencoder k_i is d_{k_i}. The edge network connects the edge device to the near-edge accelerator; this could be the nearest wireless base station. The cloud network connects that near-edge device to the cloud. The bandwidth of the edge network and cloud network are r_E and r_C respectively. rtt_E and rtt_C represent the round-trip time (RTT) of the edge network and cloud network respectively. We model communication latency as in [9] as follows:

$$Tcomm_E = \frac{d_{k_1}}{r_E} + rtt_E/2 \tag{5}$$

$$Tcomm_C = \frac{d_{k_2}}{r_C} + rtt_C/2 \tag{6}$$

Then total communication latency is:

$$Tcomm = Tcomm_E + Tcomm_C \tag{7}$$

Thus the total latency of the system is :

$$T_{total} = Texec + Tcomm \tag{8}$$

3) Energy Consumption Model: We also profile the energy consumption of the DNN layers and autoencoder configurations on the edge, accelerator, and cloud devices.

The execution energy of nth layer or portion on the edge, accelerator, and cloud are $Eexec_{n,E}$, $Eexec_{n,N}$, and $Eexec_{n,C}$ respectively. The execution energy of encoders on the edge and accelerator are $Eexec_{enc_k,E}$ and $Eexec_{enc_k,N}$, respectively. The execution energy of the decoders on accelerator and cloud are $Eexec_{dec_k,N}$, and $Eexec_{dec_k,C}$ respectively.

Hence, execution energy on the edge is:

$$Eexec_E = \sum_{i=0}^{s_1-1} Eexec_{i,E} + Eexec_{enc_{k1},E}, \tag{9}$$

the execution energy on the near-edge accelerator is:

$$Eexec_N = Eexec_{dec_{k1},N} + \sum_{i=s_1}^{s_2-1} Eexec_{i,N} + Eexec_{enc_{k2},N}, \tag{10}$$

and execution energy on the cloud is:

$$Eexec_C = Eexec_{dec_{k2},C} + \sum_{i=s_2}^{N} Eexec_{i,C}. \tag{11}$$

Hence, the total execution energy is:

$$Eexec = Eexec_E + Eexec_N + Eexec_C \tag{12}$$

We define the communication power of the edge and cloud networks, based on published work, as q_E and q_C, respectively as in Table I. Hence, the energy consumed for communication on the edge network and cloud networks is:

$$Ecomm_E = \frac{d_{k_1}}{r_E} * q_E \tag{13}$$

$$Ecomm_C = \frac{d_{k_2}}{r_C} * q_C \tag{14}$$

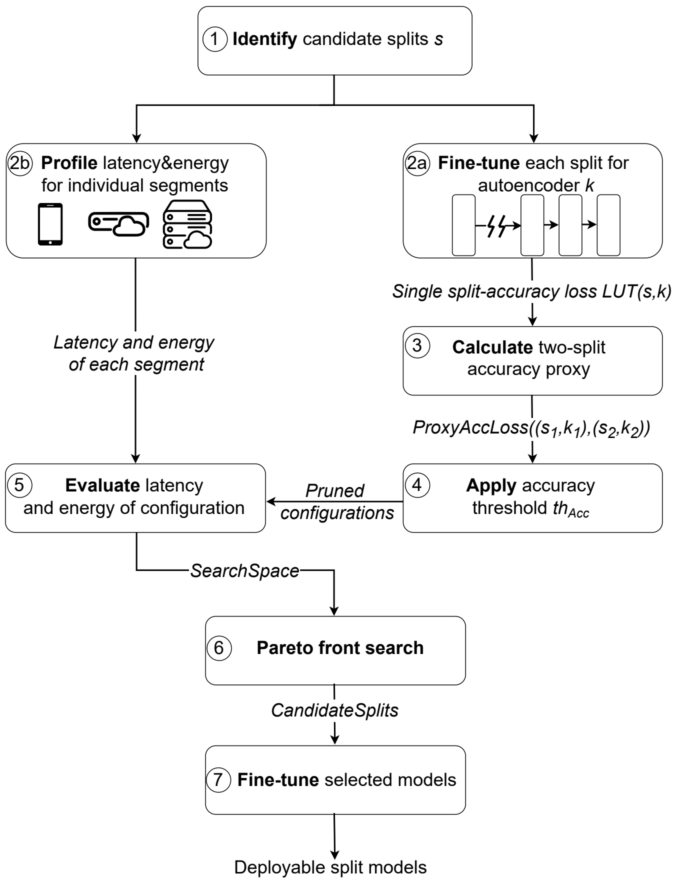
Hence, the total energy consumed for communication is:

$$Ecomm = Ecomm_E + Ecomm_C \tag{15}$$

The total energy consumption:

$$E_{total} = Eexec + Ecomm \tag{16}$$

It is worth mentioning that for the reference edge and cloud points, we execute the model entirely on the respective devices with no autoencoders inserted, while considering the communication latency as required. We assume the cloud as the sink for the DNN output though this can be modified in the model.



1 Identify candidate splits s
2b Profile latency&energy for individual segments
2a Fine-tune each split for autoencoder k
Single split-accuracy loss LUT(s,k)
Latency and energy of each segment
3 Calculate two-split accuracy proxy
ProxyAccLoss((s1,k1),(s2,k2))
5 Evaluate latency and energy of configuration
Pruned configurations
4 Apply accuracy threshold thAcc
SearchSpace
6 Pareto front search
CandidateSplits
7 Fine-tune selected models
Deployable split models

Fig. 3: The design flow for deploying a split DNN model using our proposed method.

C. DNN Splitting Strategy

We established in Figure 1 that inserting two splits and exploiting the cloud and near-edge accelerator could offer reduced latency and energy compared to edge-only, cloud-only, or edge-cloud execution. However, with N possible split positions and K possible autoencoder configurations per split, the space of possible configurations becomes $\frac{1}{2}N(N-1)K^2$. Each of these configurations results in a potential accuracy degradation, even after fine-tuning, which may or may not satisfy application requirements. This can only be determined after fine-tuning, which can be time consuming. Hence, we require a more efficient method to identify suitable candidate configurations for the trade-off, which can then be fine-tuned.

Our proposed method is shown in Fig. 3. First we determine feasible split positions in the DNN (①). There will generally be less than N feasible splits in an $N+1$-layer DNN as these should be layers which naturally have reduced feature size communicated to following layers. For example VGG16 splits are feasible after a convolutional layer or convolutional-pooling pair. Additionally for ResNet50, they can be inserted after a bottleneck block [19]. Hence for VGG16, which has 16 layers, we have $N = 12$ candidate split positions. For ResNet50, which has 50 layers, we have $N = 17$ candidate

Algorithm 1 Two-split candidate determination algorithm

```
    Input: potential split positions $\{1..N\}$, autoencoder set $K$, single split-accuracy loss LUT $Loss(s,k)$, maximum accuracy loss threshold $th_{Acc}$, latency and energy model $Eval$
    Output: candidate set of splits for fine-tuning $CandidateSplits$
 1: Initialize: $SearchSpace = \varnothing$
 2: for each $(s_1, k_1) \in \{\{1..N\}, K\}$ do
 3:     for each $(s_2, k_2) \in \{\{s_1 + 1..N\}, K\}$ do
 4:         first split loss $l_1 \leftarrow Loss(s_1, k_1)$
 5:         second split loss $l_2 \leftarrow Loss(s_2, k_2)$
 6:         $ProxyAccLoss \leftarrow max(l_1, l_2) + \frac{l_1+l_2}{2}$
 7:         if $ProxyAccLoss \leq th_{Acc}$ then
 8:             $T_{total}, E_{total} \leftarrow Eval((s_1, k_1), (s_2, k_2))$
 9:             add $((s_1, k_1), (s_2, k_2), T_{total}, E_{total})$ to $SearchSpace$
10:         end if
11:     end for
12: end for
13: $CandidateSplits \leftarrow ParetoFront(SearchSpace)$
14: return $CandidateSplits$
```

split positions.

At each split, we insert an autoencoder, configured as in [11]. This comprises an encoder-decoder pair. The encoder is a convolutional layer, a batch-norm layer, and a Sigmoid function, while the decoder is a convolutional layer, a batch-norm layer, and a ReLU function. The number of kernels applied in the convolutional layers in the autoencoder is configurable, allowing control over the size of features emitted by the encoder (affecting communication latency and energy). This results in an accuracy loss, which is mitigated to some extent by fine-tuning after inserting the split. Clearly, fine-tuning a large number of split DNN models for all potential split combinations and autoencoder configurations is infeasible.

Hence, we start by inserting different configurations of a single autoencoder at each possible single split position, then fine-tune these configurations to form a look up table (LUT) of single split-accuracy loss, $Loss(s,k)$ ($\textcircled{2a}$). Each of these model segments, comprising a minimal layer section, along with its autoencoder pair (preceeding and following) is profiled for latency and energy ($\textcircled{2b}$).

The user specifies their acceptable accuracy loss threshold, th_{Acc}. The predicted accuracy loss for each possible combination of multiple splits and autoencoder configurations is computed as per Algorithm 1 (③) then compared to the threshold and points that do not meet the threshold are filtered out (④). The pruned points are then evaluated using our model to determine latency and energy (⑤). A pareto front search (⑥) among these points is then used to identify the most promising trade-off points in terms of latency and energy, which can then be fine-tuned (⑦) for deployment. We evaluate the effectiveness of our approach in Section IV.

TABLE I: Communication model parameters.

Network	Capacity (Gbps)	Power (W)	RTT (ms)
Edge (5G)	0.13 [22]	2.5 [22]	10 [23]
Cloud	10 [24]	5.5 [24]	60 [4]

TABLE II: Training settings for CIFAR-100 dataset

Model	Stage	Epochs	(lr*0.1) @ epoch
VGG16	first	120	80
	second	160	40,100,140
ResNet50	first	90	60
	second	80	40

Hence, for a new DNN, we are only required to initially fine-tune NK model configurations ($\textcircled{2a}$) after which feasible trade-off points close to the Pareto front are found to be further fine-tuned (⑦) as needed. These fine-tuned DNN models can then be deployed.

IV. Evaluation

To verify the effectiveness of our DNN splitting strategy, we evaluate it on VGG16 [20] and ResNet50 [19] pretrained DNN models on the CIFAR-100 dataset [21]. We fine-tune all possible combinations of splits and autoencoder configurations for the full search space then compare the points chosen by our method in terms of closeness to the true Pareto front. We then show how our method provides multiple good trade-off points for the same DNNs applied to ImageNet (which is much more time consuming to train).

A. Experimental Setting

All DNN models are trained and fine-tuned using Nvidia A100 SXM and V100 SXM2 GPUs using PyTorch version 1.10.2. The total GPU hours for training all configurations of VGG16 and ResNet50 for CIFAR-100 in this paper is over 30,000. The base accuracy of VGG16 and ResNet50 on the CIFAR-100 dataset is 74.70% and 78.75%, respectively. To fine-tune the split DNN models with the autoencoders, we use a two stage strategy as in [11], i.e. first train the autoencoders while fixing other DNN model parameters, then fine-tune the whole DNN model including autoencoders. We use the SGD optimizer, with initial learning rate 0.01, momentum 0.9, and weight decay 5e-4 for both first and second stages of training [25]. Other settings are shown in Table II.

We consider a setting where the edge device produces 30 frames per second of video to be processed by the DNN. The edge device is an Nvidia Jetson Orin Nano 8GB. The near-edge accelerator is an Nvidia Jetson AGX Orin 64GB. The cloud device is a datacenter class Nvidia A100 80GB SXM. We profile the latency and energy of each layer, encoder, and decoder on each of these devices. For CIFAR-100, we use a

TABLE III: Execution latency of base models.

		Latency (ms)				**Energy (mJ)**			
		Jetson Orin Nano	Jetson AGX Orin	A100	Raspberry Pi 4	Jetson Orin Nano	Jetson AGX Orin	A100	Raspberry Pi 4
VGG16	CIFAR-100	18.59	14.23	0.4	4591.7	63.1	85.9	324.1	22958.7
ResNet50		107.2	67.3	3.2	13830.0	389.8	464.7	1313.7	69164.7
VGG16	ImageNet	1015.9	520.3	18.5	—	3645.5	3816.1	7235.3	—
ResNet50		828.3	221.4	10.7	—	2810.3	1528.0	4266.9	—

batch size of 32, 128, 512 on the edge, near-edge accelerator, and cloud, respectively. Table III shows the execution latency and energy measured for the execution of the full base models on the different devices used in our experiments. We also profiled the Raspberry Pi 4, which showed poor energy efficiency due to the long execution times resulting from a lack of hardware acceleration.

For the communication model, we model the capacity and transmission power and RTT of the edge network (based on 5G) and the cloud network (based on high bandwidth fiber) as in Table I.

B. Accuracy Proxy Evaluation

A key requirement for such a large search space is to determine candidate points that meet a required accuracy loss threshold th_{Acc}. That is, a configuration of split positions and autoencoder configurations that achieves an accuracy within this threshold of the original DNN, while offering better latency and energy consumption. We propose a proxy for the combined accuracy loss for two splits, as shown in Algorithm 1. Having trained all possible configurations for our initial analysis, we can determine how well this proxy performs. Table IV shows how our proxy performs against the proportion of prunable points based on the real accuracy for three different accuracy loss thresholds, th_{Acc}. True positives (TP) are split strategies where the true accuracy loss exceeds th_{Acc} and which are successfully pruned. True negatives (TN) are strategies successfully retained as they have real accuracy loss less than th_{Acc}. We would like to maximise these two. False negatives (FN) are strategies where the real accuracy loss exceeds th_{Acc} but which are wrongly kept. These just increase the number of points to evaluate. False positives (FP) are strategies that are erroneously pruned, which, in reality, exhibit accuracy losses less than th_{Acc}. We would like to minimise these since they could be promising solutions. We see that this proxy offers a good approximation.

C. Split Position Search

In Figure 4 and 5 we show the resulting configurations identified by our search. Each figure shows three plots for different accuracy loss thresholds that might be requested by the user. Full execution on the edge and cloud are shown in orange and blue, respectively. The axes are normalised to execution on the edge. The grey points show the full set of possible configurations, each of which we have fine-tuned to determine the final accuracy. This is just for the purposes of evaluating our approach. Points that meet th_{Acc} are included in the plot. The green points represent the true Pareto front of points meeting the required accuracy, and could only be determined through exhaustive search. Our method finds the points shown in red, which are close to the green points. Some of these red points when fine-tuned do not in fact meet th_{Acc} (shown with a circle), while most do (shown with a star). For both DNN models, we see that our approach identifies candidate points that are close to the Pareto front, offering a trade-off between latency and energy, while meeting th_{Acc}. In many cases, these are the exact same configurations determined by full search. Hence, our approach significantly narrows down the number of configurations that needs to be trained compared to the brute force approach. As ResNet50 is around 4× as computationally demanding as VGG16 in our profiling experience, we see that the computational capability of the devices offers a smoother Pareto front of points between edge and cloud for that model.

For CIFAR-100, we were able to exhaustively train all configurations in order to determine the final accuracies to validate our approach. With a larger, more complex dataset like ImageNet [26], this was not feasible, even with a large number of GPUs. The base accuracy of VGG16 and ResNet50 for the ImageNet dataset is 73.36% and 80.35%, respectively. To fine-tune the ImageNet dataset, we train 20 epochs for both the first and second stages with a batch size of 64 on two Nvidia A100s. For both stages, the learning rate is scaled by 0.1 after 10 epochs. The learning rates for the first and second stages of VGG16 are 0.1 and 0.01, while for ResNet50, they are 0.01 and 0.001, respectively. We use the SGD optimizer with momentum, and weight decay 0.9 and 1e-4, respectively. Fine-tuning a configuration on ImageNet takes 20 times as long as for CIFAR-100, hence the need for a more efficient search method to identify suitable configurations. For profiling of ImageNet, we use a batch size of 1, 32, 128 on the edge, near-edge accelerator, and cloud, respectively.

In Figure 6, we show the candidate points identified by our approach for $th_{Acc} = 0.03$ for the ImageNet dataset for both VGG16 and ResNet50. The plot shows that our method successfully identifies candidate points that offer a trade-off between latency and accuracy, while significantly improving on cloud or edge execution, and without the need for ex-

TABLE IV: Evaluation of two-split accuracy proxy on CIFAR-100. Note this is the accuracy of the proxy prediction in pruning search space points, not the resulting model accuracy.

Model	**Filter**	$th_{Acc} = 0.01$				$th_{Acc} = 0.02$				$th_{Acc} = 0.03$			
		Pruned		Kept		Pruned		Kept		Pruned		Kept	
		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
VGG16	Real	56.36%		43.64%		35.43%		64.57%		22.11%		77.89%	
	Proxy	43.37%	7.05%	36.59%	12.99%	16.21%	2.97%	61.60%	19.22%	11.59%	1.90%	75.99%	10.52%
ResNet50	Real	38.58%		61.42%		19.02%		80.98%		11.30%		88.70%	
	Proxy	20.74%	1.30%	60.12%	17.84%	13.15%	0.84%	80.13%	5.88%	9.49%	1.68%	87.02%	1.81%

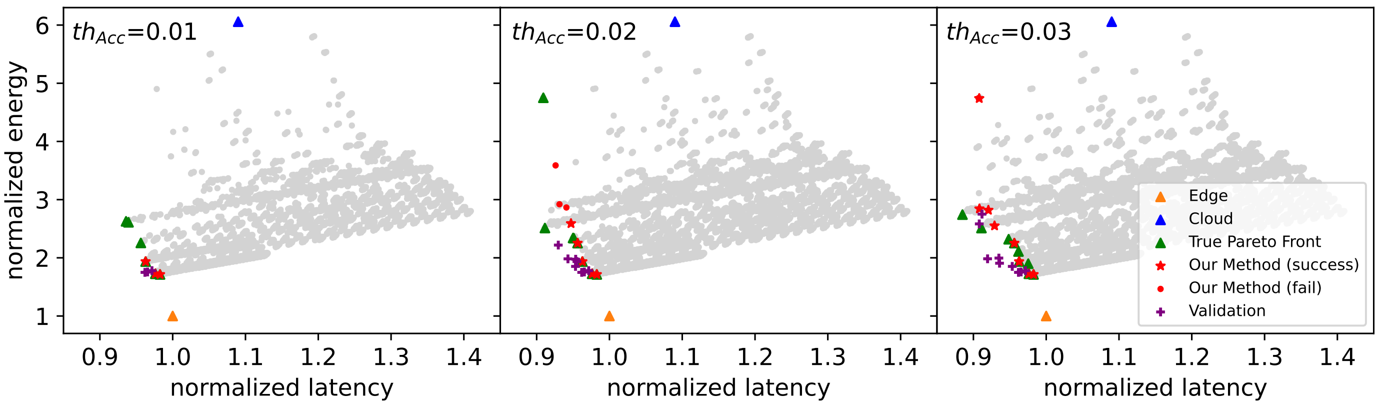


Fig. 4: Experimental results for VGG16 with the CIFAR-100 dataset.

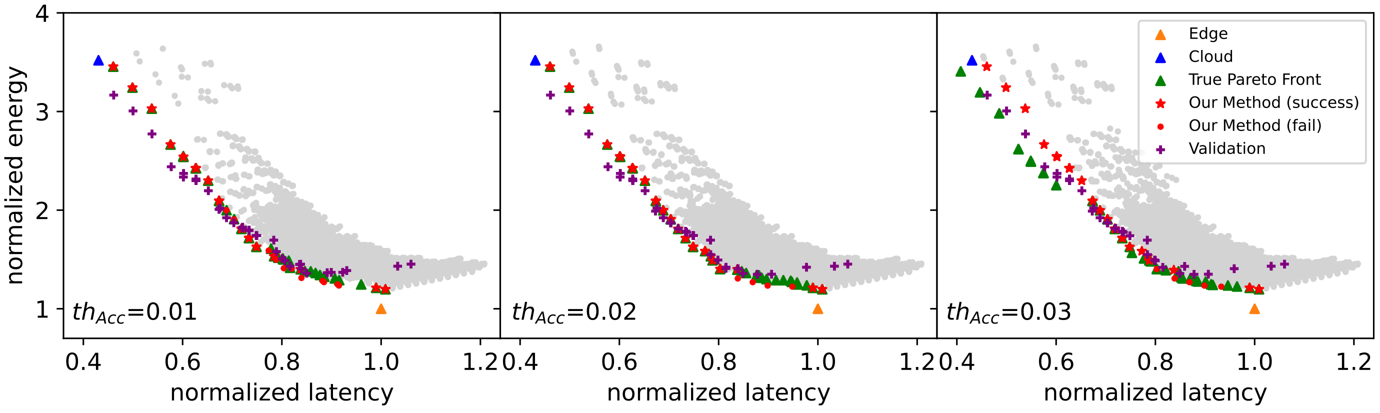


Fig. 5: Experimental results for ResNet50 with the CIFAR-100 dataset.

haustively evaluating multi-split configuration accuracies. The purple validation points show that the model is accurate while over-predicting energy in most cases for ResNet50. Our search approach consumes between 0.3 and 2.6 seconds to execute and identify the candidate points to be fine-tuned.

We also validated the energy and latency values for the Pareto-optimal split schemes by running those segments of the model on the different devices, with the resulting points shown in purple. These show good correspondence with the predicted latency maintaining monotonicity and energy also tracking reasonably. The difference between the modelled latency and energy and that measured in the final deployment was $\pm 1\%$ and $\pm 6\%$, respectively, both networks on the ImageNet dataset, and for ResNet50 on CIFAR-100. Since VGG16 on CIFAR-100 has much lower computational cost, errors were higher for that.

V. Conclusions and Future Work

In this paper, we have proposed an approach for split inference of DNNs across the edge, a near-edge accelerator, and the cloud, showing that this can offer a better trade-off of latency against energy while still meeting accuracy requirements. Our approach relies on some initial profiling, followed by a pruned search that identifies suitable positions

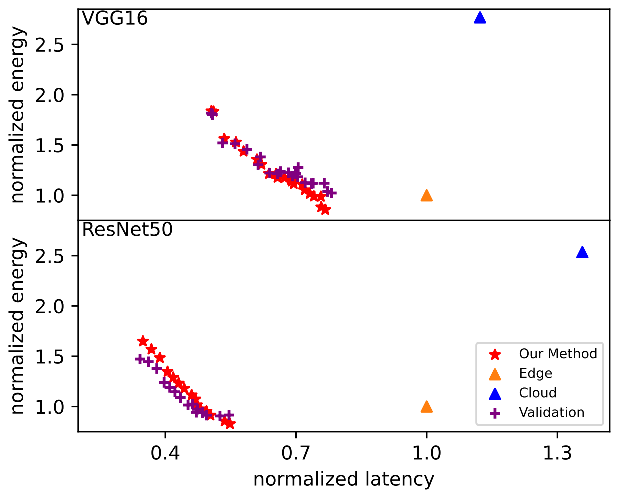


Fig. 6: Resulting splits on ImageNet with $th_{Acc} = 0.03$

for two autoencoder splits to be inserted into the DNN. The result is a much quicker determination of promising points that we show are close the the exhaustive Pareto front. We confirmed this with exhaustive experiments on VGG16 and ResNet50 on the CIFAR-100 dataset. We then showed that for the more complex ImageNet configurations of VGG16 and ResNet50, a suitable set of trade-off points could be arrived at. We finally confirmed that the split execution latency models match those from real experiments within a 1% margin.

We are now investigating scaling this method to an arbitrary number of splits, which requires an enhanced accuracy proxy to prune the search space further. We are also interested in applying this approach to more complex DNN models including Transformers. We believe new embedded ML accelerators will enable a much wider range of deployment scenarios requiring improved approaches for mapping DNN workloads to them and an increasing reliance on distributed inference.

VI. Acknowledgement

This work was supported by the King Abdullah University of Science and Technology through the Competitive Research Grant program under grant URF/1/4704-01-01.

References

[1] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[2] L. Wang, X. Ye, S. Wang, and P. Li, "Ulo: An underwater light-weight object detector for edge computing," *Machines*, vol. 10, no. 8, 2022.

[3] M. Finger and E. Portmann, "What are cognitive cities?" *Towards Cognitive Cities: Advances in Cognitive Computing and its Application to the Governance of Large Urban Systems*, 2016.

[4] A. Cartas, M. Kocour, A. Raman, I. Leontiadis, J. Luque, N. Sastry, J. Nuñez-Martinez, D. Perino, and C. Segura, "A reality check on inference at mobile networks edge," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*, 2019, pp. 54–59.

[5] R. A. Cooke and S. A. Fahmy, "A model for distributed in-network and near-edge computing with heterogeneous hardware," *Future Generation Computer Systems*, vol. 105, pp. 395–409, 2020.

[6] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, pp. 615–629, 2017.

[7] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, 2019.

[8] S. Wang, X. Zhang, H. Uchiyama, and H. Matsuda, "Hivemind: Towards cellular native machine learning model splitting," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 2, pp. 626–640, 2021.

[9] D. Luger, A. Aral, and I. Brandic, "Cost-aware neural network splitting and dynamic rescheduling for edge intelligence," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*, 2023, pp. 42–47.

[10] X. Dong, B. De Salvo, M. Li, C. Liu, Z. Qu, H.-T. Kung, and Z. Li, "Splitnets: Designing neural architectures for efficient distributed computing on head-mounted systems," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

[11] J. Shao and J. Zhang, "Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems," in *IEEE International Conference on Communications Workshops*, 2020.

[12] A. E. Eshratifar, A. Esmaili, and M. Pedram, "Bottlenet: A deep learning architecture for intelligent mobile cloud computing services," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2019.

[13] G. Li, L. Liu, X. Wang, X. Dong, P. Zhao, and X. Feng, "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *International Conference on Artificial Neural Networks*, 2018, pp. 402–411.

[14] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "Distilled split deep neural networks for edge-assisted real-time systems," in *Proceedings of the Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019, pp. 21–26.

[15] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, "Bottlefit: Learning compressed representations in deep neural networks for effective and efficient split computing," in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2022, pp. 337–346.

[16] M. Jankowski, D. Gündüz, and K. Mikolajczyk, "Joint device-edge inference over wireless links with pruning," in *IEEE International Workshop on Signal Processing Advances in Wireless Communications*, 2020.

[17] S. Yao, J. Li, D. Liu, T. Wang, S. Liu, H. Shao, and T. Abdelzaher, "Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency," in *Proceedings of the Conference on Embedded Networked Sensor Systems*, 2020, pp. 476–488.

[18] Y. Tian, Z. Zhang, Z. Yang, and Q. Yang, "Jmsnas: Joint model split and neural architecture search for learning over mobile edge networks," in *IEEE International Conference on Communications Workshops*, 2022, pp. 103–108.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[21] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," in *Citeseer, Tech. Rep.*, 2009.

[22] D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma, "Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 479–494.

[23] N. Alliance, "5g white paper," *Next generation mobile networks, white paper*, 2015.

[24] P. Bertoldi *et al.*, "Eu code of conduct on energy consumption of broadband equipment: Version 6," 2017.

[25] "pytorch-cifar100," https://github.com/weiaicunzai/pytorch-cifar100, 2020-2023.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "Imagenet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.