





# DNN Model Deployment on Distributed Edges

Eunho Cho<sup>(✉)</sup> , Juyeon Yoon, Daehyeon Baek, Dongman Lee,  
and Doo-Hwan Bae 

Korea Advanced Institute of Science and Technology (KAIST),  
Deajeon, Republic of Korea  
{ehcho,bae}@se.kaist.ac.kr, {juyeon.yoon,bdh0404,dlee}@kaist.ac.kr

**Abstract.** Deep learning-based visual analytic applications have drawn attention by suggesting fruitful combinations with Deep Neural Network (DNN) models and visual data sensors. Because of the high cost of DNN inference, most systems adopt offloading techniques utilizing a high-end cloud. However, tasks that require real-time streaming often suffer from the problem of an imbalanced pipeline due to the limited bandwidth and latency between camera sensors and the cloud. Several DNN slicing approaches show that effectively utilizing the edge computing paradigm effectively lowers the frame drop rate and overall latency, but recent research has primarily focused on building a general framework that only considers a few fixed settings. However, we observed that the optimal split strategy for DNN models can vary significantly based on application requirements. Hence, we focus on the characteristics and explainability of split points derived from various application goals. First, we propose a new simulation framework for flexible software-level configuration, including latency and bandwidth, using dockercompose, and we experiment on a 14-layered Convolutional Neural Network (CNN) model with diverse layer types. We report the results of the total process time and frame drop rate of 50 frames with three different configurations and further discuss recommendations for providing proper decision guidelines on split points, considering the target goals and properties of the CNN layers.

**Keywords:** DNN partitioning · Distributed edge · Edge computing

## 1 Introduction

Including visual analytics or augmented/virtual reality, there are various applications for utilizing the edge computing for machine learning. Researchers [2, 5] found that there are various benefits for the deployment of machine learning model on edge devices like reducing the time by offloading the processing.

Currently, researches for utilizing the distributed system on machine learning concentrates on orchestration or collaborative schemes among clients, edges, and

---

E. Cho and J. Yoon—These authors contributed equally.

© Springer Nature Switzerland AG 2022

M. Bakaev et al. (Eds.): ICWE 2021 Workshops, CCIS 1508, pp. 15–26, 2022.

[https://doi.org/10.1007/978-3-030-92231-3\\_2](https://doi.org/10.1007/978-3-030-92231-3_2)

clouds working to create a costly deep learning system. However, there are still many challenges to effectively utilizing ML/DL tasks on edges, such as managing latency and disconnected operation (achieving network transparency), handling heterogeneity of multiple edge devices, and deploying pipelines of (re)training or inference scenarios.

Various studies have suggested frameworks or models that utilizes existing production models and hardware architectures. Although there have been multiple conclusions drawn in research about the DNN deployment framework, such as the early-exit scheme of Distributed Deep Neural Network (DDNN) [5] and production-ready slices [2], there remain limitations that constrain the introduction of DNN slicing on edges in the real world. For example, DDNN requires specially tuned model architecture that cannot be quickly adapted to a realistic environment. Couper claims to generate acceptable quality of DNN slices using general DNN models, but there is space for much more improvement regarding performance. The main limitation of Couper is that we can tune the model and computation power of edges to fit with target application goals. Couper assumes a fixed specification of edge and cloud machines and does not consider the computational ability of the device. Moreover, both previous approaches prematurely generalize the requirements of diverse applications, which can vary significantly.

This paper analyzes the characteristics of split points with representative network configurations that can give us the necessary insight to design a proper DNN slicing scheme concentrating more on actual application requirements. To do so, we first implement a software-configurable DNN inference pipeline on a distributed environment, utilizing Docker and `docker-compose`. Based on the experiment with the simulator, we will provide an analysis of how to find better split points according to the various CNN layers concerning the network environment. Our main contribution is providing a way to determine a more optimized DNN slicing strategy by considering the characteristics of DNN layers and different potential combinations of computational power distribution and network configuration.

The paper is structured as follows. Section 2 describes existing works and the background of our work. The details of our simulation environment are outlined in Sect. 3. Next, the experiment setup and its results are presented in Sects. 4 and 5. Finally, we conclude the paper in Sect. 6.

## 2 Related Works

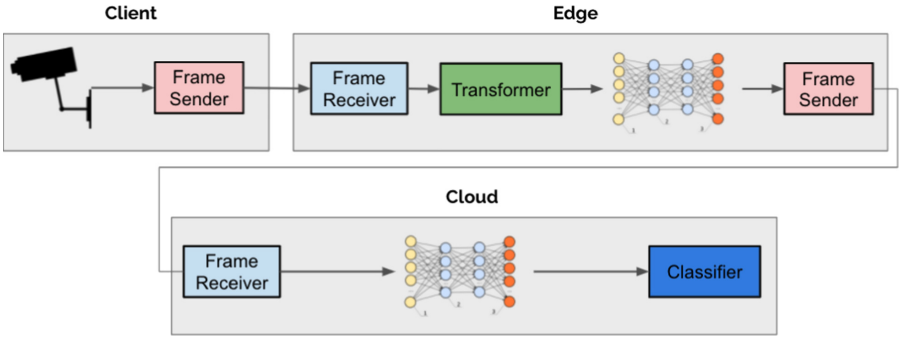
Edge cloud orchestrator (ECO) [6] supports federated learning, transfer learning, and staged model deployment by separating the burdens of training and inference. The proposed architecture leverages both edge and cloud by providing an abstraction of the control path. However, their approach still needs to deploy the whole DNN model on a single node, which can burden low-end edge devices in managing large and complex state-of-the-art CNN models.

Model light-weighting techniques presented in the research of Wang et al. [9] and Talagala et al. [6] include other directions that adapt to and specialize

in target models to fit with the given environment. By pruning an insignificant portion of the structural part or connection of the models, both the model's size and inference time can be significantly lowered.

However, model compression or pruning techniques have an explicit limitation of reduction to maintain accuracy as acceptable as the original. In contrast, the model partitioning approach that supports deploying each part of the model to the heterogeneous computing environment can benefit the deep learning system on edges and clouds. eSGD [7] supports collaborative training on a CNN model by synchronizing parameters with sparse gradient updating.

One study [1] used specialized, spatially distributed micro-architecture, specifically multiple FPGA chips, for DNN training and inference in a parallel manner.



**Fig. 1.** Structure of the Couper framework for image classification when utilizing edge-cloud environment. [2]

However, at present still, the most practical solution would be to keep the typical hardware architecture and better utilize the existing machines that are distributed geographically. Hsu et al. proposed a solution named *Couper*, which is a framework that creates slices of DNNs and deploys them on the docker-based edge pipelines. *Couper* analyzes the given DNN model and generates the partition for the distributed deployment. To do this, *Couper* supports the automated slicing and deployment on the edge-cloud environment. Figure 1 shows the *Couper* framework on image classification task. On three nodes, client, edge, and cloud, the *Couper* provides the DNN evaluator and two pairs of frame sender and receiver. Another approach by Teerapittayanon et al. [8] proposed distributed deep neural networks that measure the confidence of classification results during the intermediate layers and support an earlier exit of inference, so that more than 60% of the inference tasks could be prematurely completed on edge. In contrast, it still needs a specialized DNN architecture that supports the early exit scheme.

Similar to the architecture of *Couper*, several studies have evaluated the likelihood of the distributed DNN model's deployment on the modern network

environment. Ren et al. [4] showed the distributed DNN’s deployment on the mobile web with a 5G network based on the distributed DNN model’s deployment. This work also shows the adaptive mechanism to figure out the optimal point of partitioning the DNN model. However, this mechanism only depends on the network environment, and so it does not take into account the characteristics of each DNN layer. Lockhart et al. [3] present *Scission*, which automatically determines optimal partitions by layers while considering user-defined constraints about cloud-edge settings. However, it does not support pipelines with multiple edges and does not consider the effect of varying packet sizes and network congestion caused by the different partitioning points.

Although these studies have provided various frameworks for utilizing deep learning on edge machines, there was a lack of discussion on optimizing the framework. Especially for *Couper*, the performance is hardly dedicated to the model and its split point. Therefore, we try to determine the optimal split point based on the various features of the DNN layers and environment.

### 3 Simulation Implementation

In this section, we introduce our simulation environment. We plan to conduct experiments using an implemented pipeline DNN model simulator, inspired by Couper’s model slicing concept.

#### 3.1 Overall Architecture

The architecture uses the latest version of **Tensorflow**, and any model contains the structure of the **tf.Graph** (Tensorflow Graph) could be used as the target model to deploy. Each result tensor produced from the split layer is serialized with **ProtoBuffer** and **TensorProto** definition provided by a Tensorflow official implementation.

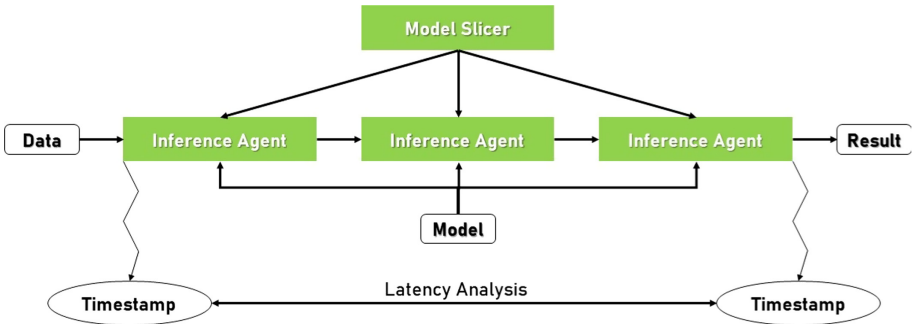


Fig. 2. Overall architecture of the simulation

Figure 2 shows the overall architecture of this implementation. There are two main components. The **Model Slicer** uses the split point as an input and sends the exact target split point to the inference agent. The **Inference Agent** has a full model and split the model by the split point from the model slicer. The inference agent represents a single edge, and each agent has information about the split DNN deployment’s pipeline structure. Each inference agent uses a docker with the same container images but a different split DNN model. Based on this architecture, we can analyze based on the frame drop rate probed from the results and conduct a latency analysis based on the total time for processing the specific amount of data.

### 3.2 Model Slicer

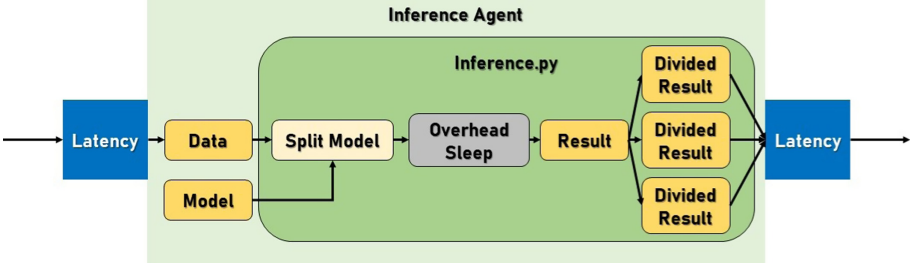
The model slicer initiates every inference agent based on the given split point and pipeline structure. The split point can be based on the operation level or the layer level. The operation level can produce more slices and a better set of slices that is more close to the true optimal. However, many models are complicated and hence contain many layers. In that case, layer-wise slicing can reduce the search time for the split points, and there is no need to discard invalid slices due to the possible branches (this is only present in the operation-level representation of DNN). In this research, we decided to split models by the level of layers. The given split point can consist of a single split point or multiple split points.

### 3.3 Inference Agent

Our inference pipeline consists of a set of sequential edges that collaborate to complete an inference task. Each split model except for the last one produces intermediate results of inference, and the result is serialized and transmitted to the next edge in the pipeline. Hence, any ‘central’ edge’s roles as both the gRPC client that requests the next inference to the neighbor and the gRPC server help complete the remaining inference task in response to the other client’s request. In the future, we aim to adopt more lightweight methods for serialization in terms of conversion cost and transmission size overhead. However, for this study, the built-in serialization method of Tensorflow was used. The inference agent splits the model when the model slicer requests it with the split point. In the current implementation, the agent splits the model using the Tensorflow method.

Figure 3 shows the detailed architecture of the inference agent. We considered three elements to develop a realistic inference. The first is the latency. We set an artificial latency by using Linux traffic control, ‘tc.’ The command was set to each docker container with modified latency. This latency applies to both inbound and outbound of each inference agent. Therefore, based on our environment, which is composed of three inferences—device, edge, and cloud—the cloud has high latency, while the others have low latency.

We also considered the overhead to show the hardware’s limitations. The realistic simulation needs to show better performance in the cloud and poor performance in the device. The overhead is implemented through the Python



**Fig. 3.** Architecture of the inference agent

sleep method. When the agent process the data, it measures the processing time. With the given overhead ratio, it calculates the overhead time based on the formula,  $(OverheadTime) = (OverheadRatio) * (ProcessTime)$ .

The last consideration was throughput, which roughly simulates the maximum network bandwidth and congestion status. We limit the maximum throughput and divide the tensor if the size exceeds the throughput. In that case, of course, the next inference agent collects it and reassembles it.

## 4 Experiment Setting

### 4.1 Research Questions

In this section, we outline our research questions to find valuable insights that explain the consequences of different split points and network configurations

- **RQ 1:** How do the different choices of split points in the CNN model layers influence the inference quality?
- **RQ 2:** How does change in network configurations affect different split point settings?
- **RQ 3:** What would be the best decision strategy for choosing better split points while considering application goals?

### 4.2 Experiments Setup

We used a sequential CNN model consisting of 14 layers for this experiment. Table 1 shows the layer construction of the model in `tf.keras` type. This model is constructed with five convolutional layers (Conv2D) and three pooling layers (MaxPooling2D). We set up the two split points for the experiment, one for between the device and edge and the other for between edge and cloud.

We set several candidates for each split point (both device-side and clouddside) based on our prior knowledge of CNN layers. The split point between the device and edge has two candidates: after the first Conv2D layer (layer 1) and after the first MaxPooling2D layer (layer 3). The choice of either split point involves a

**Table 1.** The elementary CNN model for experiment

#	Layer	Output shape	Param #
1	Conv2D	(None, 26, 26, 32)	320
2	Conv2D	(None, 24, 24, 64)	18,496
3	MaxPooling2D	(None, 12, 12, 64)	0
4	BatchNormalization	(None, 12, 12, 128)	256
5	Conv2D	(None, 10, 10, 128)	73,856
6	Conv2D	(None, 8, 8, 128)	147,584
7	MaxPooling2D	(None, 4, 4, 128)	0
8	BatchNormalization	(None, 4, 4, 128)	512
9	Conv2D	(None, 2, 2, 256)	295,168
10	MaxPooling2D	(None, 1, 1, 256)	0
11	Flatten	(None, 256)	0
12	BatchNormalization	(None, 256)	1,024
13	Dense	(None, 512)	131,584
14	Dense	(None, 10)	5,130

trade-off: a split point after layer 1 can achieve better performance because the device has much more overhead (x10) than edge and cloud. However, choosing to split after layer 3 affects the network bandwidth because the size of tensors passed through the pooling layer would become much smaller. The split point between the edge and the cloud has four candidates: after the second Batch-Normalization layer (layer 8), after the 5th Conv2D layer (layer 9), after the 3rd MaxPooling2D layer (layer 10), or after the Flatten layer (layer 11). A split point after layer 8 or layer 9 would utilize better computation power on the cloud but tends to have greater network latency because of the high tensor dimension. On the other hand, a split point after layer 10 or 11 would reduce the transmitted tensor size while having to compute more on medium-end edges.

**Table 2.** Experiment configuration

Config	Variable	Device	Edge	Cloud
All	Overhead	x10	x5	x1
	Frames	50		
Type 1	Latency (ms)	2.5	2.5	250
	Max throughput	500,000	500,000	500,000
Type 2	Latency (ms)	2.5	2.5	250
	Max throughput	<b>50,000</b>	<b>50,000</b>	<b>50,000</b>
Type 3	Latency (ms)	2.5	2.5	<b>1000</b>
	Max throughput	500,000	500,000	500,000

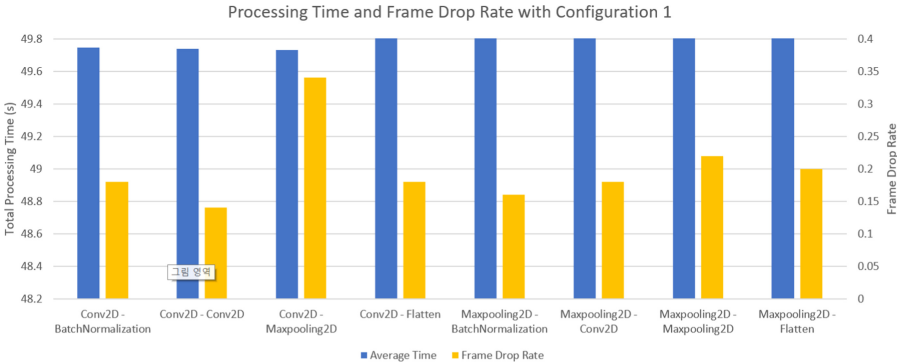
Table 2 shows the configuration of the experiment. The unit of the max throughput set is the length of the serialized string of tensors. The overhead is  $x10$ ,  $x5$ , and  $x1$  for each inference to reflect each device’s performance limitations, edge, and cloud. Each test is conducted with 50 data frames. We set three configuration types. The first configuration is a primary setting with a latency of 2.5 ms and 250 ms with a maximum throughput of 500,000. The second configuration has an environment with low maximum throughput, causing the packet size to be smaller and the transmission frequency to be higher. The last configuration has high latency (1000 ms) in the case of the wide area network (WAN) suffering from low quality or congestion. We referred to the general range of the latency of WAN as being from 250 ms to 500 ms.

### 5 Result

Here, we report the measured processing time of 50 frames and frame drop rate with three configurations explained in the previous section (Figs. 4, 5 and Tables 3, 4, 5).

**Table 3.** Experiment result with configuration 1

Split point	Time (s)	Frame drop
Conv2D - BatchNormalization	49.7454	0.18
Conv2D - Conv2D	49.7391	0.14
Conv2D - MaxPooling2D	49.7301	0.34
Conv2D - Flatten	49.8100	0.18
MaxPooling2D - BatchNormalization	49.8244	0.16
MaxPooling2D - Conv2D	49.8072	0.18
MaxPooling2D - MaxPooling2D	50.2666	0.22
MaxPooling2D - Flatten	49.5615	0.20

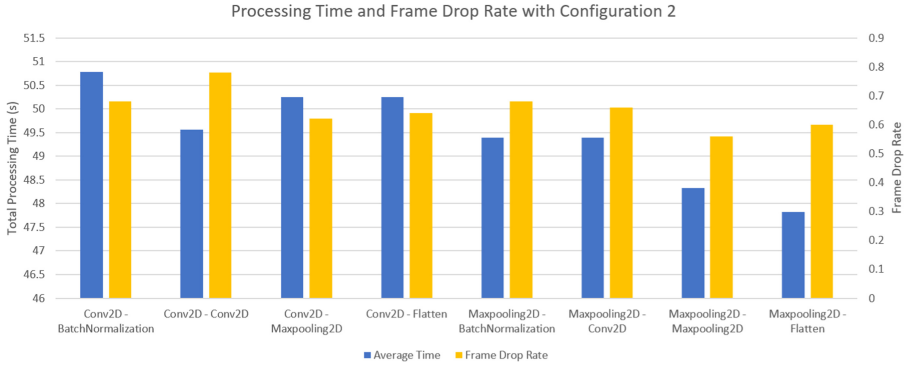


**Fig. 4.** Graph of processing time and frame drop rate with configuration 1



**Table 4.** Experiment result with configuration 2

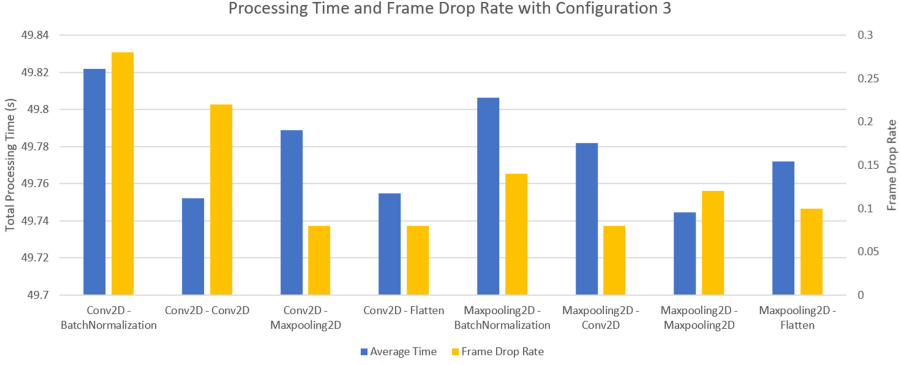
Split point	Time (s)	Frame drop
Conv2D - BatchNormalization	50.7831	0.68
Conv2D - Conv2D	49.5615	0.78
Conv2D - MaxPooling2D	50.2550	0.62
Conv2D - Flatten	50.2487	0.64
MaxPooling2D - BatchNormalization	49.3895	0.68
MaxPooling2D - Conv2D	49.3894	0.66
MaxPooling2D - MaxPooling2D	48.3246	0.56
MaxPooling2D - Flatten	47.8205	0.60

**Fig. 5.** Graph of processing time and frame drop rate with configuration 2**Table 5.** Experiment result with configuration 3

Split point	Time (s)	Frame drop
Conv2D - BatchNormalization	49.8217	0.28
Conv2D - Conv2D	49.7521	0.22
Conv2D - MaxPooling2D	49.7888	0.08
Conv2D - Flatten	49.7549	0.08
MaxPooling2D - BatchNormalization	49.8062	0.14
MaxPooling2D - Conv2D	49.7817	0.08
MaxPooling2D - MaxPooling2D	49.7446	0.12
MaxPooling2D - Flatten	49.7721	0.10

### 5.1 RQ1: Better Split Points Regarding the Type of CNN Layer

Here, to answer the first research question, we verify our initial assumption on the trade-off between network cost and computational overhead on each edge/device from our experiment result.



**Fig. 6.** Graph of processing time and frame drop rate with configuration 3

**Split After Conv2D Layer/MaxPooling Layer.** We initially predicted that splitting after the MaxPooling layer would reduce transmission overhead significantly. This tendency is shown especially in Configurations 2 and 3, simulating a low-throughput or high-latency WAN network. **MaxPooling2D-Conv2D** resulted in a higher processing time compared to the split point of **MaxPooling2D-MaxPooling2D**. In this setting, the high cost of WAN overhead exceeds the computing overhead of a processing pooling layer on a slower edge (as compared to cloud). In Configuration 1 with lower WAN latency, these split points do not show a significant difference. Additionally, choosing a split point between device and edge did not reveal much difference because we set LAN latency as only 2.5 ms (although this value is in the range of general LAN latency) (Fig. 6).

Overall, the result was more affected by the network configuration than by the characteristics of layers. However, with a greater penalty on large tensor transmission, the benefit of transmitting more processed tensors by doing more computation on device/edge would be discovered.

## 5.2 RQ2: Effect of Network Configurations

As shown in the previous subsection, different network configurations would result in divergent processing time and frame drop rate. With Configuration 2, lowering the max throughput caused high frame drop rate because packets flooded the network. Again, reducing the transmission size by processing more layers on the device/edge would prevent the total processing time from going up too high, but it could not reduce the frame drop rate.

## 5.3 RQ3: Application Goal-Driven DNN Slicing

The processing overhead (fixed in this experiment) could be tuned because nowadays, both cloud and edge devices can scale using a useful hypervisor technique (e.g., Docker). Hence, based on the simulation results and target applications'

requirements, we can manipulate allocated computing power on each machine rather than sacrifice its goal. Preserving the application goal is essential because, for safety-critical applications like visual inference in an autonomous car, frame drops in a relatively short period can result in severe situations (e.g., not identifying pedestrians in front of a car).

## 5.4 Limitation and Future Work

Initially, we planned to investigate a more diverse setting with varied computing overhead settings, different throughput for LAN and WAN, and more diverse distribution of latency. Unfortunately, we found that there are too many parameters to effectively tune. Hence, we decided to focus on a specific and representative set of latency and throughput settings and tried to explain the consequences of different split points from the perspective of the target application’s requirements and the characteristics of the CNN model. Our settings based on fixed values can constrain the expressiveness of the realistic environment in our simulator. However, we concluded that we could still produce a subset of valuable characteristics in combining various split points and network configurations, including high latency and low throughput on WAN. We set a slightly extreme value to observe the difference more clearly. (In a low-throughput setting, every batched frame was split up into 20 packets, and the value of 1000 ms is quite a harsh value simulating a very congested or erroneous network.)

In future work, we plan to implement a more fine-grained model slicer for the general Tensorflow model that contains more complex components, such as residual blocks to cope with a more realistic model and dataset. Supporting non-sequential models required non-trivial engineering, which is hard to accomplish. However, doing so would make our simulation environment more practical for pre-validating split points and preliminarily deciding which models to use and how much computation power is needed to achieve the required performance. We also aim to automate the steps to find the best configuration of parameters using search methods such as genetic algorithms (GAs). This approach can be uniquely conducted on our framework because we support a full software configuration in a single Docker daemon environment.

## 6 Conclusion

We propose a fully software-configurable simulation framework for DNN slicing on edges. The simulation environment works on general sequential-type DNN models, and arbitrary latency and throughput can be injected into the framework. Computational overhead on each machine can be simulated as well. Through an experiment with three different network configurations, we gained insight on how to consider CNN layer characteristics when deciding on split points. The experiment also showed that the total processing time and frame drop rate significantly depend on network capacity. We put a value in our simulator to provide an effective means of cost estimation in constructing a DNN

inference pipeline utilizing edges—this achieves the idea of *adapting machines* for the application’s requirements.

**Acknowledgement.** This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-2020-0-01795) and (No. 2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation).

## References

1. Fowers, J., et al.: A configurable cloud-scale DNN processor for real-time AI. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 1–14. IEEE (2018)
2. Hsu, K.J., Bhardwaj, K., Gavrilovska, A.: Couper: DNN model slicing for visual analytics containers at the edge. In: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, pp. 179–194 (2019)
3. Lockhart, L., Harvey, P., Imai, P., Willis, P., Varghese, B.: Scission: performance-driven and context-aware cloud-edge distribution of deep neural networks. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), pp. 257–268. IEEE (2020)
4. Ren, P., Qiao, X., Huang, Y., Liu, L., Dustdar, S., Chen, J.: Edge-assisted distributed DNN collaborative computing approach for mobile web augmented reality in 5G networks. *IEEE Network* **34**(2), 254–261 (2020)
5. Teerapittayanon, S., McDanel, B., Kung, H.T.: Distributed deep neural networks over the cloud, the edge and end devices. In: IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 328–339 (2017)
6. Talagala, N., et al.: ECO: harmonizing edge and cloud with ML/DL orchestration. In: USENIX Workshop on Hot Topics in Edge Computing (HotEdge 2018) (2018)
7. Tao, Z., Li, Q.: ESGD: communication efficient distributed deep learning on the edge. In: USENIX Workshop on Hot Topics in Edge Computing (HotEdge 2018) (2018)
8. Teerapittayanon, S., McDanel, B., Kung, H.T.: Distributed deep neural networks over the cloud, the edge and end devices. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 328–339. IEEE (2017)
9. Wang, X., Luo, Y., Crankshaw, D., Tumanov, A., Yu, F., Gonzalez, J.E.: IDK cascades: fast deep learning by learning not to overthink. arXiv preprint [arXiv:1706.00885](https://arxiv.org/abs/1706.00885) (2017)