# System support and mechanisms for adaptive edge-to-cloud DNN model serving

Matthias Reisinger, Pantelis A. Frangoudis, and Schahram Dustdar
Distributed Systems Group, TU Wien, Vienna, Austria
Email: e1025631@student.tuwien.ac.at, p.frangoudis@dsg.tuwien.ac.at, dustdar@dsg.tuwien.ac.at

***Abstract*—We present an orchestration scheme for Deep Neural Network (DNN) model serving, capable of computation distribution over the device-to-cloud continuum and low-latency inference. Our system allows automated layer-wise splitting of DNN structures and their adaptive distribution over compute hosts, providing an execution environment for collaborative inference. Model deployment and its self-adaptation at runtime are implemented by optimization algorithms supported in a plug-in manner. These follow service and infrastructure provider criteria and constraints, expressed via well-defined interfaces. Our framework can serve diverse neural architectures, including DNNs with early exits, with zero to minimal modifications.**

## I. Introduction

Deep Neural Networks (DNNs) have proven to be a potent tool for large-scale data analytics. Advances in hardware have spurred the design of deeper and more powerful DNN architectures which enabled their application for learning tasks in traditionally complex domains such as image classification, video analytics, and speech recognition.

Due to the limited computational capacity of end-user or IoT devices, DNNs are typically operated as centralized services in remote cloud data centers that provide the needed compute resources. This design conflicts with application scenarios that depend on a high degree of responsiveness or lack the required network bandwidth for streaming large amounts of data to the cloud. In contrast to such a monolithic service design, edge computing follows a more distributed approach that moves data processing closer to the end-devices where data originate. The layered architecture of DNNs inherently lends itself to be mapped over such a distributed compute hierarchy. This can help balance load and reduce inference latency. Recent research [1] also proposes new architectures that facilitate the distributed execution of DNN inference: Instead of using only one classifier at the final network layer, additional side-exit classifiers are introduced at intermediate layers that allow to obtain inference results at earlier points in the network.

DNN model serving over a device-to-cloud compute continuum however comes with non-trivial challenges, which are the focus of this work. These include (i) providing the appropriate execution environment on-device, at the edge and in the cloud, for distributed deployment, and zero-touch orchestration and life-cycle management of DNN model serving, and (ii) addressing the algorithmic aspects of optimal deployment and self-adaptation responding to a volatile operating environment.

To this end, we design and implement a full orchestration framework which allows service providers to deploy purposely-built (e.g., multi-exit models with built-in support for our framework) or pre-existing DNNs over hosts with diverse capabilities, expressing target performance goals such as minimizing inference latency. Different placement and orchestration strategies can be activated to distribute DNNs layer-wise, supported as plugins and executed by a host runtime environment. The latter also instruments infrastructure monitoring, which feeds critical input to the aforementioned orchestration strategies, allowing to optimize for diverse criteria (e.g., inference latency, workload distribution and fairness), subject to infrastructure- and service-level constraints.

The state of the art either allows a single DNN split point at a time and does not explicitly consider early-exit architectures [2]–[4], or assumes the full model is available both on-device and on a remote server [2], [5], or does not focus on system support for DNN orchestration [3], [6]–[8].

## II. System architecture

**Model partitioning:** We provide a programming model building on top of PyTorch which allows developers to compose and train neural network architectures, potentially with early exits, and define distributable layers. A DNN produced this way can be passed on by the service provider to a controller, which implements the core of our orchestration logic and which takes care of the distribution of its layers to compute hosts. We also provide an automated model slicing mechanism that operates on a pre-trained, vanilla PyTorch model (in particular, on TorchScript,[1] an intermediate representation of serialized PyTorch modules), scans its computational graph, and automatically identifies split points. The processed model can be submitted for serving and is treated by the controller in an identical manner as the ones built using our developer facilities. Therefore, our system can orchestrate existing, pre-trained models, without modifications.

**Runtime environment:** Each compute node implements a runtime that allows to receive DNN layers, perform the respective computations, communicate with other nodes in the compute hierarchy, trigger inference, and perform monitoring. Communication tasks are carried out over gRPC. This includes exchanging serialized intermediate results of DNN inference between layers. Each host periodically measures the available bandwidth and latency towards other known hosts. This infor-

[1] https://pytorch.org/docs/stable/jit.html

2021 IEEE International Conference on Cloud Engineering (IC2E) | 978-1-6654-4970-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/IC2E52221.2021.00046

978-1-6654-4970-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IC2E52221.2021.00046

mation is also updated during inference, when the node has to exchange information with nodes hosting other DNN layers.

**Controller:** Nodes initially register with the controller, which then contacts them periodically to determine their availability and resource state. It maintains a latency and bandwidth matrix with the respective metrics for any pair of compute nodes, which it updates based on the information reported by node runtimes. It exposes both southbound and user-facing APIs; the latter are used as an entry point to deploy a DNN model.

**Scheduler:** This component executes the key orchestration functionality. It contacts the controller in regular intervals to obtain the current state of the compute hierarchy. Upon startup, an offline profiling step is performed to estimate the resource requirements of the DNN layers. Based on that information, it then repeatedly computes a placement and deploys each DNN layer to its assigned compute host. From a software architecture perspective, we provide a plugin framework for implementing different deployment, resource allocation, and adaptation strategies. This way, different optimization goals can be pursued. Our system readily supports a number of them.

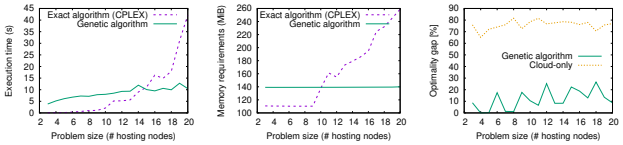
**Layer placement strategies:** When a model is provided to the scheduler for serving, the latter needs to decide on an appropriate placement of layers over nodes in a compute hierarchy. This decision can be driven by different criteria. We briefly describe latency-driven strategies, but others are also possible, such as those aiming to balance inference load fairly across nodes or minimize energy consumption. We define inference latency as the end-to-end latency from submitting input data to receiving an inference result. This is broken down to communication- and computation-related latency. Communication latency is determined by the underlying network links and the volume of data that need to be exchanged between nodes hosting different DNN layers. Computation latency depends on the computational requirements of each inference task, and the capabilities and load of the processor executing it. A latency-driven strategy aims to determine the appropriate DNN split points and deploy layers of the DNN on hosts to minimize inference latency, subject to capacity (node and network link) and layer ordering constraints. To acquire the necessary input to the algorithm, we combine the following information: (i) the computational requirements per layer (in FLOPS) and the incurred traffic volume, for which we have implemented DNN profiling facilities, (ii) the computation capabilities (in FLOPS) of different compute node hardware models, which we acquire either via existing studies [9] or via offline benchmarking (for new devices), (iii) monitoring information, and (iv) the architecture of the DNN to deploy.

When the number of potential host nodes is small, this problem can be solved to optimality fast. However, when deploying very deep neural architectures over large-scale infrastructures with many user-controlled nodes and deep fog computing hierarchies, deriving the optimal DNN layer distribution is computationally expensive. We formulate different variations of the optimization problem, shown to be NP-hard, and implement alternative algorithms to solve it, which trade solution quality for execution time.

**Triggering self-adaptation:** The execution cost of DNN (re)deployment is important when considering runtime adaptations (e.g., cross-host layer migration). Currently, a full re-deployment takes place after periodic monitoring by the controller, which might fire adaptation triggers when the conditions change, or when an updated model is available. We target more sophisticated self-adaptation schemes, e.g., using reinforcement learning [10], for future work.

## III. Experimental Results

Fig. 1 shows the performance of exemplary placement strategies on GoogLeNet [11], a 22 layers deep Convolutional Neural Network. For small-scale problems, the optimal deployment can be derived in acceptable time using CPLEX, but larger compute hierarchies induce significant compute and memory resource requirements. A genetic algorithm (without significant tuning) demands less resources for large problem instances, with a reasonable loss in solution quality. A cloud-only solution incurs significant inference latency.




(a) Execution time. (b) Memory requirements. (c) Performance (solution quality) loss vs. the optimal.

Fig. 1: Solving the distributed DNN deployment problem.

## IV. Conclusion

We provided a brief overview of our orchestration framework for distributed DNN model serving over device-to-cloud compute infrastructures. Various open issues are currently under our study, with the design of low-overhead algorithms and mechanisms for runtime self-adaptations topping our list.

## References


[1] Y. Matsubara *et al.*, "Split computing and early exiting for deep learning applications: Survey and research challenges," *CoRR*, vol. abs/2103.04505, 2021.

[2] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proc. ACM ASPLOS*, 2017.

[3] C. Hu *et al.*, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE INFOCOM*, 2019.

[4] H. Jeong *et al.*, "PerDNN: offloading deep neural network computations to pervasive edge servers," in *Proc. IEEE ICDCS*, 2020.

[5] S. Laskaridis *et al.*, "SPINN: Synergistic progressive inference of neural networks over device and cloud," in *Proc. ACM MobiCom*, 2020.

[6] S. Teerapittayanon *et al.*, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE ICDCS*, 2017.

[7] S. Scardapane *et al.*, "Why should we add early exits to neural networks?" *Cogn. Comput.*, vol. 12, no. 5, Jun 2020.

[8] E. Li *et al.*, "Edge AI: on-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wirel. Commun.*, vol. 19, no. 1, 2020.

[9] The top 50 fastest computers in the VMW research group. [Online]. Available: http://web.eece.maine.edu/~vweaver/group/machines.html

[10] Z. Zhao *et al.*, "EdgeML: An AutoML framework for real-time deep learning on the edge," in *Proc. ACM/IEEE IoTDI*, 2021.

[11] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE CVPR*, 2015.