Fog Computing for Deep Learning with Pipelines

Antero Vainio, University of Helsinki
Akrit Mudvari, Yale University
Diego Kiedanski, Yale University
Sasu Tarkoma, University of Helsinki
Leandros Tassiulas, Yale University

***Abstract*—In this article, we introduce a fog system design for processing data collected from edge devices, such as mobile, sensor, and extended (mixed, augmented, virtual) reality equipment. Our system enables the network to provide hardware-accelerated processors for resource-intensive computations on data gathered from remote locations, such as 5G and beyond mobile networks. By splitting heavy computations into pipelines, and distributing them among processors in the edge, fog and the cloud, our design benefits from the processing power of the cloud, while utilizing fog devices with a lower network latency.**

We implement our design, and use it for distributed training and inference with industry-grade deep learning models for computer vision. We deploy our architecture in infrastructure including cloud and edge servers supporting GPU-accelerated computations. We benchmark pipelines in various deployment settings to study the overhead that they introduce. Our contributions are a new design for wide-area data processing, a framework that realizes this design and provides means of developing applications that are optimized in terms of infrastructure and hardware. These contributions are complemented with our benchmark results, which reveal the potential causes of processing overhead.

***Index Terms*—fog computing, stream processing systems, deep learning**

I. Introduction

Mobile devices constitute the largest share of devices connected to the Internet, and according to CISCO forecast, their share will keep steadily increasing. Modern Internet-based applications, such as multimedia streaming and environmental sensing, produce massive amounts of data, and according to the current trends, they are expected to constitute the majority of the network traffic in the future [1].

6G networks are envisioned to improve the broadband connectivity in wireless networks drastically [2]. The peak data rate is expected to increase from 20 Gbps in 5G networks, to 1 Tbps in 6G, and the end-to-end latency is expected to reduce from 1 ms closer to .1 ms. These, among other quality improvements, enable new emerging applications, such as holographic teleportation, autonomous cyber-physical systems, intelligent industrial automation, as well as smart infrastructure and environments [3]. While 6G networks will be able to collect more data for the purposes of the applications, in doing so, they also increase the traffic in the backbone network, in case all of the data are transmitted to centralized data centers, such as cloud platforms. However, as most of the computational resources are currently centralized, collecting the data is the primary alternative for computationally demanding tasks, such as data analytics and machine learning.

Edge computing provides an alternative for centralized data collection, by utilizing compute hardware located at the edge of the network [4]. By avoiding the need for communication, edge devices can run computations with a lower latency, and have the potential to improve users' privacy. However, the edge is limited by the resource-constraints of mobile devices; certain computations may be practically infeasible to execute in the edge altogether. When developing applications that utilize the edge, one needs to consider the details of the device, such as the processor architecture as well as the available memory and disk space. [5]

Fog computing is another alternative to cloud and edge computing. It complements the two, by scattering compute resources within the network. Edge devices can utilize the fog architecture, by offloading computations to the fog hardware instead of directly to the cloud. Compared to cloud computing, fog computing provides a lower expected network latency for offloading. Fog computing has also advantages over edge computing, since the fog devices need not be mobile; static devices have less physical space constraints than mobile devices, and can include more compute power. Furthermore, static devices do not need to rely on batteries for power and offloading appropriate computations from mobile devices can save their batteries.

When it comes to real-world deployment scenarios, the primary considerations in fog computing are **resource placement and allocation**. Firstly, in order for the fog devices to provide lower network latency than cloud, they need to be geographically closer to the users. For instance, in mobile edge computing (MEC), this is guaranteed by deploying compute hardware to the mobile base stations, which the users connect to in order to get access to the Internet [6]. Secondly, due to the highly distributed nature of fog computing, the deployed hardware must be sized according to the demand; under-provisioning hardware negatively affects the quality of service, while over-provisioning makes fog computing economically infeasible.

In this paper, we investigate fog computing for geo-distributed data processing, by using software virtualisation in combination with hardware-acceleration. We present a lightweight system design for integrating edge, fog and cloud devices seamlessly via pipelined execution, and the associated programming framework, that enables the application-developer to create processing pipelines by defining each subtask as a function. We use our framework to implement two deep learning applications for computer vision, and prototype our system design with devices, that provide GPU-accelerated processing for deep learning. We conduct benchmarks to compare the performance of the pipelines in various deployment scenarios. The source code for both our system and our example applications is open-sourced.

2023 IEEE 7th International Conference on Fog and Edge Computing (ICFEC) | 979-8-3503-2288-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICFEC57925.2023.00017

979-8-3503-2288-0/23/$31.00 ©2023 IEEE
DOI 10.1109/ICFEC57925.2023.00017

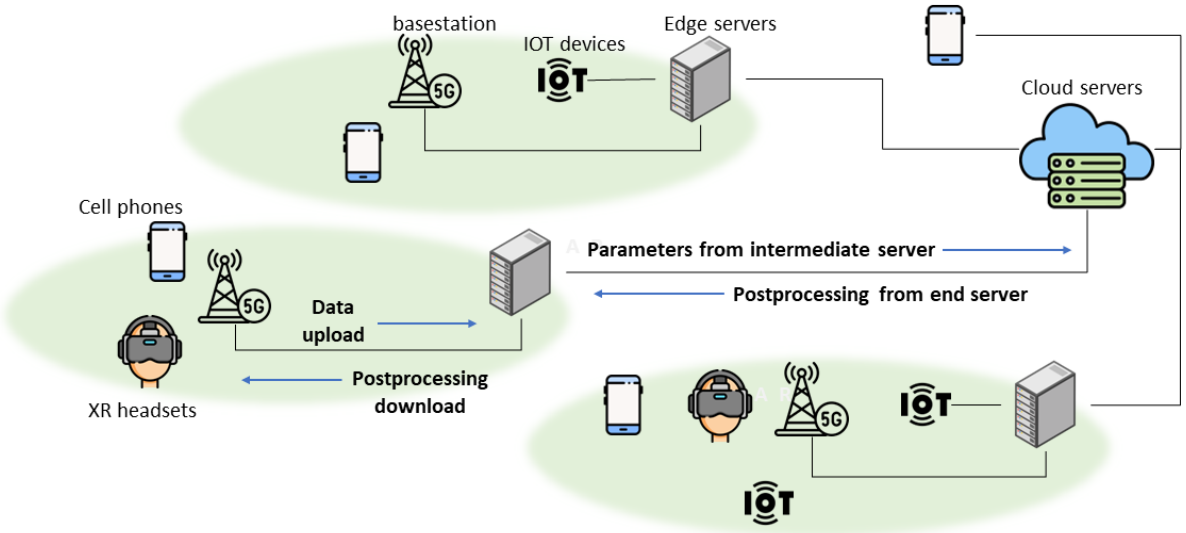


Fig. 1: An overview of data processing in the cloud continuum where next-generation applications like XR and IoT may utilize 5G and other communication methods

Combined our work provides answers to the following questions:

1) In what conditions can the fog support data processing in the edge-cloud continuum?
2) What are the main causes and symptoms of processing overhead in the fog?
3) How does communication overhead involved in fog processing compare to that in cloud processing?

The paper is organized as follows. In Section II we discuss the background and motivation for our work. In section III, we present the design, implementation and the deployment of our architecture, as well as the two deep learning pipelines using our framework: model training for image classification, and inference for object detection. In section IV we show the results from our system evaluation, where we measured the runtime performance of the two deep learning pipelines. In section V we provide additional discussion about our results. Finally, we give our final conclusion in section VI.

II. Motivation & Related Work

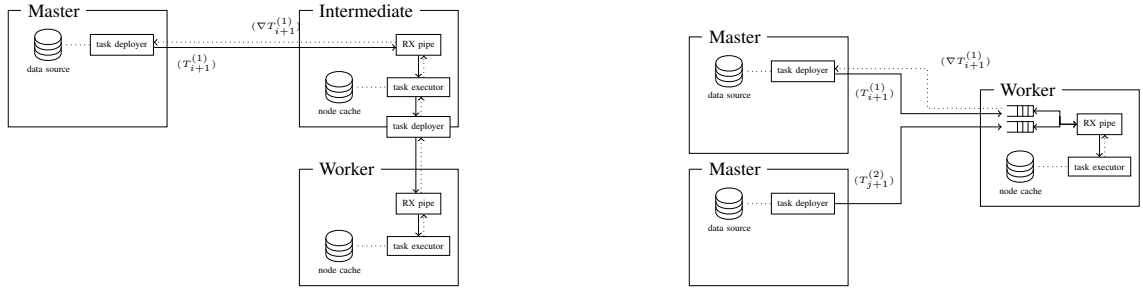
An overview of the set of future use cases in networked applications is illustrated in Figure 1. Here, different end devices, be it environment-sensing IoT, AR/VR headsets, or cell phones, will be utilizing the 5G infrastructure to access the computing resources in the edge-cloud continuum.

A. Cloud, Edge and Fog Computing

Cloud computing and big data have paved the way of distributed data processing. Since the primary bottleneck for data processing tasks results from the data transfer speeds, which tend to be orders of magnitude slower than the processing speed of the chips, in cloud computing large computational tasks are parallelised by partitioning the data among the available servers. This approach can even the gap between the slow data access speed and the fast data processing speed of contemporary computer systems.

The existing data processing cluster solutions are primarily designed for data center environments, and there are no widely adapted data processing clusters designed for fog infrastructure. Cloud-based clusters generally take advantage of distributed file systems, such as HDFS [7], which rely on high-bandwidth and low-latency networks, and may exploit a data center network topology designed according to the needs of a specific data infrastructure [8]. Due to these factors, using the Internet as the network for cloud-based data processing clusters is not possible, or it significantly affects their performance, and new cluster designs aim to optimize the distributed execution of data processing tasks over wide-area networks [9].

The next generation cellular communication, in the form of 5G [10] and beyond, has become an important enabler for the edge and the fog. While fog computing previously lacked the infrastructure for deployments, now with the proliferation of



(a) A three-node pipeline.

(b) Two two-node pipelines using the same worker node.

Fig. 2: Sparse architecture for pipelines. The offloaded tasks' input parameters are $T_j^{(i)}$ and the result values are $\nabla T_j^{(i)}$.

5G infrastructure and drafting of the MEC standards [11], the monopoly of the cloud is giving way to the edge and the fog. However, in this new 5G edge-cloud continuum, the number of alternatives for deployment settings is higher, and the right selection of infrastructure may not always be obvious.

As individual processor are limited by physical constraints, and Moore's Law has gradually ceased to improve the per-chip compute performance, new ways to improve the performance of data processing tasks are needed [12]. While using hardware dedicated for specific computations used to be the norm before the popularity of general purpose CPUs, this direction has gained a lot of recent interest, and today most of the cloud service providers are either manufacturing their own chips, or using customized chips. By using specialized processors, more computational performance can be gained per watt, however the gain depends on the task at hand. As with the selection of infrastructure, the proper choice of specialized processors is application-specific.

B. Distributed Machine Learning

As the size of the models used for state-of-the-art machine learning has increased, their computational requirements have increased as well, resulting in a need to distribute their computations. For neural networks with billions of parameters, model training in a feasible time requires cloud-scale infrastructure [13]. Model training within a data center is usually distributed by partitioning the training data and/or model parameters, and parallelising access to them. However, centralized processing requires uploading the training data, which may be subject to privacy regulations, to the cloud. Federated learning [14] avoids this problem, by instead bringing the model to the mobile devices for training. While avoiding the need to collect raw user data, federated training generally convergences slower due to having to aggregate model updates by averaging gradients computed from non-IID data. The model size in federated learning is also constrained by the mobile device, which must be able to compute the model update locally. To combine edge processing and cloud offloading, authors in [15] suggest splitting neural network vertically, and offloading part of the processing to the cloud. New machine learning methods are constantly being explored, and even entire data systems have been designed particularly for distributed machine learning [16], [17], so the current landscape of machine learning has become highly distributed.

Our work builds upon the existing solutions for cloud-based data processing, and edge computing with hardware-acceleration, while considering the emerging landscape of 5G and beyond mobile networks, as well as distributed deep learning. In addition to outlining a practical system design, we set out to develop a ubiquitous programming interface that can help data scientists to find the optimal configuration for data processing applications, both in terms of the infrastructure and the hardware.

III. Sparse: Task Execution Pipelines

Unlike in cloud computing, where all of the resources are located in a few global regions, the placement and allocation of compute resources is a major challenge in fog computing, where the end users' locations continuously change. In fog paradigm, compute resources are deployed sparsely in the network to provide maximal coverage area. As a trade-off, the fog devices cannot have processing power comparable to cloud environments.

As a solution to improve the resource-constrained fog hardware utilization, we study **networked processing pipelines**, that, in line with OpenFog Reference Architecture [18], originate from the data source and progress among the fog devices, until either the task is completed, or the pipeline reaches the cloud. In practice, the end user offloads processing to the closest fog device, which runs part of the computation, and offloads the rest to another nearby fog device. This makes it possible for the fog devices to distribute computations among each other, enabling devices that are not being directly consumed by end users to be utilized by other nearby locations. By utilizing pipelines, the fog environment becomes more robust to continuously changing demands, and avoids the need to physically relocate hardware. As a trade-off, having to partition computer program over the wide area network adds overhead. In this work, we study the details of task pipeline execution, identify and explain some causes of overhead, and how they can be avoided. We introduce a novel programming framework to realize fog computing. We call our framework Sparse: Stream Processing Architecture for Resource-Subtle Environments.

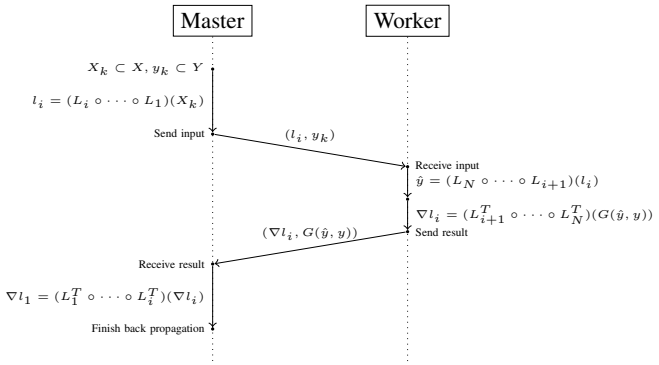


Fig. 3: Communication flow during model training pipeline. The execution of the offloaded task is awaited without blocking the execution of other tasks.

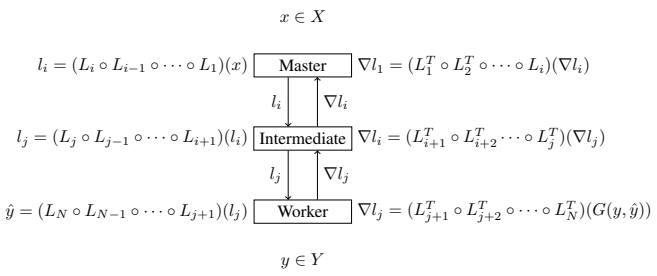


Fig. 4: Extended split learning data flow

A. Pipeline Design

The general design of a basic three-node pipeline in Sparse is depicted in Figure 2a. A Sparse cluster is formed by nodes located either in the cloud, the fog or in the edge. The nodes can create new processing tasks, or execute tasks created by other nodes. To execute tasks, a node assumes the role of a *worker*. To offload tasks, a node assumes the role of a *master* and connects to a nearby worker node. In pipelines with three or more nodes, the *intermediate* nodes are acting as both masters and workers simultaneously. A worker node can serve any number of masters, as is depicted in Figure 2b.

In order to submit new tasks, a master node includes a *task deployer* component. When submitting a new task, the task deployer opens a network connection to one or more workers, and starts sending the task's input data. The worker node includes an *RX pipe* which accepts the connection and receives the input data. Once all the needed data are transmitted, RX pipe transfers them to the *task executor*, which runs the computation. In practice a task executor may be a hardware-accelerator. The RX pipe then responds to the master node with the result of the computation.

Since nodes can cache some of the data used in consecutive computations, such as machine learning model parameters, pipelines can reduce the need for communication. This is particularly useful in iterative algorithms, like machine learning model training.

B. Implementation

We implement our framework with Python, and deploy it in containers using Docker. The source code for the implementation is available in GitHub[1]. The implementation uses an asynchronous programming pattern for concurrency. The implementation includes a module for deep learning, using PyTorch, which supports general purpose CPUs, as well as hardware-accelerators like GPUs and TPUs.

The prototype currently implements RPC for communication. The communication flow during model training pipeline (explained later in more detail) is displayed in Figure 3. A new connection is established for each task, and it is kept open during the entire offloaded processing. The nodes serialize messages using Python's Pickle library, which, unlike in text-based serialization schemes like JSON, instead converts Python objects directly into byte streams, providing lower transformation overhead for large payloads.

The when serving offloading requests, a worker node uses first-come-first-serve policy for task execution. The task executor is implemented as an abstract Python class, and it provides access to the node functionality, such as the task deployer in case the nodes is also offloading tasks. When using the framework, an application developer only needs to implement the task executor. In our deep learning pipelines, we implement executors for training a model, and for using a trained model for inference. These two executors are included in the Sparse library, and should work out-of-the-box for basic deep learning training and inference.

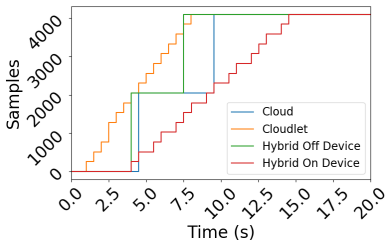
While the Sparse framework is mainly designed for creating pipelines, *there is no application level primitives included for pipelines, but instead pipelines are created by connecting the nodes with the partitioned data*. For dynamic configuration of pipelines, the framework can be interfaced with external orchestration tools, like Kubernetes or Docker Swarm.

C. Deep Learning Pipelines

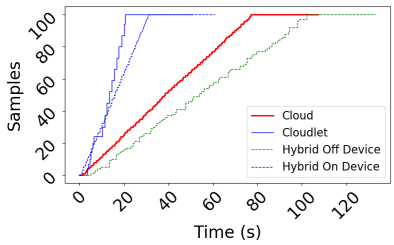
We use our framework to implement two deep learning pipelines: one for training, and another for inference. The training pipeline takes the split learning [15] approach to partition the used model based on neural layers, so that the training can be performed collaboratively by nodes, which do not have enough resources to operate on the full model. The inference pipeline splits the model similarly, but does not include backward propagation.

Figure 4 illustrates an example of a three node training pipeline, where a neural network, consisting of layers $L_1, \ldots L_N$ is trained by three nodes, by using a feature vector $x \in X$ and a label $y \in Y$. The model is trained with standard forward/backward propagation approach. First a prediction is computed for x, using the model's current parameters, in a sequential fashion. Since the model is split, the computation for the Intermediate node would happen in the following fashion:

[1]https://github.com/AnteronGitHub/sparse



(a) Training VGG



(b) Inference YOLOv3

Fig. 5: Task completion time measured in the data source. The curve step height corresponds to the used batch size (1 without batching).

$$l_j = (L_j \circ L_{j-1} \circ \cdots \circ L_{i+1})(l_i) \tag{1}$$

, including $j - i$ layers of neural network. The input for the Master node would be the feature vector x, and the output of the Worker node is the final prediction $\hat{y}$. This forward propagation is illustrated on the left side of Equation 1.

Following the approach in supervised learning, the correct label $y \in Y$ for the prediction is assumed to be known. In practice this can be achieved by manual labeling, or by extending the system for Reinforcement Learning (RL) [19]. The correct label is used along with a cost function $G : Y \times Y \to \mathbb{R}$, to find adjustments for the model parameters that best minimise the cost function, fixed for x. The most common technique is to use the chain derivation rule for the function described in Figure 4. Since L_k is usually not differentiable, the intermediate results, gained during forward propagation, are used to compute the derivative numerically. Following similar notations to what is used in [15], we will denote the computations involved in backward propagation simply L_k^T. During the computation across layers, ∇l_k is computed for each layer $k = 1 \ldots N$, and they constitute the final model update. The back propagation for the Intermediate node takes the gradient of the split layer it sent during the forward propagation as an input, calculated as:

$$\nabla l_i = (L_{i+1}^T \circ L_{i+2}^T \circ \cdots \circ L_j^T)(\nabla l_j) \tag{2}$$

For the Worker node, the calculation of the gradient includes discerning the loss function between the prediction $\hat{y}$ and the true label y. This calculation is denoted as:

$$\nabla l_j = (L_{j+1}^T \circ L_{j+2}^T \circ \cdots \circ L_N^T)(G(y, \hat{y})) \tag{3}$$

In our framework, we define each model partition as a separate function to create the pipeline. Forward and backward propagations computations use hardware-accelerated task executors, that cache model parameters in memory during the entire training. We also batch training data to boost the training on GPUs.

IV. Pipeline Benchmarks

To study the processing overhead introduced by our pipelines, we benchmark the execution of the same task in different pipeline configurations. As an example case, we use our deep learning pipelines for computer vision, a widely popular task in machine learning. In particular, computer vision models often involve billions of parameters [13], making it challenging to deploy them in resource-constrained devices. We measure the task completion time and the network usage in the data source, as well as the energy consumption and the hardware utilization in the fog device.

A. Methodology

Our testbed consists of a cloud server, and an SoC as a cloudlet. As a data source we use a laptop, that runs PyTorch data loader. Both the cloud server and the cloudlet support CUDA for GPU-accelerated processing. The hardware specifications of the devices are shown in Table I. The estimated peak performances are based on the manufacturer's specifications. The ping latency was measured before running the benchmarks.

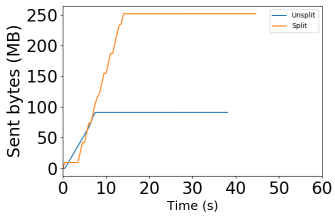
We evaluate the system with industry grade deep learning models for computer vision. For training, we use Visual Geometry Group (VGG) model architecture [20] for image classification, and the CIFAR-10 [21] dataset. For inference, we use the YOLOv3 [22] object detection model, and the COCO dataset.

We ran various benchmarks suites, each utilizing a different combination of cloud, edge and fog resources. Two of the suites split the task, while two process everything in the same node. Throughout this section, we will use the following names for different benchmark suites used in the system evaluation:

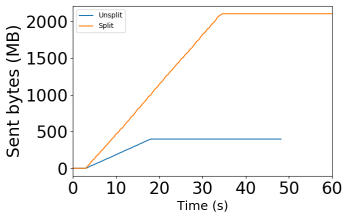
- *Cloud:* A two-node pipeline including data source and a cloud server. The data source offloads entire processing to the cloud server, without splitting the task.
- *Cloudlet:* Similar to the cloud suite, except the processing is offloaded to the cloudlet instead.

TABLE I: Hardware specifications of the devices used in the testbed.

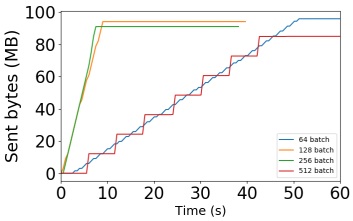
Device	Peak Perf. (INT8)	Ping latency
Cloud	130 TOPS	$\approx$ 9 ms
Cloudlet	22 TOPS	$\approx$ 1 ms



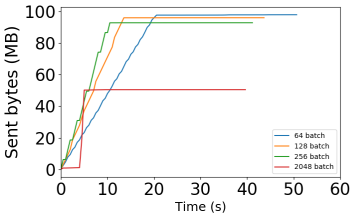
(a) Training VGG in the Hybrid On-Device suite.



(b) Inferring with YOLOv3 in the Hybrid On-Device suite.



(c) Training VGG in the Cloudlet suite.



(d) Training VGG in the Cloud suite.

Fig. 6: Network usage measured in the data source.

- *Hybrid Off-Device* A three-node pipeline, where the data source offloads the processing to the cloudlet and the cloud server, which split the task.
- *Hybrid On-Device* A two-node pipeline, where the data source also processes the first split, and offloads the rest to a cloudlet.

During benchmarks, we measure the task completion time in the data source, representing the task throughput, as well as hardware statistics in the cloudlet. Benchmarks are measured by an additional daemon process running at each cluster node. After a task has been processed, the monitored node sends a JSON message to the monitor daemon via a Unix socket, without waiting for the reply. For more details, see Appendix A for the pseudocode of the benchmarks. The source code for the monitoring is also included in Sparse source repository.

B. Results

Figure 5 shows the CDF of processed samples in training and inference. Appendix B shows, how the batch size affects the training time in each, and essentially it shows the batch sizes that provided the fastest training for each benchmark. Even though nodes load the model parameters into memory before the benchmarks are started, Figure 5 reveals that there is also a slow start involved in our pipelines; in all cases the first batch is the slowest to compute. While the slow start does not affect our main results, we note that it may be an important consideration in practice, and advocates executing pipelines for longer periods of time.

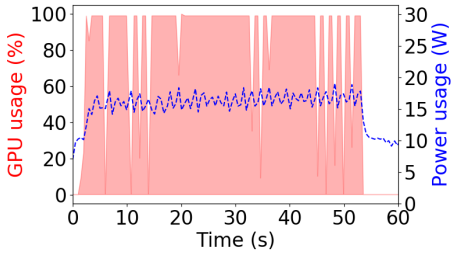
For training, by using a three node pipeline between the cloud and the edge, we were able to increase the batch size for the pipeline, enabling the hybrid off-device suite to achieve the fastest overall training time, slightly beating the cloudlet suite. Even though out of all the suites hybrid off-device uses the most hardware, and as a result more power, this result demonstrates that the fog infrastructure can complement cloud-based data processing to improve task throughput.

In contrast to training however, for inference without batching, the off-device hybrid deployment achieved the worst overall throughput. Since batching was the only method that we used to fine-tune the workloads, our results for inference mainly display the effectiveness of split neural network traversal. Even though inference in the cloudlet without splitting the model provided the highest throughput, combination of on-device processing and edge offloading was faster than cloud offloading.

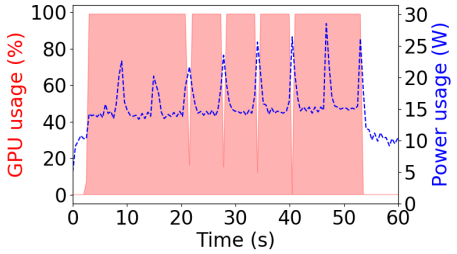
The communication overhead of neural network splitting in the pipelines is displayed for training in Figure 6a and for inference in Figure 6b. The added communication results mainly from the fact that intermediate layers in feed forward networks tend to be of higher dimension than the feature vectors. In our experiments, the network load of communicating the intermediate layer outputs is roughly four times higher than communicating the raw data.

Figures 6c and 6d show the effect of batch size for communication overhead during training without task splitting in the cloudlet and the cloud. For the resource-constrained cloudlet, using too big or too small batch size reduces the average transmission rate significantly. Essentially, there is a period during each batch, when no data is being communicated, suggesting that the pipelines become compute-bound. In contrast, when using an appropriate batch size, the transmission rate stays more constant.

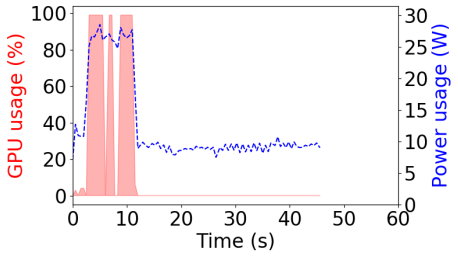
Figure 6d highlights, how the overall communication overhead reduces as batch size increases. As the cloud GPU supports a larger batch size for training the same model as the cloudlet, it avoids a lot of additional communication during training, such as gradients which are computed per batch. For a massive batch size of 2048, the overall network traffic drops



(a) 64 batch size (too small)



(b) 512 batch size (too big)



(c) 128 batch size (optimal)

Fig. 7: GPU and power usage in the fog device during the Cloudlet training suite.

by almost half in comparison to a more conservative batch size of 256. While this would be ideal in terms of overall data traffic, it has the biggest potential to congest the backbone network, and the fact that with a batch size smaller than 2048, cloud node shows signs of becoming compute-bound makes cloud offloading seem challenging to optimize without having to transmit massive amounts of data. Nevertheless, our results demonstrate how proper batching has the potential to reduce communication overhead during neural network training.

Figure 7 displays the GPU usage and the power usage measured in the fog device during the training with different batch sizes. The measured GPU usage is at maximum for most of the training, even when using a suboptimal batch size. However, different batch sizes make a clear difference in the device power usage. The base power consumption is approximately 10 W, which is the system power use when the GPU is not being used. When training with too small batches, the overall power usage increases only slightly, indicating that the GPU is not being fully utilized. Similar patterns can be seen when using too big batches, with the exception that there are small peaks in power usage; these peaks likely result from the details of the VGG model architecture. In comparison to training with a too small batch size, when using the optimal batch size, the power usage increases to almost twice as high, but the training lasts for less than a quarter.

Similar measurements for inference are displayed in Figure 8. Similarly to training, the optimal workload in inference has the highest power consumption during processing, while the overall processing time is smaller. Overall, for inference without batching, all of the benchmark suites show signs of hardware underutilization, and none of the benchmarks reach acceptable end-to-end latency, although these result also from the used model and the dataset, which are not adjusted for an XR use scenario.

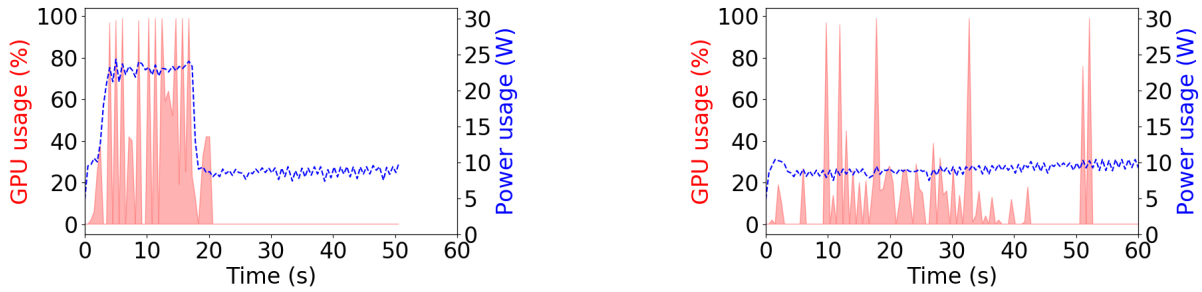
V. Discussion

Our benchmark results demonstrate the difference, that the selection of infrastructure and hardware make for data processing in the edge-cloud continuum. They highlight the impact of proper hardware utilization to the throughput of deep learning tasks. In practice, the applications determine the available optimization methods, and we noticed how the infrastructure that performed the best for batch training provided the worst performance for inference without batching.

Some of the results in our study are explained by the fact that we used mainly GPUs, which are designed for computations that can be parallelized, by including a high number of cores: as a result of this design choice, the processing speed of individual GPU cores is generally lower than cores in a CPU. Without batching, some other means of parallelizing individual inference tasks are needed to fully utilize GPUs, which can be seen in our benchmark results.

Task splitting proved to be a viable option to scale operations on feed forward networks, particularly for the purposes of model training. The freed compute capacity can this way be used to increase throughput of tasks, at least when tasks can be batched. Even without batching, by starting the processing in the data source, our hybrid on-device pipeline reached nearly as fast processing as edge offloading, and even outperformed cloud offloading.

We notice in Figure 6c, how suboptimal configuration results in a step-like curve in network traffic CDF. Essentially, there are periods during which no data is being transmitted,



(a) Cloudlet (fastest)

(b) Hybrid Off-Device (slowest)

Fig. 8: GPU and power usage in the fog device during inference suites.

and instead the device is processing previous task. Combined with the GPU and power measurements, the network usage reveals that when using too small or too big a batch size, the task becomes compute-bound, whereas with an optimal batch size, the task is primarily communication-bound.

By implementing pipelines, we have provided benchmark results for two real-world use cases utilizing deep learning for computer vision. Our results show the difference in the hardware utilization, when using different infrastructure for processing, leading to a significant difference in the completion time of computations.

VI. Conclusion

With many variables affecting the performance of pipelines, it is difficult to draw conclusions about which combination of infrastructure and hardware would be universally the best. While this study shows some cases where the fog is or is not useful for enhancing cloud-based data processing, we found our framework to be useful for prototyping such applications, and helpful in finding these optimal configurations. As such, we will keep using and developing the framework, and hope that it will be helpful for other researchers working in this domain as well.

To answer our first research question, the benchmark results demonstrate, that if the fog hardware can be fully utilized, it will improve the throughput of heavy workloads. To reach high hardware utilization, the workloads need to be appropriately sized. For neural network training, the workload size can be fine-tuned, for instance with the choice of the batch size, as we demonstrated. In practice this is achievable in fog computing, where the processing hardware is shared by multiple users, making aggregating input data is easier than in on-device edge computing.

Pipelines introduce overhead to processing, if the aforementioned condition cannot be met, in which case the intermediate processors create bottlenecks in the pipeline. The bottlenecks can turn ideally communication-bound processing pipelines compute-bound. While the most crucial symptom of the bottlenecks is a drop in task throughput, other symptoms include uneven communication flow, and oscillating or lower processor power consumption during the pipeline execution.

Interestingly, even though the split processing pipelines that we studied included more communication than their unsplit counterparts, in some instances, including the optimal configurations, the resulting overhead was mitigated by the benefits gained in task throughput. Due to these results, we conclude that a smaller overall communication does not guarantee that a task is processed faster, but note that it may be desirable for other reasons, such as avoiding network congestion.

Based on our current results, it seems that longer pipelines are best utilized for background tasks, that are expensive to compute, but do not have strict latency requirements, such as model training. Unlike training however, inference tasks have strict latency requirements, making batching an infeasible solution in many situations, since it results in additional expected waiting time for most of the individual requests. On the other hand, there may not be a need to rely on offloading for inference in the first place; as an example, model compression seeks to reduce the size and the computational cost of deep learning models [23]. However, this approach sacrifices models' representation capabilities, and as a result their accuracy.

Overall, we haven't seen similar studies elsewhere, since most of the pipeline studies we know (e.g. [24]) focus on input pipelines and not end-to-end processing. As we expect the future networked applications to keep becoming more data-driven, our results show great promise for using fog computing to enhance or even replace cloud-based data processing.

Acknowledgments

This research was in part supported by the Academy of Finland (grant number 345008), and the National Science Foundation CNS AI Institute (grant number 2112562), as well as the NSF-AoF FAIN project (grant number 2132573).

References

[1] "Cisco annual internet report (2018–2023) white paper," accessed: 2022-1-21. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] N. Rajatheva, I. Atzeni, E. Bjornson, A. Bourdoux, S. Buzzi, J.-B. Dore, S. Erkucuk, M. Fuentes, K. Guan, and Y. Hu, "White paper on broadband connectivity in 6G," *arXiv preprint arXiv:2004.14247*.

[3] I. F. Akyildiz, A. Kak, and S. Nie, "6G and beyond: The future of wireless communications systems," *IEEE access*, vol. 8, pp. 133 995–134 030, 2020, publisher: IEEE.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016, publisher: IEEE.

[5] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," *GetMobile: Mobile Computing and Communications*, vol. 25, no. 2, pp. 5–13, 2021, publisher: ACM New York, NY, USA.

[6] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017, publisher: IEEE.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[8] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 123–137.

[9] F. Lai, J. You, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Sol: Fast Distributed Computation Over Slow Networks," in *17th ${$USENIX$}$ Symposium on Networked Systems Design and Implementation (${$NSDI$}$ 20)*, 2020, pp. 273–288.

[10] A. Ghosh, A. Maeder, M. Baker, and D. Chandramouli, "5g evolution: A view on 5g cellular technology beyond 3gpp release 15," *IEEE access*, vol. 7, pp. 127 639–127 651, 2019.

[11] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[12] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017, publisher: IEEE.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, and K. Yang, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[14] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.

[15] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 2018, publisher: Elsevier.

[16] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, and M. I. Jordan, "Ray: A distributed framework for emerging ${$ai$}$ applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[17] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan, and others, "Towards federated learning at scale: System design," *arXiv preprint arXiv:1902.01046*, 2019.

[18] "Ieee standard for adoption of openfog reference architecture for fog computing," *IEEE Std 1934-2018*, pp. 1–176, 2018.

[19] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

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

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

[22] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[23] "tinyml foundation," accessed: 2022-9-12. [Online]. Available: https://www.tinyml.org/

[24] M. Kuchnik, A. Klimovic, J. Simsa, V. Smith, and G. Amvrosiadis, "Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 33–51, 2022.

APPENDIX A
BENCHMARK PSEUDOCODE

Algorithm 1 Data source training completion benchmark

```
procedure BENCHMARK TRAINING(epochs)
    monitor_client.start_benchmark()
    for i ← 1 to epochs do
        for X, y in batch do
            grad, loss ← await offload_training(X, y)
            monitor_client.batch_processed(batch_size(X))
        end for
    end for
end procedure
```

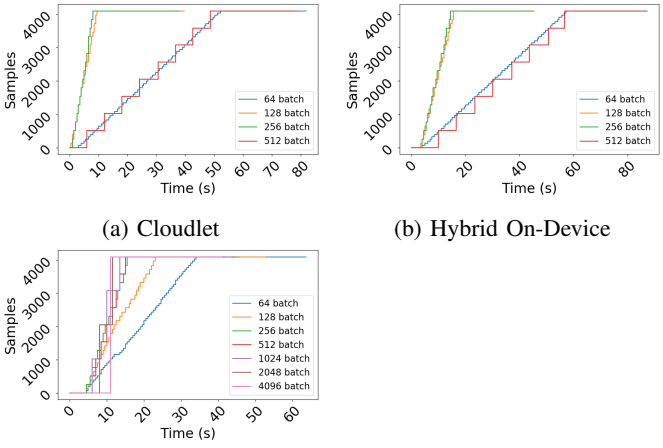
Algorithm 2 Data source inference benchmark

```
procedure BENCHMARK INFERENCE
    monitor_client.start_benchmark()
    for x ∈ samples do
        await offload_inference(x)
        monitor_client.batch_processed(1)
    end for
end procedure
```

Algorithm 3 Worker task completion benchmark

```
procedure RECEIVE TASK(input_reader)
    if ¬ benchmark_started then
        monitor_client.start_benchmark()
    end if
    input_data ← await input_reader.read()
    await process_task(input_data)
    monitor_client.task_processed()
end procedure
```

APPENDIX B
BATCH OPTIMIZATION FOR TRAINING



(a) Cloudlet

(b) Hybrid On-Device

(c) Hybrid Off-Device