Journal of Systems Architecture 118 (2021) 102214

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An open source framework based on Kafka-ML for Distributed DNN inference over the Cloud-to-Things continuum

Daniel R. Torres, Cristian Martín *, Bartolomé Rubio, Manuel Díaz

ITIS Software, University of Málaga, Arquitecto Francisco Peñalosa, 18, 29071 Málaga, Spain

ARTICLE INFO

Keywords:
Distributed deep neural networks
Cloud computing
Fog/edge computing
Distributed processing
Low-latency fault-tolerant framework

ABSTRACT

The current dependency of Artificial Intelligence (AI) systems on Cloud computing implies higher transmission latency and bandwidth consumption. Moreover, it challenges the real-time monitoring of physical objects, e.g., the Internet of Things (IoT). Edge systems bring computing closer to end devices and support time-sensitive applications. However, Edge systems struggle with state-of-the-art Deep Neural Networks (DNN) due to computational resource limitations. This paper proposes a technology framework that combines the Edge-Cloud architecture concept with BranchyNet advantages to support fault-tolerant and low-latency AI predictions. The implementation and evaluation of this framework allow assessing the benefits of running Distributed DNN (DDNN) in the Cloud-to-Things continuum. Compared to a Cloud-only deployment, the results obtained show an improvement of 45.34% in the response time. Furthermore, this proposal presents an extension for Kafka-ML that reduces rigidness over the Cloud-to-Things continuum managing and deploying DDNN.

1. Introduction

Artificial Intelligence (AI) and Deep Neural Networks (DNN) [1] are key contributors to autonomous decision and prediction processes in multiple domains, ranging from manufacturing systems to self-driving cars. Currently, thousands of data sources originated during the Internet era, such as the Internet of Things (IoT), provide data streams. Streaming data [2] feeds these application domains that, in the end, depend on Cloud platforms due to the large amount of data collected and processed. Cloud platforms [3] provide infrastructures and platforms as a service (IaaS and PaaS, respectively) to access computing, storage, and connectivity. Nonetheless, this dependency means a high transmission latency since data centers are located far from the end devices. Therefore, it challenges the real-time monitoring of physical objects featured in the IoT.

Architectures based on Edge and Fog computing represent promising alternatives that complement Cloud-based systems and make them more capable of ensuring a rapid response to emergencies. The purpose of the Fog computing paradigm is to extend the Cloud capabilities (storage, network, and computation services) and bring them closer to the edge of the network [4]. Fog systems can be considered a geographically distributed computing architecture connected to multiple heterogeneous devices (mini data centers, network devices, lightweight servers), that forms a bridge between the Cloud and the Edge so as to meet the time-sensitive requirements of IoT applications. Some authors claim that Edge computing can be interchangeable with Fog computing [5]. However, the key difference between these two paradigms may be seen in the location where the data processing is performed. In Fog computing, the processing is performed as close as possible to the IoT devices, while Edge computing pushes the limits even further by allowing connected gateways and IoT devices to process data locally. These two paradigms reduce bandwidth consumption and latency in a sequence of processing known as the Cloud-to-Things continuum [6]. The Cloud-to-Things continuum, shown in Fig. 1, is defined as a set of processing units, such as Edge devices and Fog servers. These processing units, located between the IoT and the Cloud, optimize response times and bandwidth consumption in time-sensitive applications. For instance, a deployment in this context could consist of IoT devices generating information connected to gateways or Edge devices and Fog servers processing before sending the information received to the Cloud.

Distributed DNN (DDNN) [7] combine DNN for complex pattern detection with a distribution of the DNN layers over the Cloud-to-Things continuum to optimize latency. For instance, in Structural Health Monitoring (SHM) [8], DDNN can assess the global state and detect structural problems of civil infrastructures. These mission-critical scenarios require minimal response latency for real-time evaluation of civil infrastructures and population safety. Although DDNN have evolved over

* Corresponding author.
E-mail addresses: drtorres@lcc.uma.es (D.R. Torres), cmf@lcc.uma.es (C. Martín), tolo@lcc.uma.es (B. Rubio), mdr@lcc.uma.es (M. Díaz).

https://doi.org/10.1016/j.sysarc.2021.102214
Received 17 January 2021; Received in revised form 31 May 2021; Accepted 9 June 2021
Available online 16 June 2021
1383-7621/© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

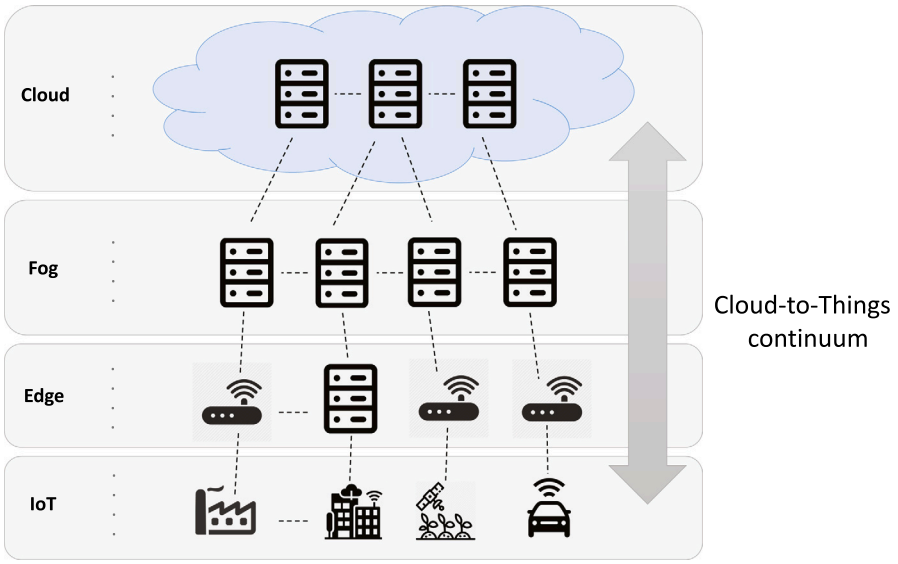

Fig. 1. Cloud-to-Things continuum.

the last few years, we argue that DDNN are still at a low technology readiness level since there is a lack of:

1. frameworks for the fault-tolerant distribution of DNN and the management and monitoring of DDNN over heterogeneous hardware,
2. effective communication layers to interconnect, discover, and communicate neural networks, and
3. solutions to manage AI pipelines from a DDNN model training until it is ready for inference.

To address the lacks mentioned in 1) and 2), in this article, a low-latency and fault-tolerant framework for enabling the flexible distribution of DNN over the Cloud-to-Things continuum is introduced.

The purpose of this framework is to manage and distribute DNN with fault-tolerant guarantees and provide adequate communication layers with low latency to interconnect them. This framework is designed considering container technologies, such as Docker [9], to facilitate rapid deployment and mobility with lightweight scaling and reallocation components, applications, and services. It also considers Apache Kafka, the present state-of-the-art solution for scalable dispatching of data flows.

The framework uses models based on BranchyNet [10], which provides a novel approach that promotes fast inference through early-exit DNN branches. These branches are complementary outputs located throughout a neural network structure. BranchyNet-based branches allow intermediary predictions that, when accurate enough, can make a DNN inference stop at that point, saving time by avoiding inference over the upper layers of a DNN model. As a result, during the inference process, all the DNN-model layers are processed only if none of the early exits in the model is accurate enough, i.e., the probability of being a class returned by the model for a prediction made by an early exit is higher than a specified threshold.

In this work, an extension of Kafka-ML [11] is also proposed to address the lack 3), by providing an open-source framework to manage and deploy AI pipelines using data streams.

The main contributions of this work are summarized as follows:

1. A framework that reduces location rigidness over the Cloud-to-Things continuum by managing and deploying DDNN applications,
2. effective communication layers to interconnect DDNN applications, and
3. the combination of the BranchyNet approach and the Edge-Cloud architecture concept, which shows an improvement of 45.34% of the response time compared to a Cloud-only deployment.

The rest of the paper is organized as follows. The motivation for this proposal is presented in Section 2. Section 3 introduces a background on Kafka-ML and the other main technologies used in this work. The low-latency and fault-tolerant framework is described in 4. In Section 5, the implementation and its evaluation are presented and discussed. Section 6 provides an outline of the related literature. Finally, Section 7 concludes the paper and explores future work lines.

2. Motivation

As was demonstrated when the Genoa (Italy) bridge collapsed in August 2018 [12], civil infrastructure failures and malfunctions can have terrible consequences for human lives and essential civil work activities. Therefore, SHM requires proper real-time monitoring and management. Detecting and predicting any damage or vulnerability is essential to protect the population before a disaster can occur. Consequently, a number of recent studies [13,14] have shown successful applications of Deep Learning (DL) techniques in this field. However, to the best of our knowledge, these techniques are normally deployed in a monolithic way (i.e., through non-distributed deep neural networks). Moreover, these techniques are not fit for use in time-sensitive systems as required in this context. One of the reasons for the absence of DDNN could reside in the lack of available architectures and frameworks for managing and deploying DDNN applications in the Cloud-to-Things continuum.

Furthermore, modern DNN require a considerable amount of computational resources [15–17]. This implies that Cloud and Edge systems must have sufficient hardware resources to allocate multiple instances of a DNN model and accept a large number of requests per minute from multiple devices. The (un)limited capabilities such as processing and storage of cloud systems to allocate these DNN models are well-known. However, the response latency present in the communications to these platforms is far from meeting the requirements of time-sensitive applications. For these reasons, DNN are being partitioned and distributed in heterogeneous hardware and multi-layered infrastructures.

DNN layers can extract complex patterns from high volume datasets consisting of images, as notably demonstrated by Convolutional Neural Networks (CNN) [18]. However, working with images can raise several

challenges. One is related to information privacy, mainly when involving sensitive data in domains like eHealth. Although some systems that deal with sensitive data only process private images without letting people access these images, their communication over the network (especially with the Cloud) can lead to security breaches, even if these communications are safely and reliably protected. To support data sensitivity, the DNN distribution allows the transmission of inferences resulting from the intermediate layers of a DNN model, which contain less sensitive information than raw images. Moreover, DNN can also notably reduce the image size [19] thanks to the image compression performed in some layers. Therefore, DDNN could also significantly reduce bandwidth consumption by exploiting this technique.

Finally, real-time applications require adequate communication layers to fully interconnect DDNN and architectures for unified management and deployment of DDNN. Moreover, as BranchyNet proposes, early exits return a response as soon as the accuracy is good enough to interact with the lowest levels of the continuum, providing real-time outcomes. Given the results of Kafka-ML [11] managing AI applications with data streams, we have envisaged an extension of this framework to overcome the management and distribution of DDNN in the Cloud-to-Things continuum.

3. Kafka-ML: managing AI pipelines through data streams

In this paper, an extension of Kafka-ML [11] is presented. Kafka-ML, which is available on GitHub,[1] is an open-source framework that allows the management of Machine Learning (ML) and AI pipelines through data streams. Kafka-ML aims to reduce the gap between data streams and current ML and AI frameworks providing an accessible framework to harmonize their full integration.

Kafka-ML makes use of the distributed message system Apache Kafka [20]. Apache Kafka is a distributed and publish/subscribe messaging system that dispatches large amounts of data at low latency. Apache Kafka enables multi-customer distribution, which allows the connection of multiple customers to topics (e.g., for distributing information to different layers like batch and stream platforms), and a high rate of message dispatching. One of its most notable features is the consumer group, which enables the distribution and parallelism of messages in a cluster of customers.

Contrary to many distributed queue frameworks, Apache Kafka stores messages in disk with a configurable retention policy, enabling its users to retrieve data later. This is popularly known as the *distributed log*, which allows consumers to search the log as they require. In some cases, such as ML training, this feature is especially useful as all data may need to be processed at once. If a failure occurs during this process, the customer can start again without losing any data stream or having to store it in a file system (as they are stored in the distributed log of Apache Kafka). In the case of adopting a queue framework that does not support a retention policy like Apache Kafka does, data streams should be at least stored into another data storage system before training is successfully performed to ensure there is no data loss. Therefore, Apache Kafka allows Kafka-ML to use a novel approach to manage data streams, which can be reused as many times as configured, so no file system or data storage are needed for datasets. Load balancing and fault tolerance among Kafka-ML tasks that require data streams, such as training and inference, is achieved through Kafka partitions and replicas of the topics. Each topic can be allocated into multiple partitions, and each partition can have multiple replicas for fault tolerance.

Regarding ML frameworks, Kafka-ML supports TensorFlow [21]. TensorFlow is an open-source framework with a flexible ecosystem of tools, libraries, and community resources for building and deploying ML-powered applications. Kafka-ML offers also an accessible and user-friendly Web interface (following a similar approach as AutoML initiatives) to manage ML and AI pipelines for both experts and non-experts users. As its main characteristic, Kafka-ML exploits containerization (Docker [9]) and container orchestration platforms (Kubernetes [22]) to facilitate the distribution of its components and the system load, and to provide high availability and fault tolerance.

Docker, container runtime of choice in the industry, automates the application deployment inside containers enabling the execution of applications on multiple architectures (x86, AMD64, ARM) and reducing development time (e.g., dependency problems) for developers and deployment teams. A cluster of machines running Docker containers is usually managed through a container orchestration system, such as Kubernetes, which enables distributed management and coordination while providing fault tolerance, vertical (in federation mode) and horizontal scaling, and high availability. Kubernetes is an open-source system for managing containerized applications in a cluster of nodes, easing both the management and deployment of containers and providing automation and declarative configuration. Kubernetes also enables continuous monitoring of containers, including Docker and its replicas, to ensure that they continuously match the defined status.

Kafka-ML users can write some code lines that define an ML model on the Web interface of Kafka-ML to start training, evaluating, comparing, and making inferences. The pipeline of an ML model in Kafka-ML representing its life cycle is shown in Fig. 2: (1) designing and defining the ML model; (2) creating a configuration of ML models, i.e., choosing a set of ML model(s) to be trained; (3) deploying the configuration for training using containers; (4) ingesting the deployed configuration with training and optionally evaluation data streams through Apache Kafka; (5) deploying the trained model for inference in the architecture presented in this work; and (6) feeding the deployed trained model for inference to make predictions with data streams. All the steps related to feeding the ML model (e.g., inference and training) use data streams. Each task executed is deployed in a Docker container in Kubernetes. Kafka-ML is used as the primary tool for training, evaluating and deploying ML models. The extension of Kafka-ML to enable DDNN and how DDNN communicate through Apache Kafka are further discussed next.

4. Distributed deep neural networks over the cloud-to-things continuum with Kafka-ML

Apache Kafka as a distributed message system is responsible for providing an effective and fault-tolerant communication layer for DDNN inference in the proposed framework. This architectural decision has the following features and benefits.

First, since Apache Kafka works with data streams, it allows this framework to accept and work with data streams as its normal functioning and opens the way for the integration of new ones, such as the ones present in the IoT and the Internet. Furthermore, architectures that use this framework can easily scale to increase the computing capacity when required (e.g., deploying replicas of DDNN applications) thanks to the Apache Kafka capabilities for parallelism (topic partitions and consumers groups), which would automatically distribute the data stream load among Apache Kafka customers (DDNN application replicas).

Second, the fault-tolerant mechanisms provided in Apache Kafka (e.g., topic replicas) enable fine and reliable control of data streams for DDNN deployments and reduce the risk of data loss. The Apache Kafka distributed log also ensures that streaming data are available (for a predefined time or until it exceeds the available memory) to DDNN applications even after they have been consumed. Thus, this also enables fault tolerance for DDNN applications.

Furthermore, IP addressing can be a challenge, especially when having a cluster of non-high-availability nodes as those present in the continuum. In this regard, Apache Kafka facilitates the discovery of

[1] https://github.com/ertis-research/kafka-ml.

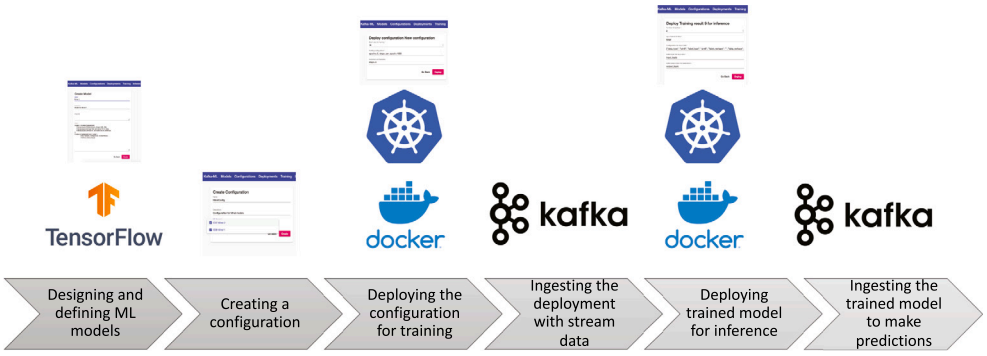

Fig. 2. ML/AI pipeline in Kafka-ML [11].

DDNN applications. They only have to know the Kafka topics where the prediction and inference results go (assuming Apache Kafka is deployed with fault tolerance and its IP is also known). Consequently, in case of a node failure (e.g., a DDNN application that waits for inference results), a new consumer subscription (along with the previous DDNN application) will be deployed in a transparent way for the DDNN application that sends the data stream (producer). Thus, Apache Kafka input and output topics have to be indicated when deploying DDNN applications. This enables high flexibility for the deployment of DDNN in this architecture, enabling the deployment of all partitioned models in one layer (e.g, the Edge or the Cloud) or in many layers as infrastructures are available in the continuum. Moreover, Kafka MirrorMaker functionality [23] enables a powerful and easy way to deal with topic synchronization among Kafka clusters as those present in the Cloud-to-Things continuum. To this end, the only requirement to be considered is that the inference of a DDDN hidden layer (non-early exits) must match the input of the next connected hidden layer in the global model definition.

As a result, the combination of DDNN and the Cloud-to-Things continuum can face the challenges presented in the Motivation section. Whereas Kafka-ML was designed for a single cluster infrastructure (e.g., a Cloud solution), this work provides a solution that allows applying this combination in the continuum. The framework has been developed to facilitate the inference of BranchyNet-based models over different heterogeneous infrastructures, such as Edge, Fog, and Cloud. Consequently, Kafka-ML now allows designing BranchyNet-based models. Therefore, Kafka-ML users can place an early exit in layers of the continuum (Fig. 3) to early stop the inference if the result is good enough (with a hit probability higher than a configurable threshold). Otherwise, the prediction is not considered reliable. In this case, the ML flow continues to the next layer in the continuum until getting a reliable prediction in the subsequent layers or reaching the last layer of the continuum (e.g., the Cloud). This allows generating predictions as close as possible to the lowest layer where a time-sensitive interaction can be required as long as the prediction is reliable according to a set threshold. This chosen threshold represents a trade-off between reliable and time-sensitive predictions. Furthermore, this framework makes the deployment of DDNN applications very flexible. The layered architecture depicted in Fig. 3 may be seen as a possible target deployment, one single layer (e.g., the Cloud), or as many layers as are available in the continuum thanks to the abstraction of the system through Apache Kafka.

To facilitate the deployment and development of DDNN applications and their portability and mobility over the continuum, this architecture, its components, and dependencies like Apache Zookeeper (required by Apache Kafka for synchronization of brokers and topic replicas) are containerized through Docker containers. This provides a portable and lightweight solution to be distributed in the continuum. Containerization also reduces development and deployment efforts since all dependencies and the source code itself are packed in containers, which can be easily deployed without dealing with the installation steps required in non-containerization deployments. Once an ML model has been defined and trained in Kafka-ML, a Docker container will be instantiated in the continuum infrastructure for each sub-model. These instances will download their corresponding trained model from Kafka-ML to start the inference process through data streams and Apache Kafka. Therefore, Docker containers (and DDNN applications) communicate each other through Apache Kafka and the topics configured. Those can also communicate each other in different clusters available in the Cloud-to-Things continuum. During training, another Docker container is deployed to train all the sub-models. For further details about these algorithms, please refer to Kafka-ML [11].

Finally, the orchestration container platform Kubernetes harmonizes the deployment of the containers that compose this architecture. Moreover, Kubernetes is used to manage a cluster of nodes that can be present in the continuum and offers other suitable features for mission-critical applications in production environments, such as fault tolerance and high availability. This allows unified management of DDNN applications and continuous monitoring to ensure a desire execution state in the available nodes. Therefore, fault tolerance and high availability are warranted for the data and control plane in this framework, through Apache Kafka with its management of data streams (data plane) and through Kubernetes with its management of the infrastructure and deployed Docker components (control plane). To avoid synchronization delays among internal clusters due to the strict requirements of consensus protocols, each available layer (e.g., Cloud, Edge, Fog) deploys an independent and isolated cluster of Kubernetes. All of these clusters can be centrally managed through the Kubernetes Cluster Federation[2] (KubeFed) functionality. Fig. 4 shows an overview of the architecture and its components deployed in two layers of the continuum (Edge and Cloud).

4.1. Time synchronization for DDNN applications

For time synchronization between DDNN layers, Apache Kafka is used. Since every message between DDNN applications is sent through Kafka (each DDNN layer has been configured with a Kafka input and output topic), all the communications in DDNN applications are available in Kafka for a predefined time or until it exceeds the available memory. Therefore, in the event that a DDNN layer has not been deployed before a message for it has arrived, messages for this layer will still be available in Apache Kafka until the layer can process them.

As DDNN applications are vertically layered, if any layer fails for whatever reason, it could also stop the whole flow of processing for a while. However, the adoption of Kubernetes and the component's isolation in Docker containers provides continuous monitoring of the available infrastructure to restart any component in case of failure.

[2] https://github.com/kubernetes-sigs/kubefed.

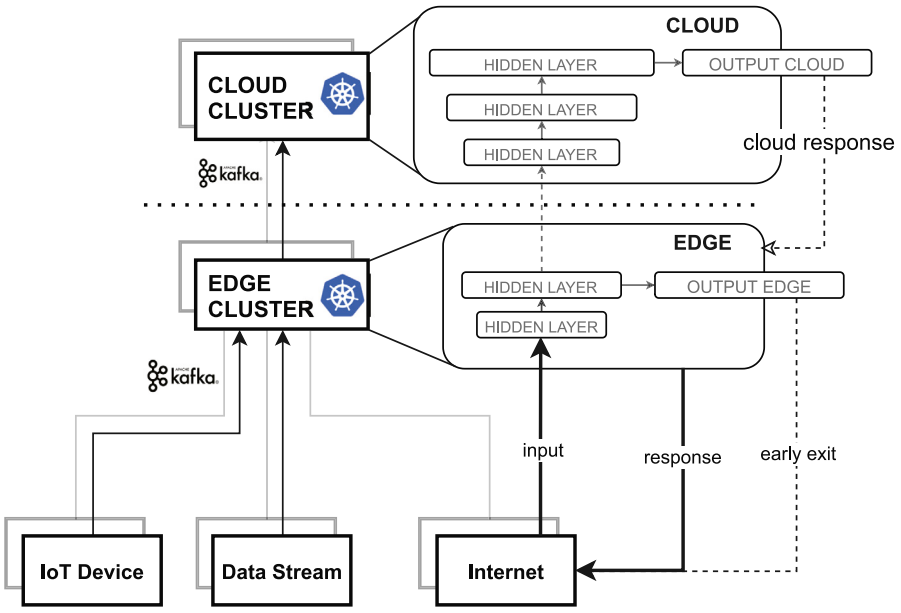

Fig. 3. Early exits and distribution of the DDNN framework in the Cloud-to-Things continuum. The Edge and Cloud contain different parts of a full neural model based on BranchyNet.

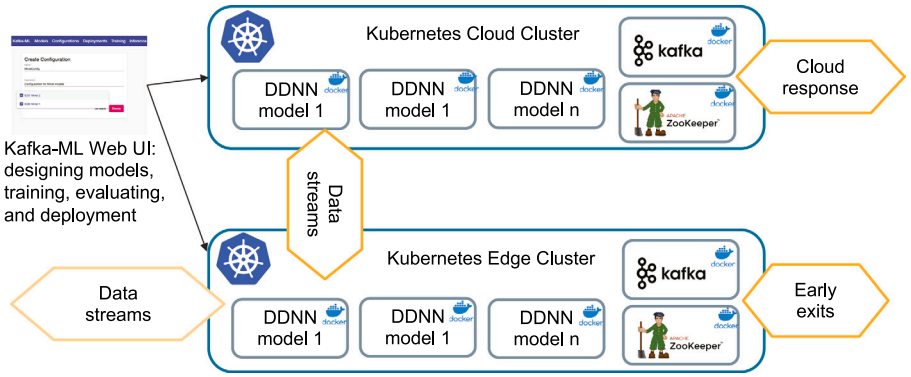

Fig. 4. Low-latency and fault-tolerant architecture for the management and deployment of DDNN applications over the Cloud-to-Things continuum based on Kafka-ML.

Moreover, the BranchyNet approach enables early exits for predictions and not always all layers have to process data streams. In this case, once a prediction is received from an early exit (i.e., hit probability is higher than a set threshold), the result is sent to the Kafka output topic configured and the DDNN communication flow ends.

5. Implementation and evaluation

5.1. Implementation

We have designed a DNN model based on VGG16 [17] and the early exit concept proposed by BranchyNet [10]. VGG16 gets a 92.7% top-5 test accuracy on ImageNet [24], and thus, we can successfully train a VGG16 model to classify images and expect more than 80% test accuracy on CIFAR10,[3] which is the dataset we have used during training and evaluation processes. Moreover, VGG16 is a well-known model and it can be easily adapted to the BranchyNet approach.

The complete resulting model is shown in Fig. 5. This model is large enough to place a sizeable workload on the Cloud, and thus, also in the Edge. In this model, we have included one early exit that corresponds to the `edge_output` layer. The branch for this early exit starts at the Flatten layer in the Edge and goes until the end of this branch, the `edge_output` layer. When placing an early exit, we have to consider the vertical traversal of the DNN stack before the start of this branch to add this early exit successfully. For instance, when designing a DNN model, we need to flatten the input of MaxPooling or Convolutional layers in order to work with Dense layers, since they work with different dimensionalities.

There is also another fact to consider when placing an early exit while following this framework, namely the additional memory required when adding a branch (i.e., including more layers) and thus the DNN model requires more resources. Although it is not a requirement, we recommend placing one early exit per architecture layer (e.g., Edge, Fog). In this way, we have one output for each layer in the architecture (e.g., Edge, Fog, Cloud) deciding whether or not to send to the upper layers of the architecture and saving in communications. For instance, if a prediction made at the early exit placed in the Edge system has a probability of being a predicted class higher than a specified threshold, this will result in the Edge system sending back this prediction to the devices rather than asking the upper layers, the Cloud in this case, for a more reliable prediction. Moreover, Edge systems have normally less available resources to allocate to a DNN model (i.e., GPUs, memory). Therefore, having an elevated number of early exits placed at the Edge level does not guarantee a faster response time. Those early exits that are placed too early in the model will not have enough prior layers in

[3] http://www.cs.toronto.edu/~kriz/cifar.html.

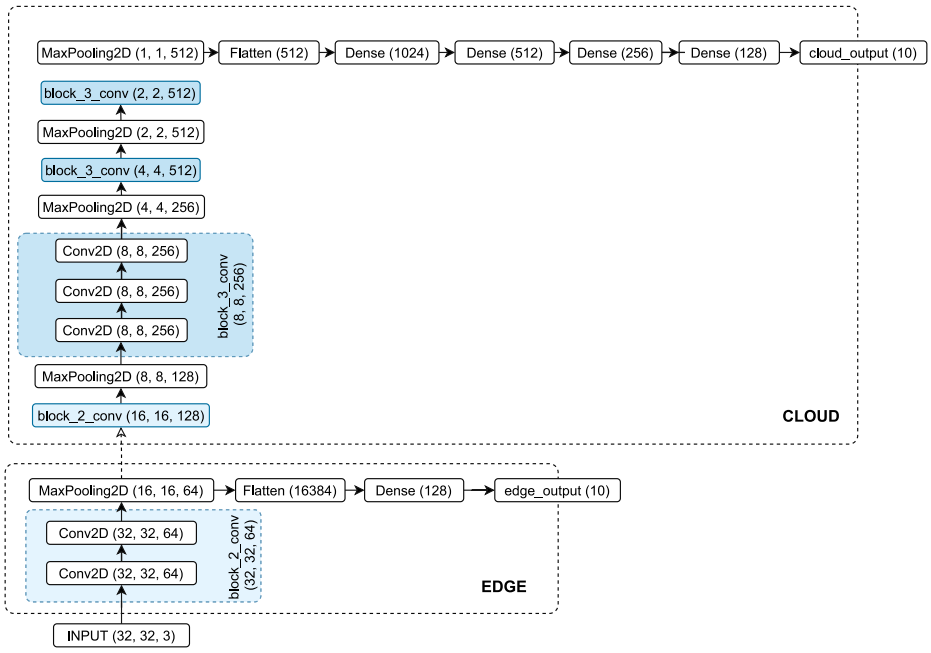

Fig. 5. DDNN model based on VGG16 and BranchyNet.

the model to provide a good prediction. Hence, we have to consider the DNN stack and the actual size of a model before introducing these branches for early exiting and partitioning the model. These aspects result in a trade-off between the accuracy and the size of the DNN model and sub-models generated once partitioned.

As a result, in our DDNN application we partitioned the model into two pieces, as shown in Fig. 5. Note that one of them is much smaller than the second one to be placed in computers that comprise an Edge cluster, while the larger one is to be placed in a Cloud cluster. We can then allocate these two parts of the model (i.e., two sub-models) in the different computers that built the evaluation environment, which will be introduced in 5.2, reaching a suitable accuracy level.

This model was trained using the CIFAR10 dataset, which consists of 60.000 32 × 32 color images in 10 well-balanced classes. First, the full model without the Edge exit was trained with 80 percent of the training set (40.000 32 × 32 color images; 10.000 images are used for the validation set). Then, all layers were blocked. The Edge exit, which consists of the Flatten, Dense and `edge_output` layers shown in Fig. 5, was added to train this branch in the same way as the rest of the model without affecting the weights already trained.

Once the model is trained (e.g., through Kafka-ML or a Jupyter notebook) and cut to the level of the early exist, each DDNN part can be deployed through Kafka-ML in different instances of this program running in different machines with Zookeeper, Kafka, and its topics correctly configured. As we include Kubernetes as management container platform, we have used Docker images for Apache Zookeeper[4] and Kafka[5] provided by third parties to easily configure them.

We have developed a container-based Python application to facilitate the inference of the BranchyNet-based model over different architectures levels by using the same technologies as Kafka-ML. These technologies are Kafka[6] and Tensorflow[7] [21]. This application is packed and executed in Kubernetes as a service, i.e., in a Docker container. Kafka-ML components (Fig. 4) are also packed in Docker containers and deployed in Kubernetes.

The code of implementation[8] and Kafka-ML[9] are both open-source projects. Furthermore, in the GitHub repository for this implementation, we have also shared the trained model based on VGG16 and BranchyNet for the sake of reproducibility, as well as the Docker and Kubernetes configuration files used during the evaluation of the proposed framework.

5.2. Evaluation

To evaluate this framework, we have used the value of 0.8 as a threshold. Selecting a threshold is a trade-off between reliable and time-sensitive predictions. As a result, we have chosen this threshold empirically due to most of the early exit responses, which return a probability higher than this threshold value, match the class returned in the Cloud. Hence, when the probability of being a class returned by the Edge early exit is higher than the specified threshold, the DDNN application in the Edge does not continue the inference process. Therefore, it does not send any images or information to continue the process in the Cloud. However, the predictions made by the sub-model placed in the Edge system may be less reliable than those made in the Cloud as the sub-model placed in the Cloud has more layers that comprise the DDNN. Note that this threshold can also be conveniently configured in the framework according to the needs of the target application.

Results are obtained using the CIFAR10 test set comprising 10.000 32 × 32 color images through an environment composed of 5 devices sending the same data at the same time, an Edge infrastructure, and a Cloud system. The Cloud deploys a Kubernetes cluster with 3 Nodes, 6 vCPUS, and 12 GB memory in Google Cloud. We have placed the Edge infrastructure at the University of Malaga, comprised of 3 computing nodes connected to the external streaming devices. These 3 computers form a cluster in Kubernetes and have different hardware configurations. The first of them works with an i7-4790 3.60 GHz and an 8 GB memory configuration. The second one has an i5-7400 3.00 GHz and 16 GB of memory, and the last of them works with an i7-10700 2.90 GHz and 32 GB of memory. The devices generating CIFAR10 data streams are deployed at the University of Malaga into 5 different computers

[4] https://github.com/31z4/zookeeper-docker.
[5] https://github.com/wurstmeister/kafka-docker.
[6] `kafka-python`: https://pypi.org/project/kafka-python/
[7] For both training and development, we have used Tensorflow 2.3.0.
[8] https://github.com/ertis-research/DDNN.
[9] https://github.com/ertis-research/kafka-ml.

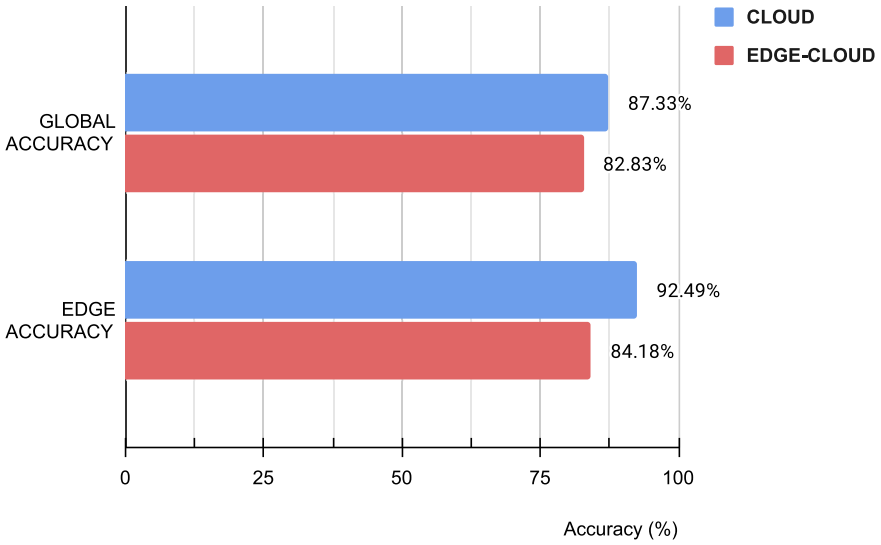

Fig. 6. Resulting accuracy for CIFAR10 test set using an Edge-Cloud architecture and an Cloud-only system.

with i7-4790 3.60 GHz and 8 GB of memory each one. Edge computers are then placed in the same network where the information is produced (University of Malaga) to reduce the network latency.

To sum up, this is the infrastructure used to deploy the DDNN application in the continuum:

- Devices deployed in 5 computers at the University of Malaga (Spain).
- Edge layer: 3 computers at the University of Malaga (Spain).
- Cloud layer: Google Cloud. Europe West 1 zone (Belgium).

We compare the results for the Edge-Cloud architecture with another architecture that depends only on the Cloud. In the Cloud-only case, the model used consists of the model shown in Fig. 5 as a single piece, i.e., without the branch for the early exit. First, we show the average accuracy for two different collections of the CIFAR10 test set in Fig. 6.

On the one hand, *global accuracy* shows the average accuracy for the whole CIFAR10 test set when using the complete model (i.e., without early exits) in the Cloud-only architecture and when using the model based on VGG16 and BranchyNet (Fig. 5), i.e., with early exits in the Edge-Cloud architecture. Despite the fact that the model evaluated is the same in both cases, in the Cloud-only case, the inference is processed using all the layers of the model, while early stopping (with an early-exit branch) in the Edge-Cloud case is considered. Therefore, as expected, the accuracy obtained in the Cloud-only case is higher than the accuracy obtained in the Edge-Cloud case. However, the results obtained by the Edge-Cloud architecture are still suitable enough.

On the other hand, we show the accuracy for the set of images that have been responded at the Edge level (early exit) of the Edge-Cloud architecture, 84.18%. This result is compared with the accuracy obtained in the Cloud-only architecture for the same set of images (92.49%). The values observed in this comparison show the main disadvantage of the BranchyNet approach. Using early exits does not increase the accuracy of the model, but decreases it while saving time during the inference process. This confirms the threshold trade-off discussed. Therefore, running a complete DNN model in the Cloud can return more reliable predictions, but considering early exits in an intermediate architecture layer (e.g., Edge) will reduce response time maintaining an acceptable degree of reliability (in this case, $> 80\%$).

Next, we have evaluated the framework comparing the Edge-Cloud environment with the Cloud-only environment in a Cloud-to-Things simple deployment by using the following configuration in both layers of the architecture (Edge on 1 computer and 1 node in Google Cloud) in order to measure the response time:

- 1× Apache Kafka broker (1 Computer)
- Kafka topics configured with 1 partition and without replication

For this test, results were obtained after sending all CIFAR10 test images from a single computer and shown in Fig. 7. As with accuracy, we show the average response time using two different sets. Using the simple deployment mentioned, the average Edge-Cloud response time is much higher than the Cloud-only architecture response time. The set of images that has been responded at the Edge corresponds to 5416 images, which are more than half of the CIFAR10 test set. Thus, the rest of the requests are much slower, since they have to go twice (for the request and response) through all the Cloud-to-Things continuum layers defined. This means that a Cloud-only architecture gives better response time results for the complete test set when using this simple deployment (with 1 computer in the Edge and a single node in Google Cloud). However, the Edge-Cloud architecture shows a significant improvement over the Cloud-only architecture regarding the response time (0.0638 s) when comparing the results for the set of images that has been responded at the Edge level. Thereby, more than 54% of the responses were provided in the Edge early exit that has reduced the Edge response time for these prediction requests.

Finally, we have evaluated the framework in a higher performance scenario. This scenario makes use of the described 3 Edge computers and the 3 nodes of Google Cloud. We have deployed a cluster of 3 Kafka brokers for each architecture level (i.e., Edge and Cloud) while varying topic replication and the number of replicas for high availability and fault tolerance. In this case, a replica of the DDNN inference module is deployed in Kubernetes with each partition to distribute the load of the system among 5 devices sending CIFAR10 data streams. Fig. 8 shows the response time of our framework versus a Cloud-only deployment.

As a result, with 1 partition, the total average Edge response time (including early exits and those that go to the Cloud) is higher due to the overhead of clients. However, with two partitions, the load is distributed among DDNN deployments, and results show lower response latency in our distributed framework than in the Cloud-only architecture. The overload in the Cloud system caused by the increase in the number of partitions and replicas may be due to the infrastructure available in Google Cloud (6 vCPUs and 12 GB of memory). Fig. 9 shows the speed-up of our architecture (early exits and total Edge time) regarding the Cloud deployment. The best result is obtained with 4 partitions and replicas, reaching a speed-up of 23× for early exits and 8× for total Edge response time.

These results demonstrate that considering early exits in DDNN combined with continuum architectures could drastically reduce response time, which is essential in AI and time-sensitive applications,

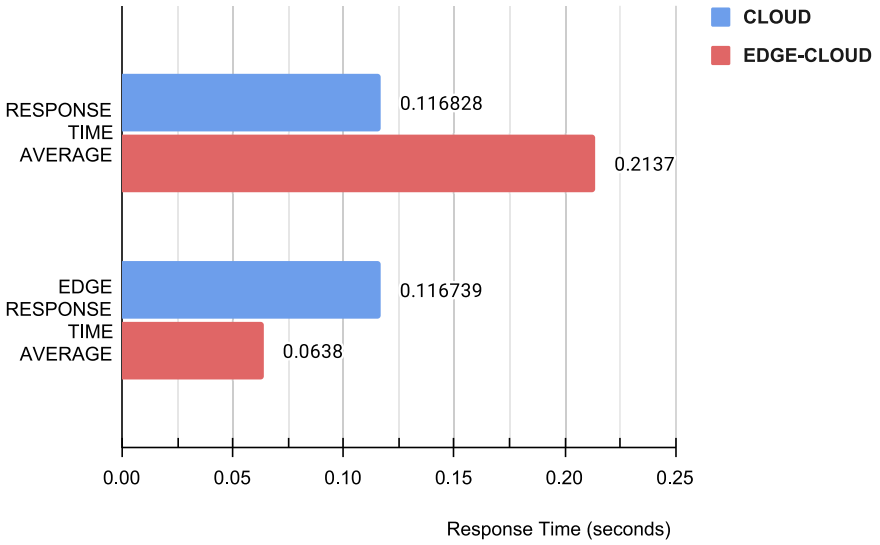

Fig. 7. Resulting response time of the distributed architecture versus an Cloud-only deployment ingesting CIFAR10 data streams. One Kafka Broker, 1 partition, and 1 client are used.

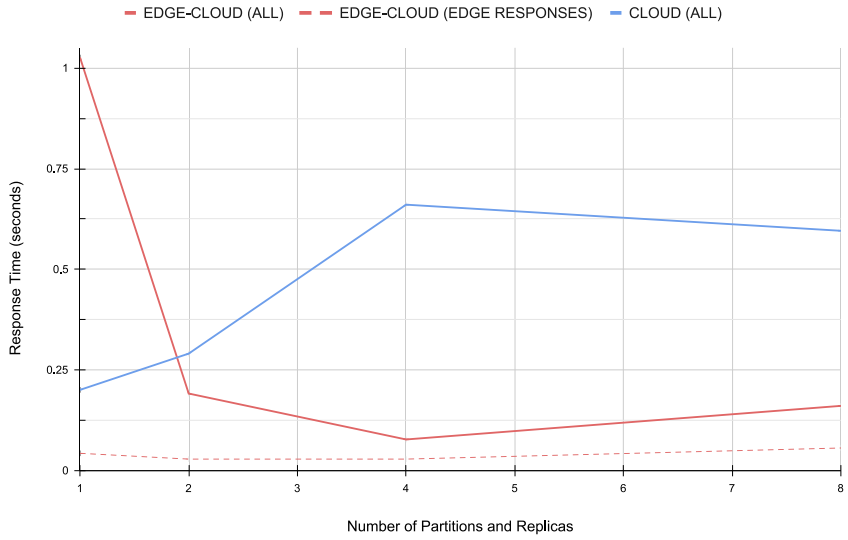

Fig. 8. Response time of the distributed architecture versus the Cloud-only deployment. Three Kafka Brokers and 5 clients are used. Partitions and replicas change.

such as self-driving, civil infrastructure monitoring, and eHealth. Despite the fact that the accuracy is slightly decreased due to early stopping during the inference process across the Cloud-to-Things continuum, it can be adjusted to satisfy the requirements of different scenarios by modifying the threshold value. Time-sensitive applications require working with response times in the order of milliseconds. Thus, the faster a system gives a prediction, the better its fitness for purpose.

6. Related work

6.1. DDNN partitioning

One of the first works on DNN branches for early exits was BranchyNet [10]. An extension of that work [7] proposed the adaption of BranchyNet to the Cloud-to-Things continuum through horizontal aggregations. This provides system fault tolerance with a 20× reduction of communication cost compared to offloading the whole computation to the Cloud. The main drawback of these approaches is that the architecture presented is statically deployed and DDNN applications do not adapt to the continuous changes. For instance, in case of a node failure in these architectures, a DDNN application would stop its service, whereas in our architecture, Kubernetes automatically would reallocate the DDNN application container into another available node to restart the service. An adaptive surgery [25] scheme dynamically splits DDNN between the Edge and the Cloud to optimize both the latency and throughput under variable network conditions. In the continuum, Fog can also play a role besides Edge and Cloud, and DINA [26] presents a fine-grained solution based on matching theory for dynamic DDNN partitioning in Fog networks. These approaches do not consider the instances when the inference stops at the middle layers (early exits), which can also reduce the network traffic [27] and the computing capacity [28].

Other studies show that response time is accelerated whilst network congestion is reduced by combining Cloud and Edge environments [29]. These approaches are also considered in video and image analysis in smart city applications, which entail a great computational

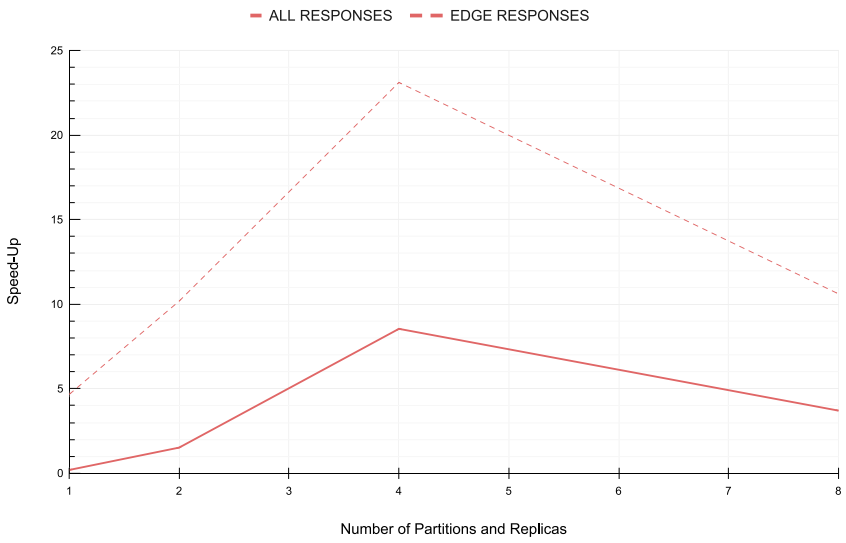

Fig. 9. The speed-up of the distributed architecture versus the Cloud-only deployment. Three Kafka Brokers and 5 clients are used. Partitions and replicas change.

cost and a huge number of data sent through the network, resulting in considerable delay reductions [30]. Yet, early exits at the middle layers are not contemplated, and they could result in a further improvement in response time and a significant decrease in the number of messages sent to the Cloud.

6.2. Cloud-to-Things continuum architectures for DDNN

As stated, Kubernetes enables the management and monitoring of all kinds of applications in a cluster of nodes. Kubeflow [31] is an open-source ML toolkit for Kubernetes. It does not provide support for data streams as the framework proposed and Kafka-ML do, but allows for the configuration of multiple steps of AI and ML pipelines, such as hyper-parameters and pre-processing. These solutions are not designed for the flexible support of DDNN over the Cloud-to-Things continuum.

EdgeLens [32] and HealthFog [33] provide frameworks to deploy deep learning-based applications in Edge-Fog-Cloud environments and improve the Quality of Service (QoS) for such applications. To reduce latency, EdgeLens scales down in resolution in order to shorten the delivery time. Both EdgeLens and HealthFog run non-distributed ML instead of adapting the DDNN themselves to the continuum.

IoTEF [34] provides a fault-tolerant architecture and unified management and monitoring for Cloud and Edge clusters; however, it does not allow the automation of DDNN in order to accomplish other QoS aspects, such as latency and inference optimization beyond fault tolerance.

7. Conclusion

This paper addresses the distribution of DNN over the Cloud-to-Things continuum for those mission-critical applications that require low-latency responses. It has been demonstrated that the partitioning of deep neural networks can be better adapted to the needs of the heterogeneous devices that host them and improve response times (early exits), security, and communication requirements. To the best of our knowledge, this work is the first to fully place the BranchyNet approach in the continuum providing a fault tolerance and low-latency framework; and an architecture and effective communication layers that offer better support to DDNN in this field. The proposed framework is open-source and available on GitHub.[10] The Kafka-ML extension, which addresses the comprehensive integration of data streams and ML frameworks and is also part of this work, is also available on GitHub.[11]

To accomplish the DNN distribution, especially in large DNN (e.g., ResNet-152), we partition the layers of the neural network. Apart from being a non-trivial task, as it constitutes a trade-off between computation and transmission costs, the partitioning of DDNN can have implications in the prediction accuracy, and the response latency could also be affected. Moreover, infrastructure capabilities may vary largely in heterogeneous and dynamic environments, which can also affect the pre-established partitioning strategy. Network conditions can also vary, e.g., the throughput can decrease by 10 times in LTE networks during peak hours [25]. Therefore, dynamic, fault-tolerant, and auto-adaptive partitioning strategies for DDNN inference acceleration should be released to ensure and maintain flexibility. We envisage that new micro-service components could be defined in Kafka-ML to continuously monitor the available continuum infrastructure [35] and dynamically decide where to cut and where to deploy DDNN applications to optimize the response latency. This will be explored as future work in order to adapt DDNN to the current status of infrastructures (hardware + networking) by allocating DDNN layers at the right place at the right time in the Cloud-to-Things continuum.

CRediT authorship contribution statement

Daniel R. Torres: Implementation of the framework and evaluation, First manuscript draft. **Cristian Martín:** Supervised the research and conceptualization, Kafka-ML development and conceptualization, First manuscript draft. **Bartolomé Rubio:** Supervised the research, Conceptualization, Manuscript review, Funding. **Manuel Díaz:** Supervised the research, Conceptualization, Manuscript review, Funding.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[10] https://github.com/ertis-research/DDNN.

[11] https://github.com/ertis-research/kafka-ml.

Acknowledgments

This work is funded by the Spanish projects RT2018-099777-B-100 (“rFOG:Improving Latency and Reliability of Offloaded Computation to the FOG for Critical Services”), PY20_00788 (“Integrados: Providing Real-Time Services for the Internet of Things through Cloud Sensor Integration”) and UMA18FEDERJA-215 (“Advanced Monitoring System Based on Deep Learning Services in Fog”). Funding for open access charge: Universidad de Malaga/CBUA. We express our gratitude to Google for their support and grant for their Cloud infrastructure.

References

[1] S. Makridakis, The forthcoming Artificial Intelligence (AI) revolution: Its impact on society and firms, Futures 90 (2017) 46–60.
[2] M.D. de Assuncao, A. da Silva Veith, R. Buyya, Distributed data stream processing and edge computing: A survey on resource elasticity and future directions, J. Netw. Comput. Appl. 103 (2018) 1–17.
[3] M. Díaz, C. Martín, B. Rubio, State-of-the-art, challenges, and open issues in the integration of Internet of Things and Cloud computing, J. Netw. Comput. Appl. 67 (2016) 99–117.
[4] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the Internet of Things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, August 13-17, Helsinki, Finland, ACM, 2012, pp. 13–16.
[5] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet Things J. 3 (5) (2016) 637–646.
[6] M. Chiang, T. Zhang, Fog and IoT: An overview of research opportunities, IEEE Internet Things J. 3 (6) (2016) 854–864.
[7] S. Teerapittayanon, B. McDanel, H.-T. Kung, Distributed deep neural networks over the cloud, the edge and end devices, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), June 5–8, Atlanta, GA, USA, IEEE, 2017, pp. 328–339.
[8] L. Alonso, J. Barbarán, J. Chen, M. Díaz, L. Llopis, B. Rubio, Middleware and communication technologies for structural health monitoring of critical infrastructures: A survey, Comput. Stand. Interfaces 56 (2018) 83–100.
[9] Docker, 2021, (accessed on 9 January 2021), Available online: https://www.docker.com.
[10] S. Teerapittayanon, B. McDanel, H.-T. Kung, Branchynet: Fast inference via early exiting from deep neural networks, in: 2016 23rd International Conference on Pattern Recognition (ICPR), Dec 04-08, Cancun, Mexico, IEEE, 2016, pp. 2464–2469.
[11] C. Martín, P. Langendoerfer, M. Díaz, P. Soltani Zarrin, B. Rubio, Kafka-ML: connecting the data stream with ML/AI frameworks, 2020, ArXiv Preprint arXiv:2006.04105.
[12] P. Milillo, G. Giardina, D. Perissin, G. Milillo, A. Coletta, C. Terranova, Pre-collapse space geodetic observations of critical infrastructure: the Morandi bridge, Genoa, Italy, Remote Sens. 11 (12) (2019) 1403.
[13] Y. Bao, Z. Tang, H. Li, Y. Zhang, Computer vision and deep learning–based data anomaly detection method for structural health monitoring, Struct. Health Monit. 18 (2) (2019) 401–421.
[14] X. Ye, T. Jin, C. Yun, A review on deep learning-based structural health monitoring of civil infrastructures, Smart Struct. Syst. 24 (5) (2019) 567–585.
[15] S. Ren, K. He, R. Girshick, J. Sun, Faster R-CNN: Towards real-time object detection with region proposal networks, 2016, arXiv:1506.01497.
[16] J. Redmon, A. Farhadi, YOLOV3: An incremental improvement, 2018, arXiv:1804.02767.
[17] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2015, arXiv:1409.1556.
[18] Z. Tang, Z. Chen, Y. Bao, H. Li, Convolutional neural network-based data anomaly detection method using multiple information for structural health monitoring, Struct. Control Health Monit. 26 (1) (2019) e2296.
[19] N. Krishnaraj, M. Elhoseny, M. Thenmozhi, M.M. Selim, K. Shankar, Deep learning model for real-time image compression in Internet of Underwater Things (IoUT), J. Real-Time Image Process. 17 (6) (2020) 2097–2111.
[20] Apache kafka, 2021, (accessed on 9 January 2021), Available online: http://kafka.apache.org/.
[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 265–283.
[22] Kubernetes, 2021, (accessed on 9 January 2021), Available online: https://kubernetes.io/.
[23] Mirrormaker 2.0, 2021, (accessed on 11 January 2021), Available online: https://cwiki.apache.org/confluence/display/KAFKA/KIP-382%3A+MirrorMaker+2.0.
[24] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, June 20-25, Miami, FL, USA, IEEE, 2009, pp. 248–255.
[25] C. Hu, W. Bao, D. Wang, F. Liu, Dynamic adaptive DNN surgery for inference acceleration on the edge, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, April 29-May 2, Paris, France, IEEE, 2019, pp. 1423–1431.
[26] T. Mohammed, C. Joe-Wong, R. Babbar, M. Di Francesco, Distributed inference acceleration with adaptive DNN partitioning and offloading, in: IEEE INFOCOM 2020-IEEE Conference on Computer Communications, April 6, Beijing, China, IEEE, 2020, pp. 854–863.
[27] R.G. Pacheco, R.S. Couto, Inference Time Optimization Using BranchyNet Partitioning, 2020, ArXiv Preprint arXiv:2005.04099.
[28] J. Zhou, Y. Wang, K. Ota, M. Dong, AaIoT: Accelerating artificial intelligence in IoT systems, IEEE Wirel. Commun. Lett. 8 (3) (2019) 825–828.
[29] L.A. Steffenel, M. Kirsch Pinheiro, C. Souveyet, Assessing the impact of unbalanced resources and communications in edge computing, Pervasive Mob. Comput. 71 (2021) 101321, http://dx.doi.org/10.1016/j.pmcj.2020.101321, http://www.sciencedirect.com/science/article/pii/S1574119220301450.
[30] A. Rocha Neto, T.P. Silva, T. Batista, F.C. Delicato, P.F. Pires, F. Lopes, Leveraging edge intelligence for video analytics in smart city applications, Information 12 (1) (2020) 14, http://dx.doi.org/10.3390/info12010014.
[31] Kubernetes, 2021, (accessed on 9 January 2021), Available online: https://www.kubeflow.org/.
[32] S. Tuli, N. Basumatary, R. Buyya, Edgelens: Deep learning based object detection in integrated IoT, fog and cloud computing environments, in: 2019 4th International Conference on Information Systems and Computer Networks (ISCON), Nov 21-22, Mathura, UP, India, IEEE, 2019, pp. 496–502.
[33] S. Tuli, N. Basumatary, S.S. Gill, M. Kahani, R.C. Arya, G.S. Wander, R. Buyya, Healthfog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated IoT and fog computing environments, Future Gener. Comput. Syst. 104 (2020) 187–200.
[34] A. Javed, J. Robert, K. Heljanko, K. Främling, IoTEf: A federated edge-cloud architecture for fault-tolerant IoT applications, J. Grid Comput. (2020) 1–24.
[35] S. Forti, M. Gaglianese, A. Brogi, Lightweight self-organising distributed monitoring of fog infrastructures, Future Gener. Comput. Syst. 114 (2021) 605–618.