Machine Learning Inference on Serverless Platforms Using Model Decomposition

Adrien Gallego
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
adrien.gallego@student.uva.nl

Uraz Odyurt
High-Energy Physics, Radboud University
Nijmegen, The Netherlands
Nikhef
Amsterdam, The Netherlands
uodyurt@nikhef.nl

Yi Cheng
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
y.cheng3@uva.nl

Yuandou Wang
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
y.wang8@uva.nl

Zhiming Zhao
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
z.zhao@uva.nl

Abstract

Serverless offers a scalable and cost-effective service model for users to run applications without focusing on underlying infrastructure or physical servers. While the Serverless architecture is not designed to address the unique challenges posed by resource-intensive workloads, e.g., Machine Learning (ML) tasks, it is highly scalable. Due to the limitations of Serverless function deployment and resource provisioning, the combination of ML and Serverless is a complex undertaking. We tackle this problem through decomposition of large ML models into smaller sub-models, referred to as *slices*. We set up ML inference tasks using these slices as a Serverless workflow, i.e., sequence of functions. Our experimental evaluations are performed on the Serverless offering by AWS for demonstration purposes, considering an open-source format for ML model representation, Open Neural Network Exchange. Achieved results portray that our decomposition method enables the execution of ML inference tasks on Serverless, regardless of the model size, benefiting from the high scalability of this architecture while lowering the strain on computing resources, such as required run-time memory.

Keywords

Serverless computing, Machine learning, Model decomposition, Inference

1 Introduction

Serverless has evolved into an emerging cloud computing paradigm for developers to run applications without managing or provisioning servers. This new cloud computing paradigm can scale to demand in real-time. Based on the incoming request volume, the cloud service provider monitors resource usage and scales the infrastructure up or down, as needed. This results in considerable cost savings for all parties involved and reduces the strain on developers.

With the rapid development of artificial intelligence, the application of Machine Learning (ML) has driven significant breakthroughs and has been widely adopted in numerous domains, such as image classification, speech recognition, and healthcare diagnostics [8, 9]. ML workloads usually consist of two main steps: model training and model inference. Model training tasks are often computation-intensive, requiring specialised hardware such as GPUs or Tensor Processing Units (TPUs) with powerful computing capacity. The inference process requires much less computational power, since it does not involve the recurrent adjustment of model parameters. ML practitioners often devote a lot of time to training and executing models on a distributed infrastructure like the cloud.

Serverless computing provides a compelling paradigm for ML tasks, inference in particular. However, deployment of complex ML workloads on Serverless environments has to address several constraints. First, the deployment package size for Serverless functions is a limitation. For instance, there are three mainstream ML frameworks: *TensorFlow 2.5.0*, with around 500 MB for the relevant Python library (without additional optional component such as GPU support) [15], *Keras 2.4.3*, as a standalone package, is rather lightweight (1 MB) [5], but since this library is part of TensorFlow, the presence of TensorFlow is also required, and *PyTorch 1.9.0*, with around 400 MB (without the additional optional components, such as GPU support) [12]. However, these frameworks might not be suitable for Serverless platforms, as their respective libraries exceed the maximum deployment package size, e.g., the maximum deployment package size of *IBM Cloud Functions* is 48 MB. Additionally, the limited amount of available runtime memory in Serverless environments is a major challenge, which can cause performance issues or cause the task at hand to crash, if not carefully managed. Yet another challenge is the limited execution time of functions on Serverless platforms. Lastly, functions must be designed within the maximum payload size and temporary storage capacity, as shown in Table 1.

To address the presented challenges, we propose a novel and efficient method for running ML models on Serverless platforms. This solution is based on the decomposition of ML models, which has been derived as a method to overcome the intrinsic limitations of Serverless offerings. The main contributions of this paper are as follows:

Table 1: Ranges of the principal limitations affecting the main proprietary Serverless platforms, AWS, Azure, GCP and IBM

Limitation	Range
Deployment package size	From 48 MB maximum for IBM Cloud Functions, up to 1 GB maximum for Azure Functions
Payload size	From not available for Google Cloud Functions, up to 5 MB maximum for IBM Cloud Functions
Temporary storage capacity	From 500 MB maximum for Azure Functions, up to 16 GB maximum for Google Cloud Functions
Memory	From 1.5 GB maximum for Azure Functions, up to 16 GB maximum for Google Cloud Functions
Execution timeout	From 10 minutes maximum for Azure Functions and IBM Cloud Functions, up to 60 minutes maximum for Google Cloud Functions

- We propose an efficient method to decompose ML models into slices of multiple layers by identifying and analysing the limitations of Serverless platforms.
- A model-agnostic implementation of the decomposition method and the demonstration of sliced ML models deployed on the AWS Serverless platform as a demonstration.
- Experimentation conducted considering three ML models with distinct characteristics to validate and analyse the performance of the method.

The remainder of this paper is presented as follows: In Section 2 we review existing related work, followed by Section 3, in which we propose an approach based on model decomposition, to deploy ML inference tasks on Serverless platforms. In Section 4, we validate the proposed approach and provide several experimental results. We discuss our solution's scope and relevant concluding remarks in Sections 5 and 6.

2 Related work

A heavy Machine Learning (ML) model can be trained on distributed infrastructure as a distributed workflow through the application of *data decomposition* and *model decomposition*.

In the data decomposition approach, each machine runs all the model layers but uses solely a subset of the data, e.g., DeepThings [19] uses a method called "Fused Tile Partitioning". The main downside of data decomposition is that the model must be executed entirely on a single machine, even though the provided data is smaller. In the case of model decomposition, the model is divided into multiple small sub-models. This allows a much smaller amount of memory required by a single machine to run the inference, which makes it a more viable solution for large models. This is achieved by splitting the layers and connections of the models, and chaining together the different inference results obtained by each machine. For instance, Hadidi et al. [3] describe a solution in which convolutional layer connections are divided and each layer is turned into a sub-task. Alternatively, MoDNN [7] deals with the VGG-16 model by dividing CNN layers and then fully connecting them. In other papers from Ishakian et al. [4] or Mahajan et al. [6], authors explore the suitability of Serverless environments for running ML inference jobs. The achieved conclusion was that latency introduced by cold starts of Serverless functions can sometimes be a limitation to this solution.

Open Neural Network Exchange (ONNX)[1] provides an efficient way for deploying and integrating distributed ML models. Marcus Turewicz describes how to use ONNX models in a Serverless environment using Azure Functions and .NET Core [17]. In this step-by-step guide, it is explained how to deploy an ONNX model as an Azure Function and how to use the function to classify images. However, this example only uses relatively small models that can fit in a single Serverless function. For heavier models, this might not be a viable solution. Another similar solution is presented by Christian Safka [13], who describes using ONNX with Python in AWS Lambda to perform inference on a ML model. The article provides instructions on how to convert a pre-trained model to ONNX format and how to deploy the function to AWS Lambda. Again, this solution is only applicable using relatively small ML models. Gopi Kumar presents a way of deploying PyTorch models on Azure Functions [10] by converting those PyTorch models into ONNX models via a native mechanism offered by the PyTorch library. The conversion from a native PyTorch model to ONNX is significantly relevant since it allows the deployment package footprint to be up to 10 times lower. This proves to be crucial for Serverless deployments since storage size is highly restrained in such an environment. Finally, another way is to use a cloud-based service that supports ONNX models, such as AWS SageMaker[2] or Azure Machine Learning[3], which allows you to deploy models to a cloud-based endpoint that can be invoked via an API. However, being vendor locked-in, those solutions are hardly reproducible for other public providers or open-source Serverless platforms.

The relevance of ML in the context of Serverless has been a controversial topic among industrial and research communities. For instance, its feasibility has mainly been discussed since ML applications often require extensive computing resources and memory to run correctly. As such, the Serverless paradigm has sometimes been described as unsuitable for this use case [2]. However, several researchers [18, 14] have argued that Serverless can prove to be beneficial for ML since users can overcome such issues by decomposing their model into sub-models, and by running them in a Serverless environment without the worry of managing the underlying infrastructure and benefiting from the high scalability.

[1] https://onnx.ai/
[2] https://aws.amazon.com/sagemaker/
[3] https://azure.microsoft.com/en-us/products/machine-learning

3 Methodology

We propose a model decomposition method to run a heavy ML inference as Serverless functions, as shown in Figure 1. We assume all the income requests are ML inference tasks.

Step 1 - Pre-processing. The pre-processing step is to convert the incoming inference model using ONNX. ONNX supports a variety of models, including neural networks, support vector machines, and decision trees, amongst others. It is noteworthy to mention that not all architectures are supported by ONNX, due to various reasons such as technical limitations, differences in model design principles, and compatibility issues. As examples of incompatible architectures, we can cite non-deep learning architectures, architectures with non-standard operations, or dynamic architectures, e.g., architectures with dynamic computation graphs that change during runtime, such as Recurrent Neural Networks (RNNs). This step checks the income requests to see if the required software libraries fit the function size required by the Serverless platform and the payload size of each layer of the neural network of the request. We explained in Section 1 that Serverless platforms often have restrictions on the function size and the function payload.

Step 2 - Decompose the model. Decompose the model into different layers based on the size of the payload. The idea is to process a pre-trained ML model composed of multiple layers through a *decomposer*. For example, if the model is composed of 10 layers, any number N between 1 and 10 will work. The decomposer then takes care of splitting the model by creating N slices. These slices can be viewed as shortened ML models with precise and restrained objectives. It will be possible to run them just like the entire model is executed, but their output will only be exploitable by the slice that follows and possibly not immediately succeeding slices in the case of parallel branching. After the data is correctly passed on from one slice to the next, the result provided by the last slice corresponds to the inference result of the entire model. This step is iterative with the next step.

Step 3 - Check the decomposition. Check the decomposed layers based on the size of required temporary storage and runtime memory. After the model has been decomposed into multiple slices, each slice will generate an output payload. For each slice, this payload is made of all the output payloads generated by the layers contained in the slice, which are needed by succeeding layers not in the current slice. In this verification step, we have to make sure that no slice produces a payload of a size greater than the payload size limit of the Serverless function or greater than the payload size limit of the Serverless workflow orchestrator. In case this maximum size is violated, a new slicing pattern is proposed and tested, and this step will have to reoccur until requirements are met.

Step 4 - Create functions. Create functions of decomposed models from the previous step. Each function will have access to its corresponding slice, either as part of the deployment package if size limitations are not exceeded, or as located in a cloud storage solution. Besides, with most cloud providers, functions will have access to the required libraries through the deployment package, even though other solutions might exist but remain highly specific to each platform.

Step 5 - Run the function flow. Run the created functions on the Serverless platform. The first function deals with the *Input workload*. This time, the input workload can be the same as in the local execution but can also be much larger. This output is often too heavy to be passed on directly to the next function. For example, in AWS Step Functions, which we have used in our implementation, payloads are limited to 256 KB. However, a relatively small decomposed model may already exceed several megabytes of data passed from layer to layer. For this reason, the solution presented in Figure 1 is to write the output payloads to a cloud data store. Since ML models are often not totally sequential, the fetched data can belong exclusively to the directly preceding function and previous ones. The last function produces the inference result.

The ONNX Python library offers a tool called *tf2onnx* [16] which can convert TensorFlow, Keras, TensorFlow.js and TensorFlow Lite (TFLite) models to ONNX. It is possible to achieve this step with a single terminal command or a single Python line of code. Other ML frameworks also offer ways of converting their models to ONNX format. For instance, PyTorch provides a solution to perform this export, as explained in their tutorial [11]. There also exists *caffe2onnx* [1] which is a Python package providing tools for model conversions from Caffe to ONNX. In our implementation, we focus on *tf2onnx* and make use of it to convert TensorFlow models to ONNX.

We prototyped the method based on *AWS Step Functions*. Each ONNX slice execution corresponds to one execution of a *Lambda function*. Since each slice requires at least one output from the previous slice, the slices have to run one after the other, i.e., sequentially. In addition, since all slices run with the same Python code, we deploy it to a single Serverless function on Lambda. Accordingly, the execution of a decomposed ML workflow consists in the execution of the same Lambda function multiple times, as many times as there are slices. The deployment to a single Lambda function makes our solution greatly scalable by fully benefiting from the high scalability offered by Serverless.

In AWS Step Functions, workflows are defined using *Amazon States Language*, a JSON or YAML-based language that allows you to specify various states, input/output data, and error handling logic of a workflow. Since we have designed our solution in such a way that only one Lambda function is needed, the definition of the workflow becomes straightforward. The workflow that meets the requirements of our solution is depicted in Figure 2.

The AWS Step Functions workflow is composed of four states, as elaborated below:

- **Init:** This state contains the definition of the first event and passes it to the next state.
- **<projectname>_<N>_slices:** Within this step resides the Serverless Lambda function. For the first slice execution, it takes as an input the first event provided by the Init state. The execution of the Lambda function returns an event as an output. The second and next times this state is entered, it takes as an input the output of the previous slice execution. The last slice execution returns an event containing the final inference result.
- **IsExecutionCompleted:** This state is responsible for checking the state of a boolean named *keep_going*, contained within

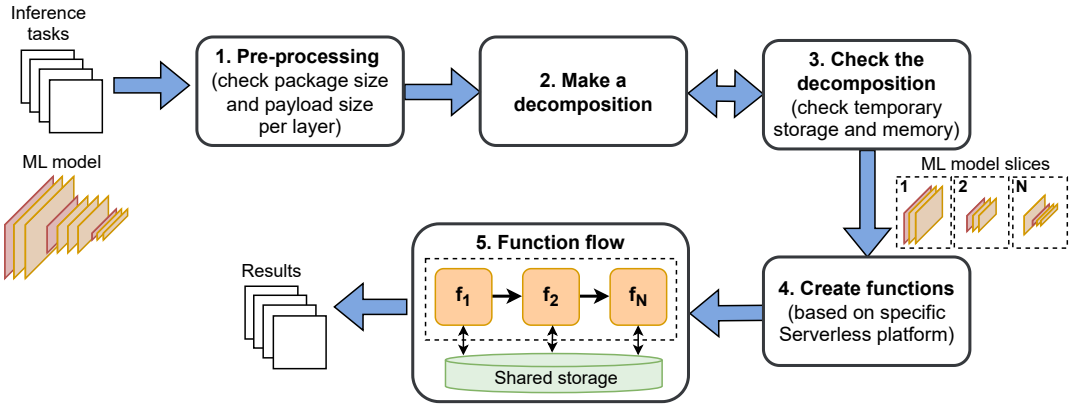

Figure 1: The high-level view of our proposed method to adapt large ML model inference tasks to Serverless functions, which are resource-constrained by design.

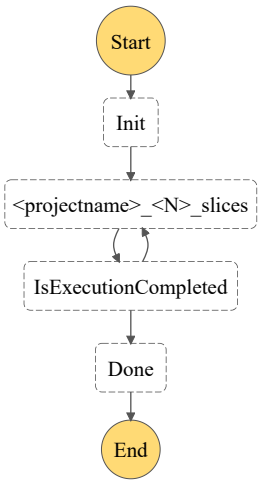

Figure 2: Graphical representation of the *AWS Step Functions* workflow responsible for a complete Serverless ML inference, defined in *Amazon States Language*

the event. As long as *keep_going* equals to true, the workflow keeps looping back to the previous state. When *keep_going* turns to false, meaning all slices have been executed, the workflow moves to the next state to provide the final result.

- **Done:** This state is responsible for displaying the final inference result.

4 Experimental results

Our experiments have been performed on AWS, using Lambda as the Serverless platform and Step Functions as the workflow orchestrator.

4.1 Impact of the slicing pattern on memory usage and execution time

Choosing an appropriate slicing pattern determines the number of slices that our model should be divided into. The following experiment aims to measure the impact of the number of slices on different metrics at our disposal. It involves the following steps:

- Selecting a model: MobileDet, ResNet50 or EfficientDet.
- Selecting several slice counts for the model slicing process: We have selected 2, 3, 5, and 10 slices. With slice counts above ten slices, due to the size of our models, we do not obtain significant improvements in any of the metrics and the execution time is considerably increased.
- Running inferences for each model and each number of slices: To obtain more reliable results, we have run each inference in batches of 10 concurrent inferences.

We have represented for the model MobileDet the evolution of the Lambda metric **maximum memory used**, as a function of the execution number, as shown in Figure 3. Each execution corresponds to one execution of the Lambda function. For each inference, the Lambda function is executed by Step Functions as many times as there are slices. Therefore, in the diagram, for each number of slices N, the total number of executions equals $10 \times N$, because we run batches of 10 concurrent inferences. For example, with a number of slices $N = 5$, we obtain $10 \times 5 = 50$ executions.

The first aspect we notice is that for the same sliced model, take MobileDet with $N = 5$ for instance, not all executions use the same amount of memory. Despite having the same amount of layers per slice, not all slices require the same memory to run. This is explained by the heterogeneous structure of ML models. When the number of slices increases, the maximum memory usage is considerably reduced. For example, with the model MobileDet, the most memory-intensive executions are toward the second half of the model, meaning that the final layers of the model are more memory-consuming than the initial ones. Memory usage, particularly of the final slices for this specific model, is considerably reduced when the number of slices increases. With ResNet50 and

EfficientDet, increasing the number of slices also directly impacts memory usage, decreasing it significantly. We deduce that, regardless of the model, a suitable way of lowering memory usage is to increase the number of slices.

In Table 2, as shown in the graphs from Figure 3, we note that as the number of slices increases, the peak of memory usage decreases. Between the non-sliced model and the 10-slice model, we observe a reduction in memory usage of 20% for MobileDet.

Table 2: Impact of the slicing pattern on memory usage and execution time for the model MobileDet

Number of slices	Peak of max memory used (MB)	Total execution time (ms)
1	220	4.925
2	200	7.538
3	190	11.785
5	178	15.379
10	175	25.799

However, the inference execution time is negatively affected as the number of slices increases. This is due to the nature of the inference process when the model is decomposed. Each slice has indeed to achieve multiple interactions with AWS S3[4], like fetching the ONNX sub-model or uploading and fetching intermediate results. These are the main causes of time overhead. Therefore, when selecting the number of slices, the user needs to find the right balance between memory consumption and execution time. To cite a few, this can depend on the limitations of the Serverless platform, the involved costs, or the requirements of the use case.

4.2 Cold start effect

Serverless platforms dynamically distribute resources to handle incoming requests. When a function is not being used, the platform will generally scale it down automatically to save resources and costs. The cold start latency is defined as the delay experienced when executing a function that has been inactive or scaled down to zero. To initialise a function, the platform needs to set up the execution environment, allocate the appropriate resources, and load the function code.

We have performed an experiment to identify the cold start effect affecting our implementation on AWS. To make sure the functions would be invoked cold, we have not run the function in the past 12 hours before the experiment. For each of our three ML models, with different slicing patterns, we have run each time 5 batches of 10 executions, with a 30-second pause between each batch. In all cases, we observed approximately the same delay for cold starts. To better identify the exact delay, it was required to measure it using the Lambda metric "Duration", because the Step Functions metric "TotalExecutionTime" is only provided as an average of all the last execution times. In addition, since not all slices are responsible for the same execution time, we have used a non-decomposed model to present the results, in order to improve the readability of the graph without affecting the results. The cold start resulting in the execution of an inference is indeed, in any case, identical whether it be a sliced model or a non-sliced model. We have represented the execution times of the corresponding Lambda function for the model *EfficientDet* in its non-decomposed version, after running inferences in 5 batches of 10 executions, with a 30-second delay between each batch invocation. These execution times are plotted in Figure 4

We observe a significant decrease in duration starting from the second batch. With the second, third, fourth, and fifth batches, the average execution time decrement compared to the first batch is 1.1 seconds, which is the delay we observed for all three ML models, regardless of the slicing pattern.

[4]https://aws.amazon.com/s3/

5 Limitations and Scope

Certain techniques applied in our implementation are not generic since particular aspects, such as the management of payloads or the storage of ONNX slices, have been designed with the selected cloud provider in mind, AWS. However, as detailed in our comparison of Serverless platforms, many limitations are rather similar amongst different platforms. Additionally, each cloud provider offers services and techniques corresponding to the ones available for AWS, with some variations.

Another threat to validity concerns the ML models we have used to validate our solution and run performance tests. For instance, a more comprehensive range of models could have been selected from other ML frameworks such as PyTorch or Caffe and with other application domains such as natural language processing or speech recognition. Although ONNX is a standardised format for representing ML models from various frameworks, particular features offered by specific frameworks are not always preserved after the conversion to ONNX. Despite these limitations, our solution remains greatly model-agnostic.

6 Conclusion

Running ML tasks on Serverless presents a unique set of challenges, which needs to be negotiated carefully to harness the potential of these two technologies and achieve seamless integration. We have investigated the adaptation of ML models to the Serverless cloud computing model. The objective was to achieve the deployment of such models on Serverless through the decomposition of the larger model into sub-models. Our method was specifically for running inference tasks. Using our implementation, we have demonstrated the applicability of our methodology to a real-world scenario using a specific public cloud provider, namely AWS. Finally, we have further elaborated the impact of our method with a performance analysis, considering three ML models of distinct characteristics. We can conclude that our method is capable of reducing resource utilisation in a targeted fashion, allowing for steered trade-offs between different resource constraints.

Acknowledgments

This work has been partially funded by the European Union's Horizon 2020 research and innovation program through the project CLARIFY (860627) and the BLUECLOUD 2026 project (101094227), by the Dutch NWO Large-scale Research Infrastructure – National

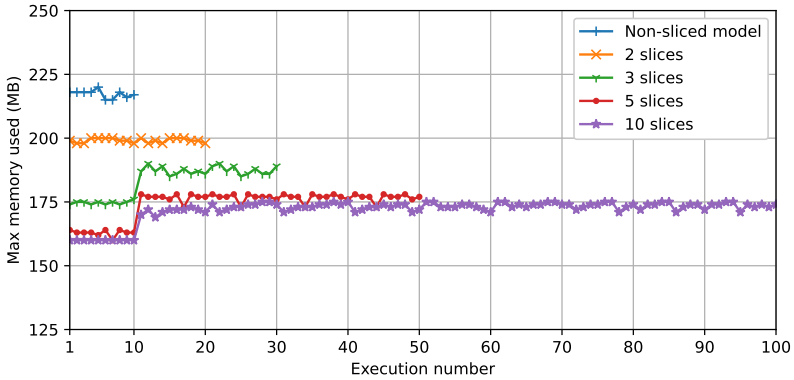

Figure 3: Impact of the slicing pattern on memory usage for the model MobileDet is shown. Note the trade-off between the maximum memory utilisation and the required execution count, i.e., execution time.

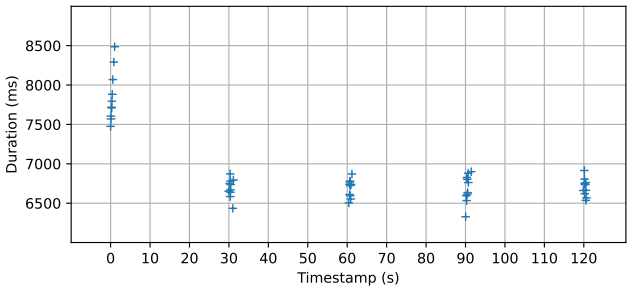

Figure 4: Identification of the cold start effect in the case of EfficientDet

Roadmap Consortia through the LITE-LIFE project and by the Life-Watch ERIC.

References

[1] 2021. Caffe2onnx 2.0.1. Retrieved Oct. 30, 2023 from https://pypi.org/project/caffe2onnx/.

[2] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 334–341. doi:10.1109/CLOUD.2018.00049.

[3] Ramyad Hadidi, Jiashen Cao, Michael S. Ryoo, and Hyesoon Kim. 2020. Toward collaborative inferencing of deep neural networks on internet-of-things devices. *IEEE Internet of Things Journal*, 7, 6, 4950–4960. doi:10.1109/JIOT.2020.2972000.

[4] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 257–262. doi:10.1109/IC2E.2018.00052.

[5] Keras. 2023. Keras api reference. Retrieved Oct. 30, 2023 from https://keras.io/api/.

[6] Kunal Mahajan and Rumit Desai. 2022. Serving distributed inference deep learning models in serverless computing. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 109–111. doi:10.1109/CLOUD55607.2022.00029.

[7] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 1396–1401. doi:10.23919/DATE.2017.7927211.

[8] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. 2021. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32, 2, 604–624. doi:10.1109/TNNLS.2020.2979670.

[9] Myeongsuk Pak and Sanghoon Kim. 2017. A review of deep learning in image recognition. In *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, 1–3. doi:10.1109/CAIPT.2017.8320684.

[10] PyTorch. 2020. Efficient serverless deployment of pytorch models on azure. (2020). Retrieved Oct. 30, 2023 from https://medium.com/pytorch/efficient-serverless-deployment-of-pytorch-models-on-azure-dc9c2b6bfee7.

[11] PyTorch. 2023. Exporting a model from pytorch to onnx and running it using onnx runtime. Retrieved Oct. 30, 2023 from https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.

[12] PyTorch. 2023. Pytorch documentation. Retrieved Oct. 30, 2023 from https://pytorch.org/docs/stable/.

[13] Christian Safka. 2021. Onnx inference with python in aws lambda. Retrieved Oct. 30, 2023 from https://becominghuman.ai/onnx-inference-with-python-in-aws-lambda-f09d08530e87.

[14] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Comput. Surv.*, 54, 11s, Article 239, (Nov. 2022), 32 pages. doi:10.1145/3510611.

[15] TensorFlow. 2023. Tensorflow documentation. Retrieved Oct. 30, 2023 from https://www.tensorflow.org/api_docs.

[16] 2023. Tf2onnx. Retrieved Oct. 30, 2023 from https://github.com/onnx/tensorflow-onnx.

[17] Marcus Turewicz. 2020. Serverless image classification with onnx, .net and azure functions. Retrieved Oct. 30, 2023 from https://www.marcusturewicz.com/blog/serverless-image-classification-with-onnx-dotnet-and-azure-functions/.

[18] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. 2022. λDnn: achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, 71, 2, 450–463. doi:10.1109/TC.2021.3054656.

[19] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37, 11, 2348–2359. doi:10.1109/TCAD.2018.2858384.