Auto-CNNp: a component-based framework for automating CNN parallelism

Soulaimane Guedria[1,2], Noël De Palma[1], Félix Renard[1,2] , and Nicolas Vuillerme[2,3]

[1]Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2]Univ. Grenoble Alpes, AGEIS, Faculté de Médecine, 38700 La Tronche, France
[3]Institut Universitaire de France, Paris, France
*E*mail: {soulaimane.guedria, noel.depalma, felix.renard, nicolas.vuillerme}@univ-grenoble-alpes.fr

***Abstract*—Effectively training of Convolutional Neural Networks (CNNs) is a computationally intensive and time-consuming task. Therefore, scaling up the training of CNNs has become a key approach to decrease the training duration and train CNN models in a reasonable time. Nevertheless, introducing parallelism to CNNs is a laborious task in practice. It is a manual, repetitive and error-prone process. In this paper, we present Auto-CNNp, a novel framework that aims to address this challenge by automating CNNs training parallelization task. To achieve this goal, the Auto-CNNp introduces a key component which is called *CNN-Parallelism-Generator*. The latter component aims to streamline routine tasks throughout (1) capturing cumbersome CNNs parallelization tasks within a backbone structure while (2) keeping the framework flexible enough and extensible for user-specific personalization. Our proposed reference implementation provides a high level of abstraction over MPI-based CNNs parallelization process, despite the CNN-based imaging task and its related architecture and training dataset. We introduce the design and the core building blocks of Auto-CNNp. We further conduct an extensive assessment of our proposal that not only shows its effectiveness in accelerating the process of scaling up CNNs training, but also its generalization for a wider variety of use cases.**

I. INTRODUCTION

Deep neural networks (DNNs) and particularly convolutional neural networks (CNNs) trained on large datasets are getting great success across a plethora of paramount applications such as medical image analysis [1], [2], speech recognition [3], video processing [4] and many other interesting fields. These recent deep learning breakthroughs have been achieved thanks to the increasing computing power that is available nowadays. However, training CNNs is time- and resource-intensive task. In a typical CNN training process, multiple CNN architectures have to be investigated. Concurrently, an hyperparameters optimization process [5] has to be performed for every CNN candidate architecture. The hyperparameters optimization task aims to select the optimal set of hyperparameters in order to optimize the CNN performance. It involves performing various hyperparameters optimization strategies [6] which generally require executing multiple training runs. For instance, training GoogleNet with the ImageNet dataset requires a total of 21 days using a single Nvidia K20 GPU [7]). Therefore, decreasing the training duration of CNNs throughout scaling up the training process has become one of the most active areas of research making deep learning converge to high performance computing (HPC) problems. Nonetheless, setting up distributed training of CNNs is a tedious task entailing a significant degree of experience and expertise in both (1) deep learning and (2) distributed optimization approaches. Moreover, introducing parallelism to CNNs training is a manual, redundant, time-consuming and error-prone process. For instance, even though *Tensorflow* natively includes a standard built-in parallelization approach[1], going distributed using it is a laborious and challenging task [8]: It requires a large amount of knowledge from the user of a considerable low level abstractions of *Tensorflow* and a lot of manual code modifications. The aforementioned problems led us to realize the importance and the need for not only automating routine tasks to avoid duplication of effort while scaling up CNNs training, but also to adopt a component reuse approach while considering the software extensibility principle.

We alleviate the aforementioned issues by introducing Auto-CNNp (Automatic CNN parallelization), a component-based framework that fully automates scaling up CNNs training task in order to bring skill-intensive distributed deep learning to non-experts users. As far as we know, the present work is the first that aims to tackle this challenge by introducing a new component-based approach. We present the design and the core building blocks of our proposed framework. The *CNN-Parallelism-Generator* component encapsulates and hides typical CNNs parallelization routine tasks while being extensible for user-specific customization. The user defines the specific framework behavior through an easy-to-understand configuration file.

Our contribution lies within the proposal of a standard component-based approach to parallelize CNNs training regardless of the (1) CNN-based image processing task, (2) its corresponding CNN architecture and (3) training dataset. Furthermore, although our proposed Auto-CNNp Proof-of-Concept (POC) reference implementation is based on both (1) *Ring-Allreduce* parallelism approach and (2) *MPI* communication protocol, it is indeed possible to port the framework to additional CNN parallelism and communication

[1]More information on distributed *Tensorflow*: https://www.tensorflow.org/guide/distribute_strategy

approaches as the framework's fundamentals remain valid.

The evaluation result of our proposed automated component-based approach on a couple of medical imaging segmentation [9] use cases are promising. It shows that a significant speedup in the CNN parallilization task has been achieved to the detriment of a negligible framework execution time, compared to the classic manual parallelization strategy.

This paper is organized as follows: Section II provides background information on distributed deep learning training approaches and reviews some related work. Section III describes our approach to automatize the parallelization of our POC MPI-based CNNs training. We present the evaluation of our proposal in Section IV, and conclude in Section V.

II. BACKGROUND AND RELATED WORK

In this section, we first introduce some basic background terminologies. After that, we present the main approaches for distributed DNN training, before reviewing some related work.

A. Terminology

Throughout this paper, the term 'CNN architecture' refers to the global structure of the neural network (i.e., the number, order, size, etc., of each network's layer). The term 'hyperparameter' refers to a variable which is required to be defined before the CNN training task begins. Also, the term 'CNN model' denotes the output of the training process of a specific CNN architecture on a particular training dataset and hyperparameters.

B. Distributed Training of Deep Neural Networks

Distributed training approaches of DNNs are mainly divided into three different categories: model, data and hybrid parallelism techniques.

1) Model Parallelism: Some deep neural network models have a considerable size, and hence, they are not adapted to the memory size of an individual training device (one GPU for instance) [10]. These models require to be partitioned across all the nodes in the distributed system and every node trains a different part of the model on the whole training dataset. For instance, as can be seen in Figure 1, every node performs the training of only a specific subset of the model. This parallelization schema is known as *model parallelism* technique [11]–[13].

2) Data Parallelism: The second distributed training strategy of deep neural networks is called *data parallelism* approach. As illustrated in Figure 2, all nodes in the distributed system in have the same complete copy of the model. However, the training is done independently on each node using a different subset of the whole training dataset, at the end of every training iteration, the results of computations from all the nodes are combined using different synchronization approaches [11]–[13].

3) Hybrid Parallelism: It is possible to combine both previously mentioned distributed training approaches (i.e. model parallelism for every node and data parallelism across nodes [13]). However, data parallelism is the most used parallelization strategy in practice [13]–[17].

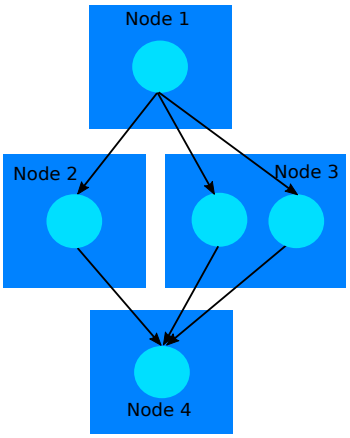


Fig. 1. Model parallelism

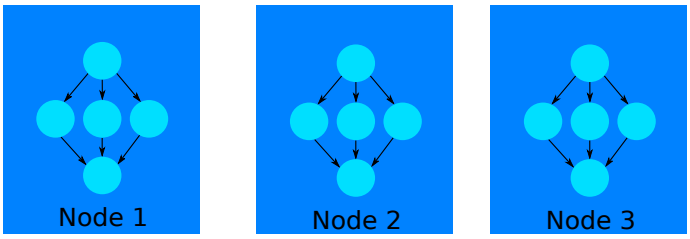


Fig. 2. Data parallelism

4) Other parallelism approaches: The Stochastic Gradient Descent (SGD) optimization algorithm (i.e., which is a variant of the gradient descent [18] optimization algorithm) is widely-used for the distributed deep learning training process. It is also important to denote that the subsequent introduced fundamentals for the DNN parallelism using SGD remain valid also for a set of popular optimization algorithms (e.g., Adam [19]). Actually, scaling up the training of DNNs evolves around strategies that aim to parallelize the computation and synchronization of the gradient during the SGD [18] optimization method. Hence, distributed training of DNNs approaches can be also roughly classified depending on the model consistency synchronization strategies [13] (e.g., synchronous [20], stale-synchronous [21] and asynchronous [12] techniques) and the parameter distribution and communication centralization methods [13] (e.g., Parameter Server (PS) [22], Shared PS [23] and decentralized strategies [24]).

C. Related Work

Component-based Software engineering (CBSE) [25] is far from being a recent research area. Indeed, it aims to build software systems by composition of software components building blocks. It has become a paramount approach to accelerate the development, deployment, management of large and complex software systems. The *component-based* approaches have been adopted in a wide range of relevant fields of applications, such as e-commerce [26], robotics software [27] and web applications development [28].

Component-based parallel systems development is a not a novel concept neither. Bramley et al. [29] introduced a component-based approach to build scientific and engineering applications. Also, *COMDES-II* [30] is a framework to develop parallel real-time control applications.

Other parallel systems implementations tools exist. For instance, *JaSkel* [31] is a Java framework for parallel and grid applications implementation. It shares some common concepts with Auto-CNNp. Particularly, encapsulating recurring parallelism routines and hiding low-level implementation details. Yet, JaSkel is not a component-based system.

The previously cited systems are not DNN-based solutions. However, With the recent growing interest to deep learning, a lot of distributed deep learning frameworks have emerged (e.g., TensorFlow, Horovod[2], DL4J[3], BigDL[4]). Nevertheless, to the best of our knowledge, no existing solution offers all the features of Auto-CNNp. In particular:

- Auto-CNNp adds an additional high level of abstraction over MPI-based CNNs parallelism techniques by fully automating the scaling up process for various CNN-based image processing task, regardless of its corresponding CNN architecture and training dataset.
- Our proposal is the first easily extensible component-based deep learning parallelism framework.

Hence, our proposed framework accelerates the research in the CNN-based field by prototyping and exploring cutting-edge and not yet investigated CNN configurations and architectures through an iterative and adaptive experimentation approach.

III. System description

This section describes Auto-CNNp our proposed framework. First, we provide a global overview on Auto-CNNp main scope, design and system architecture, before diving into the details of its building blocks and core components.

A. Framework Scope

Figure 4 pinpoints the overall scope into which Auto-CNNp operates. Indeed, the operating-system-level environment deployment on the training nodes is currently out of scope of the Auto-CNNs framework. We suppose that the training is performed on an all-set, already deployed distributed system environment (i.e., in the context where the operating system was already sat up on beforehand using tools like *SaltStack* [5] or *Puppet* [6]).

Also, we take advantage of a containerization technique to package the distributed deep learning application with its related default runtime environment (i.e., libraries, binaries and dependencies). It is indeed within this specific range of execution context where our proposed framework operates. Particularly, Auto-CNNp provides a backbone for a new way to *automatically configure and customize* the specific libraries of a distributed CNN-based application runtime environment (i.e., mainly by (1) setting up the configuration of communication libraries and (2) establishing the related deep learning user-specific execution schema).

Regarding the distributed deep learning application level, and as stated previously in section II, multiple CNN parallelism approaches exist. We decided to adopt a *decentralized synchronous Ring-Allreduce data parallelism strategy* for our proposed reference implementation for the following reasons:

- Considering that the level of scalability of the data parallelism method is naturally determined by the minibatch hyperparameter size [13], and since recent published works [15], [17] have succeeded to considerably increase the minibatch size without significant segmentation accuracy loss, data parallelism has become the most common distributed training approach when the the model size complies with the training device's memory size constraint.
- We decided to adopted a synchronous parallelism approach. Our selection criterion for the latter chosen strategy is the trade-off between the CNN model accuracy and the training speedup. In fact, synchronous methods achieve better results regarding the accuracy of the CNN models compared to the asynchronous approaches [13], [32], particularly, with a short synchronization period [33].
- The *Ring-Allreduce* algorithm (see Figure 3) is built on a HPC approach proposed in 2009 by Patarasuk and Yuan [34]. It is a highly scalable and bandwidth optimal approach as it remarkably reduces the network communications overhead [8]. Moreover, Since the network bandwidth is classified among the rarest resources in datacenters [35], and even if the centralized parameter server is one of the popular approaches in distributed machine learning with better fault tolerance, it suffers from a bandwidth bottleneck especially with large scale systems [13], [35].

Also, we adopted MPI as a communication protocol for the framework reference implementation. Indeed, MPI communication libraries have achieved remarkable performances in distributed deep learning applications due to the similar characteristics between distributed deep learning and HPC applications [13].

Nevertheless, as stated previously, it is possible to **port and extend** the framework implementation to support other parallelism approach and communication mechanisms as the framework's core principals remain well-founded. However, further modifications should be applied due to the eventual dependencies between the CNN training parallelism methods and the adopted communication protocols. These dependencies will be further discussed in subsection III-D.

[2]https://eng.uber.com/horovod/
[3]https://deeplearning4j.org/
[4]https://bigdl-project.github.io
[5]More informations can be found at https://www.saltstack.com/
[6]More informations can be found at https://puppet.com/

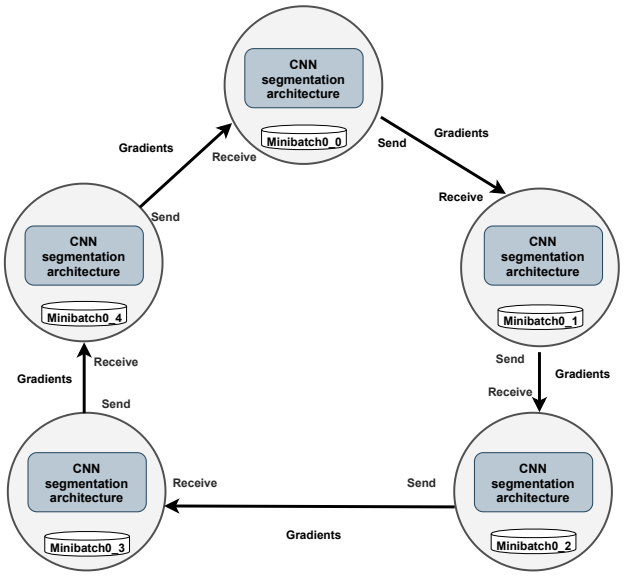


Fig. 3. Ring-Allreduce Algorithm.

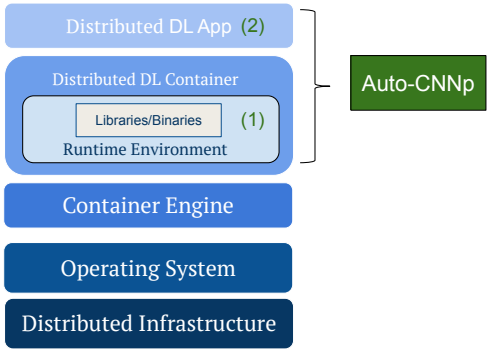


Fig. 4. Auto-CNNp operating scope: (1) runtime environment configuration and (2) user-specific deep learning execution schema definition.

B. Framework Architecture

As illustrated in Figure 5 which shows an overview of the architecture of our proposed system, Auto-CNNp framework follows a modular design. Its different building blocks are as follows:

- The **Engine** is the Auto-CNNp controller (i.e., it manages the framework's control flow). It is the central access point operating as an orchestrator of the framework's components interactions.
- The **CNN-Parallelism-Generator** is the core component of Auto-CNNp framework. It aims to simplify the task of scaling up CNNs training by separating typical parallelization strategies patterns from task-specific CNN applications. To achieve this goal, the *CNN-Parallelism-Generator* component captures common routine tasks (i.e., which are shared by all MPI-based deep learning distributed training approaches) and enables users to customize the remaining applications-specific parts.
- The **Run & Manage** component applies the final execution schema of the framework once all the training agents are ready for the distributed training. Indeed, the *Run & Manage* component is activated by the *engine* in order to initiate and launch the distributed training process.
- The **Training Config File** contains a set of an rules used by the *engine* to govern the execution mechanism of the Auto-CNNp framework.
- As its name suggests, the **distributed training infrastructure** is the execution infrastructure for the distributed training of CNNs.

Further details regarding the aforementioned Auto-CNNp core components are given later in this section.

C. Framework Execution Flow

The Auto-CNNp framework is a configuration-driven framework. The framework's execution flow steps are the followings.

1) The framework user provides an XML-based *training configuration file*.
2) The framework's *engine* parses the aforementioned *configuration file* and extracts the user-specific application behavior.
3) The *CNN-Parallelism-Generator* component is deployed and/or updated on all the training agents. Concurrently, the *run & manage* component is only deployed on the training node which initiates the training.
4) Lastly, when all the training nodes are ready, the end-user activates the *run & manage* component through the *engine* in order to start the distributed training.

D. Component Detail: The CNN-Parallelism-Generator

The *CNN-Parallelism-Generator* is the paramount component of the Auto-CNNp framework. It encapsulates and hides reusable CNNs training parallelization patterns to the framework end users in order to provide a higher level of abstraction. As shown in Figure 6, the *CNN-Parallelism-Generator* has a linear design following the typical workflow of steps to parallelize our MPI-based CNNs (presented in greater detail later). It is composed of a tree of hierarchically classified building blocks. The different abstractions used in the *CNN-Parallelism-Generator* are:

- `Components` are the building blocks of the *CNN-Parallelism-Generator*. They are classified into two categories : either (1) **modules** or (2) **composites** components. The modules do not contain other components while composites components might be composed of one or several composites components and/or modules. Also, components are connected by so-called binding connectors. Composite components are a standalone components which can be reused and replaced without affecting the framework's fundamentals.
- As stated previously, `modules` are a primitives components. They contains a set of task-related

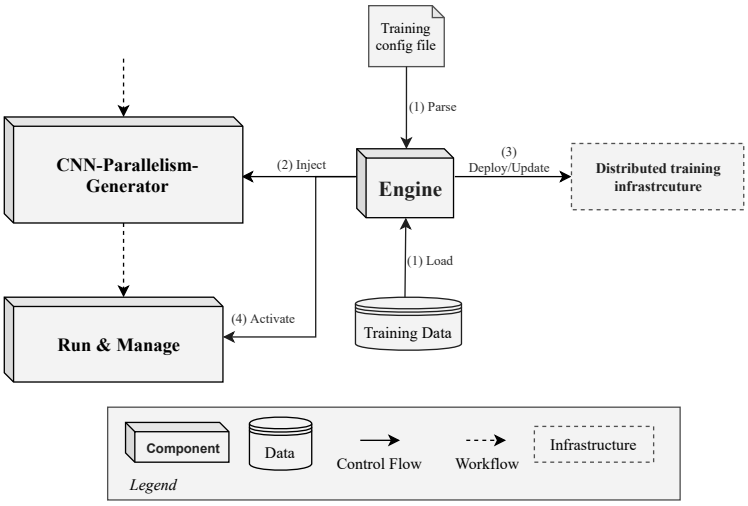


Fig. 5. Auto-CNNp System Architecture Overview.

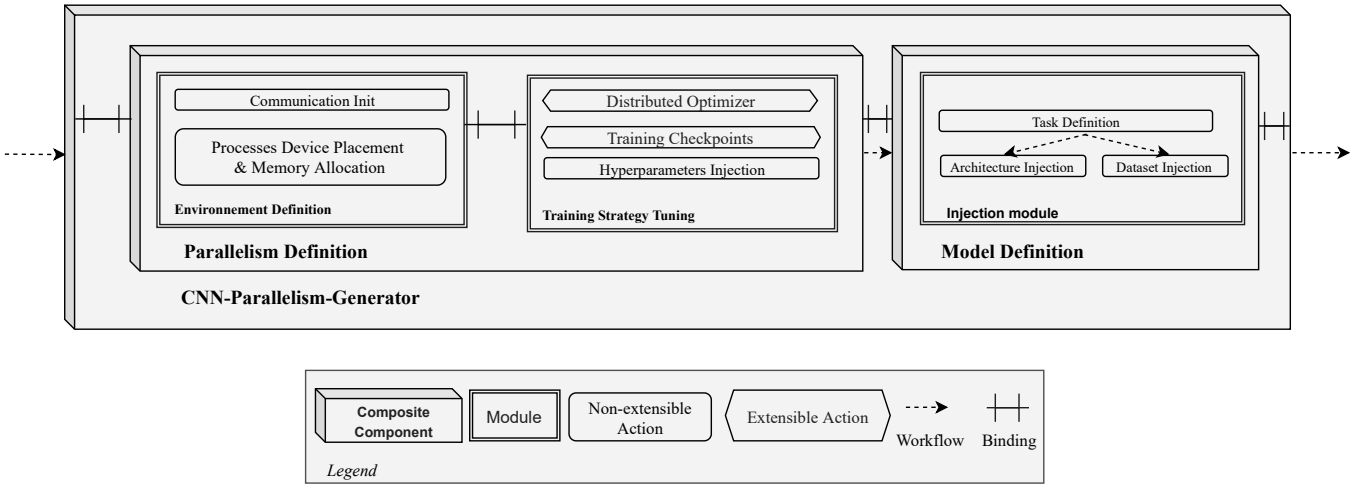


Fig. 6. CNN-Parallelism-Generator Component-Based Architecture.

`actions` cooperating towards a particular CNNs MPI-based parallelization milestone. `Modules` present an inter non-functional dependencies within each others In other words, overwriting `modules` requires the user to change/adjust the corresponding related `modules` within the same `component`.

- `Actions` are a standard collective parallelization steps. They may be classified according to their expandability property into (1) `non-extensible actions` and (2) `extensible actions`. The `non-extensible actions` constitutes a set of generic functionalities which have a unique and static implementation. They are independent from the CNNs parallelization schema, can be parametrized but cannot be extensible by the user. On the other hand, the `extensible actions` can further support extensibility throughout specific plugins which can be defined and customized by the framework's end user in order to expand or override the framework's supported functionalities.

As illustrated in Figure 6, the *CNN-Parallelism-Generator* is a composite of two components (*Parallelism Definition and Model definition*). The latter are in turn are a composite of the followings couple and single modules respectively.

1) Module Details: Environment Definition:

- *Communication Init* is a `non-extensible action` that initializes the adopted communication approach. In our POC implementation, it initializes the MPI default supported protocol.
- *Processes Device Placement & Memory Allocation* is a `non-extensible action` which establishes the custom *TensorFlow-based* processes device placement strategy on the training agents alongside with he adopted memory allocation strategy [7].

2) Module Details: Training Strategy Tuning:

[7]more informations at https://www.tensorflow.org/guide/using_gpu

- *Distributed Optimizer* is an `extensible action` which establishes the adopted CNN parallelism strategy. The default supported approach is the *Ring-Allreduce* algorithm. However, it is possible to adopt another approach (e.g, *Parameter Server*) strategy, etc.). An example of the required modifications to change the parallelism strategy is detailed in the next section.
- *Training Checkpoints* is an `extensible action`. It enable the Framework's user to set up the custom *TensorFlow-based* training checkpoints.
- *Hyperparameters Injection* is a `non-extensible action` training hyperparameters. It specifies the user-specific training hyperparameters.

3) Module Details: Injection Module:

- *Task Definition* is a `non-extensible action` which determines the CNN-based image processing task (e.g., segmentation or classification).
- *Architecture and Dataset Injection* are `non-extensible actions`. They enable an easy loading of the CNN architecture from its corresponding config file alongside with the training dataset path.

E. Component Detail: The Engine

As illustrated in Figure 5, the architecture of Auto-CNNp is based around the `engine`. The latter implementation has to be fast, to decrease the overhead of the framework to the utmost possible degree. Its functionalities are fourfold. In particular (1) parsing the *configuration file*, (2) based on that, the *engine* establishes the *CNN-Parallelism-Generator* final shape (i.e., its final comprising sub-components and modules). In order to do so, the *engine* parameterizes, customizes and loads the *CNN-Parallelism-Generator* building blocks. Next, the *engine* deploys/updates the *cnn-parallelism-generator* on the training nodes. Lasty, it activates the *run & manage* component in order to start the distributed training task.

F. Component Detail: The Training Config File

As stated previously, the *training config file* defines the control flow of the system. In particular, it contains:

- The *CNN-Parallelism-Generator* structure definition and the interaction policy of its inner modules.
- The CNN description.
- The CNN training hyperparameters [5].
- The training dataset metadata (e.g., the training data file system location path, the format)

Listing 1 shows an example of a *training config file* of Auto-CNNp for an image segmentation use case. The *config file* defines the final shape of the *CNN-Parallelism-Generator*: (1) The training runtime environment is customized (e.g., we consider local rank strategy for the device placement and soft placement as memory allocation approach) (2) We adopt the default supplied *Ring-Allreduce* CNN parallelism approach and extend the training checkpoints with a specific plugin. (3) We define the training, validation and test datasets alongside with the CNN architecture (through python *keras-based* CNN description) and the training hyperparameters that will be loaded/injected into their adequate location in the *CNN-Parallelism-Generator* structure.

```
<system_config>
  <task name="imaging_segmentation">
    <parallelism-degree>3</parallelism-degree>
    <CNN_archi><path>/archi/U-Net.py</path></CNN_archi>
    <CNN-Parallelism-Generator>
      <module name="train_env_def">
        <action class="non_extensible" type="device_placement">
          <value>local_rank</value>
        </action>
        <action class="non_extensible" type="memory_allocation">
          <value>soft_placement</value>
        </action>
      </module>
      <module name="train_srategy_def">
          <action class="extensible" type="dist_strategy">
            <value>ring_allreduce</value>
          </action>
            <action class="extensible" type="Tr_Checkpoint">
            <value name="LRSchedule">
              <path>/data/checkpoint/ckpt1.py</path>
            </value>
          </action>
      </module>
    </CNN-Parallelism-Generator>
    <data>
       <train><path>/data/brain-train</path></train>
       <valid><path>/data/brain-validation</path></valid>
       <test><path>/data/brain-test</path></test>
    </data>
    <hyperparameters>
        <property name="lr">1e-5</property>
        <property name="optimiser">SGD</property>
        <property name="loss">dice</property>
        <property name="minibarch_size">10</property>
        <property name="start_epoch">0</property>
        <property name="end_epoch">120</property>
    </hyperparameters>
  </task>
</system_config>
```

Listing 1. Training config file example

IV. Evaluation

In this section, We first introduce our experimental environments and case studies. Afterwards, we conduct a (1) quantitative and (2) qualitative evaluation of our proposal.

A. Experimental Environments

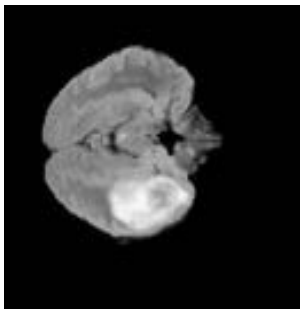
1) Hardware: We accomplished the distributed training experiments on the Nancy Grid'5000 [36] testbed site. The experiments were conducted on Grele GPU cluster which contains Dell PowerEdge R730 physical machines where each node is equipped with 2 Nvidia GeForce GTX 1080 Ti GPUs. We use the Grid'5000 Network File System (NFS) to share the training dataset and the *CNN-Parallelism-Generator* component between all training agents. The nodes are interconnected using *InfiniBand* [37] high-speed interconnect.

2) Software: We have chosen *Python* as a programming language for Auto-CNNp reference implementation. Indeed, Auto-CNNp prototype is concurrently built on top of *Tensor-Flow* and *Keras* deep learning libraries. We take

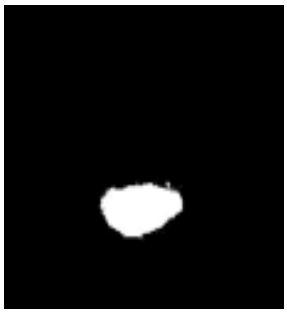
advantage also of the *Horovod* [8] implementation of the *Ring-Allreduce* algorithm in order to introduce the latter adopted synchronous data parallelism approach. In addition, we consider *Open MPI* [38] implementation of the MPI standard as a communication library. Also, we use *Beautiful Soup* python library for the xml *config file* parsing. Furthermore, to ensure research reproducibility, the *CNN-Parallelism-Generator* component alongside with its runtime environment are containerized into a debian 9 stretch-based *docker*[8] image. Lastly, we use *docker swarm* for the container orchestration task.

3) Evaluation case studies: To asses our component-based automatic training parallelism approach, we consider U-Net [39] and FCN [40] as a baseline CNN architectures applied to tackle two different medical imaging use cases, in particular:

1) The first one is a brain tumor segmentation [9] task which was proposed during the decathlon medical segmentation challenge [9]. As illustrated in Figure 7, it involves isolating the different tumor tissues in the brain from healthy ones [41]. It is a crucial and challenging task in medical image analysis because it plays an influential role in early diagnosis of brain tumors which in turn enhance treatment planning and raise the survival rate of the patients. Yet, it is a tedious and time consuming task because when it can take hours when it is manually performed by expert radiologists. The dataset which has been provided during the aforementioned segmentation challenge for the brain tumors segmentation task is a mix of two other datasets that have been initially made publicly available during the *Multimodal Brain Tumor Segmentation Challenge (MICCAI BRATS) [42] 2016 and 2017*. It contains multimodal MRI scans (i.e., 4D MRI scans [43]) of complex and heterogeneously-located brain tumors that were captured using multiple distinct MRI acquisition protocol [43] from 19 different institutional data contributors [42]. The BRATS datasets have been initially manually segmented by one to four raters, using the same annotation protocol. After that, the multimodal brain tumor MRI scans along with all their corresponding ground truth labels were manually-reexamined and approved by experienced neurologists [42].
2) The second use case is a left atrial segmentation task. It consists in isolating the left atrium body from its surrounding organs structures [44]. It plays a key role during the treatment protocol of patients with atrial fibrillation disease [44] which is the most frequent cardiac electrical disorder provoked by abnormal electrical discharges in the left atrium. The dataset has been made publicly available by Philips Technologie GmbH, Hamburg, DE, and King's College London during the 2013 LASC challenge [10]. Unlike the BRATS datasets, the left atrium segmentation dataset is a small one with wide quality levels variability as it only includes 30 mono-modal 3D cardiac MRI scans. The dataset was split such that 20 MRI scans were provided with their corresponding ground truth annotations for the training and the validation steps. The remaining 10 MRI scans were supplied as a test set. The ground-truth masks were initially annotated using automatic model based segmentation. Afterwards, a manual corrections were performed by human experts [44].



2D-MRI slice of BRATS training dataset



Corresponding brain tumor ground-truth annotation

Fig. 7. brain tumor segmentation task

4) Quantitative evaluation: We assess the cost benefit trade-off of automating CNNs parallelization task through a quantitative assessment approach. In order to do so, we measure the execution time of the Auto-CNNp *engine* for the previsouly mentioned two evaluation case studies tackled by a couple of widely used CNN architectures (U-Net and FCN). For reliability reasons, we run each experimental setup 100 times and we consider the average of the measured execution duration as our reference results. The execution times were measured using the linux `/usr/bin/time`. The outputs of the latter command are threefold: (1) *real* metric stands for the overall execution time from start to finish of the call, (2) *user* metric denotes the amount of CPU time spent in user-mode and (3) *sys* metric is the CPU time spent in kernel mode by the program.

Figure 8 depicts the evaluation results for the framework's *engine* execution time. It shows that the execution times for the four setups are approximately similar which confirms the generalizability of our proposal. The *engine's real* execution time is about 139 ms which is a negligible time compared to the typical time-consuming CNN training task duration (21 hours and 40 minutes for U-Net and 35 hours and 40 minutes for FCN for a single *Nvidia GTX 1080* GPU based training). The difference between the *real* execution time and the sum of both of *user* and *sys* times is almost 18 ms. It is due to the fact that the engine is blocked on disk I/O during the deployment

[8]More information can be found at https://www.docker.com/

[9]More informations on the decathlon segmentation challenge can be found at the following links: http://medicaldecathlon.com/ and https://decathlon.grand-challenge.org/

[10]The left atrium segementation dataset is available at the following link https://www.cardiacatlas.org/challenges/left-atrium-segmentation-challenge/

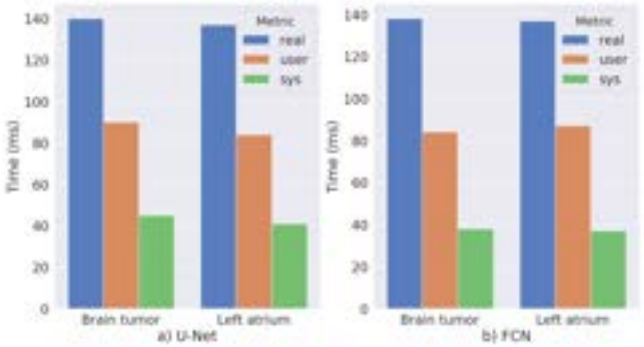


Fig. 8. : Auto-CNNp engine execution time evaluation (a) U-Net CNN architecture and (b) FCN CNN architecture for brain tumor and cardiac left atrium case studies segmentation tasks

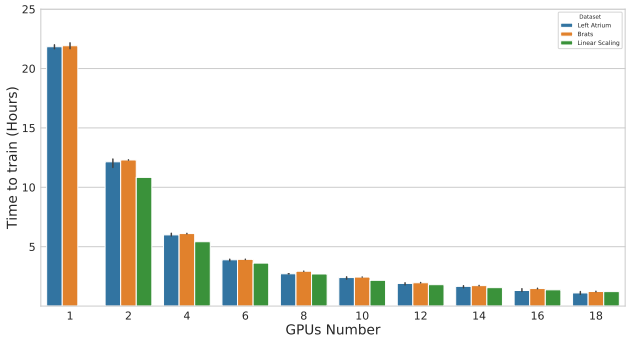


Fig. 9. Training time evolution with scale for U-Net CNN architecture for brain tumor segmentation task

or update step of the *CNN-Parallelism-Generator* component on all training agents through the NFS server.

Furthermore, in order to evaluate the framework's impact on the CNN-based task performances, we compare the obtained segmentation accuracy when scaling up CNNs training manually with the segmentation accuracy we get after using Auto-CNNp. In order to do so, we first perform the distributed training of U-Net CNN architecture for brain tumor and left atrium segmentation on 18 GPUs without using Auto-CNNp. We ran each experimental setup 10 times and we considered the average of the measured metrics as our reference results. As shown in Figure 9, we achieved an almost perfect linear scaling and **17.5x** speed-up than single-node based training for both segmentation tasks with a segmentation *dice score* of 0.886 and 0.794 for brain tumor and left atrium segmentation respectively. We achieved exactly the same results after performing the U-Net CNN parallelization using Auto-CNNp for both evaluation case studies. Hence, using Auto-CNNp does not impact the performances of the CNN parallelization process compared to the manual approach.

5) Qualitative evaluation: Auto-CNNp intents to offer a high level of abstraction over MPI-based CNN parallelism by instrumenting common routines. In order to do so, the framework is driven throughout a high level *training config file*. To qualitatively evaluate Auto-CNNp reaches, we investigate the impact of the framework in reducing the burden of practically scaling up CNNs training.

We consider the Listing 1 as a starting *training config file*. We adopt U-Net CNN architecture and *Ring-Allreduce* parallelism strategy to tackle our first evaluation use case which is the brain tumor segmentation task. After that, we aim to test a different CNN architecture (FCN) to deal with the same evaluation use case. In order to do so, we only need to change the `<CNN_archi>` tag in the *config file* and its related CNN architecture file. Also, if the framework's end user wants to tackle a different segmentation use case using the same initial CNN architecture, he exclusively needs to change the `<data>` tag siblings in the *config file*. All of this shows the easiness with which the framework's user can switch from one training dataset use case to another and/or to test different CNN architectures by minimal code changes.

As mentioned earlier, the adopted *Ring-Allreduce* algorithm constitutes a POC example for our proposal implementation. It is indeed possible to adopt another CNN parallelism approach. In order to do so, the *Distributed Optimizer* `extensible action` needs to be overwritten alongside with its corresponding *Training Strategy tuning* module. Also, the *Environment Definition* module might require to be replaced since it shares the same *CNN-Parallelism-Generator* component as the *Training Strategy tuning* module. Yet, the *Model Definition* component can be reused to generate the new component-based *CNN-Parallelism-Generator*. Finally, the operating-system-level environment might need to be adapted but as discussed earlier in subsection III-A, it is out of scope of our proposed framework.

V. Conclusion and Future Work

We presented Auto-CNNp, a framework which permits to automate CNNs distributed training task. Our proposed system offers a high level of abstraction over talent-intensive distributed deep learning by introducing a component-based approach. The latter provides a generic tool that encapsulates many common CNNs parallelism patterns while being sufficiently flexible to be extensible for user-specific customization. We described a POC reference implementation of Auto-CNNp while justifying our design choices. The quantitative and qualitative evaluations of our proposal on a couple of case studies confirm its validity and transferability to other use cases.

In the future, we plan to support additional CNN-based tasks by introducing the automated distributed training of other CNN-based applications (e.g., CNN-based text classification task). Also, we are in the process of porting Auto-CNNp in order to support other platforms and libraries (e.g., *PyTorch* [11]). Finally, we aim to integrate some infrastructure configuration management tools (e.g., *SaltStack* or *Puppet*) to the Auto-CNNp ecosystem.

[11]More informations can be found at https://pytorch.org/

ACKNOWLEDGMENTS

This work was supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-10-AIRT-05), the Virtual Studio RA FEDER EU project, the FSN Hydda project, EIT Health and Institut Carnot LSI. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] N. Sharma and L. Aggarwal, "Automated medical image segmentation techniques," *Journal of Medical Physics*, vol. 35, no. 1, pp. 3–14, 2010.

[2] H. Greenspan, B. van Ginneken, and R. M. Summers, "Guest editorial deep learning in medical imaging: Overview and future promise of an exciting new technique," *IEEE Transactions on Medical Imaging*, vol. 35, pp. 1153–1159, May 2016.

[3] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," *CoRR*, vol. abs/1303.5778, 2013.

[4] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," *CoRR*, vol. abs/1411.4389, 2014.

[5] I. Loshchilov and F. Hutter, "Cma-es for hyperparameter optimization of deep neural networks," *arXiv preprint arXiv:1604.07269*, 2016.

[6] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[7] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2592–2600, 2016.

[8] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018.

[9] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, pp. 888–905, Aug. 2000.

[10] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[11] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," in *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, (USA), pp. 507–514, Omnipress, 2012.

[12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1223–1231, Curran Associates Inc., 2012.

[13] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CoRR*, vol. abs/1802.09941, 2018.

[14] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), pp. 463–478, ACM, 2017.

[15] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[17] S. L. Smith, P. Kindermans, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *CoRR*, vol. abs/1711.00489, 2017.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Intelligent Signal Processing*, pp. 306–351, IEEE Press, 2001.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[20] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International conference on machine learning*, pp. 1337–1345, 2013.

[21] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 463–478, ACM, 2017.

[22] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 583–598, USENIX Association, 2014.

[23] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 571–582, 2014.

[24] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *INTERSPEECH*, 2015.

[25] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the pieces together, addison-westley*, p. 5, 2001.

[26] P. Fingar, "Component-based frameworks for e-commerce," *Communications of the ACM*, vol. 43, no. 10, pp. 61–61, 2000.

[27] C. More, L. Colaco, and R. Sardinha, "Application of component-based software engineering in building a surveillance robot," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pp. 651–658, Springer, 2015.

[28] A. Brown, S. Johnston, and K. Kelly, "Using service-oriented architecture and component-based development to build web service applications," *Rational Software Corporation*, vol. 6, pp. 1–16, 2002.

[29] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, "A component based services architecture for building distributed applications," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pp. 51–59, Aug 2000.

[30] X. Ke, K. Sierszecki, and C. Angelov, "Comdes-ii: A component-based framework for generative development of distributed real-time control systems," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pp. 199–208, IEEE, 2007.

[31] J. F. Ferreira, J. L. Sobral, and A. J. Proença, "Jaskel: A java skeleton-based framework for structured cluster and grid computing," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, vol. 1, pp. 4–pp, IEEE, 2006.

[32] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[33] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pp. 2350–2356, AAAI Press, 2016.

[34] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[35] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems*, pp. 19–27, 2014.

[36] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science* (I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, eds.), vol. 367 of *Communications in Computer and Information Science*, pp. 3–20, Springer International Publishing, 2013.

[37] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges, "Infiniband scalability in open mpi," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 10–pp, IEEE, 2006.

[38] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 97–104, Springer, 2004.

[39] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

[40] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[41] L. Zhao and K. Jia, "Multiscale cnns for brain tumor segmentation and diagnosis," *Comp. Math. Methods in Medicine*, vol. 2016, pp. 8356294:1–8356294:7, 2016.

[42] B. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer, K. Farahani, J. Kirby, Y. Burren, N. Porz, J. Slotboom, R. Wiest, L. Lanczi, E. Gerstner, M.-A. Weber, T. Arbel, B. Avants, N. Ayache, P. Buendia, L. Collins, N. Cordier, J. Corso, A. Criminisi, T. Das, H. Delingette, C. Demiralp, C. Durst, M. Dojat, S. Doyle, J. Festa, F. Forbes, E. Geremia, B. Glocker, P. Golland, X. Guo, A. Hamamci, K. Iftekharuddin, R. Jena, N. John, E. Konukoglu, D. Lashkari, J. Antonio Mariz, R. Meier, S. Pereira, D. Precup, S. J. Price, T. Riklin-Raviv, S. Reza, M. Ryan, L. Schwartz, H.-C. Shin, J. Shotton, C. Silva, N. Sousa, N. Subbanna, G. Szekely, T. Taylor, O. Thomas, N. Tustison, G. Unal, F. Vasseur, M. Wintermark, D. Hye Ye, L. Zhao, B. Zhao, D. Zikic, M. Prastawa, M. Reyes, and K. Van Leemput, "The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)," *IEEE Transactions on Medical Imaging*, vol. 34, pp. 1993–2024, Oct. 2014.

[43] A. Isin, C. Direkoglu, and M. Sah, "Review of mri-based brain tumor image segmentation using deep learning methods," *Procedia Comput. Sci.*, vol. 102, pp. 317–324, Dec. 2016.

[44] C. Tobon-Gomez, Geers, *et al.*, "Benchmark for algorithms segmenting the left atrium from 3d ct and mri datasets," *IEEE transactions on medical imaging*, vol. 34, no. 7, pp. 1460–1473, 2015.