The DeepHealth Toolkit: A Unified Framework to Boost Biomedical Applications

Michele Cancilla*, Laura Canalini*, Federico Bolelli*, Stefano Allegretti*, Salvador Carrión†, Roberto Paredes†, Jon A. Gómez†, Simone Leo‡, Marco Enrico Piras‡, Luca Pireddu‡, Asaf Badouh§, Santiago Marco-Sola§‖, Lluc Alvarez§¶, Miquel Moreto§¶, and Costantino Grana*

*Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia, Italy
Email: {*name.surname*}@unimore.it

†PRHLT Research Center
Universitat Politècnica de València, Spain
Email: {salcarpo, rparedes, jon}@prhlt.upv.es

‡Data-intensive Computing Group, CRS4, Italy
Email: {*name.surname*}@crs4.it

§Barcelona Supercomputing Center, Spain
Email: {*name.surname*}@bsc.es

¶Universitat Politècnica de Catalunya, Spain

‖Universitat Autònoma de Barcelona, Spain

***Abstract*—Given the overwhelming impact of machine learning on the last decade, several libraries and frameworks have been developed in recent years to simplify the design and training of neural networks, providing array-based programming, automatic differentiation and user-friendly access to hardware accelerators. None of those tools, however, was designed with native and transparent support for Cloud Computing or heterogeneous High-Performance Computing (HPC). The *DeepHealth Toolkit* is an open source Deep Learning toolkit aimed at boosting productivity of data scientists operating in the medical field by providing a unified framework for the distributed training of neural networks, which is able to leverage hybrid HPC and cloud environments in a transparent way for the user. The toolkit is composed of a Computer Vision library, a Deep Learning library, and a front-end for non-expert users; all of the components are focused on the medical domain, but they are general purpose and can be applied to any other field. In this paper, the principles driving the design of the *DeepHealth* libraries are described, along with details about the implementation and the interaction between the different elements composing the toolkit. Finally, experiments on common benchmarks prove the efficiency of each separate component and of the DeepHealth Toolkit overall.**

I. Introduction

The ongoing European Project *DeepHealth*, funded by the EC under the topic ICT-11-2018-2019 "HPC and Big Data enabled Large-scale Test-beds and Applications," aims to create a unified framework, completely adapted to exploit underlying heterogeneous High-Performance Computing (HPC) and Big Data (BD) architectures, in order to boost biomedical applications using state-of-the-art Deep Learning (DL) and Computer Vision (CV) algorithms. One of the core elements to achieve this ambitious goal is the development of two novel, integrated software libraries: ECVL (European Computer Vision Library) and EDDL (European Distributed Deep Learning Library), respectively designed for computer vision and deep learning tasks. The two libraries, ECVL and EDDL, together with a front-end designed for non-expert users, constitute the *DeepHealth Toolkit*. The toolkit and its entire ecosystem is open source and freely available on GitHub [1] under the MIT license (more details are available in Section IX).

At this point the reader is surely wondering, "Why do you have to develop new computer vision and deep learning frameworks?" and "What will these frameworks add to existing and carefully designed ones?" Let's skip the possible analogies with questions such as "Why develop Chrome when we have Internet Explorer?", which would be great if they became reasonable in the future, and let's focus on concrete motivations.

Beyond other toolkits available at the time of writing the DeepHealth proposal, we aim to provide data scientists with a powerful tool to run distributed Deep Neural Networks (DNNs) training on hybrid HPC and cloud infrastructures in a user-transparent way, without requiring a deep understanding of how DNNs and distributed high-performance computing work. Overall, the final goal of the DeepHealth Toolkit is to increase the productivity of data scientists working in the health sector. A practitioner working together with doctors should be able to run a training job on his country-donated HPC service, without writing additional code, and without the need to set up anything. However, while these requirements —and thus the current use cases— tend to focus on the health sector, the DeepHealth libraries are both general-purpose and can be employed in different domains.

In this paper, we present the rationale behind the de-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825111, DeepHealth Project. Lluc Alvarez has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Juan de la Cierva Formacion fellowship No. FJCI-2016-30984. Miquel Moreto has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Ramon y Cajal fellowship No. RYC-2016-21104.

velopment of these new libraries and the philosophy that characterizes them. Further, we present the first qualitative and quantitative results, revealing their capabilities.

The parallel and distributed computing capabilities of the DeepHealth Toolkit abstract the computation so that the only requirement for a DeepHealth Toolkit user is to provide a file describing the *"computing service"* – *i.e.*, computing resources available to train a model. The *"computing service"* is specified by the number of CPU-cores, Graphical Processing Units (GPUs), or Field Programmable Gate Arrays (FPGAs) accessible on the computer/s where the training process will run. Remote computers are identified by their IP addresses. Within the DeepHealth project, the libraries are integrated into a range of biomedical platforms provided by partners, such as Open Innovation (PHILIPS), MigraineNet (WINGS), ExpressIF^{TM} (CEA), PIAF (THALES SIX), Open-DeepHealth (UNITO), Digital Pathology (CRS4), and everisLumen (EVERIS).

The rest of the paper is organized as follows. In Section II and Section III EDDL and ECVL are described. Section IV introduces the interaction between the two libraries within the scope of the DeepHealth Toolkit. The Python APIs and the work done to containerize the libraries are respectively detailed in Section V and Section VI. In Section VII a description of the RESTful based web service to exploit the functionalities of the libraries is provided. Section VIII presents the first tests and evaluations performed on the toolkit, while Section IX reports distribution and license details. Finally, in Section X some concluding remarks are drawn.

II. European Distributed Deep Learning Library

The ambitious goal of EDDL is to cover most of the commonly used deep learning functionalities in the health sector while preserving simple installation and configuration processes. The library aims at becoming widely employed, easy to integrate into existing and future pipelines.

EDDL allows the definition of different neural network architectures and their execution on different hardware platforms and accelerators, offering tailored implementations that can be efficiently executed on CPUs, GPUs, and FPGAs. Additionally, EDDL includes *ad hoc* methods for serializing weights and gradients in order to facilitate the network update between workers and master nodes. Providing such functionalities, frameworks such as COMPSs [2] can distribute the work across available accelerators, in a way transparent to the programmer.

Neural network import and export functionalities are included employing the Open Neural Network Exchange (ONNX) format [3]. The Google Protocol Buffers [4] library is used for model serialization (weights and/or gradients) according to the ONNX definition of layers, operators and network topologies.

Given the requirement for fast computation of matrix operations and mathematical functions, the EDDL library is being coded in *C++*. GPU specific implementations are based on the NVIDIA CUDA language extensions for *C++*.

```
layer in = Input({784});

in = LeakyReLu(Dense(in, 1024));
in = LeakyReLu(Dense(in, 1024));
in = LeakyReLu(Dense(in, 1024));

layer out = Softmax(Dense(in, num_classes));
model net = Model({in}, {out});

// Build model
build(net,
   sgd(0.01f, 0.9),           // Optimizer
   {"soft_cross_entropy"},    // Loss
   {"categorical_accuracy"}, // Metric
   CS_GPU({1}, "low_mem")     // One GPU
);

// Load training and test data
tensor x_tr = eddlT::load("mnist_trX.bin");
tensor y_tr = eddlT::load("mnist_trY.bin");
tensor x_ts = eddlT::load("mnist_tsX.bin");
tensor y_ts = eddlT::load("mnist_tsY.bin");

// Preprocessing
x_tr->div_(255.0);
x_ts->div_(255.0);

// Train model
fit(net, {x_tr}, {y_tr}, batch_size, epochs);

// Evaluate
evaluate(net, {x_ts}, {y_ts});
```

Listing 1. An excerpt of EDDL neural network definition code. This model trains and evaluates a simple multilayer perceptron using one GPU as computing service. The resources to be used can be modified by changing the *"computing service"* configuration file.

A. Tensor and Neural Networks

The EDDL API is centered around the concepts of *Tensor* and *Neural Network* model. The Tensor class concentrates all what concerns tensors – *e.g.*, matrix element-wise and linear algebra operations. Moreover, this class plays the role of Hardware Abstraction Layer (HAL): models and tensors will be created and initialized on the device(s) specified by the *"computing service"* configuration file. Subsequent operations involving tensors will be transparently performed on the specific device, whether this is a (group of) CPU, GPUs, and/or FPGAs, and using as many cores as specified in the configuration.

The EDDL library aims to provide the user with a Keras-like [5] API in order to ease the learning curve, but also providing the means to work with low-level features. The documentation of the library lists layers, metrics, losses, and optimizers currently implemented. Listing 1 shows an example of a multilayer perceptron network written using EDDL, but many other examples can be found in the documentation – for instance, more complex models like ResNet-50 [6] and U-Net [7].

Data augmentation can be seamlessly integrated as a set of layers that can be stacked in the neural network topology. This way, EDDL simplifies the implementation of such operations and provides a unified framework to deal with data augmentation and tensor operations in the same manner.

```yaml
name: MNIST
description: This is the MNIST dataset.
classes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
images:
    - location: "training/0.png"
      label: 5
    - location: "training/1.png"
      label: 0
split:
   training: [0, 1]
```

Listing 2. Excerpt of the MNIST dataset defined using the DeepHealth Dataset Format syntax.

```
SequentialAugmentationContainer
    AugResizeDim dims=(100,100)
    AugMirror p=0.5
    OneOfAugmentationContainer p=0.7
        AugGammaContrast gamma=[0,3]
        AugBrightness beta=[0,30]
    End
    OneOfAugmentationContainer p=0.4
        AugElasticTransform alpha=[30,120]
                            sigma=[3,6]
        AugGridDistortion num_steps=[2,5]
                          distort_limit=[-0.3,0.3]
    End
    AugRotate angle=[-30,30]
End
```

Listing 3. Example of ECVL augmentations. Parameters in square brackets are randomly sampled in the specified interval. `SequentialAugmentationContainer` defines a list of augmentations to be applied in sequential order. One of the augmentations grouped in the `OneOfAugmentationContainer` will be applied with probability `p`.

III. European Computer Vision Library

The European Computer Vision Library has the purpose of facilitating the integration and exchange of data between existing state-of-the-art computer vision and image processing libraries. Moreover, it provides new high-level computer vision functionalities implementing specialized and accelerated versions of algorithms commonly employed in conjunction with deep learning. As for EDDL, ECVL algorithms are also adapted to hardware accelerators. The library provides support for multiple operating systems and multiple types of scientific imaging data and data formats (*e.g.*, jpeg, png, bpm, ppm, pgm, etc.), with particular reference to medical imaging: DICOM, NIfTI and many proprietary virtual slides formats are supported natively. The counterpart to the EDDL Tensor is the ECVL Image.

A. ECVL Image and Functions

The *Image* class represents the core of the entire ECVL library. It is an object that stores data (either images/videos or raw data) in a multi-dimensional dense numerical single- or multi-channel tensor. The Image provides a simple but effective Hardware Abstraction Layer which exploits generic functions for managing memory, allowing great flexibility for device differentiation (CPUs, GPUs, and FPGAs) while keeping the same user interface. The Image class has been designed for representing and manipulating different kinds of images with diverse channel configurations, providing both reading and writing functionalities for all the aforementioned data formats.

Arithmetic operations between images or between images and scalars are performed through the Image class. Of course, all the classic operations for image manipulation such as rotation, resizing, mirroring, and color space change are available. Processing functions, like noising, blurring, contour finding [8], image skeletonization [9], and connected components labeling [10], [11], [12] are implemented as well. A cross-platform GUI based on ECVL and wxWidgets [13] is also provided to allow simple exploration and test of ECVL functionalities. Furthermore, a visualizer for 3D volumes, such as CT scans, allows to observe different slices of a volume from different views.

Image processing functionalities suitable for data augmentation in a deep learning pipeline can be wrapped into *augmentation classes*. More details about their usage and advantages are provided in Section IV-B.

IV. Interaction Between the Toolkit Elements

A. DeepHealth Dataset Format

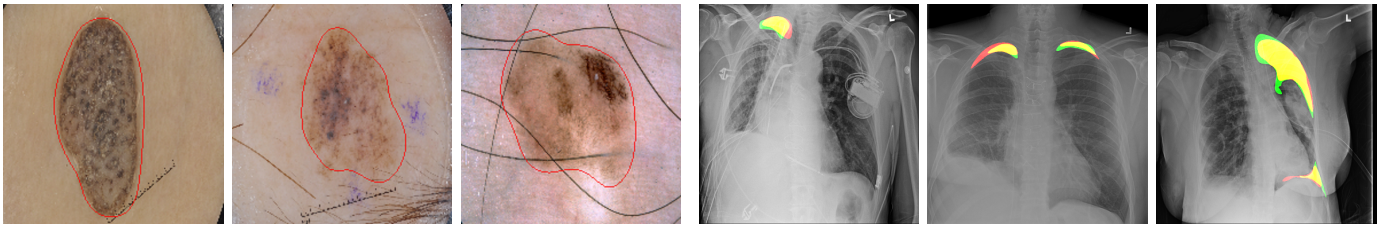
In order to make the libraries interoperable and to provide a straightforward and efficient mechanism to perform distributed DNNs training, the *DeepHealth Dataset Format* (DDF) has been defined and introduced.

DDF is based on the simple and flexible YAML [14] syntax and describes a dataset. An example is provided in Listing 2. The file format defines all the information such as `name` and `description` of the dataset, its `classes` and `features`, a list of image or volume paths, and a `split` indicating how to divide images into `training`, `validation` and `test` sets. The `classes` entry represents all the predictable categories, while `features` field describes the additional information related to each image. In the example above, MNIST dataset has ten different classes (digits from 0 to 9) and no features. The *DeepHealth Dataset Format* also allows to specify segmentation masks for each input entry, thus also supporting this fundamental task.

An ECVL module is provided to parse and load DDF-defined datasets into the specific *Dataset* class. Moreover, the library exposes a *Generator* for the automatic creation of a Dataset object by traversing a directory tree and dumping the information onto the YAML file.

B. ECVL-EDDL Interface

The cooperation between EDDL and ECVL is one of the key elements of the entire project. The libraries interface is based on two main functions that convert ECVL Image(s) into EDDL Tensor(s) and vice versa. Additionally, the generic Dataset has been extended into the so-called *DLDataset* to add specific deep learning attributes, such as the batch size or the augmentations that will be applied to the input images during the training of a neural network. In fact, ECVL provides the possibility to apply data augmentation



(a) ISIC (b) SIIM-ACR Pneumothorax

Fig. 1. Example outputs of DeepHealth Toolkit pipelines: (a) ISIC, red polygons delimit the regions of interest (skin lesions), and (b) pneumothorax segmentation masks where red indicates the prediction area, green is the ground truth, and yellow shows their intersection.

```
Image img;
if (!ImRead("examples/data/test.jpg", img)) {
    return EXIT_FAILURE;
}

// Loads the augmentation pipeline
fstream ss("augmentation.txt");
auto augs = AugmentationFactory::create(ss);

// And applies them to the image
augs->Apply(img);
```

Listing 4. Usage of data augmentation containers. In this case, the transformation list is loaded from file (see Listing 3 as an example).

```
import pyecvl.ecvl as ecvl
import numpy as np

# Image to Array
img = ecvl.ImRead(img_path)
a = np.array(img)

# Array to Image
a = np.arange(12).reshape(3, 4).astype(np.int16)
img = ecvl.Image.fromarray(a, "xyc",
  ecvl.ColorType.GRAY)
```

Listing 5. Converting images to *numpy* arrays and back.

directly during batch loading. Each dataset split can have its own augmentations defined through specific containers. The containers allow the application of the desired set of transformations, which can be either defined in compiled code or in a special Domain-Specific Language (DSL) and read from file at run time. In the latter case we can change the program behaviour without rebuilding the application. Listing 3 provides an example of a DSL-defined augmentation set. As illustrated, different types of containers are available: a `SequentialAugmentationContainer` forces the subsequent list of transformations to be executed sequentially; on the other hand, the `OneOfAugmentationContainer` forces the system to choose exactly one transformation from a specified set and to apply it with a certain probability. Listing 4 is an example of usage of the augmentation containers.

The combination of the libraries allows an easy and effective programming experience. The combined capabilities of these two libraries is showcased in the "use case pipeline" code repository (link in Table IV). This repository contains four pipeline examples using EDDL and ECVL to train Convolutional Neural Networks on three different datasets (MNIST, ISIC [15] and SIIM-ACR Pneumothorax [16]), applying different image augmentations, for both classification and segmentation tasks. Sample predictions from these pipelines are illustrated in Figure 1.

V. Python APIs

In order to make the DeepHealth libraries more accessible and easier to integrate with other libraries in ML workflows, Python APIs have been developed. The Python packages are called PyEDDL and PyECVL and are hosted on GitHub as well.

PyEDDL APIs provide access to the EDDL APIs, which can be broken down into two main functionality groups: neural networks (models, layers, regularizers, initializers, dataset handling) and tensor operations (such as creation and serialization, data copying and mathematical operations). Like in the *C++* EDDL API, the neural network training section of the Python API provides both high-level functions, such as `fit` and `evaluate`, and low-level ones that allow to fine-tune what happens in individual epochs and batches, providing much finer control albeit at a slight loss in efficiency (because more of the execution time is spent in Python code). Similarly, the PyECVL APIs expose ECVL functionalities in Python: data types, color types, device types, Image objects, arithmetic operations, image processing, image input/output, augmentations, the DeepHealth dataset parser and ECVL-EDDL interaction. To facilitate interoperability between the DeepHealth libraries and other Python libraries for data analysis or machine learning, specific bindings have been implemented to allow seamless conversion between EDDL Tensor or ECVL Image objects and NumPy arrays (see Listing 5), enabling complex workflows where data is exchanged between the different data structures.

For efficiency, the implementation of the Python APIs is based on a core binary Python extension module which provides the Python bindings for the *C++* code. The highest layer of the Python API is implemented in pure Python. This layer allows Python programmers to easily plug in new code to implement custom behavior. On the other hand, the binary core is implemented using pybind11 [17], a header-only library that allows to efficiently expose *C++* types in Python and vice

versa. The binding development process is started with the Rosetta Commons Binder [18] tool, which generates part of the code. This generated part is then manually tweaked where appropriate, and the remainder of the pybind11-based bindings are written manually (not all parts of the *C++* API are supported by Binder). This partially automated process allows us to reduce the required effort to develop the Python API and thus more closely track the development of the underlying *C++* libraries. The overall design of the Python API avoids introducing significant inefficiencies in execution speed or memory usage by keeping any computationally intensive code in *C++*, while the pure Python code acts as "glue" that joins together long-running native functions. Thus, users of this API can benefit from all the advantages of a high-level interpreted language without sacrificing speed of execution or memory usage; the performance loss with respect to the fully native implementation of the DeepHealth libraries is evaluated in Section VIII.

The code is covered by numerous unit tests and integrated by several usage examples, most of which are Python portings of the corresponding *C++* examples. PyEDDL examples include network training with various models, ONNX serialization and deserialization and conversion of tensors to/from NumPy arrays. Listing 6 shows an example of multilayer perceptron network training. ECVL examples include augmentations usage, DeepHealth dataset handling, ECVL-EDDL interaction, image processing and image I/O. Listing 7 illustrates how to perform a simple image processing task. Tests are run regularly on continuous integration services (Jenkins and Travis). In addition, the Python APIs have been tested by DeepHealth project partners, both during hackathon events and in the ongoing use cases integration into platforms.

VI. Docker images

To facilitate the use of the DeepHealth libraries, CUDA-enabled Docker container images are provided on Docker Hub (links provided in Table IV) and regularly updated. These images facilitate installation-free and reproducible usage on container-enabled HPC clusters, containerized platforms such as Kubernetes (*e.g.*, work to integrate DeepHealth in Kubeflow [19] workflows is ongoing by other partners within the project), and even on regular workstations. While Docker is supported directly, the images can also be used with Singularity by an image conversion. In all cases, the Nvidia Docker Runtime must be correctly installed on the host system(s) to run container-based workloads on GPUs supporting CUDA.

Docker images are provided for *C++*, `dhealth/libs`, and Python users, `dhealth/pylibs`. In both cases, the images contain the EDDL and ECVL libraries and their run time dependencies, including CUDA; with respect to the `libs` image, `pylibs` adds the Python bindings and the Python interpreter itself. Client scripts or applications can be grafted directly onto these images, as done for the DeepHealth Toolkit web service described in Section VII. In addition to the runtime images, a more feature-rich `-toolkit` flavor is also provided for all the images. These are built on the `devel` flavor of the `nvidia/cuda` images and add a full DeepHealth build environment. Therefore, they are suitable to compile software built on the DeepHealth libraries without installing the full set of build time dependencies on the computer. Finally, in addition to the release-tracking images just described, library-specific images that track the development process are automatically published. These are most useful for use cases that need to closely track or want to contribute to the development process of the libraries.

```python
import pyeddl.eddl as eddl
import pyeddl.eddlT as eddlT

epochs, batch_size, nclass = 10, 100, 10

# Build model
in_ = eddl.Input([784])
layer = in_
layer = eddl.ReLu(eddl.Dense(layer, 1024))
layer = eddl.ReLu(eddl.Dense(layer, 1024))
layer = eddl.ReLu(eddl.Dense(layer, 1024))
out = eddl.Softmax(eddl.Dense(layer, nclass))

net = eddl.Model([in_], [out])
eddl.build(net,
   eddl.rmsprop(0.01),               # Optimizer
   ["soft_cross_entropy"],           # Loss
   ["categorical_accuracy"],         # Metric
   eddl.CS_GPU([1], mem="low_mem")) # One GPU

# Load training and test data
x_tr = eddlT.load("trX.bin")
y_tr = eddlT.load("trY.bin")
x_ts = eddlT.load("tsX.bin")
y_ts = eddlT.load("tsY.bin")

# Preprocessing
eddlT.div_(x_tr, 255.0)
eddlT.div_(x_ts, 255.0)

# Train model
eddl.fit(net, [x_tr], [y_tr], batch_size, epochs)

# Evaluate
eddl.evaluate(net, [x_ts], [y_ts])
```

Listing 6. PyEDDL multilayer perceptron training example.

```python
import pyecvl.ecvl as ecvl
img = ecvl.ImRead("sample.png")
tmp = ecvl.Image.empty()
ecvl.Rotate2D(img, tmp, 60)
gamma = 3
ecvl.GammaContrast(tmp, tmp, gamma)
ecvl.ImWrite("sample_mod.png", tmp)
```

Listing 7. PyECVL image processing example.

VII. DeepHealth Service

The *DeepHealth Service* is a web service that exposes the functionality of the DeepHealth Toolkit. It has been developed to facilitate the use of the DeepHealth libraries. Using the DeepHealth Service, data scientists do not have to write code for the DeepHealth library APIs or directly manage computing resources, but directly use ECVL and EDDL functionalities through a RESTful web service and, optionally, a web-based GUI.

The Service provides the ability to design, train and test predictive models and to perform pre- and post- processing without writing any code. Instead, the REST interface enables *managed service* usage scenarios, where a potentially complex and powerful computing infrastructure (*e.g.*, high-performance computing, cloud computing or even heterogeneous hardware) could be transparently used to run deep learning jobs without the users needing to directly interface with it.

The typical usage scenario of the *Service* is twofold.

- *Inference*: the user can see how well a model performs on his task, by only feeding new data to a pre-trained model – *e.g.*, learning to classify melanomas using ResNet-50 [6] model already trained on ImageNet [20] and fine-tuned on ISIC [15].
- *Training*: the user trains an existing or newly devised model from scratch.

The DeepHealth Service API has been defined using OpenAPI [21] to maximize interoperability. The API service is implemented with the Django open source web framework [22]. Training states are stored in a relational database.

User job requests are distributed asynchronously by a worker microservice based on Celery [23]. The web service instances and the workers are decoupled through a RabbitMQ broker [24]. This design allows deployments of the DeepHealth Service to easily scale capacity as required. For cloud-enabled deployments, the Service has been ported to Kubernetes, and a configurable Helm chart has been created for simplified deployment.

VIII. Validation and Evaluation

A. Testing and Continuous Integration

The development of all DeepHealth libraries is supported by extensive test suites and automated Continuous Integration (CI) pipelines that execute them after every change to the source code. The pipelines run on the DeepHealth Jenkins server[1] and test the build of *C++* code with different compilers (GCC, Clang, MSVC). For the library-specific Docker container images described in Section VI, the CI pipelines also implement automated publication on Docker Hub, on condition that the image passes all tests.

B. Performance Scalability Analysis

In this section we evaluate the performance scalability of the DeepHealth Toolkit, focusing on the skin cancer melanoma classification use case based on the ISIC dataset [15]. The dataset contains $25,331$ dermoscopic images of unique benign and malignant skin lesions from over $2,000$ patients. In both training and inference phases, the dataset is split in batches that are processed in parallel. The use case employs VGG-16 [25] as reference topology, which won the ILSVRC-2014 [26] challenge. The input layer is a tensor of size $3 \times 224 \times 224$, and the output is a vector of 8 values corresponding to the 8 diagnostic categories.

[1] https://jenkins-master-deephealth-unix01.ing.unimore.it

TABLE I
Execution time breakdown of the training phase.

Function	Weight	Module
im2col	30.96%	EDDL
Eigen::internal::gebp_kernel	15.63%	EDDL
func@0x18810	10.39%	OpenMP
func@0x189a0	8.77%	OpenMP
get_pixel	8.08%	EDDL
add_pixel	7.42%	EDDL
cpu_conv2D_back._omp_fn.4	5.15%	EDDL
func@0x18190	3.34%	OpenMP
Eigen::internal::blas_data_mapper	4.97%	EDDL
cpu_conv2D_grad._omp_fn.2	1.48%	EDDL
cpu_d_relu._omp_fn.1	0.75%	EDDL
cpu_mpool2D._omp_fn.0	0.69%	EDDL
cpu_fill._omp_fn.2	0.62%	EDDL
ecvl::RearrangeChannels	0.60%	ECVL
cpu_relu._omp_fn.0	0.59%	EDDL
cpu_conv2D._omp_fn.0	0.56%	EDDL

The experiments have been performed on a dual-socket Intel Xeon Platinum 8160 CPU with 24 cores each, totaling 48 cores running at 2.10 GHz. Each core has two private L1 caches of 32 KB each (one for data and one for instructions) and a private L2 cache of 1 MB, and each CPU has an L3 cache of 32 MB shared by its 24 cores. The system is equipped with 96 GB of DDR4 memory distributed in 12 DIMMs, and runs a SuSE Linux Enterprise Server operating system.

Training. Table I shows the execution time breakdown of the training phase. The execution time percentage (weight) and the corresponding module (EDDL, ECVL or OpenMP) are reported for each relevant function.

It can be observed that most of the execution time of the training phase is spent for EDDL functions. In particular, the two most time-consuming functions are *im2col*, which takes 30.96% of the total execution time, and the *gebp* kernel of the Eigen library (which is called from EDDL) with a weight of 15.63%. The functions *get_pixel*, *add_pixel*, *cpu_conv2D_back._omp_fn.4* and *blas_data_mapper* also take a relevant portion of the execution time, between 4.97% and 8.08% each. In contrast, the EVCL library has a very small weight in the overall execution time of the training phase, being the *RearrangeChannels* the most time-consuming function with a weight of only 0.60%.

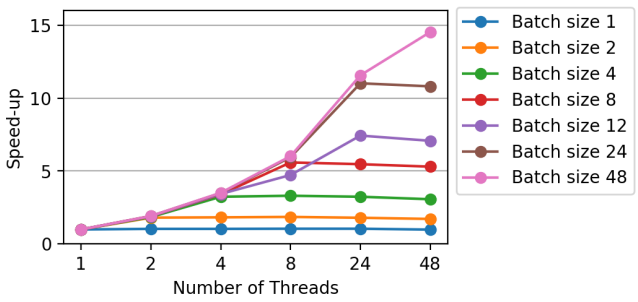


Fig. 2. Scalability of the training phase at different batch sizes.

TABLE II
EXECUTION TIME BREAKDOWN OF THE INFERENCE PHASE.

Function	Weight	Module
Eigen::internal::blas_data_mapper	46.62%	EDDL
im2col	22.37%	EDDL
get_pixel	10.78%	EDDL
func@0x18810	6.98%	OpenMP
Eigen::internal::gebp_kernel	6.48%	EDDL
func@0x189a0	2.66%	OpenMP
cpu_mpool2D._omp_fn.0	0.85%	EDDL
ecvl::RearrangeChannels	0.76%	ECVL
cpu_relu._omp_fn.0	0.72%	EDDL
cpu_conv2D._omp_fn.0	0.67%	EDDL
func@0x74b90	0.42%	OpenMP
decode_mcu	0.15%	ECVL
cpu_conv2D	0.12%	EDDL
jpeg_idct_islow	0.12%	ECVL
cpu_copy._omp_fn.1	0.08%	EDDL
fast_randn	0.07%	EDDL

In addition, it can be noticed that three OpenMP functions also have an important weight on the training step, consuming 10.39% of the execution in *func@0x18810*, 8.77% in *func@0x189a0*, and 3.34% in *func@0x18190*. These three routines are internal OpenMP functions to manage and synchronize threads: this implies that parallelization overheads are responsible for 22.5% of the total execution time.

Figure 2 details training phase scalability at different batch sizes, *i.e.* the speed-up obtained when increasing the number of threads from 1 to 48. It can be observed that the scalability improves as the batch size increases, reaching up to 14.7x speed-up with 48 threads and a batch size of 48. Another important observation is that the batch size is the main limiting factor for the scalability of the DeepHealth libraries. The Figure shows that when the number of threads is greater than the batch size no performance benefit is gained. This is due to the parallelization strategy adopted in the libraries, where each image of the batch is processed by one thread.

Inference. Table II reports how the execution time is distributed among EDDL, ECVL, and OpenMP functions during the inference phase.

As in the training phase, the execution time is dominated by function calls from EDDL. Four of the five most time-consuming functions are the *blas_data_mapper* Eigen kernel,

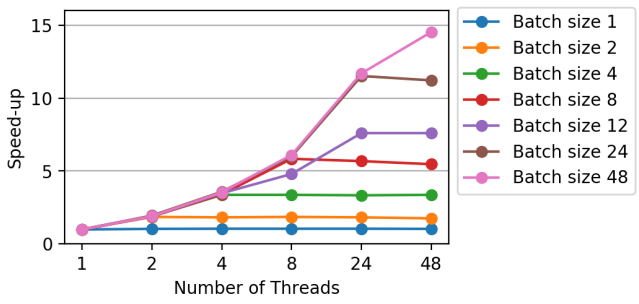


Fig. 3. Scalability of the inference phase at different batch sizes.

TABLE III
WALL CLOCK TIME IN SECONDS COMPARING *C++* AND PYTHON IMPLEMENTATIONS OF THREE EXAMPLE NETWORK TRAINING JOBS.

Job	*C++*	Python	Relative
MNIST_mlp	26.6	26.2	-1.5%
MNIST_mlp_train_batch	27.2	28.2	3.7%
MNIST_conv	332.0	330.6	-0.4%

the *im2col* function, the *get_pixel* function, and the *gebp* Eigen kernel, with respective weights of 46.62%, 22.37%, 10.78% and 6.48%. These four functions clearly dominate the execution time of the inference phase, as they combine for a total weight of 86.25%. Contrarily, the ECVL functions have a very limited weight on the execution time of the inference phase, with very modest contributions of up to 0.76% in *RearrangeChannels*, 0.15% in *decode_mcu* and 0.12% in *jpeg_idct_islow*.

Results in Table II also show that the OpenMP parallelization overheads represent a low percentage of the execution time of the inference phase. The three most time-consuming OpenMP functions are *func@0x18810*, *func@0x189a0*, and *func@0x74b90*, with weights of 6.98%, 2.66% and 0.42%, respectively. Thus, the parallelization overheads add up to only 10.06% of the total execution time.

Figure 3 shows the inference phase scalability at different batch sizes. The results are almost identical to those observed in Figure 2 for the training phase, reaching a maximum speed-up of 14.8x with 48 threads and a batch size of 48. As in the training phase, the scalability of the libraries is limited by the batch size, and using more threads than the number of images does not provide any performance benefit due to the adopted parallelization strategy.

C. Overhead of Python API

To measure the overhead imposed by the Python API with respect to the native *C++* DeepHealth libraries, we have measured the execution times of three different training jobs: MNIST_mlp is the multilayer perceptron training shown in Listing 6, MNIST_mlp_train_batch is the same job implemented with the finer grained API, MNIST_conv is a convolutional neural network training example. Table III shows the measurements from the experiment, averaged over five iterations. In all cases, the difference between the bare *C++* API and Python API is negligible; as expected, though, a slight decrease in performance can be seen when the job is implemented in Python with the lower level batch-by-batch API. All tests were run with PyEDDL 0.7.0 which maps EDDL 0.5.4a on an Nvidia GeForce RTX 2080 Ti GPU.

IX. AVAILABILITY

The entire DeepHealth Toolkit is open-source software released under the terms of the MIT license. The source code and online documentation —including installation instructions, API specification, and tutorials— are available on GitHub (see Table IV for details). Docker images of all components

TABLE IV
LINKS TO SOURCE CODE REPOSITORIES AND ONLINE DOCUMENTATION.

Source Code Repositories	
EDDL	https://github.com/deephealthproject/eddl
ECVL	https://github.com/deephealthproject/ecvl
PyEDDL	https://github.com/deephealthproject/pyeddl
PyECVL	https://github.com/deephealthproject/pyecvl
DeepHealth Service	https://github.com/deephealthproject/backend
Documentation	
EDDL	https://deephealthproject.github.io/eddl
ECVL	https://deephealthproject.github.io/ecvl
PyEDDL	https://deephealthproject.github.io/pyeddl
PyECVL	https://deephealthproject.github.io/pyecvl
Docker Images	
dhealth/libs	https://hub.docker.com/r/dhealth/libs
dhealth/libs-toolkit	https://hub.docker.com/r/dhealth/libs-toolkit
dhealth/pylibs	https://hub.docker.com/r/dhealth/pylibs
dhealth/pylibs-toolkit	https://hub.docker.com/r/dhealth/pylibs-toolkit
Applications	
Use Case Pipeline	https://github.com/deephealthproject/use_case_pipeline
ECVL Applications	https://github.com/deephealthproject/ecvl-applications

are published on Docker Hub under the `dhealth` organization. Installable pre-compiled versions of the EDDL and PyEDDL libraries are available as Conda packages under the https://anaconda.org/dhealth organization.

X. CONCLUSION

The main objective of the DeepHealth project is to offer a unified framework completely adapted to exploit underlying heterogeneous HPC and Big Data architectures to boost biomedical applications using state-of-the-art deep learning and computer vision algorithms. At the time of writing, the project is passing its mid-term review, the development of the whole DeepHealth Toolkit is progressing according to the work plan, and the estimate is that the pending developments will be completed in the remaining 18 months of the project.

ECVL and EDDL are already being used in several of the fourteen use cases of the project. The accuracy obtained by the tested models on distinct use cases shows that the implementations are correct. The implementation is not complete yet as we are halfway through the project, and more work is necessary to improve the performance (running times) of EDDL and ECVL and implement pending operators and optimizers.

An interesting outcome to highlight is the ECVL-EDDL interface, a software component that falls in the scope of both libraries and is devoted to build data-processing pipelines in order to run both training and inference procedures with data augmentations on-the-fly. The ECVL-EDDL interface improves GPU utilization by using the CPU to pre-fetch upcoming batches (loading samples from the file systems and applying data-augmentation transformations). The implemented pipelines demonstrate the advantages of this approach, addressing different classification and segmentation tasks.

Finally, we can conclude that the DeepHealth Toolkit and its HPC and cloud complements will be an alternative framework for data scientists working in the health sector, or any other sector, to implement solutions based on deep learning.

REFERENCES

[1] (2020, July) *deephealthproject* GitHub Organization. [Online]. Available: https://github.com/deephealthproject 1

[2] R. M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 3-4, pp. 32 – 36, 2015. 2

[3] (2020, July) The Open Neural Network Exchange (ONNX) format. [Online]. Available: https://github.com/onnx/onnx 2

[4] (2020, July) The Language-Neutral, Platform-Neutral Extensible Mechanisms for Serializing Structured Data. [Online]. Available: https://developers.google.com/protocol-buffers 2

[5] F. Chollet *et al.*, "Keras," https://keras.io, 2015. 2

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778. 2, 6

[7] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241. 2

[8] S. Allegretti, F. Bolelli, and C. Grana, "A Warp Speed Chain-Code Algorithm Based on Binary Decision Trees," in *4th International Conference on Imaging, Vision & Pattern Recognition*. IEEE, 2020. 3

[9] F. Bolelli and C. Grana, "Improving the Performance of Thinning Algorithms with Directed Rooted Acyclic Graphs," in *Image Analysis and Processing - ICIAP 2019*. Springer, 2019, pp. 148–158. 3

[10] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, "Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling," *IEEE Transactions on Image Processing*, vol. 29, pp. 1999–2012, 2019. 3

[11] S. Allegretti, F. Bolelli, and C. Grana, "Optimized Block-Based Algorithms to Label Connected Components on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 423–438, 2019. 3

[12] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Towards reliable experiments on the performance of Connected Components Labeling algorithms," *Journal of Real-Time Image Processing*, vol. 17, pp. 229–244, 2018. 3

[13] (2020, July) wxWidgets: Cross-Platform GUI Library. [Online]. Available: https://www.wxwidgets.org 3

[14] (2020, July) YAML: YAML Ain't Markup Language. [Online]. Available: https://yaml.org/ 3

[15] M. Combalia, N. C. F. Codella, V. Rotemberg, B. Helba, V. Vilaplana, O. Reiter, C. Carrera, A. Barreiro, A. C. Halpern, S. Puig, and J. Malvehy, "BCN20000: Dermoscopic Lesions in the Wild," 2019. 4, 6

[16] (2020, July) SIIM-ACR Pneumothorax Segmentation. [Online]. Available: https://www.kaggle.com/c/siim-acr-pneumothorax-segmentation 4

[17] W. Jakob, J. Rhinelander, and D. Moldovan. (2020, July) pybind11 – Seamless Operability Between C++11 and Python. [Online]. Available: https://github.com/pybind/pybind11 4

[18] S. Lyskov. (2020, July) Binder. [Online]. Available: https://github.com/RosettaCommons/binder 5

[19] (2020, July) Kubeflow: The Machine Learning Toolkit for Kubernetes. [Online]. Available: https://www.kubeflow.org/ 5

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255. 6

[21] (2020, July) OpenAPI Specification. [Online]. Available: https://swagger.io/docs/specification/about/ 6

[22] (2020, July) Django: The Web framework for perfectionists with deadlines. [Online]. Available: https://www.djangoproject.com 6

[23] (2020, July) Celery - Distributed Task Queue. [Online]. Available: https://github.com/celery/celery 6

[24] (2020, July) RabbitMQ - Messaging that Just Works. [Online]. Available: https://www.rabbitmq.com 6

[25] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014. 6

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. 6