DEEPHEALTH

D2.1 EDDL Library

Project ref. no.	H2020-ICT-11-2018-2019 GA No. 825111
Project title	Deep-Learning and HPC to Boost Biomedical Applications for Health
Duration of the project	1-01-2019 – 31-12-2021 (36 months)
WP/Task:	WP2/ T2.1, T2.2, T2.6
Dissemination level:	PUBLIC
Document due Date:	31/05/2020 (M17)
Actual date of delivery:	03/06/2020 (M17)
Leader of this deliverable	UPV
Author(s)	Roberto Paredes (UPV-PRHLT), Jon Ander Gómez (UPV-PRHLT) Salvador Carrión (UPV-PRHLT), Álvaro López (UPV-PRHLT) Silvia Nadal (UPV-PRHLT), Rafael Sánchez (UPV-PRHLT) Jorge Verdeguer (UPV-PRHLT), Francisco Casacuberta (UPV-PRHLT), Luca Pireddu (CRS4), Simone Leo (CRS4), Eduardo Quiñones (BSC), Philippe Dore (CEA), Olivier Bichler (CEA),
Version	1.0

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 82511

Document history

Version	Date	Document history/approvals
0.1	01/05/2020	First draft contents
0.2	28/05/2020	Document for review
0.3	02/06/2020	Corrections made from the review done by Ynse Hoornenborg as peer-reviewer
0.4	02/06/2020	Corrections made from the review done by Monica Caballero as project manager
1.0	03/06/2020	Definitive

Table of contents

Executive summary

This deliverable (D2.1) consists in the European Distributed Deep Learning Library (EDDL[1] Library), which is a software component and its documentation, publicly available in a repository on GitHub since M10 (October 2019). This document presents and describes (i) the work carried out in tasks T2.1, T2.2 and T2.6 of WP2 since M3 to M17, and (ii) the library and its documentation.

In its current status, the EDDL library covers 99% of the functionalities needed by the 7 platforms to implement and deploy the 14 use cases. A clarification is important at this point: not all pilot use cases will be implemented in all the platforms. Each use case is paired with at least one software platform as it is described in D1.1.

We would like to highlight that the documentation of the software components EDDL and PyEDDL is not going to be included in this document due to its extension (it is neither effective nor operative). Documentation of software, which also includes the API reference used by developers, is a living web document, any snapshot becomes obsolete immediately. Hence, the public repository where software and documentation are available will be referenced in this document.

This document is organised as follows:

Section 1 *"Introduction"*	summarises and reformulates the objectives of the EDDL, and puts in context the tasks T2.1 and T2.2 for EDDL and T2.6 for PyEDDL with respect to other tasks of the project.
Section 2 *"Global Overview"*	positions the EDDL (including the PyEDDL) within the DeepHealth toolkit.
Section 3 *"Development"*	explains the current status of the library design and implementation, including aspects like configuration, documentation, use of standard formats to import/export, and more details.
Section 4 *"Conclusions"*	gives a perceptual evaluation of the current development.

A final note regarding the impact of the COVID-19 pandemic on the activities relevant to this report. Fortunately, the pandemic has only had minor effects on these activities, mostly in terms of slightly reduced productivity due to the total absence of face-to-face interaction between collaborators and also as individuals work to manage personal situations caused by the imposed restrictions (e.g., closed schools and day cares). Nevertheless, the consortium has still been able to effectively organise its efforts and deliver these results according to schedule.

[1] From now on we will use the acronym EDDL because it is easier to pronounce.

1 Introduction

This document is about the design and implementation of the EDDL library including the Python wrapper (PyEDDL) being carried out in tasks T2.1, T2.2 and T2.6. This introductory section has the purpose of showing the relationships of EDDL and PyEDDL with other components of the DeepHealth toolkit, and the relationships of tasks T2.1, T2.2 and T2.6 with other tasks of the project, including tasks of other work packages.

Figure 1 illustrates the relationships between D2.1 and other deliverables. These relationships imply dependencies of other components of the project on the EDDL. Dependencies are correctly defined and solved by means of collaborations between the teams in charge of the corresponding tasks. Figure 1 also shows the periods while tasks T2.1, T2.2 and T2.6 are active.

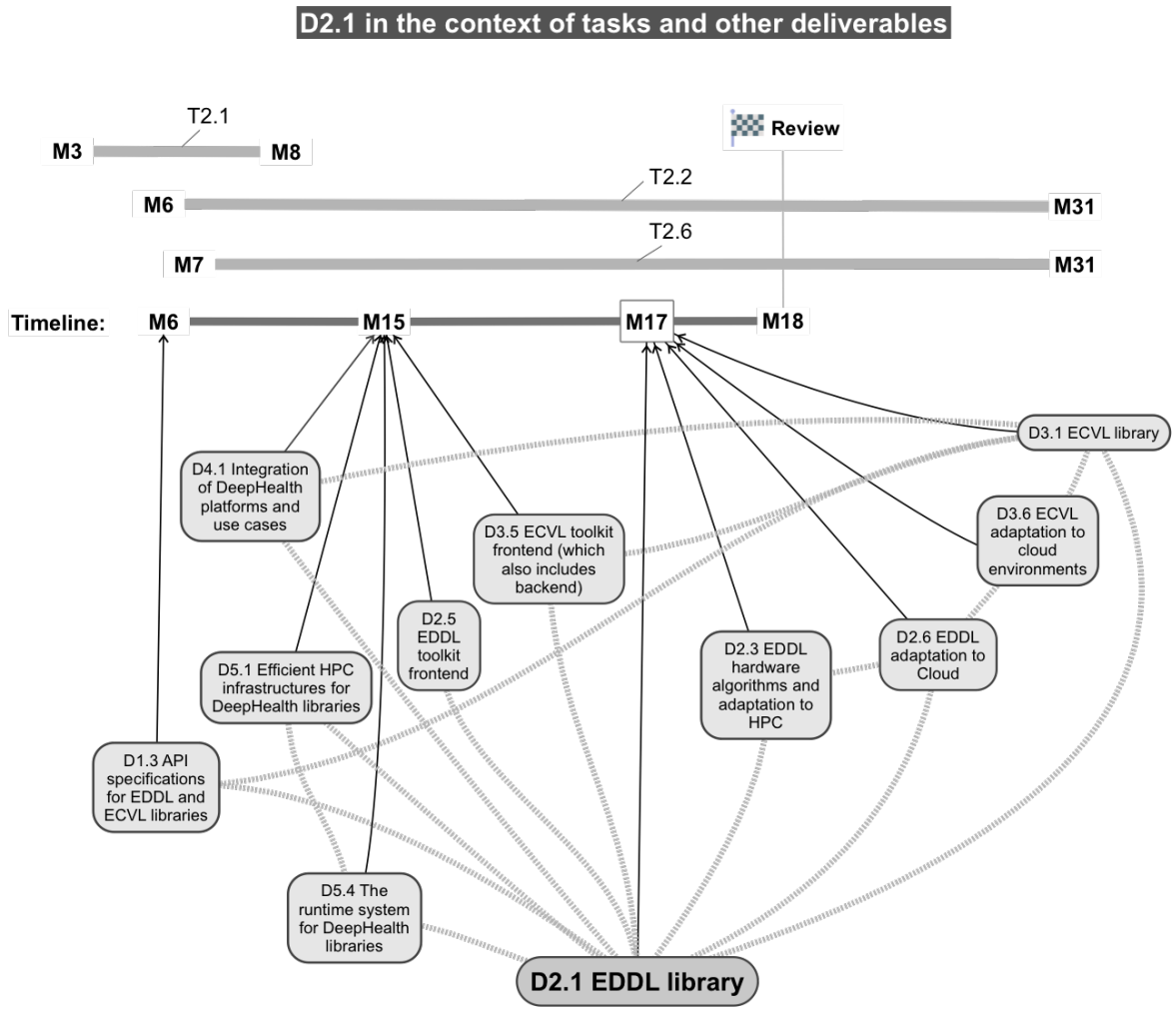


Figure 1: Context of deliverable D2.1 (a draft version of D2.2 due to M31). D2.1 includes the work carried out in tasks T2.1, T2.2 and T2.6 up to M17. Relationships with other deliverables are shown in this figure to illustrate the collaborative work.

1.1 Objectives of the EDDL library including the Python wrapper PyEDDL

The EDDL library is one of the outcomes of the DeepHealth project, that in addition to be exploited within the project when used in the 14 pilot use cases, also pursues to be used beyond the DeepHealth project and in other sectors than the Health sector. Let us summarise the objectives of this deliverable D2.1 and the related tasks T2.1, T2.2 and T2.6 in two ambitious goals:

1. Cover most used Deep Learning functionalities in the Health sector, in particular the required in the 14 pilot use cases of the DeepHealth project.
2. Become a widely used library:
   - Easy to be used and integrated by developers.
   - Easy to be installed and configured by Data / Computer scientists –we have to make it a ready-to-use software just after an easy installation.
   - Include most of the DL functionalities commonly available in other DL toolkits in addition to the functionalities needed within the DeepHealth project.

   In order to achieve this second goal EDDL+PyEDDL must be presented and published as a **General Purpose Library**. As some of the examples included in the package show, EDDL+PyEDDL is already a general purpose software that can be used to design, train, evaluate and put in production any network topology designed by a Deep Learning expert.

It is relevant to highlight at this point that an important goal of the whole DeepHealth project is to make the libraries use the available HPC and Cloud infrastructure in a transparent way for Data/Computer scientists. So the design and implementation of the EDDL+PyEDDL is also considering/including aspects in order to facilitate the transparent use of HPC and Cloud, i.e. adding the requiering functionalities in the EDDL API to make it possible the distributed training using COMPSs (see Section 3.5).

1.2 Directly related tasks

This deliverable consists in software developed in tasks T2.1, T2.2 and T2.6. Table 1 gives us the name and the time span of these three tasks.

Table 1: Tasks whose work is reported in deliverable D2.1 (this one).

T2.1	M3–M8	Design and implementation of the centralised version of the EDDL
T2.2	M6–M31	Design and implementation of the distributed version of the EDDL
T2.6	M7–M31	Design and implementation of a Python API

T2.6 was initially scheduled to finish at M12, but it has been extended to M31 in one of the amendments of the project in order to reflect the effort necessary to update the Python API every time a new version of the EDDL is released.

The period assigned to T2.1 was not enough to complete the implementation of the centralised version of the EDDL, the effort needed to complete T2.1 is being absorbed in T2.2. This is neither a problem nor a risk as the partner in charge of both tasks is UPV-PRHLT. The EDDL development plan is progressing according to the expectations. The centralised version of the EDDL is not 100% completed, but the current version covers the needs of the 14 use cases in order not to delay the start of the pilots. More details are provided in Section 3.

1.3 Indirectly related tasks

The outcomes of tasks T2.1, T2.2 and T2.6 (the ones this deliverable corresponds to) are needed in other tasks, in fact, the development of EDDL/PyEDDL and ECVL/PyECVL have been done in a close collaboration between partners. It also has been the case of the front-end, the back-end, and adaptations to HPC and cloud environments.

Frequent technical meetings have been organised between teams involved in tasks other than T2.1, T2.2 and T2.6. More specifically:

- Meetings between UNIMORE and UPV-PRHLT are taking place since the project started. Most of the meetings were specific for deciding how to use tensors and other objects from the EDDL in the ECVL. In particular, it plays an important role the conversion of images (grey or RGB) to EDDL tensors and

vice-versa, this kind of conversions are crucial to construct instruction pipelines to run training procedures where data augmentation is done on-the-fly thanks to functions provided by ECVL for loading and transforming images.

- Meetings between BSC, CRS4 and UPV-PRHLT took place, and still are taking place (e.g. the regular meetings of WP2 every month), to share concept designs and to decide how to implement specific functions in the EDDL and the PyEDDL in order to be used from COMPSs.

  The most important outcome is the serialisation of models using ONNX, the standard format for serialising deep neural networks, in order to communicate gradients and weights between worker nodes and the master node (more details in Subsection 3.5).

1.3.1 Pipeline with ECVL

As mentioned above, the training and inference procedures for many DNN topologies require the application of data augmentation techniques on-the-fly. Data augmentation for most of the DeepHealth use cases are image transformations carried out thanks to the functionalities provided by the ECVL. So making both libraries run together in a coupled way is mandatory for the project.

According to this, an important collaboration is needed between the teams working on tasks T2.1, T2.2 and T2.6 with the teams working on tasks T3.1 and T3.5. As part of this collaboration both teams are organising activities carried out collaboratively:

- `https://github.com/deephealthproject/use_case_pipeline`
- `https://github.com/deephealthproject/workshops`.

1.3.2 Back-end

UNIMORE has done an important development within T3.1 and T3.4 that has been reflected in D3.5. The back-end was not initially planned this way because in the proposal it was conceived as part of the front-end. However, during the design of the DeepHealth toolkit we realised the back-end and the front-end needed to be different software components of the DeepHealth toolkit, Figure 2 in Section 2 illustrates the toolkit's scheme.

The back-end acts as the runtime for both libraries (ECVL and EDDL), runs on server instances which can be deployed as Linux containers defined and created using Docker and orchestrated using Kubernetes (more details in D2.6 and D3.6).

In this line, the collaboration of UNIMORE and UPV-PRHLT also included few discussions about the design of the back-end according to the EDDL API, and the redesign of EDDL API when needed to facilitate the development of the back-end.

1.3.3 Front-end

Despite front-end and back-end are two different software components because their implementation is completely different, we have to point out that both components interact in the sense that all the procedures the user ask to be executed by using the front-end are actually executed by the back-end.

The front-end has been developed in T2.5 and T3.4 because, finally, the web-based graphical user interface is one, from which the expert user can choose to run functions and procedures involving ECVL, EDDL, or both.

The front-end has been described in deliverables D2.5 and D3.5, but still needs some updates as both libraries are updated during the DeepHealth project.

1.3.4 HPC and Cloud adaptation

The work carried and being carried out in tasks T2.3, T2.4, T3.2, T3.3, T5.1 and T5.4 needed, and still need, outcomes from tasks T2.1, T2.2 and T2.6. Tasks T2.4 and T3.3 finish in M17.

- Deliverable D5.1 (M15) describes the progress in the design and implementation of the kernels (very specific functions to run operations with tensors) adapted to run on FPGAs.

- Deliverable D5.4 (M15) details how the EDDL is used to train and inference in distributed HPC + Cloud architectures thanks to the adaptation to COMPSs, where one of the key points is the serialisation of models (i.e. neural networks) to report gradients from worker nodes to the master node and to send weights updates from the master node to the worker nodes.

  The serialisation of gradients and weights facilitates the definition of COMPSs tasks and data flow dependencies in order to allow COMPSs runtime to distribute the workload in the HPC+Cloud infrastructure.

- Deliverable D2.3 (M17) describes how the EDDL has been adapted to be used in HPC+Cloud infrastructures by using COMPSs. A short explanation can be seen in Subsection 3.5.

- Deliverable D2.6 (M17), corresponding to T2.4, describes how cloud environments are prepared to run training processes using EDDL/PyEDDL. Images of Linux containers have been defined and created to automatically launch instances using Docker and Kubernetes. The following link shows how to use Docker images:

  `https://github.com/deephealthproject/docker-libs`

2 Global overview

2.1 EDDL library within the DH toolkit

Figure 2 shows the scheme of the DeepHealth Toolkit with its main components. ECVL and EDDL appear at the same level, however, there is a small dependency, ECVL uses some EDDL data structures and functionalities.

Both libraries are integrated in the **back-end**, which acts like a runtime for both libraries in the sense that many (not all) of the functions provided by the libraries can be executed via the **back-end**. On the other hand, the **back-end** is running on a server in order to satisfy all the queries coming from the **front-end**. Obviously, multiple instances of the **back-end** can run on several servers thanks to docker-based Linux containers which can be instantiated dynamically according to the workload thanks to Kubernetes. Details on this are described in other deliverables: D2.3, D2.6 and D5.4.

The **back-end** runs the functions provided by the libraries in order to execute the instructions required by the user through the **front-end** (e.g. starting a training procedure which includes some image transformations in order to perform data augmentation: the DNN training is carried out by EDDL, the image transformations on-the-fly by the ECVL). Details of the **back-end** and the **front-end** are described in deliverables D2.5 and D3.5.

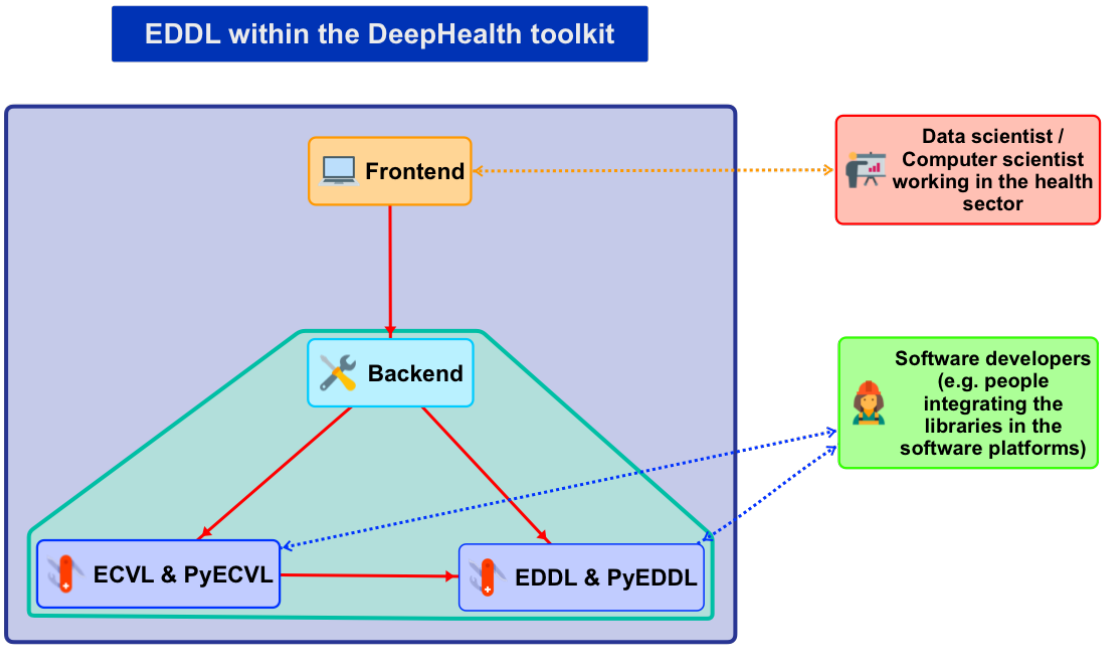


Figure 2: The EDDL library is one of the main components of the DeepHealth toolkit, providing the toolkit with all the Deep Learning functionalities. Red arrows show the dependencies between components: $A \rightarrow B$ indicates that the component A needs functionalities provided by component B.

2.2 Elements and features of the EDDL

Figure 3 illustrates a map with all the elements and features of the EDDL in addition to the design and implementation of the code. It can be observed the amount of details and aspects to be taken into account and implemented in order to make the library to be used by other developers who can choose to work with the EDDL (including the wrapper PyEDDL) instead of other toolkits like Tensorflow+Keras or PyTorch.

Next section (Section 3) gives more details about the current status of the development of the EDDL library, but it is important to highlight the effort done to integrate *Protocol Buffers* from Google in order to import/export models by using the ONNX standard format for neural networks. And the effort to make the installation as easy

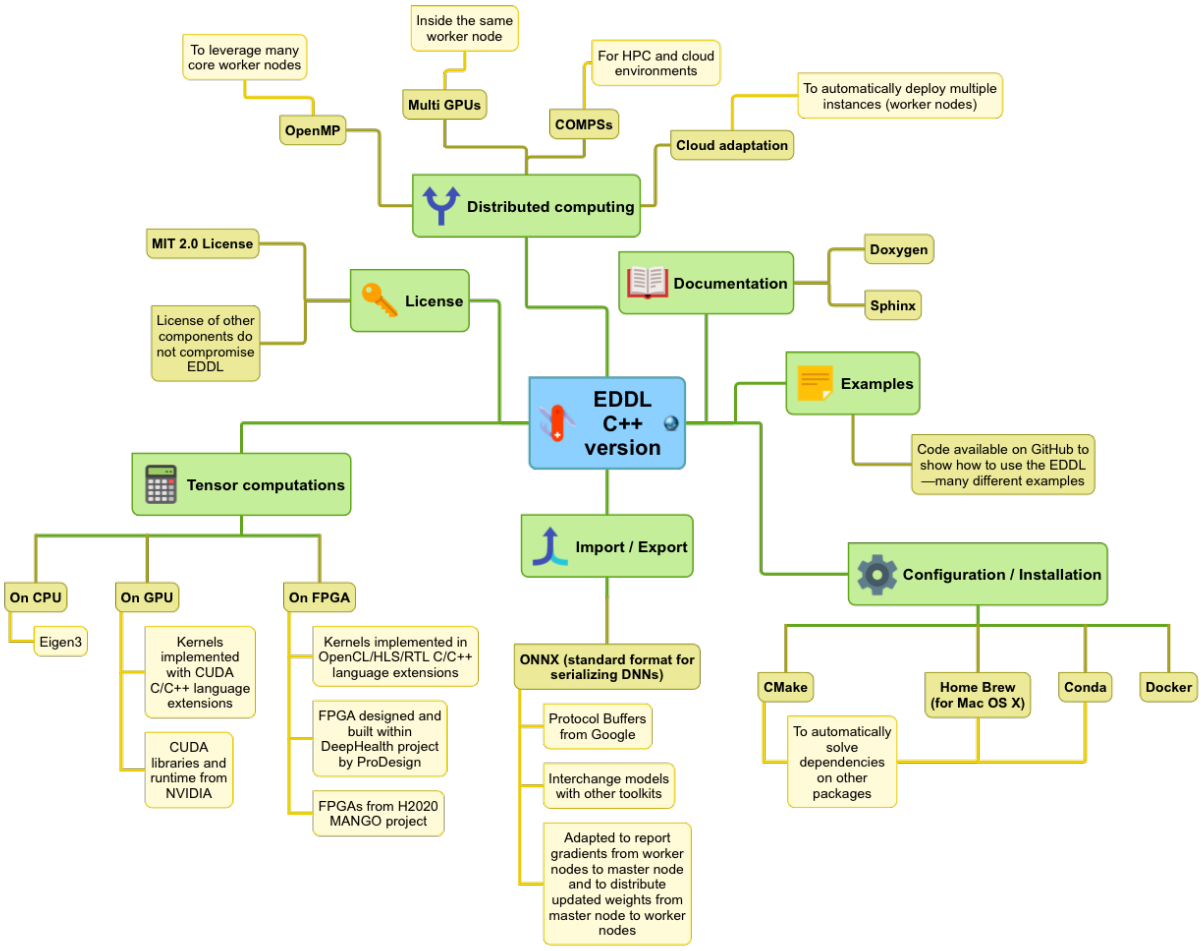


Figure 3: Map with elements and features of the EDDL in addition to its design and development.

as possible to other developers who can use the library or expert users who need just to design and evaluate neural network topologies for any particular problem/project.

The key elements of the EDDL shown in Figure 3 are:

Tensor computations for CPU and GPU have been developed in tandem. FPGA kernels are in progress and pending to be integrated. Eigen3[2] has been used for CPU and kernels developed by UPV-PRHLT in the `eddl::Tensor` class have been used for GPU. It is planned to use CUDNN from NVIDIA to reach training performances similar to TensorFlow and PyTorch when using GPUs.

Import/Export functionality uses the ONNX format for neural networks. *Protocol Buffers* from Google is the library used in C++ for serialising a whole model (with weights or gradients) according to ONNX definitions for layers, operators and network topologies. This point is crucial to make the DeepHealth toolkit compatible with other Deep Learning toolkits, but as mentioned above, there are differences in the use of the ONNX format, i.e. each toolkit introduces variants in the way they serialise some layers or operators leading to incompatibilities between toolkits.

Configuration/Installation as detailed in the next section and can be found in the documentation[3] [4], several alternative procedures for installation and configuration have been developed, and are still in progress.

[2] `http://eigen.tuxfamily.org/`
[3] `https://deephealthproject.github.io/eddl/installation/installation.html`
[4] `https://deephealthproject.github.io/pyeddl/`

This is one of the critical points for the deployment of EDDL and PyEDDL, and even for ECVL, PyECVL and the whole DeepHealth toolkit; making it easy for software developers and data/computer scientists required a great effort in person months, and will require a great effort still. As we are aware that an easy installation is crucial for achieving the goal of making these libraries widely used, a special working group with people from CRS4, UNIMORE and UPV-PRHLT has been created to address all the installation issues.

Examples EDDL and PyEDDL have available several examples at different levels of difficulty to illustrate other software developers and technical staff from partners of the DeepHealth project who are platform providers to integrate the EDDL and the PyEDDL in their platforms. Regarding the EDDL there are examples at different levels:

- Basic: `https://deephealthproject.github.io/eddl/usage/getting_started.html`
- Intermediate: `https://deephealthproject.github.io/eddl/usage/intermediate.html`
- Advanced: `https://deephealthproject.github.io/eddl/usage/advanced.html`

And regarding the PyEDDL:

- Basic `https://deephealthproject.github.io/pyeddl/getting_started.html`
- Intermediate:
  `https://deephealthproject.github.io/pyeddl/getting_started.html#additional-examples`

Although not all the examples in C++ are available in Python, we have to highlight that the examples in C++ can be translated to Python in an easy way for any developer, the opposite is a little bit more difficult, that is why there are more examples in C++.

Documentation is available online with the API reference manual for software developers. The URLs referenced above are also part of the documentation in relation to installation and examples of how to use the functions provided by the library. Other interesting parts of the documentation are:

- Troubleshot `https://deephealthproject.github.io/eddl/installation/troubleshoot.html`
- FAQ `https://deephealthproject.github.io/eddl/installation/faq.html`
- Video-tutorials `https://deephealthproject.github.io/eddl/videotutorials/developers.html`

And finally the API reference manual. Some examples are:

- Core layers: `https://deephealthproject.github.io/eddl/layers/core.html`
- Convolutional: `https://deephealthproject.github.io/eddl/layers/convolutional.html`
- Model: `https://deephealthproject.github.io/eddl/model/model.html`
- Losses: `https://deephealthproject.github.io/eddl/bundle/losses.html`
- GPU: `https://deephealthproject.github.io/eddl/bundle/computing_service.html#gpu`
- Tensors: `https://deephealthproject.github.io/eddl/tensor/create.html#`

Obviously, not all the sections of the documentation are completed so far. The generation of the documentation is automated by using Doxygen[5] and Sphinx[6].

Distributed computing is being done in a dual implementation:

- using COMPSs[7] from BSC in Python with the PyEDDL by one hand, and
- directly in C++. First attempt in C++ uses OpenMPI[8], but we are going to use another approach because OpenMPI is not flexible enough for the purposes of the DeepHealth project, one weak point is that the data transfer from the master node to many worker nodes is not actually implemented with true broadcasting. This is a difficult issue to solve even using other approaches than OpenMPI.

[5] `http://doxygen.nl/`
[6] `https://www.sphinx-doc.org/`
[7] `https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar`
[8] `https://www.open-mpi.org/`

In both cases, the distributed computing is based on the ONNX serialisation functions implemented in the EDDL which make it possible to serialise a complete model including the weights or the gradients. This facility makes it straightforward to report gradients from worker nodes to the master node and to update the weights from the master node to the worker nodes.

And also in both cases the use of the underlying HPC+Cloud infrastructure is defined in one or more configuration files.

Multi-threading on many-core computers (for CPU) and on machines with multiple GPUs is not considered distributed computing in the DeepHealth project because no data distribution is needed. Currently this has been done using OpenMP[9] in the case of CPUs, and developed internally in the case of GPUs.

License used for all DeepHealth outcomes is the MIT License:

- EDDL `https://github.com/deephealthproject/eddl/blob/master/LICENSE`
- PyEDDL `https://github.com/deephealthproject/pyeddl/blob/master/LICENSE`

The licences of the dependencies of EDDL and PyEDDL do not compromise the distribution of the EDDL and PyEDDL as open source and free software:

- ONNX: `https://github.com/onnx/onnx/blob/master/LICENSE`
- Protocol Buffers: `https://github.com/protocolbuffers/protobuf/blob/master/LICENSE`
- Eigen3: `http://eigen.tuxfamily.org/index.php?title=Main_Page#License`
- ZLib: `https://zlib.net/zlib_license.html`
- Google Test: `https://github.com/google/googletest/blob/master/LICENSE` – but it is optional and does not implies any restriction to the user of the EDDL.
- NVIDIA CUDA: `https://docs.nvidia.com/cuda/eula/index.html`

Regarding the use of *Protocol Buffers* in the EDDL, it was decided to add this dependency to the EDDL after verifying that it is the unique alternative to serialise models in C++ using the ONNX format; otherwise we would have had to implement from scratch all the required data structures and functions to serialise models.

These licenses are not restricting and allow any software developer company to use the EDDL without restrictions, i.e. for deploying their solutions that integrate the EDDL library even commercially.

[9] `https://www.openmp.org/`

3 Development

The EDDL library is being coded using the C++ programming language. All the computations are faster in C/C++, in particular the internal computations involving operations with matrices and mathematical functions. Additionally, the use of GPUs from languages like C and C++ is possible thanks to the CUDA language extensions for C/C++. Another implementation detail important to be highlighted is that the development for CPU and GPU has been and is being carried out in tandem. This has been possible thanks to the design and implementation of a C++ class named `Tensor`[10] where all the tensor operations are concentrated, including matrix element-wise operations and dot product as a particular case of 2D-tensor operations. The `Tensor` class plays the role of a hardware abstraction layer. If the model has been built to run on GPU, then tensors are created in the GPU memory and the tensor operations are done by the GPU, otherwise the operations are done in the CPU using as many cores as specified when the model is built. Which device to use and which resources of each device to use is specified with *computing service* objects[11].

The Python version of the EDDL, named PyEDDL, is in fact a wrapper to transparently use the C++ functions in Python.

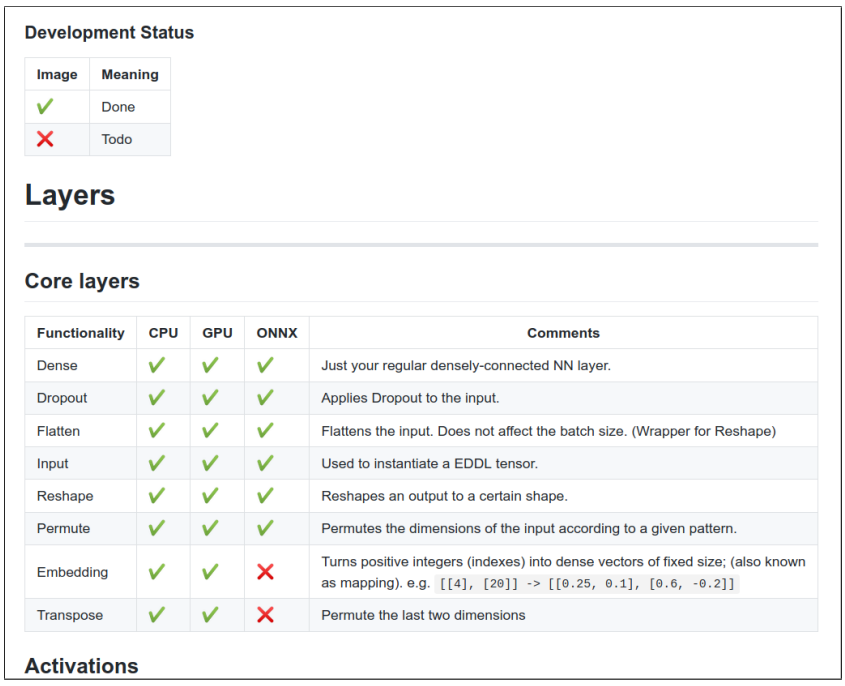
3.1 Current status

The current status of the development can be consulted at:

- EDDL development status:

  `https://github.com/deephealthproject/eddl/blob/master/docs/markdown/eddl_progress.md`

Figure 4 shows an screenshot of the progress where it can be seen how each element can be in state **done** or **todo**. The figure illustrates some functionalities of the core layers and if they are implemented for CPU and/or GPU and if are included in ONNX.



Development Status

Image	Meaning
✓	Done
✗	Todo

Layers

Core layers

Functionality	CPU	GPU	ONNX	Comments
Dense	✓	✓	✓	Just your regular densely-connected NN layer.
Dropout	✓	✓	✓	Applies Dropout to the input.
Flatten	✓	✓	✓	Flattens the input. Does not affect the batch size. (Wrapper for Reshape)
Input	✓	✓	✓	Used to instantiate a EDDL tensor.
Reshape	✓	✓	✓	Reshapes an output to a certain shape.
Permute	✓	✓	✓	Permutes the dimensions of the input according to a given pattern.
Embedding	✓	✓	✗	Turns positive integers (indexes) into dense vectors of fixed size; (also known as mapping). e.g. `[[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]`
Transpose	✓	✓	✗	Permute the last two dimensions

Activations

Figure 4: Screenshot of the web page with the current development status.

[10] In this context a tensor is a multidimensional array. The tensor documentation in the EDDL can be found at `https://deephealthproject.github.io/eddl/tensor/manipulation.html`

[11] `https://deephealthproject.github.io/eddl/bundle/computing_service.html`

3.1.1 GitHub repository

All the implemented code, examples, tutorials and other documentation is available on a GitHub public repository:

- EDDL `https://github.com/deephealthproject/eddl`
- PyEDDL `https://github.com/deephealthproject/pyeddl`

3.1.2 Online documentation

The online documentation has been already referenced in the previous section (Section 2). All the documentation is automatically generated from the comments included in the code and other configuration files by means of Sphinx-doc[12] and Doxygen[13].

- EDDL documentation main entry point `https://deephealthproject.github.io/eddl/`
- PyEDDL documentation main entry point `https://deephealthproject.github.io/pyeddl/`

3.1.3 Installation options

There are different options to install the EDDL, as commented in other parts of this document the automatic installation of the libraries is one of the most difficult tasks we faced.

Several options have been implemented in order to facilitate the installation to developers and computer/-data scientists. The goal is to achieve that any person interested in using or at least testing the EDDL and the PyEDDL can do it in few minutes. With this purpose, specific channels for DeepHealth have been created in the most popular package managers.

Next, the different options based on different package managers are listed, with the reference to the EDDL GitHub repository where all the installation steps are detailed:

- **Conda** `https://conda.io/` Figure 5 shows the command line to install the EDDL compiled for GPU, it is assumed that the user already has the **conda** package manager installed. The installation of **conda** is explained in `https://conda.io/`

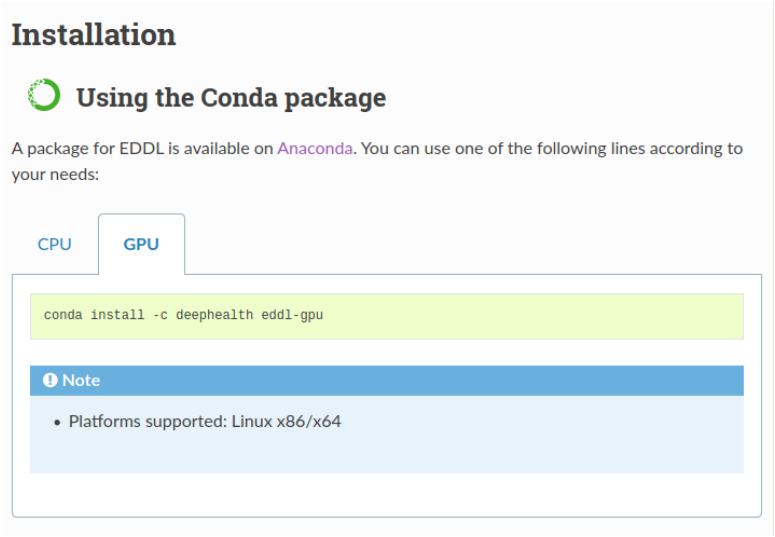


Figure 5: `https://deephealthproject.github.io/eddl/installation/installation.html`

- **HomeBrew** for Mac OS X `https://brew.sh/` Figure 6 shows the steps required to install the EDDL in Mac OS X assuming that the **homebrew** package manager was already installed.

[12] `https://www.sphinx-doc.org/`
[13] `http://doxygen.nl/`

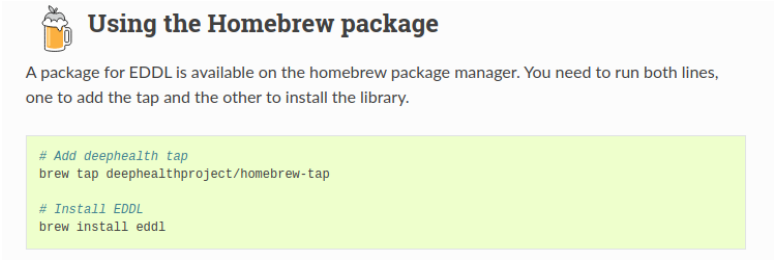


Figure 6: https://deephealthproject.github.io/eddl/installation/installation.html

- **CMake**: From source (manual installation –special for developers) https://cmake.org/

  Figure 7 shows the steps required to install the EDDL in Unix like systems, for Mac OS X is also available. This option will be the one preferred by developers, where all the source code of the EDDL can be downloaded from GitHub.

  In order to solve dependencies on other packages, here it can be used the **conda** package manager or the SUPERBUILD configuration option of the EDDL that solves the dependencies independently of package managers, but only for installation from source.

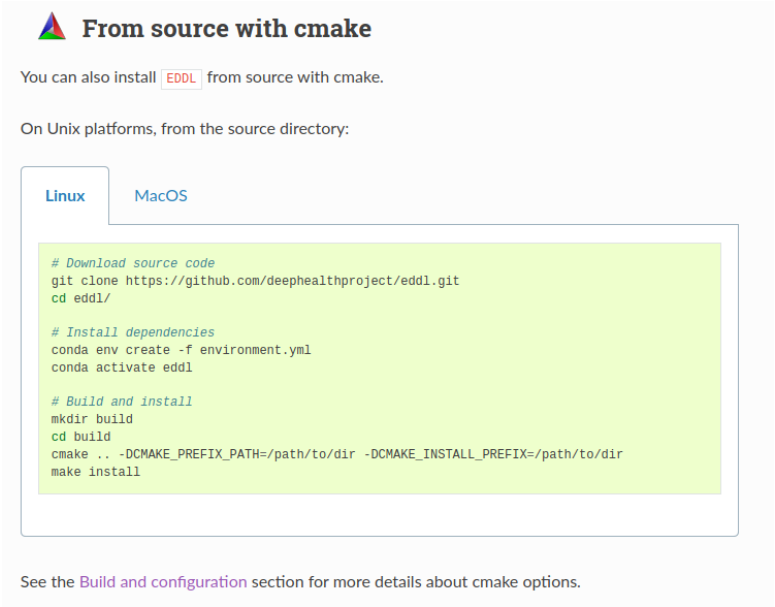


Figure 7: https://deephealthproject.github.io/eddl/installation/installation.html

- **Docker** https://www.docker.com/

  Docker images are available with different configurations of the libraries already installed.

  – https://github.com/deephealthproject/docker-libs

  Note: This is a work done by CRS4 as part of the strategy to automate the installation of the libraries in cloud environments, so this is more part of T2.4 than one of the tasks related to this deliverable.

- **Kubernetes** https://kubernetes.io/

  Not mature enough. This is not exactly an installation alternative, is complementary to **docker-libs** in order to deploy cloud instances when both DeepHealth libraries ECVL and EDDL will be ready to be used.

  – https://github.com/deephealthproject/deephealth-k8s

Note: This is a work done by TREE as part of the strategy to automate the installation of the libraries in cloud environments, so this is more part of T2.4 than one of the tasks related to this deliverable.

3.2 What has been done since M3 and what is pending

Complete and ongoing:

- EDDL library: all the code has been written from scratch. The previous developments from UPV-PRHLT (the **UPV DL library**[14] –developed by Roberto Paredes and Jon Ander Gómez from PRHLT as part of an automatic speech recognizer, not publicly available because it is registered software of the UPV– and **Layers**[15] –developed integrally by Roberto Paredes from PRHLT–) have served as a crucial experience to avoid common developing errors as well as to face the problems which appeared in the current development. **[Ongoing & according to the work plan]**
- PyEDDL – Python wrapper for the EDDL: **[Ongoing & according to the work plan]**
- Creation of the GitHub repositories. **[Complete]**
- Generation of the documentation. **[Ongoing & according to the work plan]**
- Easy installation procedures. **[Ongoing & according to the work plan]**
- Import/Export using ONNX. **[Ongoing & according to the work plan]**
- Distributed version using COMPSs with the PyEDDL the ONNX serialisation functions. **[Ongoing & according to the work plan]**

Pending:

- Bidirectional LSTM layer type.
- GRU –Gated Recurrent Unit– layer type.
- Attention model.
- Include last implemented layer types in ONNX.
- Alternative distributed implementations to carry out a comparative study focused on performance.

3.3 Which functionalities required by other partners are currently covered

DeepHealth has 14 use cases that are paired each one with at least one application platform to develop the pilot test beds. Use cases UC1 and UC5 require very specific functionalities:

- UC1 needs unidirectional LSTM and the fast $tanh(x)$ activation function defined as $f(x) = \frac{x}{1+|x|}$. **Done**.
- UC5 needs GRU and attention model to generate text sequences (what is well known as **seq2seq** models). **Pending**.

In summary, all requirements are satisfied except recurrent layers of the type GRU and the possiblity of building models to use RNNs to generate sequences with or without attention model. It is expected to have this functionalities ready to be used by the end of M19.

[14] https://aplicat.upv.es/exploraupv/ficha-tecnologia/patente_software/15415
[15] https://github.com/RParedesPalacios/Layers

3.4 Which HW is currently supported

Currently the EDDL and by extension the PyEDDL can use multiple cores of CPUs and several GPUs installed in a single computer. What is included here as a description of how GPUs are used is also included in deliverable D2.3 *"EDDL Hardware algorithms and adaptation to HPC"*. Deliverable D2.3 also includes details about the development of the code to run some specific functions of the EDDL on FPGAs, but the code to use FPGAs is not mature enough to be integrated in the stable and development versions of the EDDL.

Nevertheless, the EDDL library can increase free the amount of supported hardware by relying on other frameworks. This is quite straightforward for inference code generation thanks to the support by the EDDL of the ONNX exchange format. As depicted in D2.3 deliverable section 5.1, FPGA is supported for inference through a design flow involving both the EDDL library and the CEA N2D2 Framework. In such configuration, the EDDL library get access to every inference code generation provided after checking the capability of the framework used as a third party to import the Neural Network generated the EDDL library. Moreover, thanks to the activity carried out by CEA in WP5, the flow could be extended to distribute the inference code other several FPGAs.

Regarding CPUs it has been already commented in Section 2 that the Eigen3[16] library is used to perform algebraic operations with tensors and the use of multi-threading for leveraging many-core computers is done by means of OpenMP[17]. The user can decide how many threads to use in the `build` function:

```
build(net, optimiser, {losses}, {metrics}, CS_CPU( nthreads ) );
```

The default value of `nthreads` is -1, this indicates the EDDL to use one thread per core of the CPU.

In the case of GPUs the CUDA C/C++ extension language has been used to implement the kernels which perform all the tensor computations on GPUs. The use of one or more of the GPUs available in a single computer is explained below, where some performance figures are presented.

GPU accelerators can be selected in the process of building the neural network using the `CS_GPU` object. This is the unique difference in the program that user must consider in order to select among the different accelerators. The neural network design and the training procedure is agnostic about the hardware accelerator selected. Therefore, in the building command the user can select the *Computing Service* associated to the GPUs accelerators:

```
build(net, optimiser, {losses}, {metrics}, CS_GPU( params ) );
```

Where params are the following:

- Binary vector selecting the available accelerators (mandatory),
- Memory level requirements (optional, default=`"full_mem"`).
- Synchronisation delay parameter (optional, default=1).

  This parameter indicates the number of batches the training process will wait to merge the weights of the networks running in the GPUs and then update the weights in all the active GPUs; it is applicable when using 2 or more GPUs.

EDDL allows the user to select among the available GPUs. For instance, in a system with 4 GPUs installed the user can specify to use only the first and third GPU by means of using a binary vector in the `CS_GPU` object:

```
CS_GPU( {1,0,1,0} )
```

EDDL checks whether the system actually has these 4 GPUs installed and select only those activated by the user. Regarding the memory levels, GPUs use to have low memory with respect to the main computer memory. This GPU memory is a worth resource and EDDL propose three different memory levels:

[16] `http://eigen.tuxfamily.org/`
[17] `https://www.openmp.org/`

Table 2: Running times in minutes to complete 50 epochs on the CIFAR10 dataset using a VGG16 neural network model with a batch size of 100 samples for different values of the synchronisation delay in the case of using 2 GPUs; the synchronisation delay is not necessary when using 1 GPU. HW: Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz with 12 cores and two NVIDIA GeForce RTX 2080.

GPUs	DELAY	Time (minutes)
1	–	50
2	1	49
2	10	33
2	100	31
2	1000	31

`full_mem` EDDL tries to get as much memory as possible to ensure the fastest processing.
`mid_mem` EDDL performs some memory saving with a low speed degradation.
`low_mem` EDDL performs a strong memory saving with a significant speed degradation.

An example of a 2 GPUs system where both GPUs are selected with a `"low_mem"` setup:

```
CS_GPU( {1,1}, "low_mem" )
```

Finally, when using multiple GPUs we perform *"data parallelism"*. The neural network is replicated in all the GPUs but the data is split and distributed among them. In this scenario we must ensure that GPUs have the same parameters of the network. To this end the GPUs synchronise the parameters every batch of data processed. This synchronisation entails a significant overhead in time. Therefore, we propose to postpone this synchronisation after some batches, being this value set by the parameter of delay synchronisation.

For instance, an example of a 2 GPUs system where both GPUs are selected with a `"mid_mem"` setup and synchronised every 10 batches will be:

```
CS_GPU( {1,1}, 10, "mid_mem" )
```

3.4.1 Performance Results

Two different experiments have been carried out in order to assess the performance of the GPU accelerators regarding the different parameters involved.

CIFAR10

In the first experiment we deal with a classical problem (CIFAR10[18]) that does not entail any special memory requirements (always `full_mem`). We build a VGG16[19] neural network, train with batch size 100 and run a total of 50 epochs. Table 2 shows the total training time in minutes varying the number of GPUs and the delay parameter.

UC12 Skin lesion classification

In the second experiment we use a larger neural network topology also based on the model VGG16 and deal with the UC12 about skin classification. This experiment requires to take care of the amount of memory since the size of the images and the neural network topology are larger. Table 3 shows the time in seconds for one epoch on the UC12 about skin classification using 1 GPU varying the batch size and memory requirements. Each epoch comprises a total number of 19328 samples. The void cells represent those setups that do not fit into memory.

Table 4 shows the performance results using a system with 2 GPUs, varying the batch size, memory requirements and delay synchronisation parameter. The time is show in seconds for one epoch. The void cells represent those setups that do not fit into memory.

[18] `https://www.cs.toronto.edu/~kriz/cifar.html`
[19] Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", arXiv:1409:1556v6, `https://arxiv.org/abs/1409.1556v6`

Table 3: Running times in seconds to complete one epoch using a VGG16 neural network model with the UC12 about skin cancer classification; different batch sizes and different memory configurations were tested in the case of using 1 GPU. Missing results correspond to those setups that the model plus one batch do not fit into the GPU memory. HW: Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz with 12 cores and two NVIDIA GeForce RTX 2080.

Batch \ Mem	full	mid	low
8	746	755	669
16	–	–	647
32	–	–	615

Table 4: Running times in seconds to complete one epoch using a VGG16 neural network model with the UC12 about skin cancer classification; different batch sizes, different memory configurations and different synchronisation delays were tested in the case of using 2 GPUs. Missing results correspond to those setups that the model plus one batch do not fit into the GPU memory. HW: Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz with 12 cores and two NVIDIA GeForce RTX 2080.

	Mem								
	full			mid			low		
Batch \ Delay	1	10	100	1	10	100	1	10	100
8	3120	860	571	3292	1002	617	3231	967	616
16	2287	727	441	2408	768	478	2405	747	470
32	–	–	–	–	–	–	2654	587	382

The figures shown in Table 4 verify the expected effect of the synchronisation delay and the batch size, both configuration parameters with a significant improvement in the performance. It can be observed how the improvement from 1 to 10 is much higher than the improvement from 10 to 100. Depending on the model complexity and the dataset used to train, the delay is a parameter that must be adjusted by the computer scientist using the library in order to find a balance between convergence and performance.

Regarding the use of 1 GPU vs 2 GPUs, we have to compare the results obtained by using 1 GPU with the ones corresponding to a delay of 100 batches when using 2 GPUs, because in the case of 1 GPU no synchronisation is required.

Table 5 shows comparative results and the speedup factor reached when using 2 GPUs. The speedup factor depends on both the level of memory requirements and the batch size. It can be observed the batch size is the parameter with a higher impact. The best speedup factor is obtained with the larger batch size but with the low memory setup, these results remark the relevance of the GPU memory.

3.5 Parallelisation and distribution of EDDL training operations with COMPSs

This section describes the parallelisation of the training operation of the PyEDDL using the pyCOMPSs programming model. Figure 8 shows a snipped of the source code of the PyEDDL training operation parallelised with COMPSs (simplified for readability purposes). The COMPSs tasks (or simply tasks) correspond to Python

Table 5: Comparative results of using 1 GPU vs 2 GPUs, times are show in seconds, the values of the column corresponding to 2 GPUs also appear in Table 4. Last column shows the **speedup factor**. HW: Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz with 12 cores and two NVIDIA GeForce RTX 2080.

Config \ GPUs	1	2	speedup
`low_mem bs=8`	746	616	1.211
`low_mem bs=16`	647	470	1.376
`low_mem bs=32`	615	382	1.609
`mid_mem bs=8`	755	478	1.579
`full_mem bs=8`	669	571	1.171

```
1.  @task(is_replicated = True)
2.  def build (model):
3.    # The net model is created at each worker
4.    […]
5.    return model
6.
7.  @task(returns = [Tensor], parameters = IN, dataset = IN)
8.  def train_batch(parameters, dataset):
9.    # A train operation is executed at each worker on the net model and
10.   # the dataset specified
11.   […]
12.   return trained_weights
13.
14. def main():
15.   # A new model is created
16.   net = eddl.model([…])
17.   build(net)
18.   for i in range(num_epochs):
19.     for j in range(num_batches):
20.       weights[j] = train_batch(net.getParameters(), dataset[j])
21.     # Synchronize all weights from workers
22.     compss_wait_on(weights)
23.     # Update weights on the model
24.     update_gradients(net, weights)
```

Figure 8: Snipped of the distributed training.

functions that can be potentially executed in parallel and are identified with a standard Python decorator `@task` (lines 1 and 6). At execution time, the COMPSs runtime is responsible of distributing the execution of the COMPSs tasks among the available computing resources (defined in the `resource.xml` and `projects.xml` configuration files). The Python decorator `@task` can define the following arguments:

- `IN/OUT` (line 7) defines the data direction of the associated function parameters, and so the data dependencies existing among COMPSs tasks[20]. A task with a data element defined as in `IN` argument cannot start its execution until the previously defined tasks with the same data element defined as `OUT` complete.
- `is_replicated=True` (line 1) forces the COMPSs task to be executed in all available computing resources. This feature is used to ensure that all computing resource have the model in its local memory.

In Figure 8, the main function starts by creating and building the network model and making it available at each worker using the `is_replicated` argument (lines 16 and 17). Then, the function iterates synchronously over `num_epochs` epochs (line 18). At every epoch, a COMPSs task is instantiated (with the updated model parameters as argument) to perform a train operation over a given batch of the data-set already available on the worker node (lines 19 and 20). Given an epoch, all COMPSs tasks are synchronised with `compss_wait_on` (line 22), that makes the COMPSs master[21] wait until the completion of all tasks created at line 20. Then, the parameters trained by each task are collected and unified, and set back to all COMPSs task to start a new epoch. It is important to remark that the complete model is not transfer among computing resources, but only the parameters locally trained at each worker are transferred. The model is created locally at each computing node in the `build` function.

The task parallelism exposed by an execution of the distributed version of the PyEDDL training operation can be represented as a Task Dependency Graph (TDG) shown in Figure 9. Each node corresponds to a COMPSs task instantiating and edges represent data dependencies (and so data transfers) among tasks. Notice that `update_gradients` function is not a COMPSs task but has been included to represent the synchronisation point of `compss_wait_on`. The number of batches (defined by the `num_batches` variable at line 19 of Figure 8) determines the “width” of the graph; the number of epochs (defined by the `num_epochs` variable at line 18 of Figure 8) determines the “height” of the graph.

[20] By default, parameters are `IN` and returned values are `OUT` and so there is no need to explicitly specify them in the `@task` decorator; they are however included at lines 1 and 7 for clarification purposes.

[21] Notice that COMPSs runtime is organised as a master-worker structure, where the master is responsible of, among others, executing the main function, creating tasks and instructing workers to execute them (see D5.4 “The runtime system for DeepHealth libraries” (March 2020) for further details).

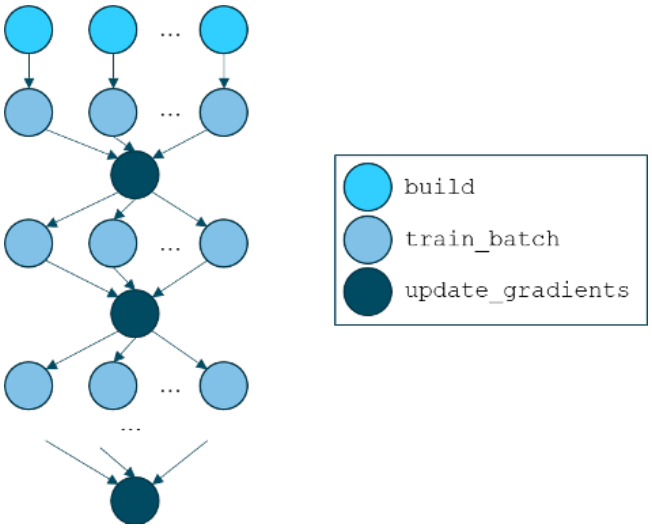


Figure 9: Tasks representation of the application presented in Figure 8.

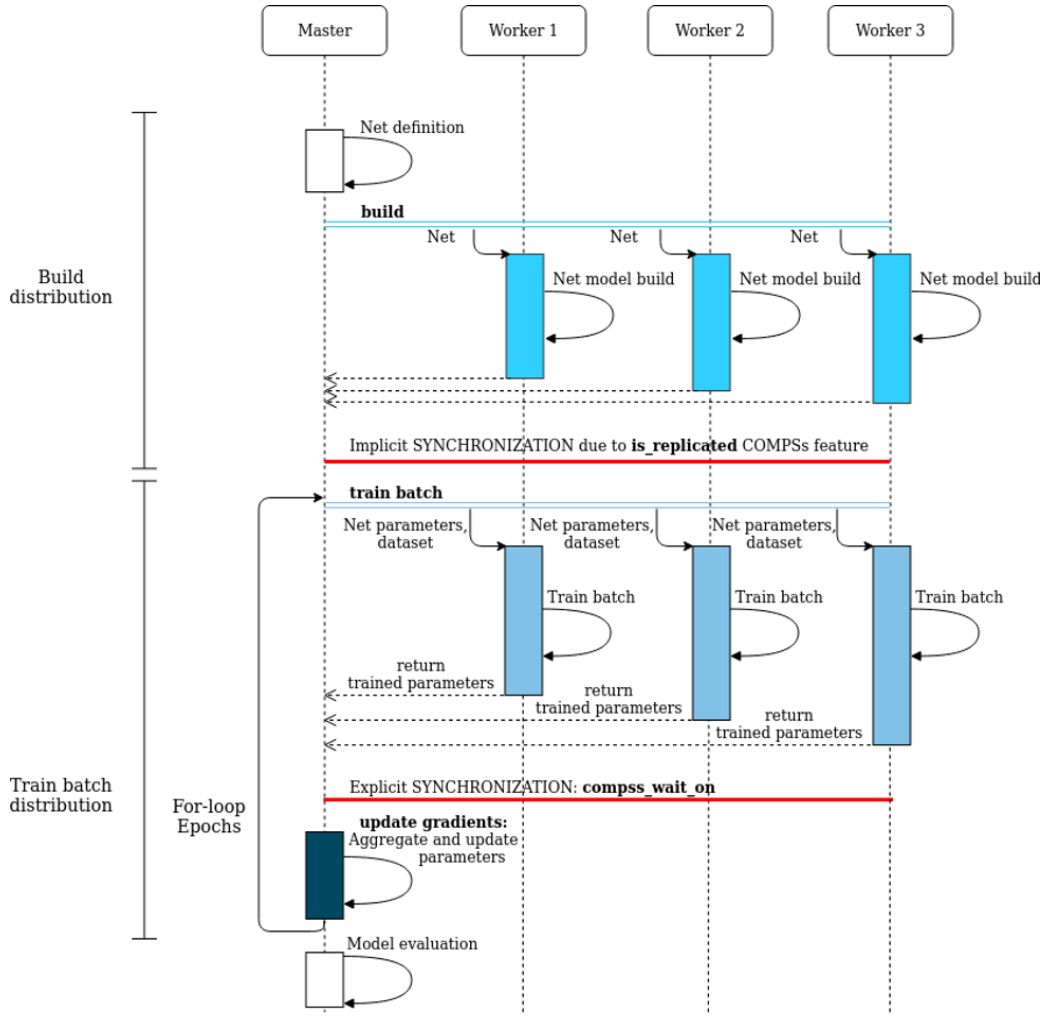


Figure 10: Execution workflow of the distributed training.

Figure 10 shows the execution timeline of the PyEDDL training operation distributed among three COMPSs worker nodes. The distributed training is divided into two parts: (1) the net model is built in all the computing nodes available so that each worker have access to it (named *build distribution* in the figure); and (2) the distributed training operation itself is performed over a given number of epochs (named *Train batch distribution* in the figure).

We refer to the reader to see deliverables D2.6 “EDDL adaptation to cloud environments” and D2.3 “EDDL Hardware algorithms and adaptation to HPC” for a preliminary performance speed-up analysis on the on-premise cloud provided by TREE and the Marenostrum supercomputer provided by BSC.

3.6 Issues addressed during development

As any ambitious project this one has also had to face unexpected problems and code refactorisations. Several code refactorisations have been carried out since the beginning with the purpose of improving the organisation of the code and making it easier for other developers to use the functions provided by the library; however, the three code refactorisations carried out so far are normal in any development project and have not represented an extraordinary effort.

What has really taken an extra effort is (a) the implementation of automatic installations and (b) the integration of ONNX format to import and export neural networks in order to be compatible with other toolkits. Making the installation easy to other partners of the project and any other people interested in using the EDDL library made us to spend many developer hours.

The different possibilities of installing the EDDL and the PyEDDL presented in subsection 3.1.3 need to solve many different issues related to:

(a) solve dependencies on packages of EDDL and PyEDDL like

- Eigen3 `http://eigen.tuxfamily.org/`
- Protocol Buffers `https://github.com/protocolbuffers/protobuf`
- ZLib: `https://zlib.net`

(b) prepare all the configuration files to install the EDDL via

- **conda** `https://conda.io/`
- **homebrew** `https://brew.sh/`
- Advanced Packaging Tool (**APT**) `https://packages.qa.debian.org/a/apt.html`
- or manually

(c) deal with bugs found in CMake[22].

3.6.1 ONNX compatibility with variants used in other toolkits/frameworks

Regarding the adoption of ONNX to import/export neural networks, the main problems have been:

1. the incompatibilities between different versions of *Protocol Buffers*. If there is an earlier version installed in the Linux system the compilation of the EDDL could fail. When installing the compiled version of the EDDL is worse, because if incompatible versions of *Protocol Buffers* co-exist in the system then the installation will fail. So achieving robust automatic installation procedures to make it easy the use of the EDDL for developers and expert users implied an important effort in person-hours in T2.1, T2.2 and T2.6.
2. Another problem related to the use of ONNX has been that each Deep Learning toolkit serialises some layers and operators in different ways; such toolkits do not strictly follow the standard ONNX. This implies that the EDDL is not able to import neural networks created or trained using other toolkits.

   At M16 we decided to strictly follow the ONNX standard file format defined at `https://github.com/onnx/onnx`. This decision was motivated to guarantee full compatibility with ONNX runtime while considering the project resources available. The reference we are following since M16 is to guarantee full compatibility with ONNX runtime `https://microsoft.github.io/onnxruntime/`.

[22] `https://cmake.org/`

In order to rapidly provide the EDDL library with support of inference code generation for FPGA, CEA has developed in n2D2 the ONNX import capability, which insure the interworking between the two frameworks. This will allow designing and training a Neural Network using the EDDL library and to generate the inference code for FPGA thanks to N2D2. The corresponding designed flow is detailed in D2.3

4 Conclusions

The development of the EDDL and the PyEDDL is in progress and according to the work plan. As it can be seen in this document (the deliverable D2.1 is the software of EDDL+PyEDDL developed so far in T2.1, T2.2 and T2.6) and others deliverables related to this one, there is a lot of work carried out with a great effort done by the development teams of the involved partners.

Thanks to the implementation of the import/export feature following the ONNX format for neural networks, the EDDL+PyEDDL is compatible with other toolkits.

In addition, the serialisation of any network model thanks to ONNX is also used for distributed training. Having the option to select weights or gradients for the serialisation, gradients can be reported from worker nodes to the master noded; the standard and default option of doing the serialisation with weights is used to update weights from the master node to the worker nodes. Specific internal functions in the EDDL have been implemented to apply accumulated gradients reported from any worker node into the network model maintained in the RAM memory the master node.

Additionally, it is important to point out how much fruitful it is being the collaboration between partners involved in T2.1, T2.2 and T2.6 with other partners involved in other tasks of WP2 and other tasks of other work packages. This is reflected in this deliverable other deliverables due to M17 and the ones due to M15.