DEEPHEALTH

# D3.5 ECVL Toolkit front-end

| | |
|---|---|
| **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:** | WP3/T3.4 |
| **Dissemination level:** | PUBLIC |
| **Document due Date:** | 31/03/2020 (M15) |
| **Actual date of delivery** | 02/04/2020 (M16) |
| **Leader of this deliverable** | UNIMORE |
| **Author (s)** | Michele Cancilla, Federico Bolelli, Costantino Grana |
| **Version** | V3.1 |

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 |
|---|---|---|
| **1** | 19/03/2020 | First proposal before review. |
| **2** | 31/03/2020 | Second version, after internal review. |
| **2.1** | 31/03/2020 | Minor changes and introduction of explanatory images. |
| **3** | 01/04/2020 | Final version, after second round of internal review. |
| **3.1** | 02/04/2020 | Ready-to-submit version. |



# Table of contents

# 1 Executive summary

The DeepHealth toolkit includes both libraries, ECVL and EDDLL, plus the front-end exposed to expert users for an efficient usage of all the functionalities of these libraries. This toolkit will be free and open-source software.

The ECVL library is devoted to image processing functionalities useful for pre- and post- processing, data loading, augmentation and image format management, while the EDDL library provides the tools for defining Deep Learning pipelines and executing them efficiently on hybrid hardware resources.

A web-based couple front-end and back-end completes the DeepHealth infrastructure and makes the toolkit a stand-alone solution.

It is extremely important to underline that ECVL and EDDL libraries strictly cooperate within the DeepHealth toolkit, every time that a training or inference process takes place in EDDL a previous images manipulation (*i.e.,* dataset loading and splitting, image augmentation, etc.) occurred. For this reason, the development solution adopted unifies ECVL and EDDL front-ends. The same applies for the back-ends.

Therefore, besides the initial title of the deliverable, this document aims at reporting the rationale behind the DeepHealth toolkit back-end functionalities for both ECVL and EDDL, providing a general description of this part of the toolkit, its interaction with the front-end, and a detailed documentation manual. On the other hand, the front-end counterpart, again for both ECVL and EDDL is reported in Deliverable 2.5.

# 2 Introduction

This deliverable is the report concerning the development of one of the key components of the DeepHealth toolkit (DHt).

The DHt is composed by three main components, the two libraries ECVL and EDDLL plus the so called front-end (see Figure 1). However, and in order to avoid confusion, the front-end of the DHt is a software module that is divided into two parts, one visible to the user (a.k.a. graphical user interface–GUI) through the navigator (Firefox, Chrome, Edge, …) and one invisible part that performs all the actions indicated by the user using the functionalities provided by both libraries. The GUI is also known as front-end. The invisible part is commonly known as back-end. In order to clarify, when we use the concept of front-end in this document (D3.5) and in the deliverable D2.5 we refer to the GUI. When we use the concept of front-end as a component of the DHt, we refer to the module that includes both the GUI and the back-end.

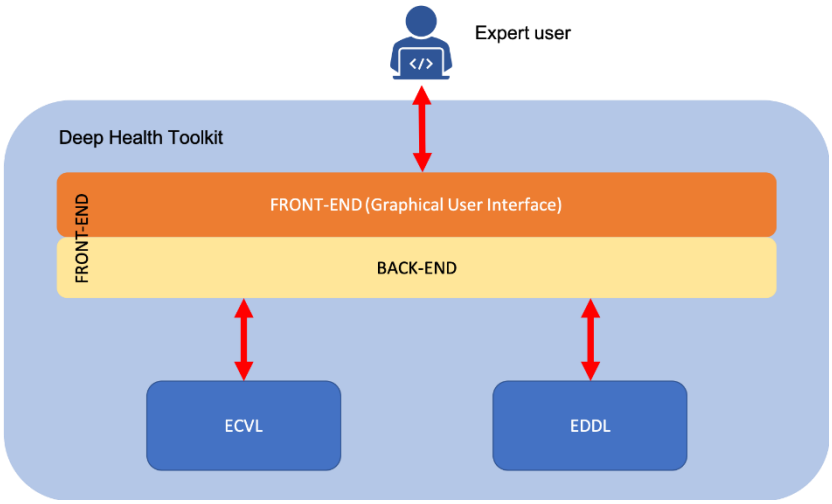


*Figure 1. Schema of the DeepHealth toolkit.*

After the previous clarification with the help of Figure 1 it is easier to understand what this deliverable describes in detail, the development and functionalities provided by the software part that runs on a web server (back-end block in Figure 1) and executes different processes by using the libraries ECVL and EDDLL. As a complement to this deliverable, D2.5 provides the descriptions of the brown block titled as “GUI (front-end–Graphical User Interface)” in Figure 1.

The design of DeepHealth toolkit follows several core principles to support a set of key requirements:

1. support a wide variety of application types in medical image analysis and computer-assisted diagnosis;
2. be simple to be used in common use cases, but flexible enough for complex use cases;
3. support parallel processing, model distribution and adaptation;
4. support best practices as data augmentation and correct data set partitioning;
5. user-friendly visualization.

The web-based graphical user interface (GUI) is the tip of the iceberg that allows the entire toolkit to follow the aforementioned core principles.

The front-end will then allow user to access, in a graphic way, to all functionalities provided by the libraries (ECVL and EDDLL) and will provide the graphical tools to exploit these functionalities in a user-friendly way. To achieve this goal, the GUI needs to be coupled with a back-end that will manage low level libraries interaction, driving the dataset loading, image manipulation and the training/inference processes.

This architecture provides several benefits:

1. expert/end users do not have to manage individual software, installing and updating them at every library update;
2. similarly, the web browser interface will delegate the pairing with the hardware infrastructure to the software maintainer, meaning that any customer can easily maximize the use of the toolkit on any combination of hardware and operating system;
3. expert users will no longer have to worry about compatibility between the graphical user interface (GUI) and the server. The graphical user interface will be intuitive, attractive, and easy to use and to learn.

The back-end includes the datasets management tool, and allows handling data/image manipulation pipelines. The former allows loading dataset images to be manipulated and that will feed the neural network model for training or testing. The format used to define and manage a dataset, including its split in training, validation and test sets is the *DeepHealth Toolkit Dataset Format,* introduced in the following of this document.

Another important result related to the front-end/back-end infrastructure is that the latter will be a key element in the cloud-based version of the toolkit. In fact, the back-end will serve as an entry point for the requests and then distribute the workloads to computational facilities through the network.

It is important to stress that ECVL and EDDLL libraries strictly cooperate within the DeepHealth toolkit.

Considering a typical scenario in which expert users interact with the toolkit, they will want to provide their data (for example an image or a dataset) and then choose a neural network model that performs the desired activity.

In this situation ECVL read, augment and expose the images to EDDLL, which is in charge of setting up the neural network model for training or inference.

Therefore, despite the initial proposal, in the adopted architectural software design, instead of implementing an ECVL specific front-end (and a specific one for EDDLL), we tightly coupled the two libraries in a single front-end.

Besides the initial title of the deliverable, this document aims at reporting the rationale behind the DeepHealth toolkit back-end functionalities for both ECVL and EDDLL, providing a general description

of this part of the toolkit, its interaction with the front-end, and a detailed documentation manual. On the other, the front-end counterpart, again for both ECVL and EDDLL is reported in Deliverable 2.5.

In the following sections, we will describe the used technologies, the envisioned interaction between front-end and back-end, details on the dataset format, and document the current status of the API. Finally, some examples of interaction with the corresponding REST API calls are shown.

# 3 Toolkit Technologies

For the development of such a project, many programming tools have been used.

The core application is based on Django, a popular Python-based free and open-source web framework. Django follows the model-template-view (MTV) architectural pattern. The Django Software Foundation (DSF), an independent non-profit organization, maintains it.

Django's primary goal is to ease the creation of complex, database-driven websites. The framework emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of don't repeat yourself. Python is used throughout, even for settings files and data models. Django also provides an optional administrative “create, read, update and delete” interface, which is generated dynamically through introspection and configured via admin models.

In order to provide the REST features, we used Django REST framework, which provides several advantages for building Web APIs. It eases development thanks to the web browsable API, it provides authentication policies including packages for OAuth1a and OAuth2, the serialization supports both ORM and non-ORM data sources. Moreover, it is customizable, it has extensive documentation, and great community support, and it is used and trusted by internationally recognized companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Data is stored and managed in a MySQL relational database. This is one of the most used and supported relational database management system, it is open-source, well maintained and its performance are definitely enterprise level.

The training and inference tasks are distributed using Celery, an open source asynchronous task queue or job queue, which is based on distributed message passing. It is focused on real-time operation, but supports scheduling as well. The execution units are executed concurrently on a single or more worker servers.

The current implementation of the back-end, retrievable at https://github.com/deephealthproject/backend, is still single server, but the cloud development is under way.

# 4 System Interaction and Use Case Scenarios

This section describes the reference scenarios envisioning the interaction between the user, which may be a human through a frontend interface or a platform service, and the back-end application.

The current status of the system is based on the idea, that model creation is demanded to programmers with administrative access to the system. Future versions, when ONNX model loading will be stable in the Python EDDL interface, will allow to load models through the APIs.

The usage scenario is twofold:

1. *Train*: the user wants to see how well a model can perform on his data, by training it from scratch, or from an existing set of weights –a pretrained model– (e.g. learn to classify melanomas with resnet50 pre trained on ImageNet). This will create a new set of weights.
2. *Inference*: the user wants to see how well a model can perform on his data, by only running the data through a model with an existing set of weights. This does not create new weights for the selected model.

In the *Train* scenario we aim at calling `/train`. This requires

1. a model: the description of the neural network architecture. This is provided as an id, obtained with `/models`.

   Optionally, the training can be provided with a set of weights. These can be millions of values, so they are identified with a *weights_id*, obtained with `/weights`. This allows for both finetuning a pretrained model and start again a training process from where we left (e.g. because of a network failure).

2. a dataset: the list of the input images and their labels. This is complex and can require many information, so it is specified with the "DeepHealth Toolkit Dataset Format" (detailed in Section 5). A list of the known datasets can be obtained with `/datasets`, or a new one can be created from a *.yml* file with a POST to `/dataset`.
3. a project: while not strictly necessary in principle this is required to keep track of the application functioning. The next time a user will start working on the same project, he will be able to see the latest operation performed, whether it's a training session or an inference. Again, *projects_id*s can be queried or created with `/projects`.
4. some properties: many options can be specified for the training process, such as the learning rate, a loss function, the input sizes, and many others. These can be retrieved for the whole system with `/properties`, while some model specific ones can be obtained with `/allowedProperties`.

In the *Inference* scenario, we instead call `/inference`. In this case the *weights_id* is mandatory (which also includes the model it refers to), along with the dataset on which to run the model and the project this activity belongs to.

In both cases the process can take a long time, so the process is identified by a UUID, and its status can be inspected with `/status`. After a process finishes, the `/output` API allows to query the results obtained by the trained model or by the inference.

# 5 DeepHealth Toolkit Dataset Format

The *DeepHealth Toolkit Dataset Format* is a simple and flexible YAML[1] syntax to describe a dataset. This definition is implemented by ECVL data loading modules, since all data format manipulation routines are specific function calls to ECVL libraries. These, are exploited by the back-end toolkit to handle dataset loading and splitting.

The format includes the definition of:

- (optional) a name for the dataset;
- (optional) a textual description;
- (optional) an array with the names of the classes available in the dataset;
- (optional) an array with the names of the features available in the dataset;
- an array with the list of images;
- (optional) a dictionary that specifies the splits of the dataset and the images assigned to them.

The first entries of the YAML file, define the basic information of a dataset such as name, description, classes and features. The `classes` entry represents all the categories predictable, while `features` describes the additional information related to each image.

[1] https://yaml.org

## 5.1 Header

```
# (optional) Descriptive string used just for pretty reporting
name: dataset_name

# (optional) Descriptive string to document the file
description: >
      This is an example of long text which describes the use of this dataset and whatever
      I want to annotate.

      You can also write multiple paragraphs with the only care of indenting them
      correctly.

# (optional) Array of class names
classes: [class_a, class_b, class_c]

# (optional) Array of features names
features: [feature_1, feature_2, feature_3, feature_4]
```

## 5.2 Images

The `images` array lists all the available images. Each image is characterized by the following parameters:

- `location`, it can be either absolute or relative path to file;
- (optional) `label`, contains one or more classes of the `classes` field in the form of:
  - a class name (e.g. in case of single class tasks)
  - a class index (e.g. in case of single class tasks)
  - an array of class names (e.g. multi-classes tasks)
  - an array of class indexes (e.g. multi-classes tasks)

  It can also be an URL to an image (e.g. in case of a segmentation task)
- (optional) `values`, are the values that a fixed feature (listed in features) can assume and it can be:
  - an array of values (with `null` when the i-th feature is not available)
  - a dictionary with the name of the feature coupled with its value

```
# Array of images. Images are listed as a couple of location (absolute or relative to this
file location) and an optional label. The location must be unique in the array
images:
# label can be a class name (string)...
# values can be an array with a positional correspondence with the features array...
      - location: image_path_and_name_1
        label: class_b
        values: [value_1, null, value_3, null]

# ... or the class index (integer) w.r.t. the classes array
# ... or a dictionary with the name of the feature coupled with its value
      - location: image_path_and_name_2
        label: 2
        values: { feature_1: value_1, feature_3: value_3 }

# In the case of multi class problems, label can be an array of class names (array of
strings) ...
      - location: image_path_and_name_3
        label: [class_a, class_c]
# ... or an array of class indexes (array of integers)
```

```
      - location: image_path_and_name_4
        label: [0, 2]

# label can be a path (string) to an image in case of a segmentation task
      - location: image_path_and_name_5
        label: path_to_ground_truth_image

# Remember that labels are optional
      - location: image_path_and_name_6
      - location: image_path_and_name_7

# When only the location is used, it can be omitted
      - image_path_and_name_8
      - image_path_and_name_9
```

## 5.3 Split

In the split section you can specify how to divide the data. This dictionary lists the indexes of the images to be used in three different phases (training, validation and test).

```
# (optional) Split is a dictionary with three arrays called training, validation, and
test. They list the indexes of the images to be used in the three different phases.
split:
      training: [0, 1, 2, 3, 4]
      validation: [5, 6]
      test: [7, 8]
```

## 5.4 A Complete Example

```
# (optional) Descriptive string used just for pretty reporting
name: dataset_name

# (optional) Descriptive string to document the file
description: >
      This is an example of long text which describes the use of this dataset and whatever
      I want to annotate.

      You can also write multiple paragraphs with the only care of indenting them
      correctly.

# (optional) Array of class names
classes: [class_a, class_b, class_c]

# (optional) Array of features names
features: [feature_1, feature_2, feature_3, feature_4]
# Array of images. Images are listed as a couple of location (absolute or relative to this
file location) and an optional label. Location must be unique in the array

images:
# label can be a class name (string)...
# values can be an array with a positional correspondence with the features array...
      - location: image_path_and_name_1
        label: class_b
        values: [value_1, null, value_3, null]

# ... or the class index (integer) w.r.t. the classes array
# ... or a dictionary with the name of the feature coupled with its value
```

```
      - location: image_path_and_name_2
        label: 2
        values: { feature_1: value_1, feature_3: value_3 }

# In the case of multi class problems, label can be an array of class names (array of
strings) ...
      - location: image_path_and_name_3
        label: [class_a, class_c]

# ... or an array of class indexes (array of integers)
      - location: image_path_and_name_4
        label: [0, 2]

# label can be a path (string) to an image in case of a segmentation task
      - location: image_path_and_name_5
        label: path_to_ground_truth_image

# Remember that labels are optional
      - location: image_path_and_name_6
      - location: image_path_and_name_7

# When only the location is used, it can be omitted
      - image_path_and_name_8
      - image_path_and_name_9

# (optional) Split is a dictionary with three arrays called training, validation, and
test. They list the indexes of the images to be used in the three different phases.
split:
      training: [0, 1, 2, 3, 4]
      validation: [5, 6]
      test: [7, 8]
```

# 6 Data Schema

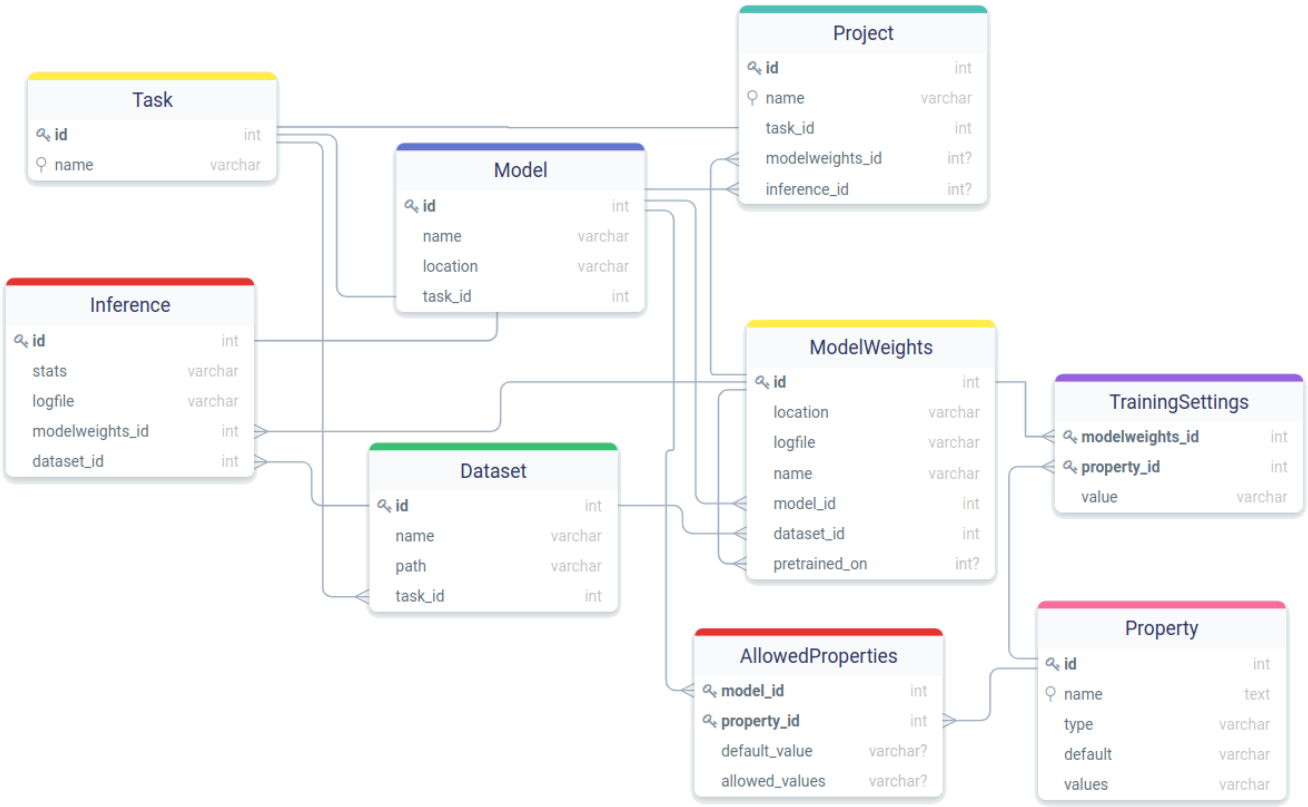


*Figure 2. DeepHealth Toolkit ER schema.*

The DeepHealth Toolkit relies on a relational database managed by MySQL, a popular open-source relational database management system. The data modelling is shown as in Figure 2 and it is browsable at https://drawsql.app/aimagelab/diagrams/api.

The entities that constitute the database are the following:

- **Project**: Memorizes the projects created by the users, storing the latest activities and modifications done within a project.
- **Task**: This entity stores the possible deep learning tasks supported by the Toolkit. The current implementation supports classification and segmentation tasks.
- **Model**: Stores the Neural Network models for addressing different tasks.
- **Dataset**: Collects all the available datasets. The path attribute points to the a DeepHealth Toolkit Dataset Format YAML file, which lists the dataset details.
- **ModelWeights**: Stores the information about a training of a neural network, namely a dataset, a model and an optional weight to load for fine-tuning tasks.
- **Property**: Collects the possible options and hyper-parameters for a Neural Network training process.
- **TrainingSettings**: Stores the properties values for a fixed training process.
- **AllowedProperties**: This table lets to redefine the default values of a Property depending a Model. For example, AllowedProperties can store a different default *Loss* Property of a model designed for classification task, setting its “default_value” field.
- **Inference**: Collects information about an inference process. It requires a Neural Network already trained.

# 7 DeepHealth Toolkit APIs Description

The DeepHealth back-end has been developed as a RESTful web service, where REST is acronym for **RE**presentational **S**tate **T**ransfer. The toolkit serves different APIs which allow a user to feed an already trained Neural Network (NN with medical images or datasets. A user can also train a NN from scratch if none of the already trained NNs fits his needs. The online interactive documentation of the APIs is available at https://app.swaggerhub.com/apis/pritt/DeepHealthToolkitAPI/1.0.2.

## 7.1 /allowedProperties

### 7.1.1 GET

This method returns the values that a property can assume depending on the model employed. It provides a default value and a comma separated list of values to choose from.

When this API returns an empty list, the property allowed values and default should be retrieved using the `/properties/{id}` API.

*Parameters:*

`model_id`: Required integer representing the model.

`property_id`: Required integer representing a property.

*Docs:*

```
GET /backend/allowedProperties?model_id=integer&property_id=integer

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "allowed_value": string,
        "default_value": string,
        "property_id": integer,
        "model_id": integer
    }
]
```

*Example:*

```
GET /backend/allowedProperties/?model_id=1&property_id=1

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "allowed_value": null,
        "default_value": "0.0001",
        "property_id": 1,
        "model_id": 1
    }
]
```

## 7.2 /datasets

### 7.2.1 GET

This method returns the datasets list that should be loaded in the power user component, on the left side of the page, in the same panel as the models list.

*Parameters*

`task_id` *(optional)*: Integer representing a task used to retrieve dataset of a specific task.

*Docs:*

```
GET /backend/datasets?task_id=integer

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "path": string,
        "task_id": integer
    }
]
```

*Example:*

```
GET /backend/datasets

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "MNIST",
        "path": "/mnt/data/backend/data/datasets/mnist.yml",
        "task_id": 1
    },
    {
        "id": 52,
        "name": "isic_segmentation",
        "path": "/mnt/data/backend/data/datasets/isic_segmentation.yml",
        "task_id": 2
    }
]
```

### 7.2.2 POST

This API uploads a dataset description in YAML format and stores it in the back-end. The URL must contain a valid reference to a dataset, e.g.:

https://www.dropbox.com/s/ul1yc8owj0hxpu6/isic_segmentation.yml?dl=1.

```
POST /backend/datasets
    {
        "id": integer,
        "name": string,
        "path": string,
    }


Allow: POST, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "path": string,
        "task_id": integer
    }
]
```

```
POST /backend/datasets
{
    "name": "isic_segmentation",
    "task_id": 2,
    "path": "https://www.dropbox.com/s/ul1yc8owj0hxpu6/isic_segmentation.yml?dl=1"
}

HTTP 200 OK
Allow: POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
    "id": 1,
    "name": "MNIST",
    "path": "/mnt/data/backend/data/datasets/mnist.yml",
    "task_id": 1
}
```

## 7.3 /models

### 7.3.1 GET

This API allows the client to know which Neural Network models are available in the system in order to allow their selection.

The optional `task_id` parameter is used to filter them based on the task the models are used for.

*Parameters:*

`task_id` *(optional)*: integer Optional integer for filtering the models based on task.

*Docs:*

```
GET /backend/models?task_id=integer

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string
    }
]
```

*Example:*

```
GET /backend/models

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "LeNet"
    },
    {
        "id": 2,
        "name": "vgg"
    },
    {
        "id": 3,
        "name": "DeepLabV3+"
    },
    {
        "id": 4,
        "name": "SegNet"
    }
]
```

## 7.4 /projects

API which provides *GET*, *PUT*, and *POST* methods:

- *GET* retrieves all the projects or one using `projects/{id}` link.
- *POST* lets to create a new project.
- *PUT* updates a project.

### 7.4.1 GET

*Docs:*

```
GET /backend/projects

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "task_id": integer,
        "modelweights_id": integer,
        "inference_id": integer

    }
]
```

*Example*:

```
GET /backend/projects/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "firstProjectName",
        "task_id": 1,
        "modelweights_id": 193,
        "inference_id": 179
    },
    {
        "id": 2,
        "name": "secondProjectName",
        "task_id": 2,
        "modelweights_id": 149,
        "inference_id": 138
    }
]
```

### 7.4.2 GET/{id}

*Docs:*

```
GET /backend/projects/{id}

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "task_id": integer,
        "modelweights_id": integer,
        "inference_id": integer
    }
]
```

*Example*:

```
GET /backend/projects/1

HTTP 200 OK
Allow: GET, POST, PUT, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept


{
    "id": 1,
    "name": "firstProjectName",
    "task_id": 1,
    "modelweights_id": 193,
    "inference_id": 179
}
```

### 7.4.3 POST

*Docs:*

```
POST /backend/projects
{
    "name": string,
    "task_id": integer,
    "modelweights_id": integer,
    "inference_id": integer
}


Allow: POST, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "task_id": integer,
        "modelweights_id": integer,
        "inference_id": integer
    }
]
```

*Example*:

```
POST /backend/projects
{
    "name": "newProject",
    "task_id": 1,
    "modelweights_id": 113,
    "inference_id": 5
}
HTTP 200 OK
Allow: POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
    {
        "id": 1,
        "name": "firstProjectName",
        "task_id": 1,
        "modelweights_id": 193,
        "inference_id": 179
    },
    {
        "id": 2,
        "name": "secondProjectName",
        "task_id": 2,
        "modelweights_id": 149,
        "inference_id": 138
    },
    {
        "id": 3,
        "name": "newProject",
        "task_id": 1,
        "modelweights_id": 113,
        "inference_id": 5
    }
]
```

### 7.4.4 PUT

*Docs:*

```
PUT /backend/projects
{
    "id": integer,
    "name": string,
    "task_id": integer,
    "modelweights_id": integer,
    "inference_id": integer
}


Allow: PUT, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "task_id": integer,
        "modelweights_id": integer,
        "inference_id": integer
    }
]
```

*Example*:

```
PUT /backend/projects
{
    "id": 3,
    "name": "newProjectName",
    "task_id": 1,
    "modelweights_id": 113,
    "inference_id": 5

}
HTTP 200 OK
Allow: PUT, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "firstProjectName",
        "task_id": 1,
        "modelweights_id": 193,
        "inference_id": 179
    },
    {
        "id": 2,
        "name": "secondProjectName",
        "task_id": 2,
        "modelweights_id": 149,
        "inference_id": 138
    },
    {
        "id": 3,
        "name": "newProjectName",
        "task_id": 1,
        "modelweights_id": 113,
        "inference_id": 5
    }
]
```

## 7.5 /properties

This API allows the client to know which properties are "globally" supported by the back-end.

A model can have different default and allowed values if the `/allowedProperties` return an entry.

*Docs:*

```
GET /backend/properties/

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "type": string,
        "default": string,
        "values": string
    }
]
```

*Example:*

```
GET /backend/properties/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "Learning rate",
        "type": "float",
        "default": "0.0001",
        "values": null
    },
    {
        "id": 2,
        "name": "Loss function",
        "type": "list",
        "default": "CrossEntropy",
        "values": "CrossEntropy,SoftCrossEntropy,MSE"
    },
    {
        "id": 3,
        "name": "Epochs",
        "type": "integer",
        "default": "50",
        "values": null
    },
    {
        "id": 4,
        "name": "Batch size",
        "type": "integer",
        "default": "1",
        "values": null
    }
]
```

## 7.6 /tasks

This API allows the client to know which tasks this platform supports. e.g. classification or segmentation tasks.

*Docs:*

```
GET /backend/tasks/

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string
    }
]
```

*Example:*

```
GET /backend/tasks/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "Classification"
    },
    {
        "id": 2,
        "name": "Segmentation"
    }
]
```

## 7.7 /trainingSettings

This API returns the value used for a property in a specific training (an instance of ModelWeights). It requires a `modelweights_id`, indicating a training process, and a `property_id`.

*Parameters*

`modelweights_id`: integer Required integer representing the ModelWeights.

`property_id`: integer Required integer representing a property.

*Docs:*

```
GET /backend/trainingSettings?modelweights_id=integer&property_id=integer

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "value": string,
        "modelweights_id": integer,
        "property_id": integer
    }
]
```

*Example:*

```
GET /backend/trainingSettings?modelweights_id=1&property_id=1

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "value": "0.005",
        "modelweights_id": 1,
        "property_id": 1
    }
]
```

## 7.8 /weights

When 'use pre-trained' is selected, it is possible to query the back-end passing a `model_id` to obtain a list of datasets on which it was pretrained.

*Parameters*

`model_id`: integer Required integer representing the model.

*Docs:*

```
GET /backend/weights?model_id=integer

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

[
    {
        "id": integer,
        "name": string,
        "logfile": string,
        "model_id": integer,
        "task_id": integer,
        "dataset_id": integer,
        "pretrained_on": integer
    }
]
```

*Example:*

```
GET /backend/weights?model_id=1

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "weight1",
        "logfile": null,
        "model_id": 1,
        "task_id": 1,
        "dataset_id": 1,
        "pretrained_on": null
    },
    {
        "id": 4,
        "name": "LeNet_MNIST_2020-02-20_17:33:49",
        "logfile": null,
        "model_id": 1,
        "task_id": 1,
        "dataset_id": 1,
        "pretrained_on": null
    }
]
```

## 7.9 /inference

This is the main entry point to start the inference. It is mandatory to specify a pre-trained model and a dataset for finetuning.

*Docs:*

```
POST /backend/inference
{
    "modelweights_id": integer,
    "dataset_id": integer,
    "project_id": integer
}

Allow: POST, OPTIONS
Content-Type: application/json

{
    "result": string,
    "process_id": string
}
```

*Example:*

```
POST /backend/inference
{
    "modelweights_id": 143,
    "dataset_id": 1,
    "project_id": 1
}

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "result": "ok",
    "process_id": "aa6cd767-a88f-4b65-8577-3714b6bcba1e"
}
```

## 7.10 /inferenceSingle

This API allows the inference of a single image. It is mandatory to specify the same fields of `/inference` API, but for dataset_id which is replace by the URL of the image to process.

*Docs:*

```
POST /backend/inferenceSingle
{
    "modelweights_id": integer,
    "image_url": string,
    "project_id": integer
}

Allow: POST, OPTIONS
Content-Type: application/json

{
    "result": string,
    "process_id": string
}
```

*Example:*

```
POST /backend/inferenceSingle
{
    "modelweights_id": 112,
    "image_url": "https://url/my-image.png",
    "project_id": 1
}

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "result": "ok",
    "process_id": "aa6cd767-a88f-4b65-8577-3714b6bcba1e"
}
```

## 7.11 /status

This API allows the frontend to query the status of a training or inference, identified by a `process_id` (which is returned by `/train` or `/inference` APIs).

*Docs:*

```
GET /backend/status?process_id=string

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

{
    "result": string[],
    "status": {
        "process_type": string[“training”, “inference”],
        "process_status": string[“running”, “finished”, “error”],
        "process_data": string
    }
}
```

*Example:*

```
GET /backend/status?process_id=63ad58f4-cfaa-4edd-8600-e6c3f99bd87e

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "result": "ok",
    "status": {
        "process_type": "training",
        "process_status": "finished",
        "process_data": "Evaluation 156/156 softmax4(cross_entropy=0.633,categorical_accuracy=0.89
5)"
    }
}
```

## 7.12 /stopProcess

Stop a training or inference process specifying a `process_id` (which is returned by `/train` or `/inference` APIs).

*Docs:*

```
POST /backend/stopProcess
{
    "process_id": string
}

Allow: POST, OPTIONS
Content-Type: application/json

{
    "result": string
}
```

*Example:*

```
POST /backend/stopProcess
{
    "process_id": "62a10034-1f2f-44b9-9014-71a1992647be"
}

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "result": "Training stopped"
}
```

## 7.13 /train

Starts the training of a (possibly pre-trained) model on a dataset. This is the main entry point to start the training of a model on a dataset. It is mandatory to specify the project, a model to be trained and a training dataset. When providing a `weights_id`, the training starts from the pre-trained model.

Moreover, using a simple text-based scripting language, the expert user can specify how images should be transformed during data loading, with the Augmentations property. This choice, contrary to just allowing him to run a Python script, allows us to integrate the data transformation operations into the batch loading, without the need of checking which operation, libraries and function calls are made by user scripts. This avoids any security concerns and any library incompatibilities.

*Docs:*

```
POST /backend/train
{
    "model_id": integer,
    "weights_id": integer,
    "properties": [
        {
            "name": string,
            "value": string
        }
    ],
    "dataset_id": integer,
```

```
    "project_id": integer
}

Allow: POST, OPTIONS
Content-Type: application/json

{
    "result": string,
    "process_id": string
}
```

*Example:*

```
POST /backend/train
{
    "model_id": 1,
    "weights_id": null,
    "properties": [
        {
            "name": "Learning rate",
            "value": "1e-3"
        },
        {
            "name": "Loss function",
            "value": "CrossEntropy"
        },
        {
            "name": "Metric",
            "value": "CategoricalAccuracy"
        },
        {
            "name": "Augmentations",
            "value": "AugRotate angle=[-5,5] center=(0,0) scale=0.5 interp=\"linear\"\n"
        },
        {
            "name": "Epochs",
            "value": "1"
        },
        {
            "name": "Input height",
            "value": "30"
        },
        {
            "name": "Input width",
            "value": "30"
        }
    ],
    "dataset_id": 1,
    "project_id": 1
}

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "result": "ok",
    "process_id": "ec63b923-850a-4d1d-aa24-e9d54a942c1b"
}
```

## 7.14 /output

This API provides information about an `inference` process. For classification task it returns the list of images and an array composed of the class prediction scores. In segmentation task it returns the URLs of the segmented images.

*Parameters:*

`process_id`: Required integer indicating a process id.

*Docs:*

```
GET /backend/output?process_id=string

Allow: GET, HEAD, OPTIONS
Content-Type: application/json

{
    "outputs": [
        [string, string],
}
```

*Example:*

```
GET /backend/output?process_id=04510596-4b21-464a-9696-4b7b9f65b806


HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "outputs": [
        [
            "/mnt/data/DATA/mnist/testing/0.png",
            "[[1.7636454003877589e-06, 9.877122408852301e-08, 7.840961188776419e-06, 0.00021923755
411989987, 1.5176419765339233e-07, 3.19655964631238e-06, 2.1568662411652895e-09, 0.999383449554443
4, 6.77434513818298e-07, 0.00038347215740941465]]"
        ],
        [
            "/mnt/data/DATA/mnist/testing/1.png",
            "[[0.0016769643407315016, 0.00033066936885006726, 0.9884131550788879, 0.00473130447790
0267, 1.3814938881751004e-07, 0.0016333448002114892, 0.0022395525593310595, 6.316236067505088e-08,
0.0009746149880811572, 1.0740181721757835e-07]]"
        ],
         …
}
```

# 8 Interaction Examples

## 8.1 Training

In this section, we provide some examples of the information needed to start a training process.

The first step is to provide an image augmentation property string, which allows flexible local image enhancements and variations. The simple augmentation format is line based, with the possibility to include initial whitespace to allow easier reading.

“Container” type augmentations allow to group together other augmentations.

Every augmentation in ECVL can be included here, and the parameters are expressed as `parameter_name=parameter_value`. Optional parameters can be omitted altogether. Parameter values can be:

- *number*: decimal value
- *string*: text (usually only specific values are acceptable, depending on the augmentation function)
- *list of values*: parenthesis enclosed sequence, to allow specifying vector quantities
- *random range*: square brackets enclosed min, max couple. This is the main type, which instructs the system to pick a uniform random number to perform the specific augmentation

Here is a complete example:

```
SequentialAugmentationContainer
    AugRotate angle=[-5,5] center=(0,0) interp="linear"
    AugCoarseDropout p=[0,0.55] drop_size=[0.02,0.1] per_channel=0
    OneOfAugmentationContainer
        AugAdditiveLaplaceNoise std_dev=[0,51]
        AugAdditivePoissonNoise lambda=[0,40]
    end
end
```

In the example, also graphically drawn in Figure 3, we specify that every image must be transformed with a sequence of image transformations (`SequentialAugmentationContainer`). First a random rotation of ±5 degrees centered in 0,0 is performed to improve robustness to small rotations (`AugRotate`); secondly, random rectangles (whose sides will be from 2% to 10% of the image) will be blackened with a random probability up to 55%, in order to improve occlusion robustness (`AugCoarseDropout`); finally, we apply some random noise (`OneOfAugmentationContainer`), by selecting one of Laplacian or Poisson noise (each with some random parameter).

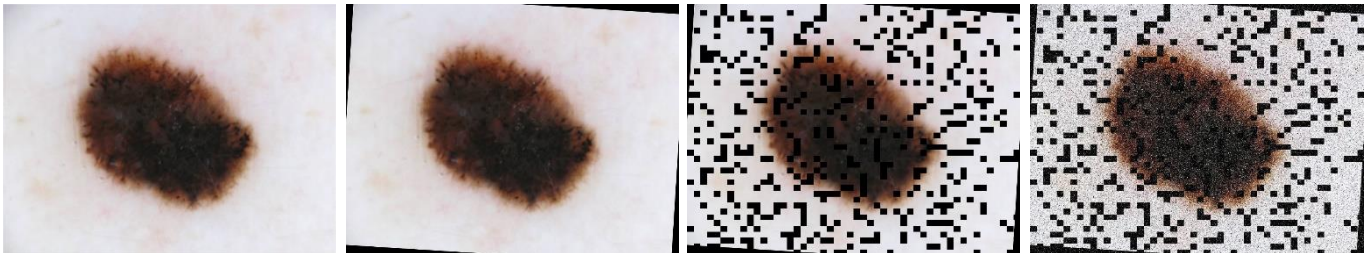

*Figure 3. Example of ECVL augmentations applied to an input image.*

Therefore, considering a scenario in which an end-user wants to train a neural network from scratch, he can start a training process calling the *train* API with a POST request, providing a JSON body like:

```
{
    "model_id": 349,
    "weights_id": null,
    "properties": [
        {
            "name": "Learning rate",
            "value": "1e-3"
        },
        {
            "name": "Loss function",
            "value": "CrossEntropy"
        },
        {
            "name": "Metric",
            "value": "CategoricalAccuracy"
        },
        {
```

```
            "name": "Epochs",
            "value": "30"
        },
        {
            "name": "Augmentations",
            "value": "SequentialAugmentationContainer\n    AugRotate angle=[-5,5] center=(0,0) interp=\"linear\"\n    AugCoarseDropout p=[0,0.55] drop_size=[0.02,0.1] per_channel=0,\n    OneOfAugmentationContainer\n        AugAdditiveLaplaceNoise std_dev=[0,51]\n        AugAdditivePoissonNoise lambda=[0,40]\n    end\nend\n"
        }
    ],
    "dataset_id": 143,
    "project_id": 24
}
```


Where *model_id* and *dataset_id* are the integer identifiers of model and dataset. *project_id* refers to the current project, while *weights_id* is an optional field which indicates whether to load a previous model checkpoint. The *properties* field specifies the hyperparameters to use and their values.

At this point the back-end performs different actions:

1. Check the request correctness and the existence of the specified fields.
2. Start an asynchronous training process.

   These lines of code show where the back-end delegates the execution of a training process to another agent which will perform it asynchronously.

   ```
   # Differentiate the task and start training
   if task_name == 'classification':
       celery_id = classification.training.delay(config)
   elif task_name == 'segmentation':
       celery_id = segmentation.training.delay(config)
   ```

   This execution policy unbinds the back-end from the execution of training or inference processes, allowing other slaves to perform these tasks.
3. Send response to client with a *process_id* of the training.

## 8.2 Inference

A user can employ pre-trained neural networks or train from scratch a new one (as shown in Section 4). Using the `/weights` API the user can retrieve all the available models training weights and choose one of them.

For example, selecting the following *weight*:


```
{
    "id": 189,
    "name": "LeNet_MNIST_2020-03-12_12:48:06",
    "logfile": "/mnt/data/backend/data/training/logs/4de6a2f5c8f34dcc8e6351ca4473c389.log",
    "model_id": 1,
    "task_id": 1,
    "dataset_id": 1,
    "pretrained_on": 113
}
```


At this point he must provide a dataset, selecting from the ones available in the system, with `/datasets` or uploading a new one calling `/datasets` with a POST request. At this point he can start an inference process using `/inference`:

```
POST /backend/inference
{
    "modelweights_id": 189,
    "dataset_id": 3,
    "project_id": 4
}
```

Alternatively, the user can use `/inferenceSingle`, which performs inference of a single image provided by URL.

```
POST /backend/inferenceSingle
{
    "modelweights_id": 112,
    "image_url": "https://url/dermoscopic-image.jpg",
    "project_id": 1
}
```

Finally, the user can retrieve the neural network predictions with `/output` which returns, for classification task, the list of images and an array composed of the class prediction scores, and for segmentation task, it returns the URLs of the segmented images.

Example of `/output` response:

```
{
    "outputs": [
        [
            /mnt/data/DATA/isic_skin_lesion/images/ISIC_0000000.jpg",
            "[[1.16545313275712e-06, 5. 235364753687456e-08, 7.840961188776419e-06, 0.000219237554
11989987, 1.5176419765339233e-07, 3.19655964631238e-06, 2.1568662411652895e-09, 0.9993834495544434
]]"
        ]
    ]
}
```

# 9 Conclusions

As indicated in the introduction, this deliverable is the report concerning the development of one of the key components of the DeepHealth toolkit (DHt). Concretely the back-end, the invisible part that runs all the actions indicated by the user using the functionalities provided by both libraries (ECVL and EDDLL).

This report includes the adopted scheme based on YAML formatted files for describing datasets in order to use them for training DL models and to evaluate them (Section 5 – Deep Health Toolkit Dataset Format). And additionally, also includes the possibility of processing single images, one at a time, by using trained models and provide feedback to medical personnel. Nevertheless, we have to point out that it is not the purpose of the DeepHealth toolkit, the toolkit is oriented to expert users, i.e. computer and/or data scientists working in the health sector. In Section 6 – Data Scheme, the relational database designed to manage all the information needed by the back-end is shortly described. Then, Section 7 – DeepHealth Toolkit APIs Description presents and show how to use all the API for executing the tasks required to train and evaluate DL models. The front-end GUI described in deliverable D2.5 will use this API to instruct the back-end to execute the different tasks. Section 8 – Interaction Examples complements Section 7.

In summary, this report describes all the work done in WP3 to make it possible expert users with no profound knowledge on Deep Learning to train and evaluate different topologies of Deep Neural

Networks on different available datasets. As indicated in the proposal, one of the goals of the DeepHealth project is to increase the productivity of expert users working in the health sector. The DHt contributes to such goal thanks that the DHt will allow many expert users to train DL models that later can be integrated in other software platforms used directly by the end-users, i.e. the doctors, who will upload an image in order to get feedback in the form of one or more values of some KPIs or a segmentation of an image by highlighting different tissues or regions of interest in order to help identify pathologies.

The back-end described in this report will be part of a pipeline that will be created in order to show the use of the DHt as a whole in the M18 review. Obviously, not all the functionalities of the whole DHt will be ready at M18.