Trabajo de Fin de Máster

Métodos de aprendizaje profundo para la segmentación semántica

Deep learning methods for semantic segmentation

Autor : ***Escobedo Gutiérrez, Rubén***

Tutor: Heras Vicente, Jónathan

MÁSTER:

Ciencia de Datos y Aprendizaje Automático

Escuela de Máster y Doctorado

UNIVERSIDAD DE LA RIOJA

AÑO ACADÉMICO: 2021/2022

Abstract

Semantic segmentation is a computer vision field that has been applied for tumour location or object recognition in traffic control systems (in autonomous cars). Nowadays, there are several libraries that allow anyone to build semantic segmentation models; nevertheless, since those libraries have been developed by different people, the way to construct, train and compare those models differs among them. During this project, we have researched how those libraries can be grouped together into a common facade, having no difference in the way of using them. For this purpose, some libraries implemented in a common base library have been selected. Since the previous steps had some drawbacks in its implementation, we decided to increase the working scope and, instead of giving a common interface for those libraries, we decided to automate semantic segmentation steps: from the dataset creation to the models evaluation, including also the data augmentation management and the models construction and training.

Resumen

La segmentación semántica es uno de los campos de la visión computador. Con ella, pueden realizarse algunas tareas, como la localización de tumores o el reconocimiento de objetos en los sistemas de control de tráfico (coches autónomos). Actualmente, existen varias librerías que permiten trabajar con modelos de segmentación semántica; sin embargo, al ser desarrolladas por distintos desarrolladores, las formas de construir, entrenar y comparar estos modelos difieren de una a otra. Durante este proyecto, se ha investigado sobre cómo se podrían agrupar todas estas librerías bajo una fachada común, de forma que no hubiera diferencia en usar una u otra. Para ello, se han seleccionado algunas librerías implementadas desde una base común. Dado que el paso anterior tuvo algunos problemas a la hora de implementarse, decidimos aumentar el alcance del proyecto y, en vez de dar una interfaz común para estas librerías, decidimos automatizar las tareas de segmentación semántica: desde la creación del conjunto de datos hasta la evaluación de los distintos modelos, pasando por la gestión del aumento de datos y la creación y el entrenamiento de los modelos.

Table of contents

1. Introduction

In this section, we provide a brief overview about semantic segmentation, the state-of-the-art methods to tackle this computer vision task, and the current challenges to apply those methods. Subsequently, we explain the aim of this project and its context.

Image segmentation is a computer vision task that is based on splitting a digital image into several segments [1]. Those segments are sets of pixels belonging to the same class, see Figure 1.1.

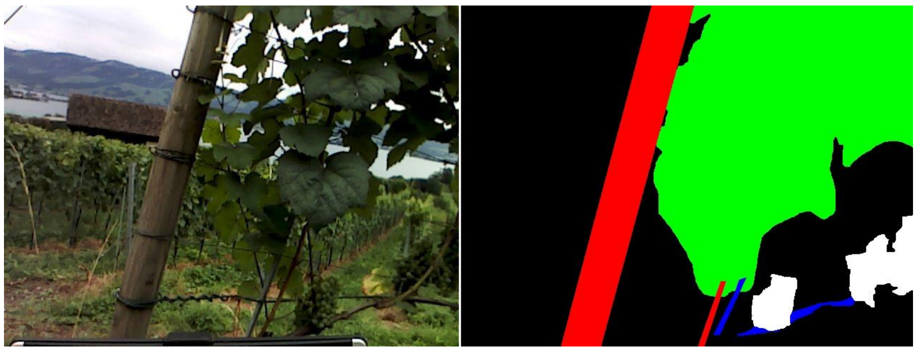


Figure 1.1. An example of image segmentation.

We can distinguish two types of image segmentation problems: semantic segmentation and instance segmentation. In semantic segmentation, the segments of the same class do not differentiate among instances of the same class. On the contrary, if the segmentation is able to distinguish each one of the entities in the segments like different instances, it is denominated instance segmentation. This is illustrated in Figure 1.2. Image segmentation methods can be applied to locate tumours and other pathologies [2], surgery planning in medicine [3], object recognition in traffic control systems [4] or video surveillance in airports [5].

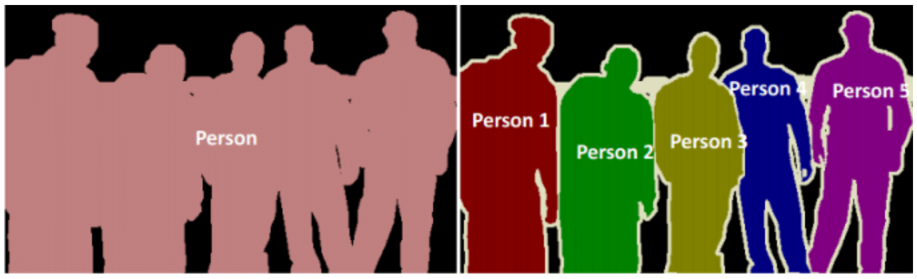


Figure 1.2. Semantic segmentation and Instance segmentation (respectively) example from the article *Improving Learning Effectiveness For Object Detection and Classification in Cluttered Backgrounds* [6].

Nowadays, image segmentation methods are mainly based on deep learning techniques (specifically, convolutional neural networks [6]). There are several image segmentation algorithms, such as *Deeplabv3+ [7], UNet [8]* or *Mask RCNN [9]*. Several libraries allow users to construct image segmentation models using such architectures. Since those libraries have been developed by different developers, their use varies: models are defined differently, input and output formats differ, errors are handled in different ways, and so on. All these issues are drawbacks to easily construct image segmentation models and compare the results obtained with them. In addition, users of those libraries, both experts and novices, have to learn how to use each specific library every time they need it; and reading, studying and understanding exactly how a library works might be a challenging task.

Therefore, in this project we aim to create a library that facilitates the construction and comparison of image segmentation models, deleting all the particular using details and giving an interface that groups all the functionality and the way to work with them under a common usage.

In the next sections, we will demonstrate: how to build an image segmentation model using the library *FastAI [10]*, why the models that can be built using *FastAI* are not enough to cover all the state-of-the-art problems, what can we do to build other models using other libraries, how can we use all of them in a generalistic and easy way, and how can we compare the results of training a neural network constructed with one or another library.

2. A brief introduction to the process of image segmentation

In this section, we will define the concepts presented in the introduction, focusing on what steps are taken in the image segmentation process. Until the end of this section, we need to keep in mind that image segmentation is a computer vision task and, therefore, some steps are shared with other computer vision tasks such as image classification or object detection. Currently, most techniques to solve image segmentation tasks are based on neural networks, and more specifically, on convolutional neural networks [6].

The concepts that we are going to describe belong to the basic image segmentation model construction process. In Figure 2.1 we show a workflow diagram to exemplify the process.

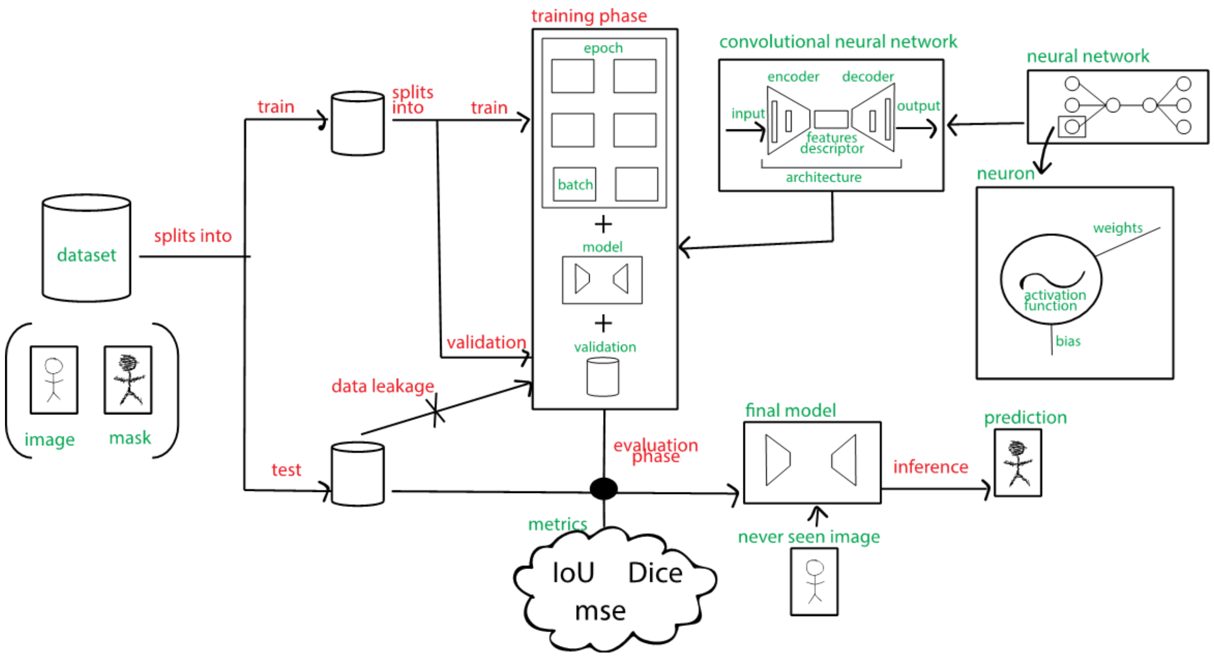


Figure 2.1. Image segmentation workflow diagram.

2.1. Background

The first task that needs to be tackled to create a model, in any supervised machine learning solution, is the definition of the *dataset*. The dataset is the data that will be used to train, validate and test the convolutional neural network. In the semantic segmentation scenario, the dataset is a set of images and their related masks; in other words, a bunch of tuples *(image, mask)*.

The *masks*, which are the labels in image segmentation, are also images. Those images group together pixels with the same semantic information (for example, every pixel that belongs to a person will have the same identifier). Masks are defined with colour codes such grey scale values or RGB tuples, but the common indexing annotation starts at zero and goes through the natural numbers. Because of this fact, it is needed to make a bijection between the codes and the identifiers, using the codes for visualisation and the identifiers for training the models. An example of those tuples *(image, mask)* is shown in Figure 2.2.

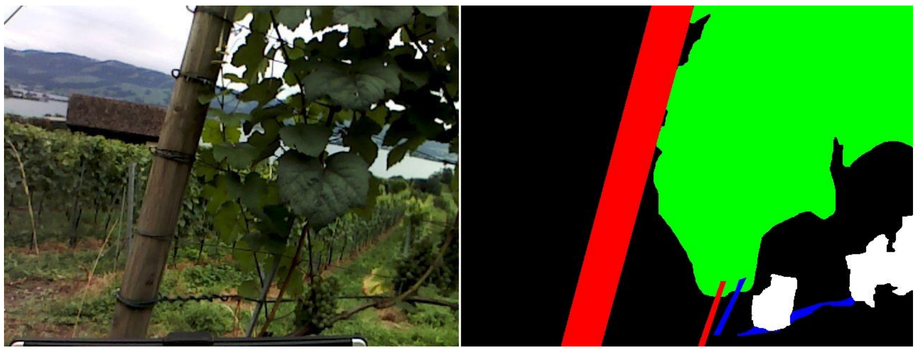


Figure 2.2. An example of *(image-mask)* tuple used for semantic segmentation.

Once the dataset is defined, we have to split it into two subsets: train and test. The *train set* is splitted again into two subsets: train and validation. While the train set is used to learn the parameters in the convolutional neural network (the weights), the validation set is used to tune its hyperparameters.

Unlike general neural networks (generally used to refer to fully connected networks), *convolutional neural networks* use *filters* (or kernels) along the layers (from the input to the output) to, finally, provide a feature matrix. Those neural networks are more resistant to overfitting since they are not fully connected (except in their, commonly, three last layers; called the fully-connected layers).

The *filters* are matrices that are convolved through the input image (or the output that results from processing the input by the previous layers), giving another representation to those images, as we show in Figure 2.3.

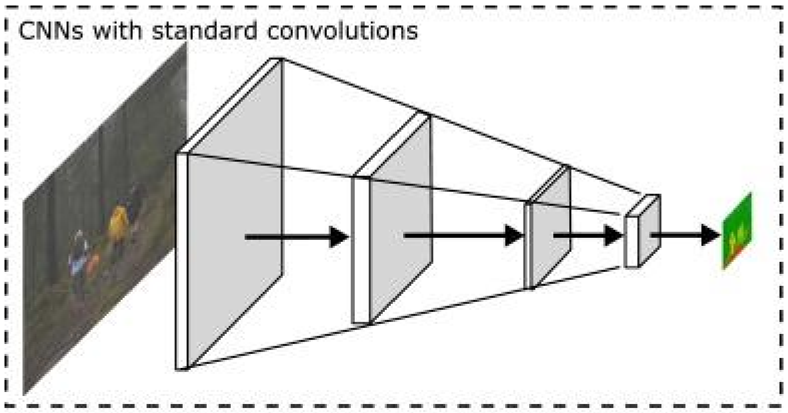


Figure 2.3. An example of a bunch of filters used in convolutional neural networks [11].

The idea behind filters is that the first filters of a convolutional neural network will recognize simple patterns, like contours; meanwhile, latest filters of the network will be capable of recognizing more complex forms, like human eyes. All those filters applied sequentially give to the network the regularisation it needs and another point of view to understand the images. The values of the filters, in the convolutional neural networks, are denominated *weights*, and they are learned during the training process.

In addition to filters, convolutional neural networks also include *activation functions* that are the key point that distinguish a network from a composed linear regression. Those activation functions bring some non-linearity to the convolutional neural network, making it capable of detecting more complex objects in each layer.

If we group filters together into layers, we will define a convolutional neural network. Those neural networks, in the context of semantic segmentation, have the singular structure shown in Figure 2.4, where blue and green layers, defined on the left side, represent the encoder and red, blue and yellow layers, defined on the right side, represent the decoder.

The *encoder* of the convolutional neural network, also called the *backbone*, is a bunch of layers that are used to encode the input image into a feature matrix. That feature matrix encodes all the important information seen in the image and it will be the input for the decoder.

The *decoder* will decode that matrix and will generate the output of the convolutional neural network (a mask in this case).

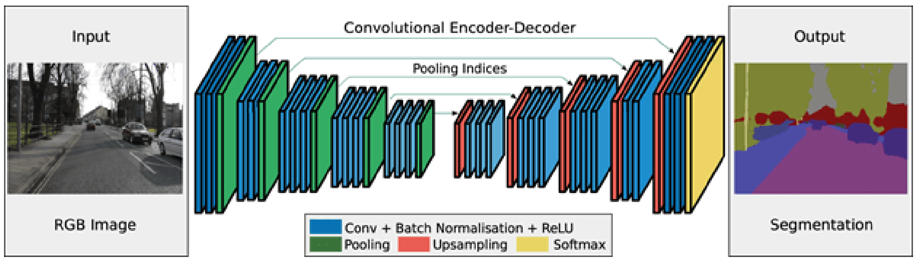


Figure 2.4. General semantic segmentation convolutional neural network structure [7].

The weights values of the encoder (that can be seen as an image compressor) are valuable information that will be used to retrain some models in a task called *fine tuning* [12]. Those encoders are usually available in the cloud and several libraries allow their users to employ them, defining new convolutional neural networks with those encoders and new decoders, making the process of training shorter and getting better results. So, indeed, *backbones* are modular components that can be replaced in the models.

To train a model, the data in the train set will be shown in *epochs*. Given the fact that those datasets may have thousands of images, they need to be packaged into small groups called *batches*. A batch is the amount of images that are parallely processed by the convolutional neural network. Since those images are generally processed by the GPU, the batch size is almost always a power of two. The predictions of those batches are used to update the weights in the training phase. Several epochs can be needed to get the expected results.

Fine tuning is the task of retraining (also called *refine*) a model with an encoder pretrained in a, generally, larger dataset. The model will learn the basic configuration with the data contained in that dataset and it will be refined with the custom dataset (the train set of the data), specialising the model into a specific task. In this process, the weights of the encoder are freezed (they will not change) for a couple of epochs while the decoder weights are trained. Then, the full model is trained as usual.

The model is trained using the train set and validated using the validation set. Once it is fitted (trained), we will have a model and the test set is used to evaluate it. This set is saved apart from the process of training to avoid *data leakage* [13]. This problem appears when some data from outside the training set is used to train the model. Doing so, it will learn some information that it otherwise would not know. Generally, data leakage is produced when training and testing or training and validating sets are mixed. The *model evaluation* process is based on getting the prediction of the model to the images in the test set and comparing the predicted masks with the real masks. Several kinds of comparisons can be used and they are called *metrics*. In semantic segmentation, the most used metric in binary class tasks (where there are only two classes) is IoU, known also as Jaccard Index [14].

The *IoU* metric compares the amount of pixels that really belong to the class with the amount of pixels of the class (predicted and real). In other words, it is the coefficient between the intersection of the masks and their union. This concept is shown in Figure 2.5. In this metric, predicting the correct identifier of the pixels is important (intersection), but it is also predicting just the identifier of the pixels (union). Those metrics formulas are shown in Figure 2.6.

$$DSC(X,Y) \;=\; \frac{2\,|X\cap Y|}{|X|+|Y|}\,;\, J(X,Y) \;=\; \frac{|X\cap Y|}{|X\cup Y|}$$

Figure 2.6. Difference between Dice and IoU metrics.

In Figures 2.7 and 2.8, we show a visual explanation of this concept. The real and predicted masks can be seen in Figure 2.7, whereas their intersection and union are represented in Figure 2.8.

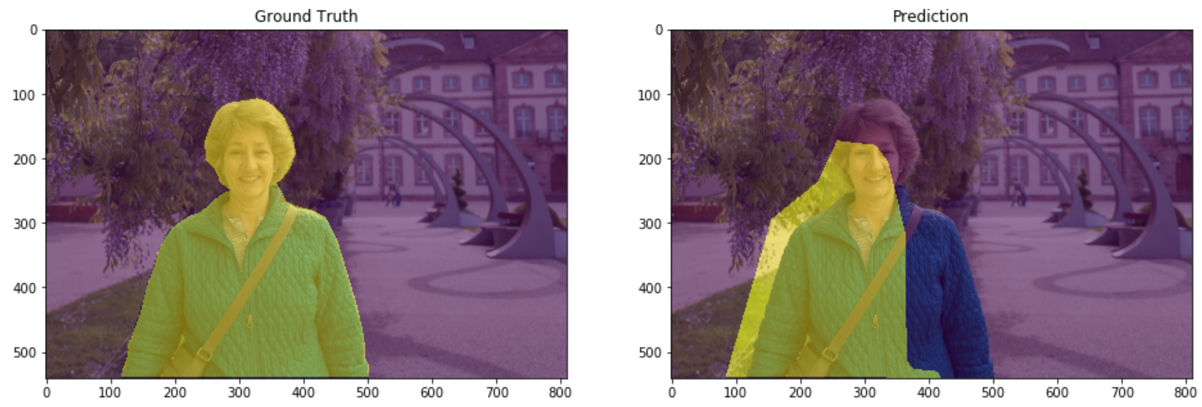


Figure 2.7. An example of real and predicted masks [15].

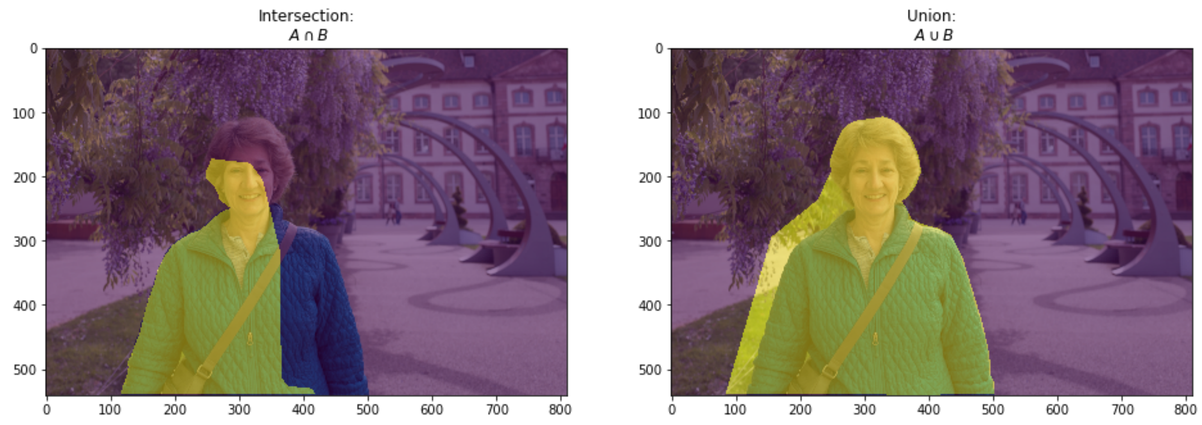


Figure 2.8. The intersection and union between the previous masks [15].

Another metric used in multiclass tasks is the *DiceMulti coefficient*. This metric is the averaged version of the *Dice* metric, that is also used in binary class tasks. Those metrics get the coefficient between two times the intersection and the sum of the elements in both classes.

Finally, with the model and a new image, inference can be done. *Inference* is the process of predicting a new mask for an image that has never been seen by the model.

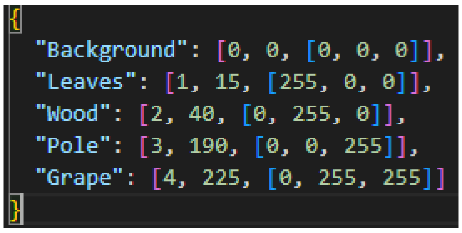
2.2. An example of building an image segmentation model

We have mentioned all the concepts related to image segmentation models, but we have not built any model yet. Now, we will explain how to build an image segmentation model using the *FastAI* library, a library whose goal is "to make deep learning as accessible as possible" [16].

2.2.1. Getting and preprocessing the dataset

Firstly, we need to get a dataset. In this case, because of the implementation of *FastAI*, the dataset will be stored in a directory tree. The tuples of *(image, mask)* will be splitted into train and test subdirectories and, then, splitted again into images and masks subdirectories. We will use a grapevine dataset with the task of segmenting the grapes, the pole, the wood and the leaves of the grapevines [17]. An example of an image and a mask in this dataset was shown in Figure 2.2.

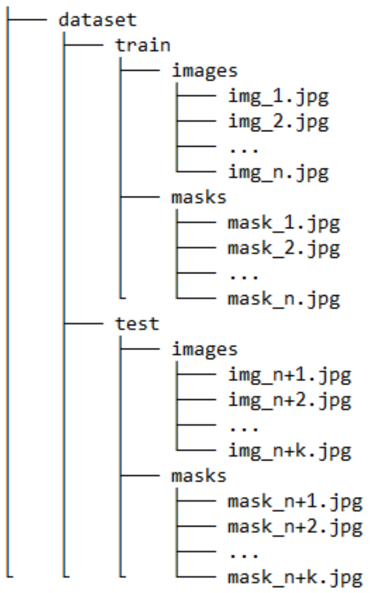
Additionally, we need to define a file that defines the codes and identifiers bijection. This file is a *json* file (a dictionary of key-value pairs) which has the name of the label and the biyection associated with the label: its natural number, its real value and its RGB tuple. An example of this file can be shown in Figure 2.9.



```
{
    "Background": [0, 0, [0, 0, 0]],
    "Leaves": [1, 15, [255, 0, 0]],
    "Wood": [2, 40, [0, 255, 0]],
    "Pole": [3, 190, [0, 0, 255]],
    "Grape": [4, 225, [0, 255, 255]]
}
```

Figure 2.9. An example of *codes.json* file.

We also need to create the directories structure that will fit with the structure shown in Figure 2.10.



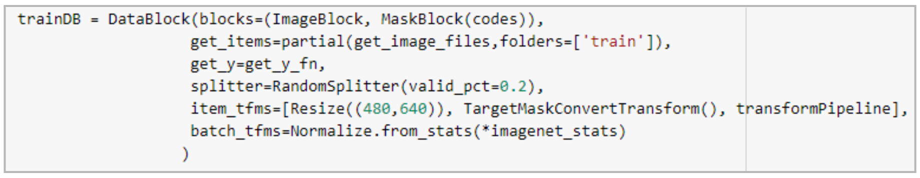
```
├── dataset
│   ├── train
│   │   ├── images
│   │   │   ├── img_1.jpg
│   │   │   ├── img_2.jpg
│   │   │   ├── ...
│   │   │   └── img_n.jpg
│   │   ├── masks
│   │   │   ├── mask_1.jpg
│   │   │   ├── mask_2.jpg
│   │   │   ├── ...
│   │   └   └── mask_n.jpg
│   ├── test
│   │   ├── images
│   │   │   ├── img_n+1.jpg
│   │   │   ├── img_n+2.jpg
│   │   │   ├── ...
│   │   │   └── img_n+k.jpg
│   │   ├── masks
│   │   │   ├── mask_n+1.jpg
│   │   │   ├── mask_n+2.jpg
│   │   │   ├── ...
└   └   └   └── mask_n+k.jpg
```

Figure 2.10. Structure of directories after the preprocessing of the dataset.

2.2.2. Defining the DataBlock and the DataLoader

Once the dataset is built, we need to define the way to load the data into memory, so that the convolutional neural network can learn from it. *FastAI* implements an API that helps the users to define this “data loading pipeline”. The principal unit of this API is the DataBlock.

Given some blocks (how to load a single type of element inside the images subdirectory – there are several ways to load an image, but it can also load text or other types using different blocks), a list of image-paths, a function that can find the mask related to the loaded image-path, a splitter and some transformations, we can define a DataBlock that will be able to load all the images, their annotations, make a validation split, and do some augmentations (transformations applied to the images and masks, like flips, crops or dropouts) to improve the training results. An example of this DataBlock is shown in Figure 2.11.



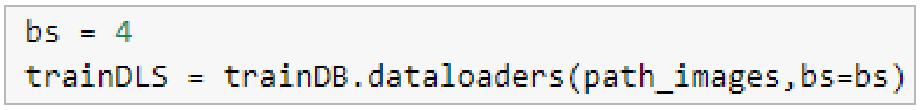
```
trainDB = DataBlock(blocks=(ImageBlock, MaskBlock(codes)),
                   get_items=partial(get_image_files,folders=['train']),
                   get_y=get_y_fn,
                   splitter=RandomSplitter(valid_pct=0.2),
                   item_tfms=[Resize((480,640)), TargetMaskConvertTransform(), transformPipeline],
                   batch_tfms=Normalize.from_stats(*imagenet_stats)
                  )
```

Figure 2.11. The DataBlock is defined to load the grapevine dataset.

In the DataBlock of Figure 2.11, we can see the following components:

- The blocks that we need are the ImageBlock to load the images and the MaskBlock to load the masks (using the codes defined in the *codes* file).
- We will use the function get_image_files defined in *FastAI* to get all the images in a group of directories recursively (train subdirectory in this case). We need to define a partial function because get_items hyperparam is expecting a function with only one parameter. The function partial returns a new function fixing a bunch of hyperparams (like folders in this case).
- The map between images and masks is defined by the function get_y_fn, defined by us. Given an image-path, it replaces the parts of the path as needed to get the related mask-path. In our scenario, and given the directory structure defined in Figure 2.10, these replacements are: “images” to “masks”, “img” to “mask” and “jpg” to “png”.
- The splitter hyperparameter is the way to split our data into train and validation datasets. The validation dataset is used to evaluate the neural network through the epochs. In this case, it will take twenty percent of the input training data and use it as validation.
- The item_tfms and batch_tfms are the transformations applied to the data. In this case, we use a Resize of the image, a mask conversion (from the grey-scale image into a identifier map matrix – using the bijection defined in the *codes* file) and several *Albumentations* [18] composed in the variable *transformPipeline*.

Given the DataBlock, we can easily get the DataLoader using the function dataloaders as is shown in Figure 2.12. The relevant hyperparameters of this function are, in this case, the path to the directory and the batch size.

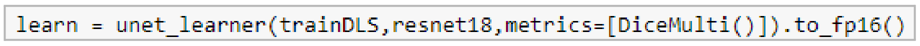


```
bs = 4
trainDLS = trainDB.dataloaders(path_images,bs=bs)
```

Figure 2.12. Defining the DataLoader.

2.2.3. Training the `Learner`

Finally, with the `DataLoader`, the backbone and some metrics (that depend on the problem – only `DiceMulti` in this case –), we can define our network as shown in Figure 2.13.



```
learn = unet_learner(trainDLS,resnet18,metrics=[DiceMulti()]).to_fp16()
```

Figure 2.13. Defining the `Learner`.

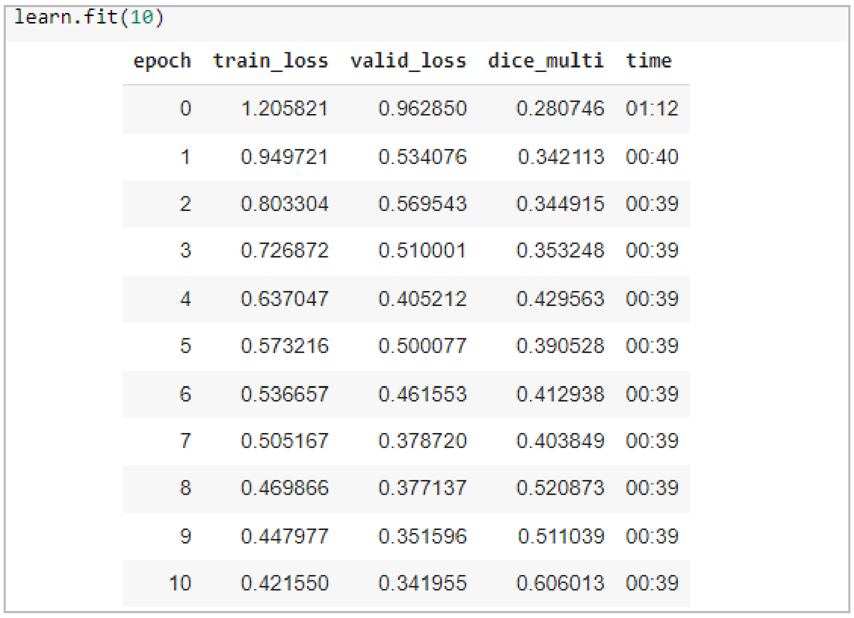
We are defining a neural network with the UNet architecture and the ResNet18 pretrained backbone, but we could also have used the possible architectures that can be consulted in the github repository of *FastAI [19]*.

We can modify the hyperparameter of the backbone and use one of the several backbones availables. It is important to know that not all the backbones can be used with one architecture. Those incompatibilities must be known to correctly define the neural network. If we try to construct it with an incompatible architecture and backbone pair, an error will be raised in the construction or training phases.

Finally, once the convolutional neural network is defined, it is time to train it. There are several ways to do it by using *FastAI*, but the most well known are the methods: `fit`, `fit_one_cycle` and `fine_tune` from the `Learner` object.

The `fit` method is the most basic training loop: it uses the input data to get the prediction and updates the weights of the network using the `forward` function and the static learning rate value. The `fit_one_cycle` method uses the 1cycle policy [20]: this technique uses discriminative learning rates (modify the learning rates from high values to small values through the epochs) to improve the common training loop. Finally, the `fine_tune` method uses the technique of fine tuning explained before.

An example of training using the `fit` function is shown in Figure 2.14.



```
learn.fit(10)
```

epoch	train_loss	valid_loss	dice_multi	time
0	1.205821	0.962850	0.280746	01:12
1	0.949721	0.534076	0.342113	00:40
2	0.803304	0.569543	0.344915	00:39
3	0.726872	0.510001	0.353248	00:39
4	0.637047	0.405212	0.429563	00:39
5	0.573216	0.500077	0.390528	00:39
6	0.536657	0.461553	0.412938	00:39
7	0.505167	0.378720	0.403849	00:39
8	0.469866	0.377137	0.520873	00:39
9	0.447977	0.351596	0.511039	00:39
10	0.421550	0.341955	0.606013	00:39

Figure 2.14. The training of the `Learner`.

Currently, *FastAI* only supports the UNet architecture, but there are other architectures that may get better results. Those architectures are provided in other libraries, but their use is not as straightforward as with the *FastAI* library.

Given this fact, we need to search which are the libraries that implement the state-of-the-art architectures and backbones for semantic segmentation, and put them together in a common library. Doing so, it will be possible to use all the architectures and backbones defined in those libraries with a common endpoint and a way to use them.

3. Creating a library to build image segmentation models

In this section, we propose the creation of a library to build image segmentation models based on the *FastAI* features. The aim is to take advantage of all the functionalities provided by the FastAI library to train different architectures provided by several libraries. As we have indicated previously, there are several libraries that implement image segmentation models (like *SemTorch [21]* or *segmentation models for PyTorch [22]* – henceforth, *smp*). Nevertheless, the way to build and use those models depends on the library. The aim for this section is to create an abstraction of all those libraries that will allow us to generalise and simplify the way to use them as much as possible.

3.1. Analysis of existing libraries

Image segmentation models are defined by means of an architecture and a backbone; therefore, we can suppose that the most important choice taken to create them (regardless how they are trained) is the choice of those components. There are various architectures scattered on the libraries (even repeated) that use several backbones, the first step to create our abstraction is to analyse and collect all those libraries, and the architectures and backabones available in them.

3.1.1. An exhaustive search for image segmentation libraries

To train image segmentation models, we need an implementation that covers their definition and training. That functionality is collected and provided by several libraries and we need to search for the one that best fits with our task. In this case, we need to take one step further and think about what libraries can better help the users of our library. For this purpose, we need to look for all the libraries that we can find.

In Table 2.1, we show the name, the underlying library, the number of provided architectures, the number of provided backbones, and the last update for each found library in an exhaustive search [23–30]. We use the word *several* to refer to a number greater than fifty. In the table, the libraries that will be used in the future to build the abstraction are highlighted.

Name	**Base library**	**Architectures**	**Backbones**	**Last update**
FastAI	PyTorch	2	Several	Some days
MIScnn	TensorFlow, Keras	1	* Unlimited	4 months
SegmenTron	PyTorch	28	Several	2 years
SMP	PyTorch	9	18	3 months
PaddlePaddle		45	9	Some days
SM	TensorFlow, Keras	4	10	2 years
SemTorch	PyTorch, FastAI	5	34	1 year
MMSegmentation	PyTorch	33	Several	Some days

Table 2.1. Libraries found and library selection to the abstraction.

(*) *MIScnn* backbones are created using the *Keras API*, so they are custom (unlimited).

To make the first implementation easier, we decided to select a subset of libraries that are based on the same library. In this case, we have taken all the libraries that are implemented using *PyTorch*. Since such a library is also the underlying library for *FastAI*, we can build *FastAI* objects and use its methods, so they will have a common part in their use (they will build and train the models in a similar way). Those libraries have been also selected because of their amount of available backbones and last update.

Since there exist incompatibilities between some architectures and backbones, we need to check the relation between them in our selected libraries.

We show in Table 2.2 the possible combinations for *SemTorch*. This table is extracted from its documentation, where it can be found with more information [21].

Architecture	**Backbones**
UNet	resnet18 resnet34 resnet50 resnet101 resnet152 xresnet18 xresnet34 xresnet50 xresnet101 xresnet152 squeezenet1_0 squeezenet1_1 densenet121 densenet169 densenet201 densenet161 vgg11_bn vgg13_bn vgg16_bn vgg19_bn alexnet
DeepLabV3+	Resnet18 resnet34 resnet50 resnet101 resnet152 resnet50c resnet101c resnet152c exception65 mobilenet_v2
HRNet	Hrnet_w18_small_model_v1 henet_w18_small_model_v2 hrnet_v18 hrnet_w30 hrnet_w32 hrnet_w48
MaskRCNN	resnet50
U^{2}Net	Small, normal

Table 2.2. Architectures and backbones used in *SemTorch*.

The documentation of *Segmentron [31]* does not have a table of those relations, so we tried all the possible combinations of architectures and backbones supplied by this library. The majority of the architectures use specified backbones, but there are other architectures that have a different construction (they build their own network instead of using a backbone); those libraries are marked with a hyphen. We group together the usable combinations in Table 2.3 and 2.4.

Architecture	**Backbones**
BiSeNet	resnet18 resnet34
CGNet	-
ContexNet	-
DABNet	-
DeepLabv3+	mobilenet_v2 resnet18 resnet34 resnet50 resnet101 resnet152 resnet50c resnet101c resnet152c xception65

Table 2.3. Architectures and backbones used in *SegmenTron*.

Architecture	**Backbones**
DenseASPP	mobilenet_v2 resnet18 resnet34 resnet50 resnet101 resnet152 resnet50c resnet101c resnet152c
DFANet	resnet34
FCN	resnet50 resnet101 resnet152 resnet50c resnet101c resnet152c xception65
FPNet	-
HRNet	hrnet_w18_small_v1
LEDNet	-
OCNet	resnet50 resnet101 resnet152 resnet50c resnet101c resnet152c xception65
PSPNet	resnet50 resnet101 resnet152 resnet50c resnet101c resnet152c xception65
UNet	-

Table 2.4. Architectures and backbones used in *SegmenTron* (continuation).

The architectures [32] and backbones [33] of *smp* can be found in its documentation (we do not include a table of them because it would extend for several pages). Unlike the previous libraries, *smp* allows us to select, in addition to the architecture and the backbone, the dataset that was used to pretrain the backbone that we are going to use (like *ImageNet* [34] or *Instagram* [35]). This parameter could have relevance, so our library should include it.

Finally, the last library that we will include is *MMSegmentation* [36], from *OpenMMLab* [37]. This library is the largest by far. It has several architectures and pretrained backbones, using different datasets to train its backbones. Given this fact, we do not include the table either. Unfortunately, *MMSegmentation* is very different from the other libraries, so we will need to take some different steps to train its models.

Now, with all the collected information, we can finally start to create the library.

3.2. Library design

Given the fact that the aim for this section is the creation of a library that will simplify the process of creating models with the selected libraries, the *facade design pattern* [38] fits perfectly with the final product we want to produce. This design pattern is based on creating a simple interface for a group of complex components in a system, implementing just the functionality that the user will employ.

In our library, the library itself is the interface, the other libraries are the “complex components” and the functionality needed by the user is the construction of the model (regardless of the library used to do it).

In a completely arbitrary choice, we will name our interface (our library) *SegmentationManager*.

We need to select which parameters are the most important ones to build our library. As we have explained in the analysis of libraries, we need to provide: the architecture, the backbone and the name of the dataset used to train the backbone.

We know that *MMSegmentation* uses two files (to build the convolutional neural networks) where the backbone and the weights are defined. So, to work with this library, we will need to know where these files are stored. Generally, the user will know the name of the architecture, but not the name of the files used by *MMSegmentation*, so we will need an auxiliary function that will search the files that defines the architecture and its backbone (using the known name, find the path to the files).

Additionally, we should note that there are some architectures that are replicated in those libraries (for instance, *UNet, DeepLabV3+* or *HRNet*). For those cases, we need to choose which library will be used to build the requested model. Taking this decision manually for all the duplicated implementations of architectures or backbones in the libraries can be a complex task for the final user of our library. To avoid this issue, we will look up for the libraries that can be used to build the requested model and, if there is more than one, one library will be taken randomly to build it.

Finally, we need to implement the function that builds the model. This function should return a `Learner` object in order to take advantage of all the implemented *FastAI* functionality, but we need to remember that *MMSegmentation* does not work with this object, but with another object named `EncoderDecoder`. Because of that, we need to group together all the functionality to use them likewise. Hence, we need to make a decision: create an object that groups together the `Learner` and the `EncoderDecoder` or look up for a way to transform the `EncoderDecoder` object into a `Learner` object.

In order to simplify the use of our library, we consider that the returned object must be a `Learner` from *FastAI*, which has a lot of functionality already implemented and explained.

When the library is built, we will be able to use it to request the construction of a model. The way to use it would be an endpoint (a simple function) that receives an architecture, a backbone and name of a dataset and returns a `Learner` from *FastAI.* This object could be trained using the functions explained in Section 2.

3.3. Library implementation

The first step is the installation of all the packages needed to build our library, like libraries or github repositories. We have installed *SemTorch* and *smp* libraries from pypi [39], the Python package index; and downloaded *SegmenTron* and *MMSegmentation* repositories.

In this first step, we found three issues:

- The repository of *SegmenTron* is not updated and we need an non-existent configuration file to use it.
- *MMSegmentation* library is in the developing phase and it is changing continuously. We need to define the way this library is used, updating the local files, getting the updates dynamically or downloading the files of the last stable version.
- Some of the libraries that we want to abstract use distinct versions of other basic libraries (like *torch* or *torchvision).* Those versions are, generally, incompatible.

To solve the first issue, we modified the repository of *SegmenTron* to build the models without the configuration file. This work was already done since there exists a forked version of *SegmenTron* that changes the code as we need. Another option would have been to create the configuration file on our own. Although this solution is the cleanest (we do not need to change the code of the original repository), it would have taken much more time (there is not a documentation that specifies how to create this file and it would have been an inverse engineering process from the existing code). To use the modified version, we forked the original repository and added the necessary modifications[1].

To solve the second issue, and keeping in mind that we want a robust abstraction, we chose to use the last stable version of *MMSegmentation*. The initial idea was to manage the download of *MMSegmentation* automatically if it was detected that the files do not exist. Nevertheless, if we want to make this process completely transparent to our users, we need to also detect if the files have changed (it will mean that there is an updated version). If so, the files should be updated (delete the old ones and download again "the new version", which now is another version).

Solving the third issue, which is related to the incompatibility of the libraries, was a patience test. If we check which are exactly the versions that allow or reject the libraries we are going to use, we note that there is an installation order that allows us to use all of them without incompatibilities. This order is:

1. *smp (library).*
2. *SemTorch (library).*
3. GitPython (*auxiliary library)*.
4. PrettyTable (*auxiliary library)*.
5. *SegmenTron (library).*
6. mmcv-full *(library).*
7. torch (1.6.0 + cuda101) (*auxiliary library)*.
8. torchvision (0.7.0 + cuda101) (*auxiliary library)*.

Once all the libraries are installed, the second step is to create the facade that will allow us to use the selected libraries in a general way. For that, we need to implement the functions described in the design phase.

3.3.1. Selecting the library

When someone uses our abstraction, they will request a specific configuration (for example, a convolutional neural network with *UNet* architecture and *Resnet18* backbone). In order to determine which is the most appropriate, we define a function named `_choose_library_`. We can use the known particularities to do this task. For example, we know that the names of the backbones from *MMSegmentation* library ends with ".py", so we will use this library if the last three characters of the backbone name are ".py". In the case of *MMSegmentation*, we need to take the names of the architectures directly from the files, so we will look up for the available architectures in the `configs` directory.

This function will return the name of the library that is available and can build the model. If there is no library to build it, the function will raise an exception.

3.3.2. Searching the architecture and backbone files

When *MMSegmentation* is used to build a model, two parameters are needed: the config and the checkpoint file (the backbone and its weights). These files can be found in the downloaded repository. Indeed, each config file has a checkpoint file related, so we only need to know about the first one to look up for the other. In other words, the users only need to know what architecture they want to use.

[1] The github repository can be visited in https://github.com/ruescog/SegmenTron

The function `_search_files_` will use the config file of the requested model to search what file of weights may be used and, then, it will return the paths of this configuration and the related backbone file (that probably needs to be downloaded).

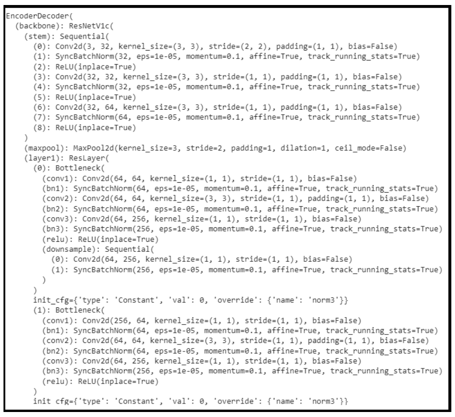
3.3.3. Creating the model

Finally, we create the function that returns the model to the user. For *SemTorch*, regardless of the requested architecture, we only need to call one function. For *smp*, *SegmenTron* and *MMSegmentation* we need to use the specific constructors. However, as we have mentioned before, the result of calling the constructor of the library *MMSegmentation* is an `EncoderDecoder` object, which does not have the same methods as a `Learner` object.

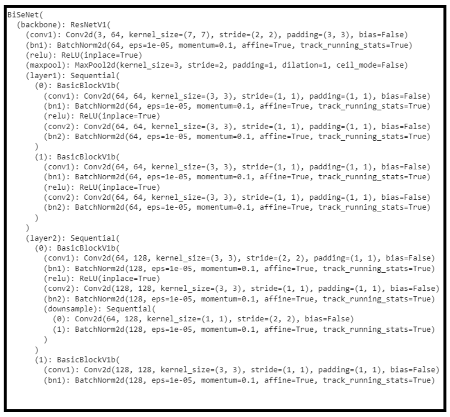
3.3.3.1. Trying to convert the EncoderDecoder into a Learner

As we have mentioned in the design phase, one of the options to combine the `Learner` and the `EncoderDecoder` was to create a new object that defines all the methods for both types of objects. This is a partial solucion: in the future, there may be other kinds of objects defining a convolutional neural network that will not be covered in our library due to this design, so we propose to transform the `EncoderDecoder` into a `Learner`. Moreover, creating a new object adds another layer in our abstraction, making the facade less transparent to the users (they will need to know what the new object is and how it works). For all these reasons, we decided to implement the second idea: the `create_model` function will return a `Learner` from *FastAI*.

At this point, we have an `EncoderDecoder` that we want to transform into a `Learner`. The first questions we ask ourselves are: Can we really do this transformation? Is the `EncoderDecoder` similar to the structure used to build a `Learner`? To solve them, we can build both objects and look up their differences. See Figure 3.1.



```
EncoderDecoder(
  (backbone): ResNetV1c(
    (stem): Sequential(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (1): SyncBatchNorm(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): SyncBatchNorm(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
      (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (7): SyncBatchNorm(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): ReLU(inplace=True)
    )
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1): ResLayer(
      (0): Bottleneck(
        (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): SyncBatchNorm(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): SyncBatchNorm(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): SyncBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): SyncBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      init_cfg={'type': 'Constant', 'val': 0, 'override': {'name': 'norm3'}}
      (1): Bottleneck(
        (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): SyncBatchNorm(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): SyncBatchNorm(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): SyncBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      init cfg={'type': 'Constant', 'val': 0, 'override': {'name': 'norm3'}}
```



```
BiSeNet(
  (backbone): ResNetV1(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1): Sequential(
      (0): BasicBlockV1b(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): BasicBlockV1b(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlockV1b(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): BasicBlockV1b(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

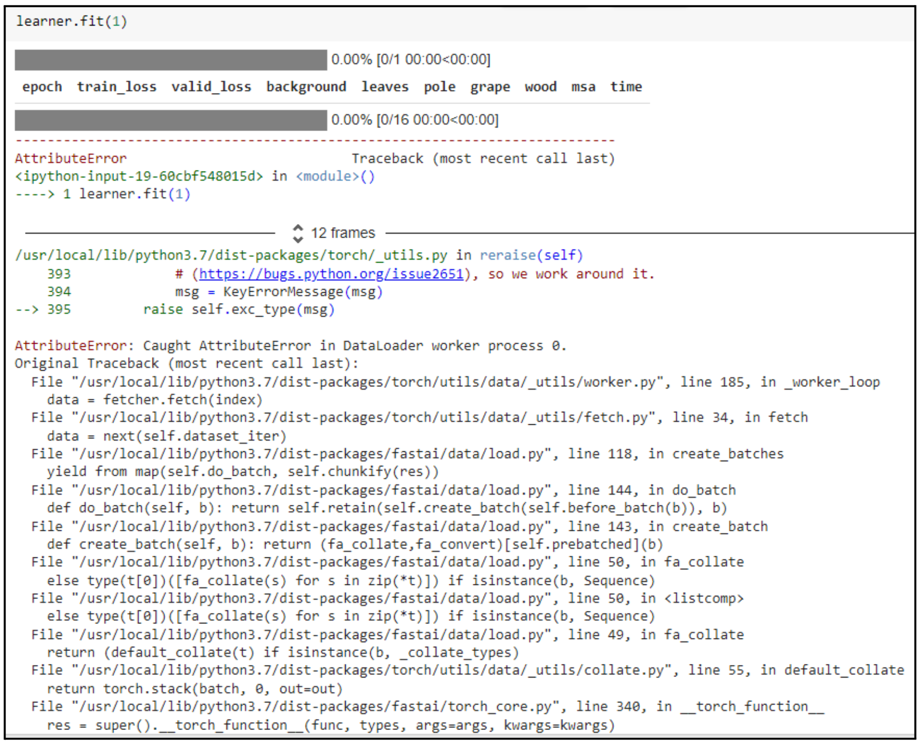
Figure 3.1. Differences between an `EncoderDecoder` and the structure used to build a `Learner`.

As we have shown in Figure 3.1, the model used to build the `Learner` object has the same structure (in this case, the structure of the backbone) as the `EncoderDecoder` object. Given this fact, we can suppose that an `EncoderDecoder` object can be used to build a `Learner`. To try it, we will need a `DataLoader` (used in Section 2.2.2) and a convolutional neural network (the structure, their weights…). Suprisely, given the `DataLoader`, we can build a `Learner` using the `EncoderDecoder`. Nevertheless, if we try to train the built `Learner`, we get the exception shown in Figure 3.2.

So we ask ourselves, again: Is this conversion possible? Have we built our `DataLoader` correctly? The answer to these questions will guide the solution to this problem.

If we try to solve the first question, the solution might be to build a dummy `Learner`, in other words, a kind of an empty skeleton, and add the configuration of the requested architecture and backbone to this skeleton. From both objects (`EncoderDecoder` and `Learner`) we have access to the structure itself, so we can modify it. The key point is that we need to know, exactly, what we need to modify. The layers of those models handle the information sequentially, like a conveyor, so the input from the second layer is the output from the first one. Given this fact, it is needed to be extremely careful modifying those layers or it will contain shape incompatibilities.

Since modifying the existing model is not a recommended task, we propose to investigate another solution for this issue. For example, there could exist some default functionality that can help us to do the transformation. Indeed, we found an existing function in the library *MMDetection* [40] (from the same group *OpenMMLab*) that converts `EncoderDecoders` into `Learners`. However, this function works with specific architectures: for example, it converts *RetinaNet* [41] models into `Learners`. Although these functions are useful, our issue persists: the libraries we are handling have dozens of models and there is not a function for each model. Furthermore, they are not general functions that can handle any kind of architecture, but only the ones they are related to.



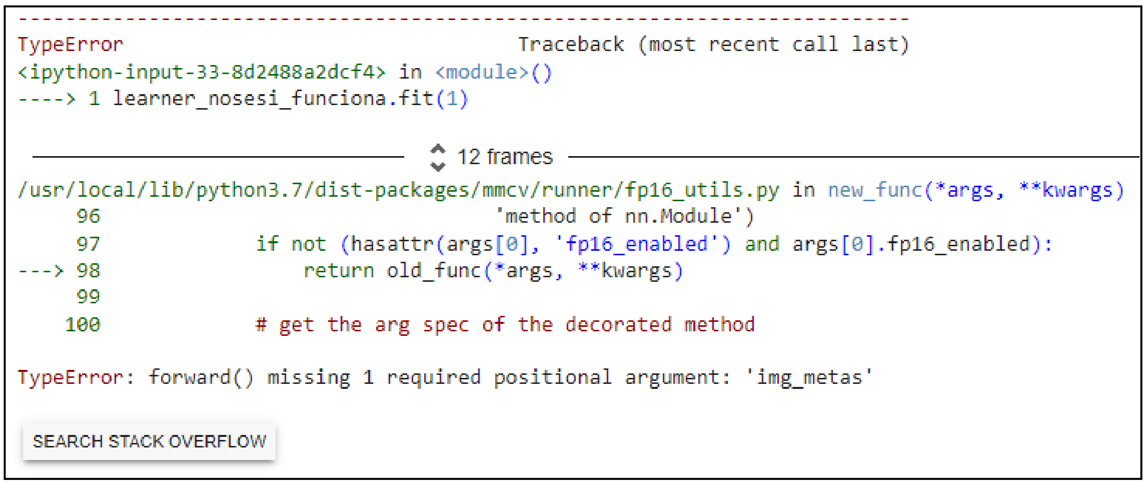
```
learner.fit(1)

0.00% [0/1 00:00<00:00]
 epoch  train_loss  valid_loss  background  leaves  pole  grape  wood  msa  time

0.00% [0/16 00:00<00:00]
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-19-60cbf548015d> in <module>()
----> 1 learner.fit(1)

12 frames
/usr/local/lib/python3.7/dist-packages/torch/_utils.py in reraise(self)
    393             # (https://bugs.python.org/issue2651), so we work around it.
    394             msg = KeyErrorMessage(msg)
--> 395         raise self.exc_type(msg)

AttributeError: Caught AttributeError in DataLoader worker process 0.
Original Traceback (most recent call last):
  File "/usr/local/lib/python3.7/dist-packages/torch/utils/data/_utils/worker.py", line 185, in _worker_loop
    data = fetcher.fetch(index)
  File "/usr/local/lib/python3.7/dist-packages/torch/utils/data/_utils/fetch.py", line 34, in fetch
    data = next(self.dataset_iter)
  File "/usr/local/lib/python3.7/dist-packages/fastai/data/load.py", line 118, in create_batches
    yield from map(self.do_batch, self.chunkify(res))
  File "/usr/local/lib/python3.7/dist-packages/fastai/data/load.py", line 144, in do_batch
    def do_batch(self, b): return self.retain(self.create_batch(self.before_batch(b)), b)
  File "/usr/local/lib/python3.7/dist-packages/fastai/data/load.py", line 143, in create_batch
    def create_batch(self, b): return (fa_collate,fa_convert)[self.prebatched](b)
  File "/usr/local/lib/python3.7/dist-packages/fastai/data/load.py", line 50, in fa_collate
    else type(t[0])([fa_collate(s) for s in zip(*t)]) if isinstance(b, Sequence)
  File "/usr/local/lib/python3.7/dist-packages/fastai/data/load.py", line 50, in <listcomp>
    else type(t[0])([fa_collate(s) for s in zip(*t)]) if isinstance(b, Sequence)
  File "/usr/local/lib/python3.7/dist-packages/fastai/data/load.py", line 49, in fa_collate
    return (default_collate(t) if isinstance(b, _collate_types)
  File "/usr/local/lib/python3.7/dist-packages/torch/utils/data/_utils/collate.py", line 55, in default_collate
    return torch.stack(batch, 0, out=out)
  File "/usr/local/lib/python3.7/dist-packages/fastai/torch_core.py", line 340, in __torch_function__
    res = super().__torch_function__(func, types, args=args, kwargs=kwargs)
```

Figure 3.2. Exception obtained trying to train the "Frankenstein" `Learner`.

At this point, we ran out of ideas to follow this line of research, so we reconsider the task we are working on. We have been trying to create new `Learners` from existing ones, but perhaps the problem is not in the creation of the model, but in the associated `DataLoader`. In fact, if we go back to the exception shown in Figure 3.2, we can see a line that reads as follows: "*Caught AttributeError in DataLoader worker process 0*". In other words, the `DataLoader` creation is not complete. This issue is caused by using the `DataBlock` wrongly, because we are using an outdated version from *FastAI*.

To update it, we found a library that manages all the installation process automatically, solving all dependencies issues (instead of doing it manually, as we have solved the third issue in the first part of this section). This library is *openmim [42]*. We update all the libraries using it and build again the `DataBlock`, but we still get an exception when we try to fit the `Learner`. See Figure 3.3.

As we can see in Figure 3.3, the `forward` function takes one additional parameter that we are not supplying. To solve this, we need to modify our `DataBlock` to indicate that there exists one more input: the *img_metas* (a dictionary with information about the image: its shape, scale and flip ratio). This modification can be handled with several options, we consider to modify the amount of parameters this function takes or to create a pipeline that will produce two outputs (the image and its metainformation).

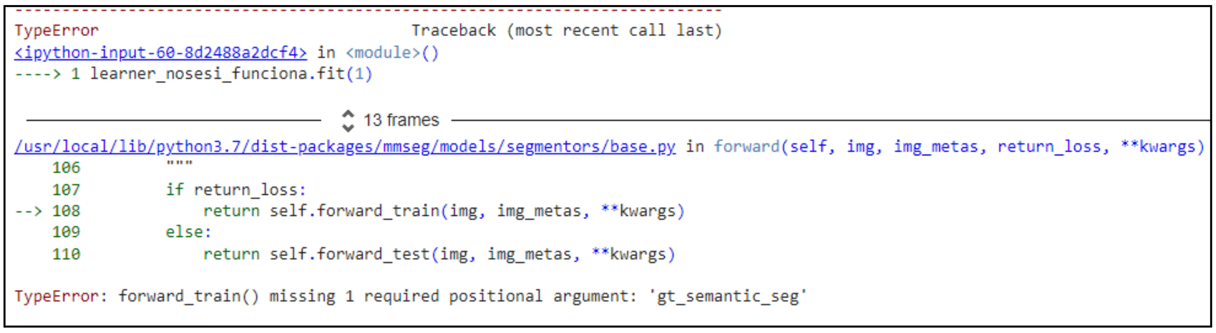


```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-8d2488a2dcf4> in <module>()
----> 1 learner_nosesi_funciona.fit(1)

                              12 frames
/usr/local/lib/python3.7/dist-packages/mmcv/runner/fp16_utils.py in new_func(*args, **kwargs)
     96                                    'method of nn.Module')
     97             if not (hasattr(args[0], 'fp16_enabled') and args[0].fp16_enabled):
---> 98                 return old_func(*args, **kwargs)
     99 
    100             # get the arg spec of the decorated method

TypeError: forward() missing 1 required positional argument: 'img_metas'
```

Figure 3.3. Exception in the forward function: one parameter is missing.

We tried to create a pipeline, but we fastly noted that is not a possible solution to our problem since those pipelines are used to transform the input data, not to create new parameters, so we still have the exception shown in Figure 3.3.

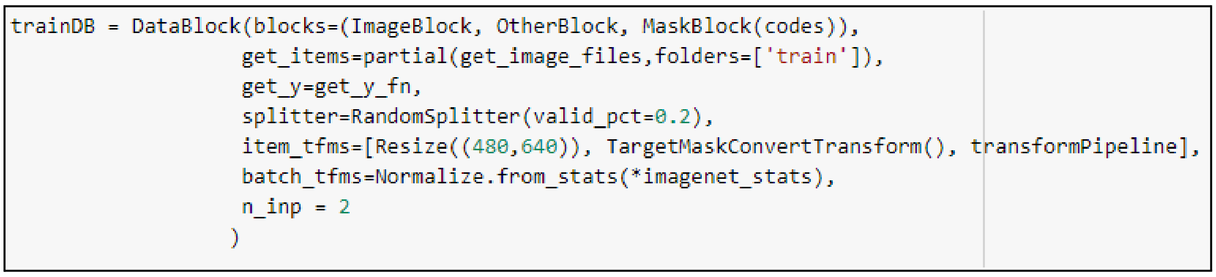
Given this fact, we only have one option left that consists in modifying the input parameters to the `forward` function (what it seems to be trivial: modify one building parameter in the `DataBlock`), but we got a new exception shown in Figure 3.4.



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-60-8d2488a2dcf4> in <module>()
----> 1 learner_nosesi_funciona.fit(1)

                              13 frames
/usr/local/lib/python3.7/dist-packages/mmseg/models/segmentors/base.py in forward(self, img, img_metas, return_loss, **kwargs)
    106         """
    107         if return_loss:
--> 108             return self.forward_train(img, img_metas, **kwargs)
    109         else:
    110             return self.forward_test(img, img_metas, **kwargs)

TypeError: forward_train() missing 1 required positional argument: 'gt_semantic_seg'
```

Figure 3.4. Exception in the `forward` function: one parameter is missing (again).

As we can see in Figure 3.4, the `forward` function takes all what it needs, but the `DataBlock` does not know where the information of the masks (*gt_semantic_seg*) is. This is caused because we have two inputs and one output parameters (*input image, img_metas* and *mask*), whereas the `DataBlock` only loads two of them (*input image* and *mask*). To solve this issue we need to create a new Block, allowing the DataBlock to take all the input data it needs. This Block will be a new personalised Block, it will not load an image, but a dictionary, as is shown in Figure 3.5.



```
trainDB = DataBlock(blocks=(ImageBlock, OtherBlock, MaskBlock(codes)),
                   get_items=partial(get_image_files,folders=['train']),
                   get_y=get_y_fn,
                   splitter=RandomSplitter(valid_pct=0.2),
                   item_tfms=[Resize((480,640)), TargetMaskConvertTransform(), transformPipeline],
                   batch_tfms=Normalize.from_stats(*imagenet_stats),
                   n_inp = 2
                  )
```

Figure 3.5. DataBlock with three Blocks and two inputs.

Again, we find an exception: since the DataBlock has multiple inputs, the way to group together all the input elements (*get_items* and *get_y*) must be defined in a list of functions. Nevertheless, even defining the *get_items* parameter correctly, we get the same error shown in Figure 3.5.

At this point, we ran out of ideas (again), so we need to step back again and look up for other solutions.

If we go back to the beginning of this section, we can retry an idea discarded before: build an encapsulating object. To do so, we need to create the convolutional neural network and box it inside an object that has more information. To create an EncoderDecoder object, we need to use several methods of this library and then modify the default configuration to specify the characteristics of the data and the model (input classes, training pipeline…). Then, a personalised EncoderDecoder will be built and saved into the encapsulating object.

Keeping in mind the key idea followed in this section (there may exist a relationship between EncoderDecoder and the Learners), we are going to build a DataSet that will define the way the data is accessed and, using this DataSet, we will build the DataLoader (without using the DataBlock). Then, with the DataLoader, we will be able to build the Learner and train it.

Since it is a new point of view to solve the principal issue of this section, we have new drawbacks too: DataSets from *FastAI* and datasets from *MMSegmentations* are different objects, they have different information. Given this fact, we can not use the *FastAI* methods to access the information of the datasets from *MMSegmentation*.

And this is as far as we have come. We need to find another way to finally move forward.

4. A library that covers the training process for image segmentation models

In this section, we are going to tackle the initial idea (make the process of training a semantic segmentation model as simple as possible) from another point of view. Instead of managing just the constructed model, we are going to facilitate the whole process: from the dataset construction to the model evaluation.

Obviously, all the work done before was not wasted time: we are going to use all that we have learned from the libraries and from the process of constructing models with them to build a new facade. This new facade will cover more functionality that the previous one: we are not only providing an endpoint to build the models, but to facilitate the whole process.

As we have seen before, *FastAI* and *MMSegmentation* workflows differ and they need to take different steps to construct and build their models successfully. In our scenario, three of the libraries abstracted use the *FastAI* way to work with the models (*SemTorch, Segmentron and smp*), meanwhile *MMSegmentation* has its own way. Given this fact, and knowing that they will be our first selected libraries to the first implementation of the facade, we need to face the difficulties of those libraries, especially the *MMSegmentation* ones. We have previously seen how to build a *FastAI* model in Section 2.2 and, now, we are going to construct a *MMSegmentation* model (that is something that we have not explained before) to show the differences with FastAI.

4.1. Building a *MMSegmentation* model

In order to show how to build a *MMSegmentation* model, we are going to use the library tutorial [43] and the dataset with grapevine images previously presented. The aim is the same as before: build a model that knows how to segment a grapevine image identifying the classes “background”, “leaves”, “wood”, “pole” and “grape”.

4.1.1. The *MMSegmentation* particularities

Before explaining the grapevine example, we are going to face the *MMSegmentation* particularities.

First of all, an image and its associated mask must have the same name. *MMSegmentation* calls *name* to the first common part in the name of the dataset files. Additionally (and to avoid the operating system name restrictions), the files will have a suffix. This suffix is, generally, the format of the file (such as *png* or *jpg*), but in some cases it can be a code and the format (for instance, *mask.png* or *image.jpg*). In both cases, the file name will be the concatenation of the common part and the specific suffix.

Another issue is that at the time to create the training and testing splits, it is necessary to create two text files having the base name[2] of the images that belong to the training set and the name of those that belong to the test set respectively. In this case, the validation set is managed by *MMSegmentation* in an automated way, so we do not need to care about it.

Finally, we need to modify the base *MMSegmentation* model configuration in which we are going to base our training, indicating: the number of classes of the dataset, the data augmentation processes we are going to use and the training and testing configuration. Even if we need to construct and train

[2] The base name of a file is the name itself with the file format. Usually, when we use the term *file name*, we refer to the concatenation of its directory structure and its base name. For instance, *image.jpg* is a file base name while *dataset/image.jpg* is a file name.

a model from scratch, it is also recommended to use a defined configuration file and indicate that the backbone is not pre-trained (so its weights will be initialised randomly).

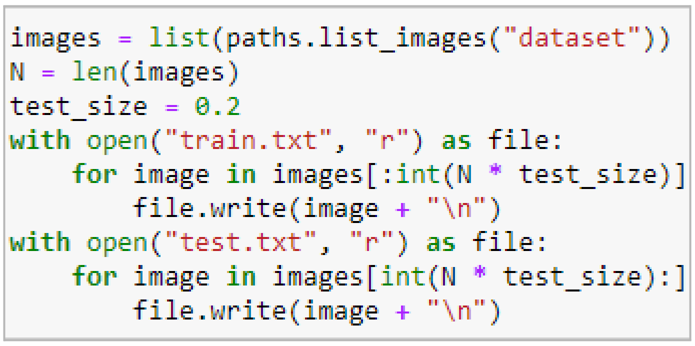
4.1.2. A *MMSegmentation* model construction example

Once all the *MMSegmentation* particularities are exposed, we can construct a model in order to get an example of its construction and training phases.

First of all, the grapevine dataset is divided into two directories (images and masks). Every image has the prefix *color_* and the masks have the prefix *gt_*, meanwhile they share a suffix with a number and the format. This number (o code) tracks the relation between an image and its mask. In other words, given an image and its mask, they will have the same code (but not the same format). So, to work with *MMSegmentation*, it is needed to delete those prefixes having a common base name.

Subsequently, it is needed to modify the mask format. This format modification is not a *MMSegmentation* particularity (indeed, we have also done it in the *FastAI* example, implementing a class that could transform a RGB image into a bidimensional matrix of identifiers). In this example, we need to do the same but, instead of using a pipeline, we are going to modify the saved files in the disk (not only in memory). This mapping must be known by the user, so it is an issue that we will take into account when developing our library.

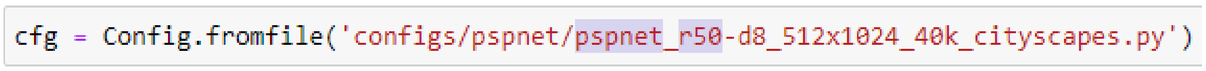
Then, as we want to train a model, it is needed to split the dataset into training and testing sets. To this aim, we have to create two new files and save inside of them the names of the files corresponding to each set. The code to conduct this process is shown in Figure 4.1.



```
images = list(paths.list_images("dataset"))
N = len(images)
test_size = 0.2
with open("train.txt", "r") as file:
    for image in images[:int(N * test_size)]
        file.write(image + "\n")
with open("test.txt", "r") as file:
    for image in images[int(N * test_size):]
        file.write(image + "\n")
```

Figure 4.1. Making the splits for *MMSegmentation.*

Finally, we need to modify the base model configuration file (from the model that we are going to use to do fine tuning – in our case, we use as architecture *PSPNet* and *ResNet50* as backbone –, see Figure 4.2) indicating that: we have five classes, we are going to use the default data augmentation process (the one defined in the base model) and the path to the training and testing sets. Some of those modifications are highlighted in Figure 4.3.



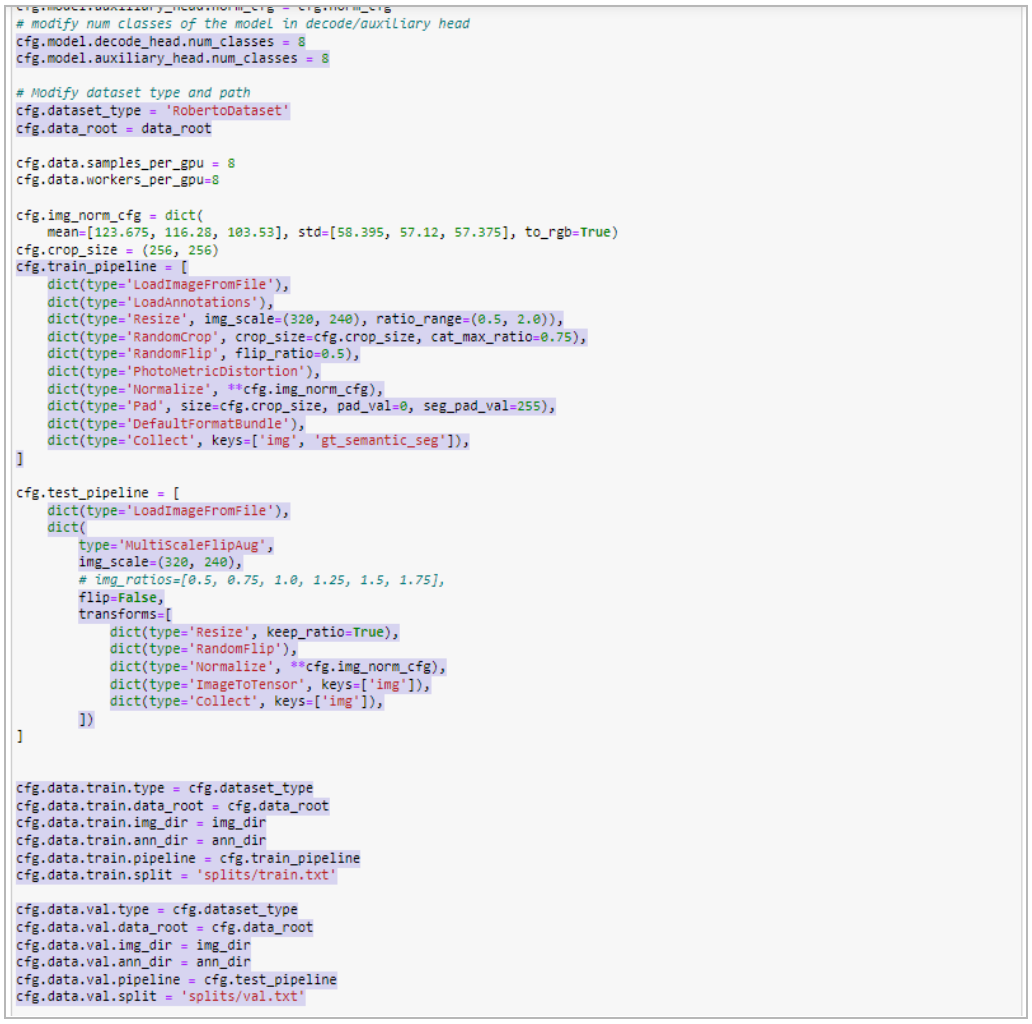
```
cfg = Config.fromfile('configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py')
```

Figure 4.2. Base model configuration loading.

Then, we can train our model, which result is shown in Figure 4.4. To this aim, we have to:

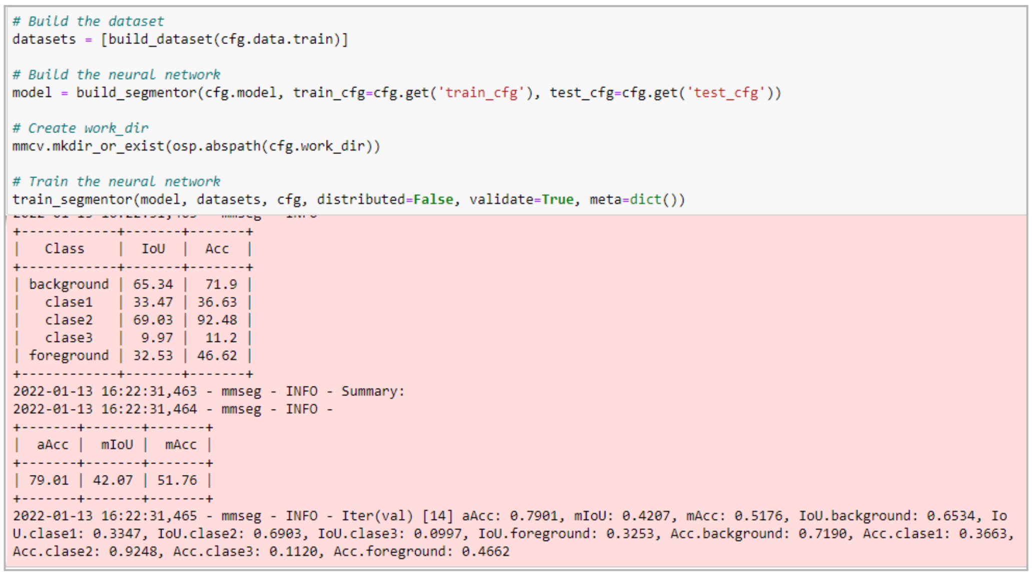
1. Load the dataset.
2. Construct the model from the modified configuration file.
3. Create a temporal file to save the training results.

4. Train the model.



```
cfg.model.auxiliary_head.norm_cfg = cfg.norm_cfg
# modify num classes of the model in decode/auxiliary head
cfg.model.decode_head.num_classes = 8
cfg.model.auxiliary_head.num_classes = 8

# Modify dataset type and path
cfg.dataset_type = 'RobertoDataset'
cfg.data_root = data_root

cfg.data.samples_per_gpu = 8
cfg.data.workers_per_gpu=8

cfg.img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
cfg.crop_size = (256, 256)
cfg.train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations'),
    dict(type='Resize', img_scale=(320, 240), ratio_range=(0.5, 2.0)),
    dict(type='RandomCrop', crop_size=cfg.crop_size, cat_max_ratio=0.75),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='PhotoMetricDistortion'),
    dict(type='Normalize', **cfg.img_norm_cfg),
    dict(type='Pad', size=cfg.crop_size, pad_val=0, seg_pad_val=255),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_semantic_seg']),
]

cfg.test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(320, 240),
        # img_ratios=[0.5, 0.75, 1.0, 1.25, 1.5, 1.75],
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **cfg.img_norm_cfg),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]


cfg.data.train.type = cfg.dataset_type
cfg.data.train.data_root = cfg.data_root
cfg.data.train.img_dir = img_dir
cfg.data.train.ann_dir = ann_dir
cfg.data.train.pipeline = cfg.train_pipeline
cfg.data.train.split = 'splits/train.txt'

cfg.data.val.type = cfg.dataset_type
cfg.data.val.data_root = cfg.data_root
cfg.data.val.img_dir = img_dir
cfg.data.val.ann_dir = ann_dir
cfg.data.val.pipeline = cfg.test_pipeline
cfg.data.val.split = 'splits/val.txt'
```

Figure 4.3. An example of the base model configuration modification.



```
# Build the dataset
datasets = [build_dataset(cfg.data.train)]

# Build the neural network
model = build_segmentor(cfg.model, train_cfg=cfg.get('train_cfg'), test_cfg=cfg.get('test_cfg'))

# Create work_dir
mmcv.mkdir_or_exist(osp.abspath(cfg.work_dir))

# Train the neural network
train_segmentor(model, datasets, cfg, distributed=False, validate=True, meta=dict())
```

```
+------------+-------+-------+
|   Class    |  IoU  |  Acc  |
+------------+-------+-------+
| background | 65.34 |  71.9 |
|   clase1   | 33.47 | 36.63 |
|   clase2   | 69.03 | 92.48 |
|   clase3   |  9.97 |  11.2 |
| foreground | 32.53 | 46.62 |
+------------+-------+-------+
2022-01-13 16:22:31,463 - mmseg - INFO - Summary:
2022-01-13 16:22:31,464 - mmseg - INFO - 
+-------+-------+-------+
|  aAcc |  mIoU |  mAcc |
+-------+-------+-------+
| 79.01 | 42.07 | 51.76 |
+-------+-------+-------+
2022-01-13 16:22:31,465 - mmseg - INFO - Iter(val) [14] aAcc: 0.7901, mIoU: 0.4207, mAcc: 0.5176, IoU.background: 0.6534, Io
U.clase1: 0.3347, IoU.clase2: 0.6903, IoU.clase3: 0.0997, IoU.foreground: 0.3253, Acc.background: 0.7190, Acc.clase1: 0.3663,
Acc.clase2: 0.9248, Acc.clase3: 0.1120, Acc.foreground: 0.4662
```

Figure 4.4. Training result for the grapevine dataset using *MMSegmentation*.

Each one of the aforementioned steps is conducted with one line of code as shown in Figure 4.4.

Once we have faced all the *MMSegmentation* construction and training process, we can start to design our facade taking in consideration all its restrictions, and those seen when using the FastAI library.

4.2. Library design

Designing a library that covers all the training steps for building a semantic segmentation model is an ambitious project that needs the definition of several components. For example: How is the data collected? Which transforms are applied? Which models are used? How is the validation made? All these questions must be answered separately, as they are independent. Hence, the answers to those questions give rise to new modular components.

Having some modular and independent components may hinder the implementation of our library, because we are going to work with “something like” instead of “something that really is”. However, overcoming this abstraction, we will be able to easily replace the library components if needed. So, if a new library appears, the adaptability of this facade will be as simple as extending our code (never change it).

To this aim, we have to implement several components:

- `Utils`. This component will define some generic methods that will be used in the rest of the components, such as the logging system or the custom exceptions.
- `DatasetManager`. This component will manage the input data. This includes checking whether the input data is correct and usable, creating new mapping functions between the images and the masks, and creating colour maps that can be used to visualise the masks and defining the necessary objects for the data treatment that will be used by the other components.
- `TransformManager`. This component will manage all the data transformations that take part for image processing (for example, resizing all the images) or for data augmentation (such as rotating the twenty-five percent of the images ninety degrees). This component must provide enough functionality to perform all the needed transformations to the images and their masks.
- `ValidationManager`. This component will manage the set up of the validation rules for the models. In other words, solving the next questions:
    - How do we optimise the hyperparameters?
    - Do we use two fixed sets for training and validating or a *kfold* strategy [44]?

  This component must manage the use of the chosen strategy.
- `ModelManager`. This component will manage the construction and training of those defined models.
- `SegmentationManager`. This component will manage the determination of which library is going to be used to build the specified model, and the presentation of the training results.

In Figure 4.5 we show the class diagram that corresponds to the design of our library. In such a diagram, we can see not only the described components, but also their implementations (for instance: for the component `ModelManager`, we will have two implementations: `ModelManagerFastai` and `ModelManagerMMSegmentation`).

Furthermore, we can see a class named `DataLoaderManager` used by the `SegmentationManager` component. This class is going to be used to load the data in the *FastAI* form (we are going to describe it in further sections).

Finally, as we have described before, the `Utils` class is going to be transversal to our code and used by all the other components (we do not link it in the class diagram for the sake of clarity).

In the next subsections, we present the design of all those components with more detail.

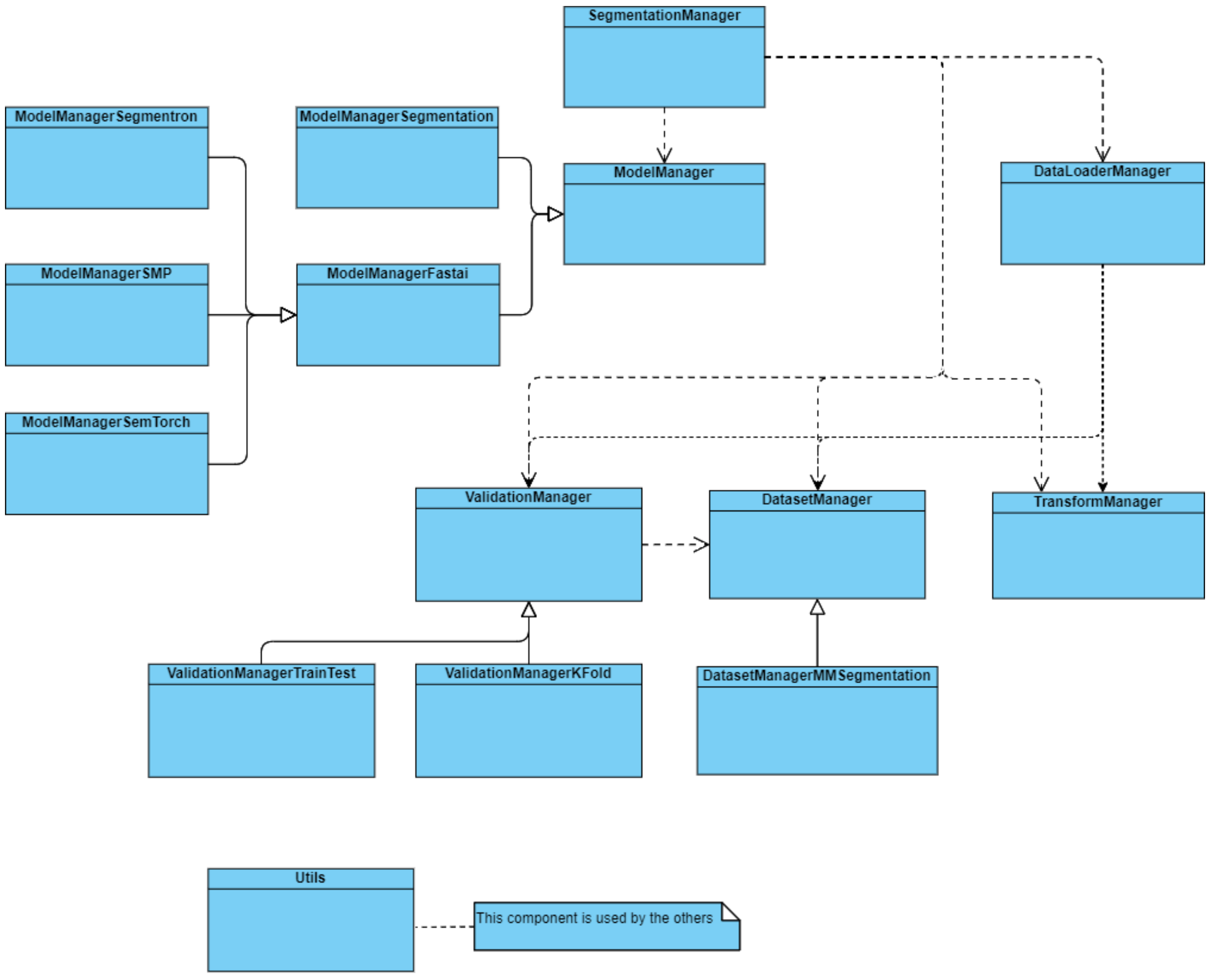


Figura 4.5. The class diagram of the library.

4.2.1. Utils (error management)

The first functionality we need to implement is the one that covers error management. This behaviour is going to be crucial because of our target users, who might not have too much experience on programming, machine learning or using one of the available libraries. Hence, error management must be verbose enough to guide those users when problems arise.

This management, even more, is transversal to all the components of the developed library and fits perfectly with the idea of the *Aspect Oriented Programming paradigm* (henceforth, POA) [45]. The key idea of this paradigm is that there exist some functionality, or components, that are used along the code (for example, a logging component). This implies that all the other components have some code, inside their functions, that needs something defined in this transversal component. The POA paradigm splits those functionalities (transversal and specific components), getting a non mixed code.

In our library, the log messages will be preceded by the text "[LOGGER]", making it clear that they are informative messages about the process that is being executed. Likewise, exception management will be also transparent to the user, who will be able to see which errors are happening. Those messages will be preceded by the text “[WARNING]” if the exception is not disruptive (it does not end the execution of the program) or “[ERROR]” if it is. Furthermore, the exception manager will show which function is raising the warning or error.

4.2.2. DatasetManager

The `DatasetManager` will be the component that manages the input data. The input dataset will be saved in a local directory which will be referred as the “root directory”. This directory will have all the files needed to train our models.

This directory will be a directory tree that contains, at least, two folders and a *json* file. Those folders will contain respectively the images and masks of the dataset; and the *json* file (named *codes*) will define the colour codes used by the masks to identify the different categories of the images.

The `DatasetManager` aims to check structure issues: whether the structure provided is correct, whether the relationship between images and mask is bijective (each image has one -and only one- mask and *vice versa*) and whether the masks have the correct format.

Additionally, it will define useful methods to get the data information: for instance, a list of all the images included in the dataset, or the way to build a `Dataset` object that will be mandatory to build the *MMSegmentation* models. Since the behaviour of the `DatasetManager` depends on the concrete library that will be used for building a model, `DatasetManager` will be an abstract class inherited by its specific implementations.

In particular, the common part for each class that inherits `DatasetManager` will be the constructor, that will get the following parameters:

- Root path: the directory where all the structure exists.
- Prefixes and suffixes for the images and masks.
- If it is needed to remove those prefixes and suffixes.
- The maps between the images and masks.
- If it is needed to check the bijective relationship between images and masks.
- What to do if an image or mask does not meet this relationship (used only if the check is needed).
- If it is needed to transform all the masks to a correct format.
- The default class used to map the identifiers without an assigned class after the transformation (used only if the transform is needed).

Therefore, the `DatasetManager` constructor will follow the next steps:

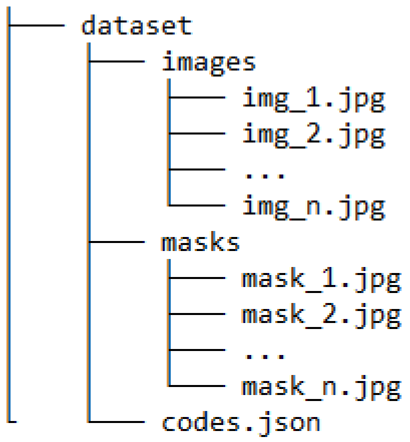
1. Check whether the required directories (images and masks) and the file (codes) exist in the root directory. This process will raise an exception if any of them does not exist.
2. Allocate the suffixes for the files (the images and the masks) if they are not supplied. This allocation will use the mode. For instance, let us suppose that we have a hundred files: twenty of them are *png* and eighty of them are *jpg*; if we do not provide the suffix, then the facade will suppose that the suffix is the mode between all those file formats, in this case *jpg*.
3. Define the mapping function between images and masks. We should note that those functions are two: one for the image to mask relation and the other for the mask to the image relation; those functions are needed to check the bijectionallity between images and masks. Additionally, image to mask relation will be used further in the process of data loading. If the mapping is not supplied, it will be calculated by default using the prefixes and

suffixes information: as all the images and masks have a prefix, a code and a suffix; the mapping between them is the function that replaces the prefixes and suffixes of the images with the prefixes and suffixes of the masks, and *vice versa*, and keeps the same code.

4. Rename all the files by deleting their prefixes if the prefix elimination is requested.
5. Check that the mapping functions are bijective, if the checking is requested. In other words: it will check that every image has one (and only one) mask associated and every mask has one (and only one) image associated. In this scenario, it will not need to check the "and only one" requisite, since there can not exist two files with the same name and format in the same directory. If this relationship is not bijective, it will apply the functions supplied by the user (one for the images and the other for the masks). If those functions are not supplied, it will apply the default function that will move the wrong files into a new directory named *map_fails*.
6. Check the state of the masks. If they need to be converted, it will convert all the masks to new masks that will really be an identifier map. If a new identifier is found on this conversion, it will be mapped into the background class (the first one by default: 0). If the conversion is not requested, it will check if the masks have the correct format to use them, showing an error message if they do not.

Once all the previous steps are completed, a new `DatasetManager` object will be defined with a correct structure. This correct structure is shown in Figure 4.6.

Nevertheless, for some libraries it will be necessary to define new subcomponents that inherit this base component and define the way to build a `Dataset`. In our scenario, *FastAI* does not need it, but *MMSegmentation* does, so we will implement the `DatasetManagerMMSegmentation` class.



```
├── dataset
│   ├── images
│   │   ├── img_1.jpg
│   │   ├── img_2.jpg
│   │   ├── ...
│   │   └── img_n.jpg
│   ├── masks
│   │   ├── mask_1.jpg
│   │   ├── mask_2.jpg
│   │   ├── ...
│   │   └── mask_n.jpg
└   └── codes.json
```

Figure 4.6. The correct structure that is checked by the `DatasetManager`.

4.2.3. TransformManager

One of the main drawbacks for building Deep Learning models is that they need a lot of data to be properly trained. In other words, if we want to train a model to segment the cats in an image, we will need a lot of cat images (and their masks too) to teach the model to segment each one of those cats.

Nevertheless, for a computer, those images are nothing more than n-dimensional arrays of real numbers. Hence, modification of those images (such as a rotation, or changes in colour) will completely change its representation. Behind this idea lies *data augmentation* [46].

Data augmentation is based on the application of several transformations (such as flips, smudges, cuts, resizes…) to an image in such a way that the resulting image must be different enough to be considered as a new image to train the model. In the context of semantic segmentation, this new image will be already labelled (its mask can be constructed by applying the same transformations to the mask).

In general, those image transformations are randomly applied on the fly to the images while training the model; however, those transformations can also be applied to normalise or resize the images to all the images of the dataset.

Both kinds of transformations will be implemented in our library by means of a `TransformManager` component that will store the transformations that will be applied to the images.

There exist several ways to implement those transformations, but we are going to use the *albumentations* library [15]. Using such a library, we will be able to define the transformations, including composition between them (for instance, appling a crop and then a padding). Furthermore, each one of those defined transformations will happen with a certain probability.

4.2.4. ValidationManager

The model validation is the process in which the model is tested and the model results are compared with the expected ones. The performance of the model is completely related to the problem that is tackled and it can be measured with several metrics. The available metrics in our library will be explained in the explanation about the evaluation process.

In order to validate our models, we are going to split the data into some groups (training, validation and testing). Then, we will apply some validation strategies such as *train-val-test* validation[3] or *k fold* validation[4].

The strategy will be implemented and applied using the `ValidationManager`.

4.2.5. DataLoaderManager

As we have mentioned before, some libraries will return a `Learner` object from *FastAI*. This object is common between all of these libraries, so its use can be generalised.

In order to use those `Learner` objects while training the models, we will need to load the data from the disk and provide it to the model. That work is going to be covered by this component, that is going to be used by the `SegmentationManager` if the library needed to build the requested model is one of those that use *FastAI*. In the case of *MMSegmentation*, the data will be loaded directly, so additional components are needed.

4.2.6. ModelManager

Once the data is loaded and splitted, we will need to define how to build, train and save the models. This behaviour is provided by this component.

As we have mentioned in Section 1, image segmentation models are mainly defined by an architecture and a backbone that may also have an associated weights file.

[3] Three sets. Training is used to train the model, validation to validate it and test to test it.

[4] *K* sets are created. Then, the model is trained with K - 1 sets and validated using the last one. The test set is also used to test the model at the end of the training of all the folds.

In this scenario, we are going to put together all this information (that we will denote as the *metamodel*) inside a new component called `ModelManager`. This component will define several functions needed to train the model (such as `fit`, `lr_find`...), that will be abstract and it will be implemented by all the subclasses that will inherit from it (each of them will represent a different library that will be included in our facade).

Doing so, our facade will not be associated with one library, but it will be extensible to new ways to train the models (using inheritance).

4.2.7. SegmentationManager

The `ModelManager` components (itself and its subclasses) are able to build a model knowing its structure (that is, knowing the metamodel). Nevertheless, the aim of the facade is to train several models in a simple and centralised way. For instance, we might want to compare the results of several models trained using the *UNet* architecture with different *ResNet* backbones (18, 34, 50, 101 and 152).

This means that we have to train several models defined using different `ModelManager` objects. Additionally, if we are using a *kfold* strategy with *k* equals to five, we will be training six models for each backbone (the five folds and the final model); knowing that there exist five *ResNet* backbones (in their simplest version), we have a total of thirty models to train. Since dealing with such an amount of objects might be cumbersome, our task here is to simplify this process into a single component that can manage the training process of all those models. This component will be the `SegmentationManager`.

This component will determine what `ModelManager` component must be used to train each model requested by the user. In other words, what library is necessary to build the defined metamodel. Such a component will be very verbose, indicating what is happening in each training step to the user with messages like: "training the fold *k* of the model *m*".

After training all the models, one of the most important steps is the study of the results. Given this fact, this component will be capable of managing the training log to show it to the user in a friendly way.

4.3. Library implementation

Once all the components are designed, it is time to implement them. To do so, and for testing the code, we are going to use *Python* with *Jupyter notebooks* in a server with a GPU with 10GB RAM[5].

All the code mentioned during this section can be found in the following github repository: https://github.com/ruescog/SegmentationManager.

4.3.1. Logging information, alerts and errors

As we have mentioned before, the logging management is going to be tackled using the AOP paradigm. Idyllically, AOP paradigm allows us to uncouple the transversal methods to the other methods. This is, in real life, another file where the AOP methods are implemented. Nevertheless, to

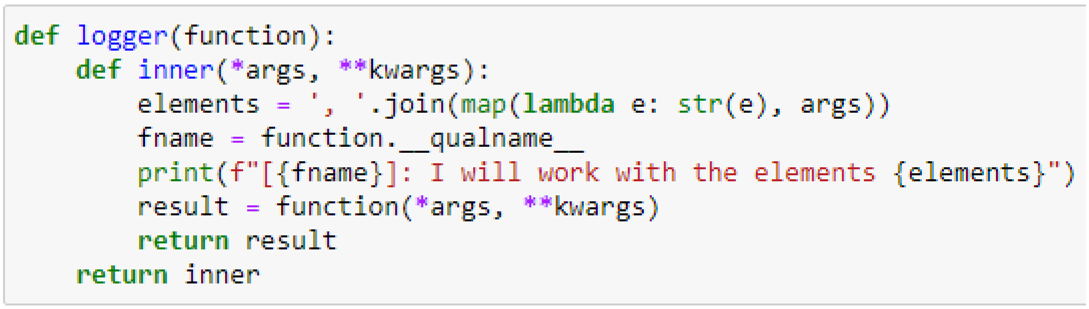
[5] Random-access Memory [47]

achieve this, we would need to use another library similar to *JAspect [48]* for *Java*, but there does not exist anything similar in *Python*[6].

Given this fact, we need to use a custom implementation. This approximation will not be purely AOP, but will separe (as much as possible) the transversal code (the logging) from the other components. To do so, we are going to use *Python decorators* [51].

A decorator is a function that has another function as input parameter and adds to it some new functionality.

Let us suppose that we have several functions that do the basic mathematical operations (such as additions, subtraction…). We want to show a message every time that one of those functions is called, showing "I am doing this operation with those elements". This message, in the AOP paradigm, should not be added to the function implementation, because it is not necessary to its correct execution. Instead, this functionality must be added using a decorator as it is shown in Figure 4.7.

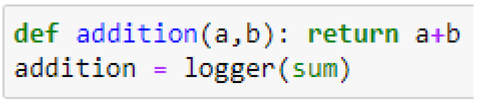


```
def logger(function):
    def inner(*args, **kwargs):
        elements = ', '.join(map(lambda e: str(e), args))
        fname = function.__qualname__
        print(f"[{fname}]: I will work with the elements {elements}")
        result = function(*args, **kwargs)
        return result
    return inner
```

Figure 4.7. Defining the decorator.

Without explaining it in detail, we have defined a decorator to add the logging behaviour to some functions. As we can see, the last two lines of the `inner` function call the original called function and return its result. The other lines are used for logging. In this scenario, we want a message like: "[addition]: I will work with the elements 1, 2". The name of the function, *addition*, is going to be obtained with the *__qualname__* attribute.

Once this decorator is defined, we must call it with those functions that we want to decorate, like we show in Figure 4.8.

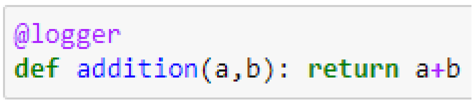


```
def addition(a,b): return a+b
addition = logger(sum)
```

Figure 4.8. Using the decorator.

Doing this with all the functions that we want to decorate with the new decorator can make our code dirty (and we need to remember that if we are doing this is just for the opposite goal), so we will use the syntactic sugar provided by *Python*, adding the key symbol @ and the name of the decorator over the function that we are going to decorate, as it is shown in Figure 4.9.

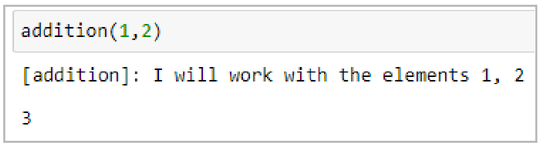
[6] When we say that "there does not exist anything similar in *Python*" we mean that those libraries are not developed enough or they are out-of-date and can bring more issues than solutions. Those libraries are *springpython [49]* and *lemonframework* [50].



```
@logger
def addition(a,b): return a+b
```

Figure 4.9. Using the decorator using the syntactic sugar of *Python.*

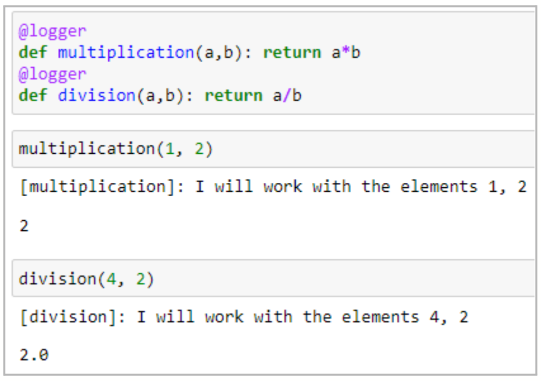
Finally, if we call the `addition` function, we will get the output shown in Figure 4.10.



```
addition(1,2)

[addition]: I will work with the elements 1, 2

3
```

Figure 4.10. Output for `addition` function.

Additionally, we can also apply this decorator to other functions as shown in Figure 4.11.



```
@logger
def multiplication(a,b): return a*b
@logger
def division(a,b): return a/b

multiplication(1, 2)

[multiplication]: I will work with the elements 1, 2

2

division(4, 2)

[division]: I will work with the elements 4, 2

2.0
```

Figure 4.11. Using the decorator to manage the logging of the operations.

As we have suggested, this AOP solution is not pure (it is partially mixed with the code), but its code is not inside the function either. It is a trade-off that we consider enough for our scenario.

For our real decorators (the ones used for the facade components) we will include some additional functionality. For instance, it will be possible to determine when the message is going to be logged (before or after the function call). Furthermore, the message may have the names of the parameters used to call the function in capital letters, and their values will be logged replacing those positions in the log message.

To manage the exceptions, another decorator will be defined. This new decorator will capture the exceptions and show a message of *WARNING* or *ERROR* depending on the exception severity. When a message has the category of *ERROR*, the exception will be thrown again and, potentially, the program execution could be stopped.

An example of one of those decorators is shown in Figure 4.12. In this example, we can observe that three decorators are going to be used sequentially. First of all, the decorator that is next to the function (the one with the message "The best lr value is VALUE") is going to add that functionality to the `lr_find` function. We should note that the *VALUE* text (in capital letters) is going to be replaced with the value returned by that function after the execution (with the *when = after* default behaviour). Subsequently, the decorator that is just above will encapsulate the previous one and the function, adding a message logged *before* the execution of the function. Finally, if this function throws a `FileNotFound`, it is going to be catched by the *excepter* decorator showing an *ERROR* message and stopping the execution.

```
@AOP.excepter(FileNotFoundError)
@AOP.logger("Searching the best lr.", when = "before")
@AOP.logger("The best lr value is VALUE")
def lr_find(self):
```

Figure 4.12. An example of a decorator.

4.3.2. DatasetManager

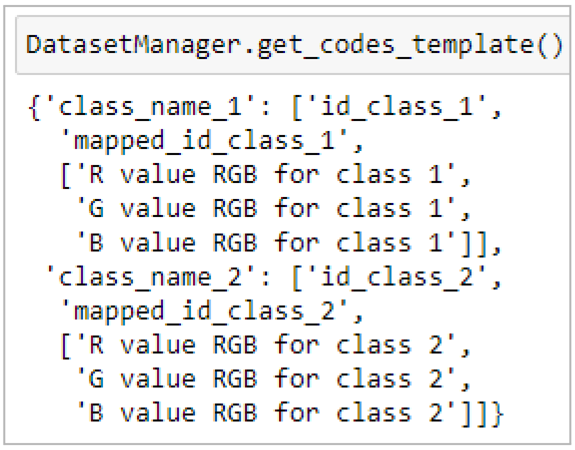
The implementation for this component has followed the steps presented in the design, but we should point out some details considered in the implementation.

First of all, we note that the masks conversion method is a bottleneck at the time to build the dataset. We were forced to improve its efficiency using *numpy* [52] and its boolean mask indexing methods, as shown partially in Figure 4.13, improving the transformations time to one mask each eight milliseconds. We do not discard that this method will need to be improved again.

```
# convert all the masks using the codes file
for mask in masks:
    # loads the image
    mask_data = Image.open(mask)
    converted_mask = mask_data.convert("P")

    # creates a np array with the image data
    x = np.array(converted_mask)

    # maps it
    for value in codes.values():
        class_value = value[0]
        map_value = value[1]
        x[x == map_value] = class_value

    # looks for noise in the conversion
    real_classes = np.unique(x)
    if len(real_classes) > len(codes):
        for real_class in real_classes:
            if real_class not in codes_ids:
                x[x == real_class] = background_class
                noise += 1

    # saves the result
    converted_mask = Image.fromarray(x)
    converted_mask.save(mask)
```

Figure 4.13. A fragment of the implementation of the method `__convert_masks__`.

Additionally to the functionality described in the design phase, we have defined another function called `get_codes_template`. This function will be useful to obtain a template of the *codes* file, which will be a custom file written by the user and it needs to follow some structural rules (it must be a *json* with the class names as keys, and the id of the class, its mapped id and it colour map as values). This template is shown in Figure 4.14.


```
DatasetManager.get_codes_template()
```

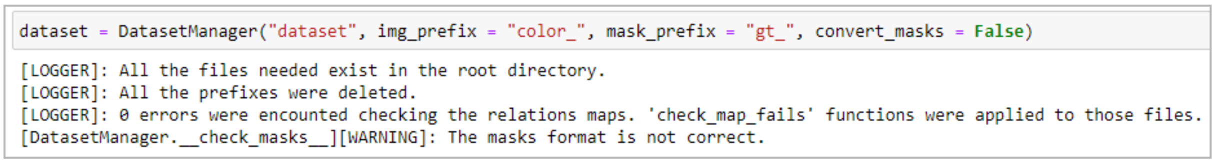
```
{'class_name_1': ['id_class_1',
  'mapped_id_class_1',
  ['R value RGB for class 1',
   'G value RGB for class 1',
   'B value RGB for class 1']],
 'class_name_2': ['id_class_2',
  'mapped_id_class_2',
  ['R value RGB for class 2',
   'G value RGB for class 2',
   'B value RGB for class 2']]}
```

Figure 4.14. Template returned by the `get_codes_template` function.

On a separate topic, we will need a name to define and register the *MMSegmentration* dataset in the loaded datasets, which is a list with all the classes that inherits from the `CustomDataset` abstract class in *MMSegmentation*. Those classes define new datasets and how to work with them, but they need to be registered in the datasets list. This name must be unique, so if it is not supplied, or the name supplied is not unique, the dataset will be registered with a unique identifier obtained using the actual time (and it will return that name to use it).

It will not be necessary to build a `DatasetManagerFastai` object because *FastAI* models can be built in many ways and, the one that we will use, does not use those objects.

We show an example of a dataset construction in Figure 4.15.


```
dataset = DatasetManager("dataset", img_prefix = "color_", mask_prefix = "gt_", convert_masks = False)
```

```
[LOGGER]: All the files needed exist in the root directory.
[LOGGER]: All the prefixes were deleted.
[LOGGER]: 0 errors were encounted checking the relations maps. 'check_map_fails' functions were applied to those files.
[DatasetManager.__check_masks__][WARNING]: The masks format is not correct.
```

Figure 4.15. An instance of a dataset construction

As we can observe in Figure 4.15, the dataset structure is correct, all the prefixes have been deleted (we have requested it), no error has been found between images and their masks and there exists at least one mask with an incorrect format. This message is shown since we have requested to not convert the labels and, when this happens, it checks if all the masks have the correct format.

4.3.3. TransformManager

In order to define the transformations, we are going to use the *albumentations* library which can be used directly with the models that use `Learner`s from *FastAI*. Nevertheless, it cannot be used for the models of *MMSegmentation*, since this library uses another language for its transformations.

However, there exists the possibility to define new transformations in *MMSegmentation*, so the way to apply *albumentation* transforms to a *MMSegmentation* model is to define a new transformation that will apply the composition of all the transformations that have been defined with *albumentations*. We clarify this tongue twister in Figure 4.16.

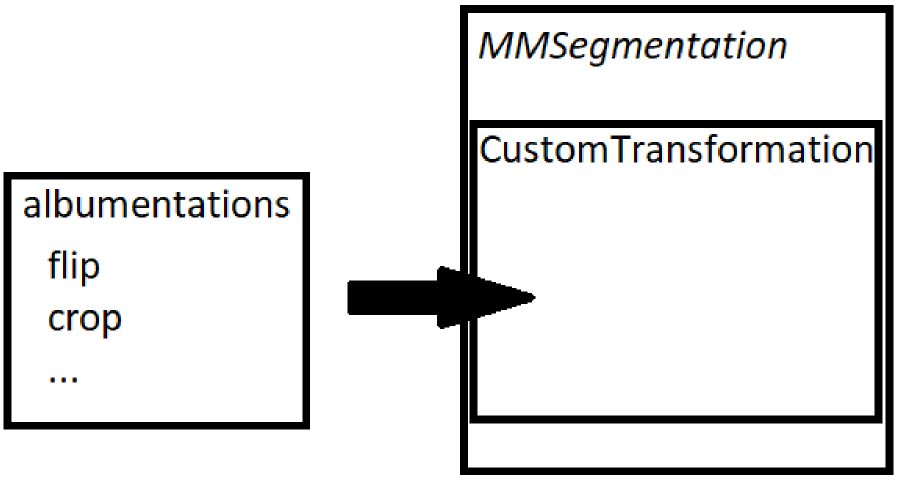


Figura 4.16. Using *albumentation* in *MMSegmentation*.

Additionally, we should note that the data augmentation process is managed in memory. Given this fact, all the sintetic images created in this way are not saved anywhere and they can differ from one execution to another.

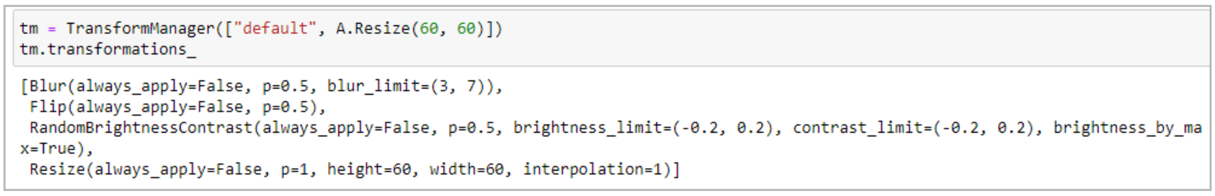
As we have described, the management of the transformations will depend on the library which have been used to construct the model, so it has sense to define two subclasses and implement the abstract method `get_pipeline` defined in `TransformationManager`, those two classes are `TransformManagerFastai` and `TransformManagerMMSegmentation`.

Whereas in `TransformManagerFastai` the method `get_pipeline` will just compose the selected transformations, in `TransformManagerMMSegmentation`, additionally to the composition, the result will be encapsulated into a *MMSegmentation* class that will be registered in the transformation module (likewise done with the `DatasetManager`).

In both implementations, all the transformations must be applied to both the images and their masks. Additionally, it will be also needed to implement another method that can build a specific class object (for *FastAI* or *MMSegmentation)* providing the generic `TransformManager`.

In order to simplify their use, we have defined a couple of default transformations that will be applied if the user requests it by using the keyword *default*. Those default transformations are a blur transformation, a flip, and a change of brightness contrast, using the objects `Blur`, `Flip` and `RandomBrightnessContrast` from the albumentations library respectively. All of them are applied with a probability of fifty percent.

If the user wants to use those default transformations and some additional ones, they will be able to use the *default* keyword to define the `TransformManager` object. For instance, if the user uses the following list of transformations to build a `TransformManager` object, it will get the result of Figure 4.17.


```
tm = TransformManager(["default", A.Resize(60, 60)])
tm.transformations_
```

```
[Blur(always_apply=False, p=0.5, blur_limit=(3, 7)),
 Flip(always_apply=False, p=0.5),
 RandomBrightnessContrast(always_apply=False, p=0.5, brightness_limit=(-0.2, 0.2), contrast_limit=(-0.2, 0.2), brightness_by_ma
x=True),
 Resize(always_apply=False, p=1, height=60, width=60, interpolation=1)]
```

Figura 4.17. An instance of `TransformManager` construction.

4.3.4. ValidationManager

As suggested in the design phase, the available strategies defined inheriting the `ValidationManager` abstract component will be *TrainTest* and *KFold*. Such strategies are going to be implemented using the `splits` abstract method from the `ValidationManager` class.

This approach allows us to inherit the `ValidationManager` abstract class in other particular classes like `ValidationManagerTrainTest` or `ValidationManagerKFold` to implement these strategies. Furthermore, nothing prevents us from defining more specific classes in the future to implement other splitting strategies such as *one-out* or *repeatedKFold*.

The `splits` method will create a new folder inside the root directory called *splits* that will have a new folder to define each of the training sets and, finally, a file with the names of the testing images.

Each of those folders will have two files: one with the names of the training images and another with the names of the validation images. Those names will be the basename (the file name without the pathing nor the format) of those images. Such a method will create the training, validation and testing structure and it will return a dictionary object where the built structure will be represented.

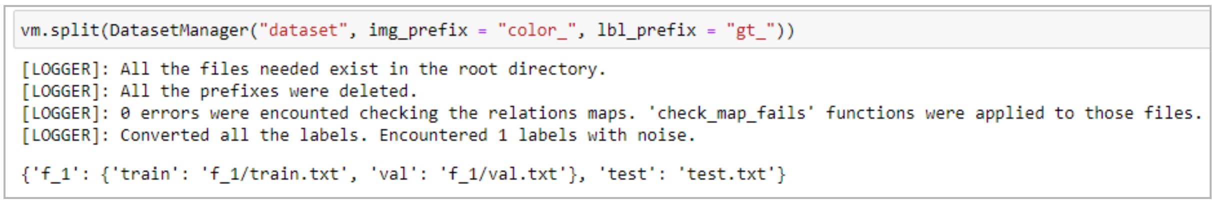
In Figure 4.18 we show an example of the dataset structure after applying one of those validation techniques (we should note that the result of applying *TrainTest* strategy is just one folder of training – since the model will be trained just once –, whereas the *KFold* strategy will create *K* folders).

```
├── dataset
│   ├── images
│   │   ├── img_1.jpg
│   │   ├── img_2.jpg
│   │   ├── ...
│   │   └── img_n.jpg
│   ├── masks
│   │   ├── mask_1.jpg
│   │   ├── mask_2.jpg
│   │   ├── ...
│   │   └── mask_n.jpg
│   ├── splits
│   │   ├── f1
│   │   │   ├── train.txt
│   │   │   └── val.txt
│   │   ├── f2
│   │   │   ├── train.txt
│   │   │   └── val.txt
│   │   ├── ...
│   │   ├── f_k
│   │   │   ├── train.txt
│   │   │   └── val.txt
│   │   └── test.txt
└   └── codes.json
```

Figure 4.18. Dataset structure after applying the *KFold* validation strategy.

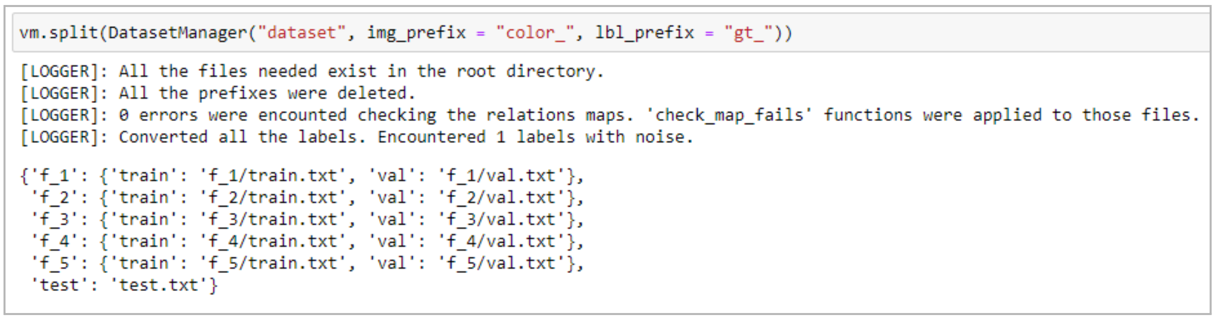
Additionally, we have noticed that there are some common parts in the implementation of those strategies. For instance, in both of them it is checked that the relative size of each set is one and it is normalised, the sets indexes are calculated from the total size of the dataset, the *slices* are created using those sets indexes, and the structure is generated. In order to avoid code duplication, all this functionality has been implemented as auxiliary functions in the `ValidationManager` superclass.

In Figure 4.19 and 4.20 we show the result of applying those strategies.



```
vm.split(DatasetManager("dataset", img_prefix = "color_", lbl_prefix = "gt_"))

[LOGGER]: All the files needed exist in the root directory.
[LOGGER]: All the prefixes were deleted.
[LOGGER]: 0 errors were encounted checking the relations maps. 'check_map_fails' functions were applied to those files.
[LOGGER]: Converted all the labels. Encountered 1 labels with noise.

{'f_1': {'train': 'f_1/train.txt', 'val': 'f_1/val.txt'}, 'test': 'test.txt'}
```

Figure 4.19. An instance of applying the *traintest* strategy.



```
vm.split(DatasetManager("dataset", img_prefix = "color_", lbl_prefix = "gt_"))

[LOGGER]: All the files needed exist in the root directory.
[LOGGER]: All the prefixes were deleted.
[LOGGER]: 0 errors were encounted checking the relations maps. 'check_map_fails' functions were applied to those files.
[LOGGER]: Converted all the labels. Encountered 1 labels with noise.

{'f_1': {'train': 'f_1/train.txt', 'val': 'f_1/val.txt'},
 'f_2': {'train': 'f_2/train.txt', 'val': 'f_2/val.txt'},
 'f_3': {'train': 'f_3/train.txt', 'val': 'f_3/val.txt'},
 'f_4': {'train': 'f_4/train.txt', 'val': 'f_4/val.txt'},
 'f_5': {'train': 'f_5/train.txt', 'val': 'f_5/val.txt'},
 'test': 'test.txt'}
```

Figure 4.20. An instance of applying the *kfold* strategy with a *k* value of five.

4.3.5. DataLoaderManager

This component, which manages how the data is loaded to train the models, will use the result of the previous component (`ValidationManager`) in order to define the objects needed to build the models.

As `splits` method returns a dictionary with the structure that defines the path to each set in the *splits* folder, picking up those images to build the `DataBlock`s objects is trivial. We just need to:

1. Get the classes names defined in the `DatasetManager`.
2. Get the training and validating images defined inside the *splits* folder.
3. Get the masks associated with the previous images using the bijective function defined in the `DatasetManager`.
4. Load the training and validation sets.
5. Define the way to apply the requested transformations to the images.

Additionally, another set will be created exclusively with the testing data to evaluate the model. The construction of the `DataBlock` used to test the model is very similar to the previous ones. The only difference is that all the data will be loaded indicating that the whole dataset will be used for validation. In other words, this set will not have any images for training, but all of them will be used for testing.

Once all the `DataBlock`s are defined, their transformation into `DataLoader`s is conducted by using the `dataloaders` method: it is only needed to indicate how many images are going to be used at the same time during the process[7].

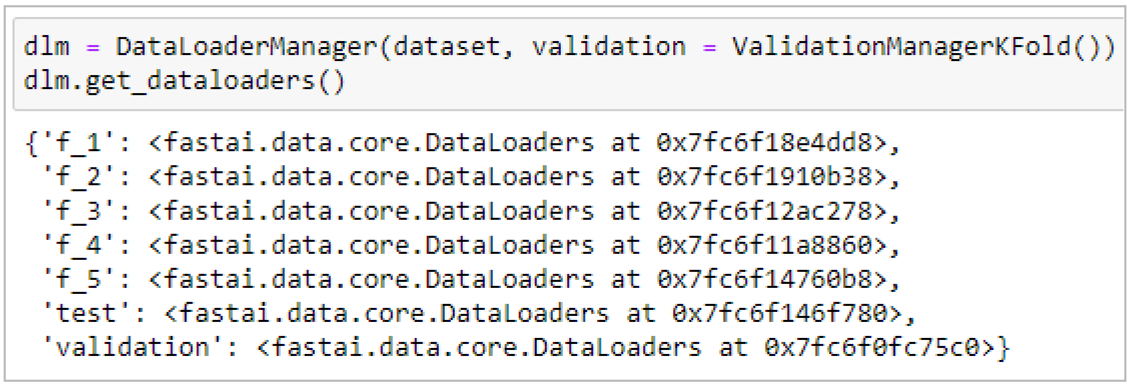
[7] Since the GPU is being used in the training phase, matricial operations can be done in a more efficient way. Additionally, due to the physical architecture of the GPUs, this process is absurdly parallelable, so the model can be trained with multiple images at the same time. The amount of images that are parallel shown by the model defines the *batch size*.

As we have mentioned, this component uses the results from the previous three components, so an instance of them is needed to define this new one (this component boxes the previous ones). In other words: a DatasetManager, a TransformsManagerFastai (from *FastAI*, because is the library that uses DataBlocks and DataLoaders) and a ValidationManager are going to be used to define the DataLoaderManager.

During this component implementation, we have noticed that it will be interesting to create an additional set to retrain the model when using the *kfold* strategy. In other words, the dataset is divided into several folds, each of them is trained and tested and, finally, the model is trained using all the folds mixed (this one is the new set, called *validation*) and it is tested with the testing set. This final model is the one that is going to be compared with the others in order to determine which model obtains the best results.

Defining this new set is simple, we just load all training, validation and testing data and we do a *traintest* split training.

Since the previous component returns a path dictionary to each set location, this component will return a dictionary with each of the DataLoaders defined, as shown in Figure 4.21.



```
dlm = DataLoaderManager(dataset, validation = ValidationManagerKFold())
dlm.get_dataloaders()
```

```
{'f_1': <fastai.data.core.DataLoaders at 0x7fc6f18e4dd8>,
 'f_2': <fastai.data.core.DataLoaders at 0x7fc6f1910b38>,
 'f_3': <fastai.data.core.DataLoaders at 0x7fc6f12ac278>,
 'f_4': <fastai.data.core.DataLoaders at 0x7fc6f11a8860>,
 'f_5': <fastai.data.core.DataLoaders at 0x7fc6f14760b8>,
 'test': <fastai.data.core.DataLoaders at 0x7fc6f146f780>,
 'validation': <fastai.data.core.DataLoaders at 0x7fc6f0fc75c0>}
```

Figure 4.21. An instance of using the DataLoaderManager component.

Each dictionary value is a DataLoader that is going to be used to train a model.

Finally, it is important to note that this component is the first one that is going to use the GPU if it exists, because the DataLoader objects are built there.

4.3.6. ModelManager

We have defined this component in order to implement how to build, train and save a model.

Given the fact that some libraries (such as *SegmenTron*, *smp* and *SemTorch*) use the *FastAI* training procedure, we believe that is convenient to group them in a single component that will define several of the abstract methods (not all of them, but the ones related with the training phase). This component will be called ModelManagerFastai.

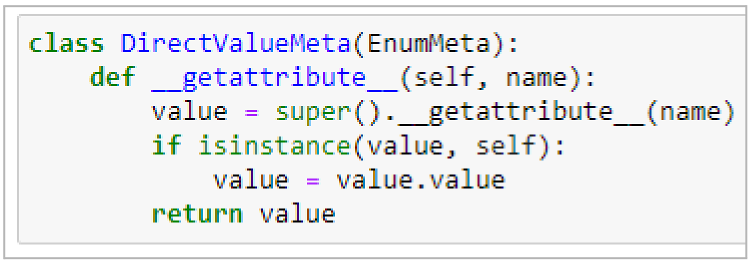
Subsequently, three subcomponents will be defined (inherit from ModelManagerFastai) and will contain the way to build their objects and the valid configuration that gives a model using their constructors: ModelManagerSemtorch, ModelManagerSegmentron and ModelManagerSMP.

We must not forget about `ModelManagerMMSegmentation`, that will also define their methods to build and train the models from *MMSegmentation*, and it will inherit directly from `ModelManager`.

Each one of those previous libraries is capable of constructing a model for some defined architectures and backbones. Those combinations are generally shown in the documentation of each library (or in our Section 3), but sometimes they are not. Since checking manually which libraries can build which combinations (of architecture, backbone and weights) can be a tiresome task, we will provide three enumeration classes that will group together all the possibilities. Each `ModelManager` subclass will define a dictionary named *valid_config* (accessible using the method `get_valid_config`) that will save the possible combinations for that library using those enumeration values. So, seeing what combinations are available for a library is as simple as calling the `get_valid_config` method from its `ModelManager` component implementation.

Those enumeration classes will be useful at the time to construct a model: the architecture, backbone and weights will be provided and it will be only needed to check if that combination is in the *valid_config* dictionary. Furthermore, the enumeration class can be used itself to specify an architecture, backbone or weight. For instance, we can provide a string to define an *UNet* with a *ResNet18* metamodel or we can provide their enumeration values, using *ARCHITECTURE.UNET* architecture with *BACKBONE.RESNET18* backbone and WEIGHT.NONE weights.

For using those enumeration types as their values, we need to change the default behaviour of the *Python* enumeration class, inheriting it as a metaclass[8] and changing what happens when those values are accessed. By default, the returned value is an enumeration object (not a string), so we need to change that enumeration object and unbox the string that it saves inside. We show, in Figure 4.22, the implementation of that custom enumeration class.



```python
class DirectValueMeta(EnumMeta):
    def __getattribute__(self, name):
        value = super().__getattribute__(name)
        if isinstance(value, self):
            value = value.value
        return value
```

Figure 4.22. An enumeration class that returns *string*s when called.

In particular, the distinct component that inherits from `ModelManager` will allow us to:

- Know if an architecture, backbone and weights can be used to build a model using that library.
- Build the model.
- Search for the best learning rate to use in the training phase.
- Train the model using different strategies (such as `fit`, `fit_one_cycle` or `fine_tune`).
- Save the state of the model.
- Load the state of the model.

We need to be careful defining the enumeration classes: such classes that list all the architectures, backbones and weights may contain duplicates because different libraries may call the same architecture, backbone or weight in a different way. Since enumeration classes can not contain

[8] Using metaclasses, the default behaviour of the classes that use them can be changed dynamically.

duplicated enumeration values, we need to solve it in a different way. Namely, we will define a list of synonyms. When a user employs our enumeration class, the value used will contain all the possible values defined for all the libraries (it will be a `tuple`), so detecting which name is really used is a task that must be done by the library itself, with the function `is_buildable`.

This function will desglose recursively all the possible combinations between the 3-tuple (architecture, backbone, weights) and it will choose the first result that is not *None*. For that goal, `Utils` class will define a function called `coalesce` that will return the first not *None* value in a list.

4.3.7. SegmentationManager

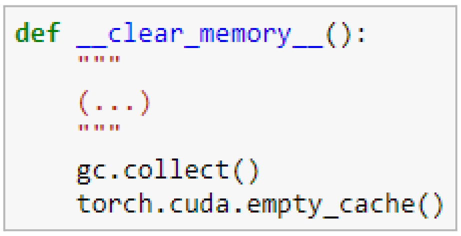
Finally, we will implement the component that will manage the library selection and model construction, training and evaluation for each defined metamodel.

To request the training, the method `multiple_train` will be used, indicating the metamodels:

- An unique identifier for the model.
- The architecture.
- Optionally: the backbone, the weights, the learning rate and the library.

This metamodel will be supplied in a tuple form (for instance: *[("model1", ARCHITECTURES.UNET, BACKBONES.RESNET18]*)). The default values are: for the backbone parameter, *BACKBONE.NONE*; for the learning rate parameter, *best* and, for the library, *None*. Additionally, if the learning rate is *best*, it will be automatically calculated before training and, if the library parameter is not *None*, the facade will be forced to use the supplied library. Hence, the tuple elements define a metamodel, meanwhile the list of metamodels defines the models that are going to be trained.

On another topic, defining multiple models (as we do) has a clear drawback since all those models will be defined using the GPU memory. To avoid the overload of this memory, we will manage manually the *Python garbage collector* and the reserved memory in the GPU. After training each model, we will delete the object that represents it, freeing the GPU. This process is done by the `__clear_memory__` method inside the `SegmentationManager` component, as shown in Figure 4.23. We are forced to do this in this way because otherwise the facade would train models until the GPU collapses.

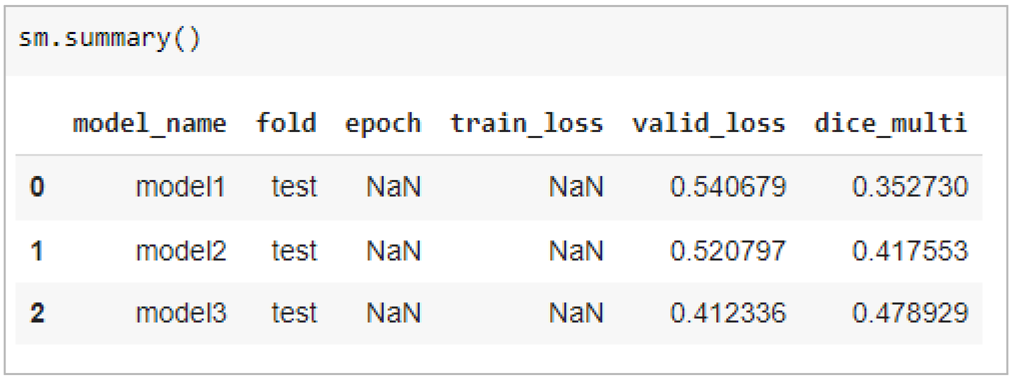


```
def __clear_memory__():
    """
    (...)
    """
    gc.collect()
    torch.cuda.empty_cache()
```

Figure 4.23. Memory management.

In order to log the results of the training, the method `summary` will return a `DataFrame` of the results as we show in Figure 4.24 and `plot_train_valid` will show them in a box plot, comparing all the trained models, as can be shown in Figure 4.25. It is very important to design some filters to those functions, so the user can choose what models (even what folds, epochs or metrics) they want to show the results.

In any case, the result of the `summary` method is a `DataFrame`, so it will be used to create more graphics if the returned by the `plot_train_valid` method is not enough.



```
sm.summary()
```

	model_name	fold	epoch	train_loss	valid_loss	dice_multi
0	model1	test	NaN	NaN	0.540679	0.352730
1	model2	test	NaN	NaN	0.520797	0.417553
2	model3	test	NaN	NaN	0.412336	0.478929

Figure 4.24. Using the function `summary`.

```
sm.plot_train_valid(metric = "dice_multi")
```

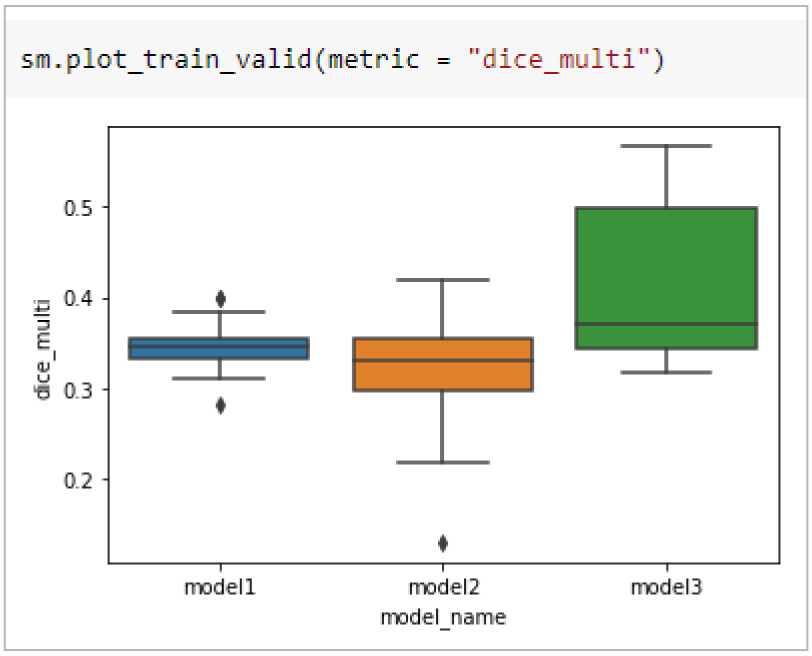


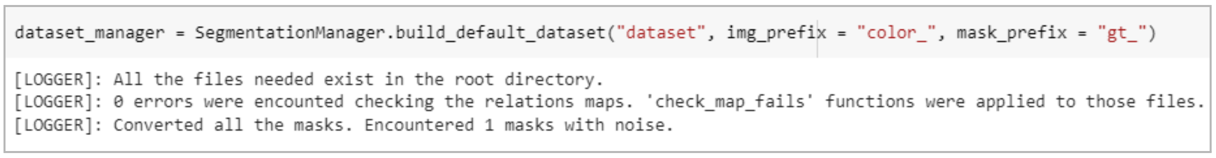
Figure 4.25. Using the function `plot_train_valid`.

4.4. A running example

We have defined how we have implemented our library. Now, we are going to demonstrate how to use the facade in order to train multiple models using just one endpoint and the well-known dataset of grapevines. This demonstration can be also consulted in the repository.

First of all, we need to define the main components in order to define a `SegmentationManager`. Those components, as it was said before, are the `DatasetManager`, the `TransformManager` and the `ValidationManager`. In order to simplify their use, `SegmentationManager` has three methods to build those components by default (`build_default_dataset`, `build_default_transformation` and `build_default_validation`).

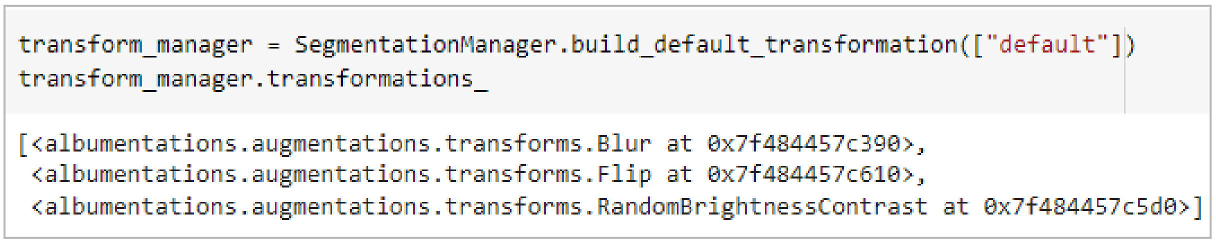
In this example, we are going to use them. We will construct our dataset from a *dataset* directory with some images and their masks. We know that, in this dataset, there is a mask with an incorrect format, but our facade will detect it automatically. See Figure 4.26.



```
dataset_manager = SegmentationManager.build_default_dataset("dataset", img_prefix = "color_", mask_prefix = "gt_")

[LOGGER]: All the files needed exist in the root directory.
[LOGGER]: 0 errors were encounted checking the relations maps. 'check_map_fails' functions were applied to those files.
[LOGGER]: Converted all the masks. Encountered 1 masks with noise.
```

Figure 4.26. Building the `DatasetManager`.

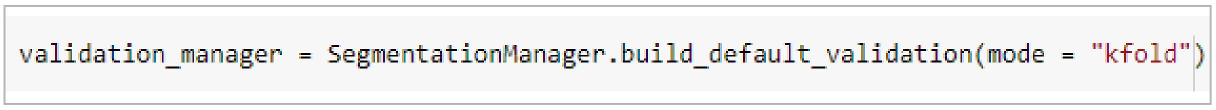
Subsequently, we can define the `TranformManager`. At this point, we should note that all the components that we are defining are the abstract ones. It is not the user, but the facade, who is going to particularise and construct the subclasses needed for the libraries in order to train the requested models. In this example, we are going to use the default transformations. See Figure 4.27.



```
transform_manager = SegmentationManager.build_default_transformation(["default"])
transform_manager.transformations_

[<albumentations.augmentations.transforms.Blur at 0x7f484457c390>,
 <albumentations.augmentations.transforms.Flip at 0x7f484457c610>,
 <albumentations.augmentations.transforms.RandomBrightnessContrast at 0x7f484457c5d0>]
```

Figure 4.27. Building the `TransformManager`.

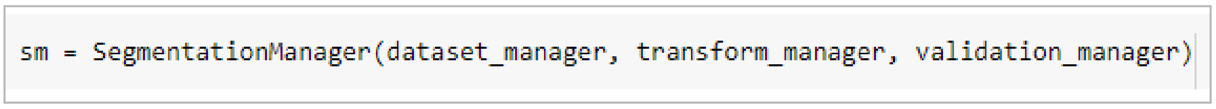
Finally, we need to define the strategy used to validate our models (all of them will share it). In this case, we are going to use a *kfold* validation with a *k*-value of five, which is not the default value (by default, the strategy used is the *traintest* split). See Figure 4.28.



```
validation_manager = SegmentationManager.build_default_validation(mode = "kfold")
```

Figure 4.28. Building the `ValidationManager`.

Once all the components are defined, we can assemble them and construct the `SegmentationManager`, as shown in Figure 4.29.



```
sm = SegmentationManager(dataset_manager, transform_manager, validation_manager)
```

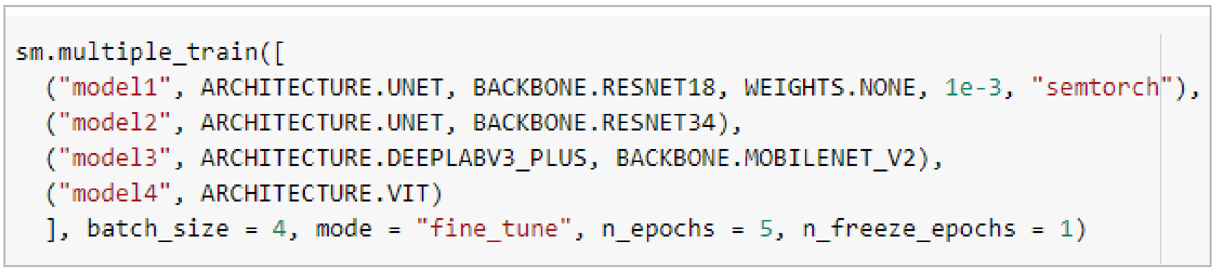
Figure 4.29. Building the `SegmentationManager`.

As we can show, the `ModelManager` and the `DataLoaderManager` component are not used directly by the user. Again, it is the facade which will use them to do the process.

Finally, we will use the defined `SegmentationManager` to train some models, for this example (see the code in Figure 4.30):

- An *UNet* architecture with a *ResNet18* backbone, using *SemTorch*.
- An *UNet* architecture with a *ResNet34* backbone.
- A *DeepLabv3+* architecture using a *Mobilenet_v2* backbone.
- A *VIT* architecture.

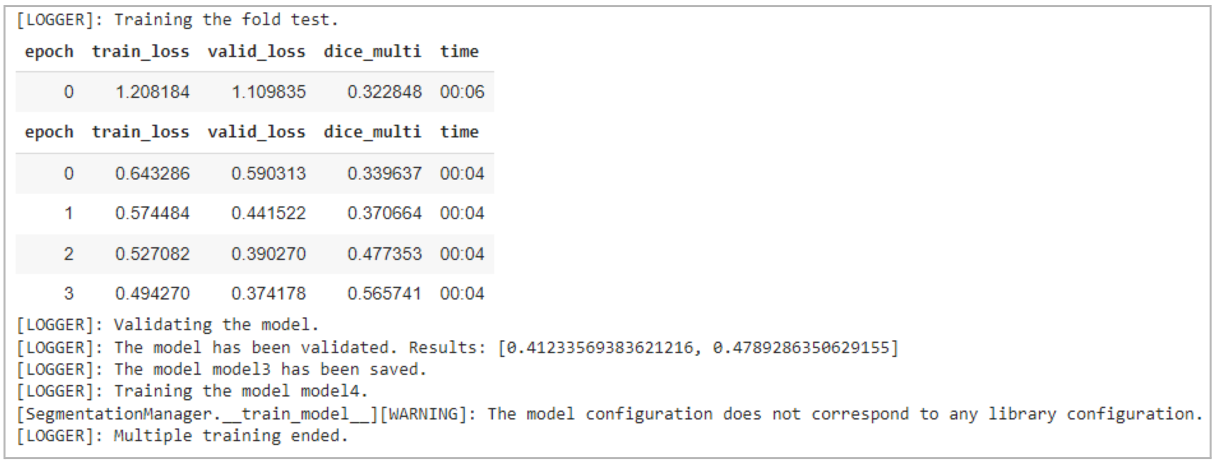
The last one does not correspond to any possible combinations in any libraries, so it should be skipped, as is shown in Figure 4.31.



```
sm.multiple_train([
  ("model1", ARCHITECTURE.UNET, BACKBONE.RESNET18, WEIGHTS.NONE, 1e-3, "semtorch"),
  ("model2", ARCHITECTURE.UNET, BACKBONE.RESNET34),
  ("model3", ARCHITECTURE.DEEPLABV3_PLUS, BACKBONE.MOBILENET_V2),
  ("model4", ARCHITECTURE.VIT)
  ], batch_size = 4, mode = "fine_tune", n_epochs = 5, n_freeze_epochs = 1)
```

Figure 4.30. Training request.

When the previous cell is executed, the process begins and the models are defined, trained and saved. The log is also saved in the *summary* file and it will be consultable using the `summary` and `plot_train_valid` methods. An example of the screen log is shown in Figure 4.31.



```
[LOGGER]: Training the fold test.
 epoch  train_loss  valid_loss  dice_multi  time
     0    1.208184    1.109835    0.322848  00:06
 epoch  train_loss  valid_loss  dice_multi  time
     0    0.643286    0.590313    0.339637  00:04
     1    0.574484    0.441522    0.370664  00:04
     2    0.527082    0.390270    0.477353  00:04
     3    0.494270    0.374178    0.565741  00:04
[LOGGER]: Validating the model.
[LOGGER]: The model has been validated. Results: [0.41233569383621216, 0.4789286350629155]
[LOGGER]: The model model3 has been saved.
[LOGGER]: Training the model model4.
[SegmentationManager.__train_model__][WARNING]: The model configuration does not correspond to any library configuration.
[LOGGER]: Multiple training ended.
```

Figure 4.31. An extraction of the log of the `multiple_train` method execution.

After training, we can see the results, see Figures 4.32 and 4.33.

```
sm.plot_train_valid(metric = "dice_multi")
```

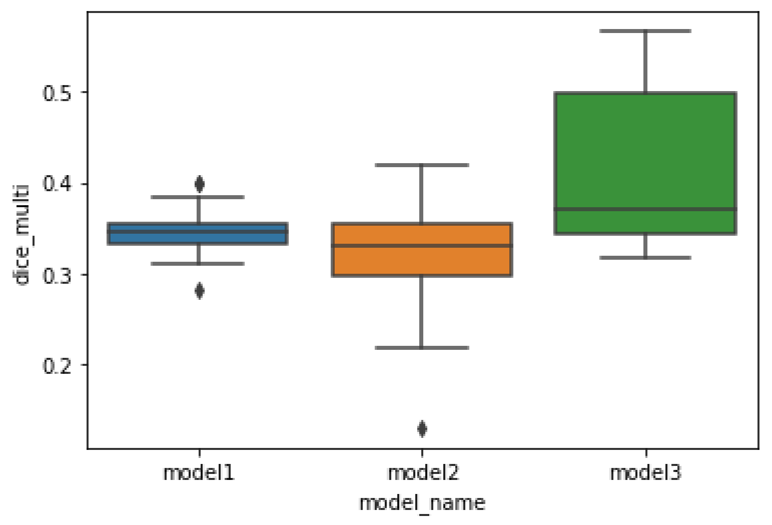
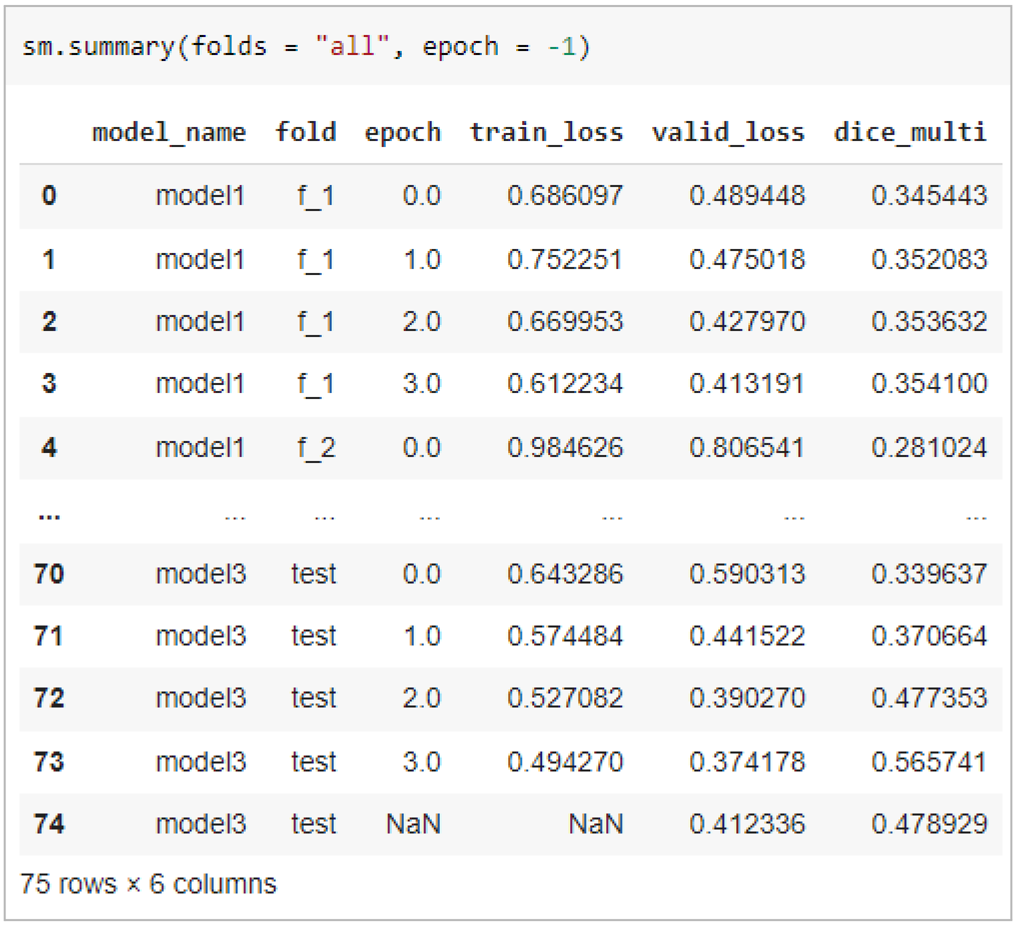


Figure 4.32. Using the `plot_train_valid` method with filters (plotting the *dice_multi* metric).


```
sm.summary(folds = "all", epoch = -1)
```

	model_name	fold	epoch	train_loss	valid_loss	dice_multi
0	model1	f_1	0.0	0.686097	0.489448	0.345443
1	model1	f_1	1.0	0.752251	0.475018	0.352083
2	model1	f_1	2.0	0.669953	0.427970	0.353632
3	model1	f_1	3.0	0.612234	0.413191	0.354100
4	model1	f_2	0.0	0.984626	0.806541	0.281024
...	...	...	...	...	...	...
70	model3	test	0.0	0.643286	0.590313	0.339637
71	model3	test	1.0	0.574484	0.441522	0.370664
72	model3	test	2.0	0.527082	0.390270	0.477353
73	model3	test	3.0	0.494270	0.374178	0.565741
74	model3	test	NaN	NaN	0.412336	0.478929

75 rows × 6 columns

Figure 4.33. Using the `summary` method with filters (all the folds and epochs).

4.5. Future work

Although we have implemented the functionality needed to work with our facade, we know that there are some details that remain as future work. We detail them in the following list:

- Future work for the logging management component:
  - Add more messages to manage more exceptions and improve the transparency of the facade.
  - Allow the log messages to be saved in a log file instead of being shown in the execution cell.
  - Allow to deactivate the log.
- Future work for the `DatasetManager` component:
  - Allow to use multiple datasets from the same `DatasetManager`.
  - Parallelise the process of mask transformation.
  - Allow the masks multiple mapping. In other words, allowing multiple colour codes could be assigned to the same class.
- Future work for the `TransformManager` component:
  - Allow to save the transformed images in the disk.
- Future work for the `ValidationManager` component:
  - Implement the *one-out* strategy or others.
- Future work for the `DataLoaderManager` component:
  - Calculates the *batch_size* automatically.

    - Abstract this component into a `PreprocessingManager` that will group together all the preprocessing steps (such as `DataLoader`s building for *FastAI*) into a new generic class. In this scenario, the *MMSegmentation* version of this component `PreprocessingManagerMMSegmentation` will not do nothing, but the `SegmentationManager` will not be needed to access directly to either the `DatasetManager`, `TransformManager` nor `ValidationManager`.
- Future work for the `ModelManager` component:
    - Make the *valid_config* dictionary dynamically, instead of defining it statically and manually, as is done at this moment.
    - Keep finding the way to transform an *EncoderDecoder* from *MMSegmentation* into a *Learner* from *FastAI*, in order to simplify and reduce the facade components.
- Future work for the `SegmentationManager` component:
    - Allow the facade to add libraries dynamically using a module registration system. This kind of system allows us to modify the libraries of the facade dynamically, adding new libraries without updating the facade code. In other words: the users will be able to customise the facade as they need.

5. Conclusions

During this project, we have researched how some libraries that cover semantic segmentation (such *SegmenTron* or *MMSegmentation*) can be grouped together into a common facade, having no difference in the way of using them.

Since all those libraries use custom *Python* objects to define a model (with custom methods to train and validate them), our first aim was to convert all those objects into a common one. Given the fact that three of the four implemented libraries use *FastAI* behind the scenes, our goal was to convert the `EncoderDecoder` objects from *MMSegmentation* into `Learner` objects from *FastAI.*

Since the previous step had some drawbacks in its implementation, we decided to increase the working scope and, instead of giving a common interface for those libraries, we decided to automate the whole process of semantic segmentation: from the dataset creation to the models evaluation, including also the data augmentation management and the models construction and training.

In order to do that, we have needed to divide the process into different components that are replaceable. Indeed, those abstract components can be used by everyone to define new particular components that complete the work that can be not implemented in our library yet.

As a result, we have a completely modular facade that manages the whole process of image segmentation and can be extended and used anytime by anyone.

6. Bibliography

1. Image segmentation. [cited 23 Nov 2021]. Available: https://en.wikipedia.org/wiki/Image_segmentation

2. Website. Available: https://www.researchgate.net/publication/359629812_Brain_Tumor_Detection_using_Deep_Learning

3. Website. Available: https://www.researchgate.net/publication/354773820_A_New_Approach_to_Orthopedic_Surgery_Planning_Using_Deep_Reinforcement_Learning_and_Simulation

4. Deep Reinforcement Learning based approach for Traffic Signal Control. Transportation Research Procedia. 2022;62: 278–285.

5. A computer vision framework using Convolutional Neural Networks for airport-airside surveillance. Transp Res Part C: Emerg Technol. 2022;137: 103590.

6. Contributors to Wikimedia projects. Convolutional neural network. Wikimedia Foundation, Inc.; 31 Aug 2013 [cited 28 Apr 2022]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network

7. DeepLabv3+. [cited 22 Apr 2022]. Available: https://wiki.hasty.ai/model-architectures/deeplabv3

8. U-Net. [cited 22 Apr 2022]. Available: https://en.wikipedia.org/wiki/U-Net

9. matterport. GitHub - matterport/Mask_RCNN: Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow. In: GitHub [Internet]. [cited 27 Apr 2022]. Available: https://github.com/matterport/Mask_RCNN

10. Home. [cited 23 May 2022]. Available: https://www.fast.ai/

11. Convolutional Filter. [cited 13 Jun 2022]. doi:10.1016/B978-0-12-820488-7.00034-7

12. Fine-tuning. Wikimedia Foundation, Inc.; 14 Sep 2004 [cited 13 Jun 2022]. Available: https://en.wikipedia.org/wiki/Fine-tuning

13. Leakage (machine learning). Wikimedia Foundation, Inc.; 13 Jan 2020 [cited 13 Jun 2022]. Available: https://en.wikipedia.org/wiki/Leakage_(machine_learning)

14. Jaccard index. Wikimedia Foundation, Inc.; 10 Jul 2005 [cited 24 May 2022]. Available: https://en.wikipedia.org/wiki/Jaccard_index

15. Jordan J. Evaluating image segmentation models. In: Jeremy Jordan [Internet]. 30 May 2018 [cited 14 Jun 2022]. Available: https://www.jeremyjordan.me/evaluating-image-segmentation-models/

16. About. [cited 30 May 2022]. Available: https://www.fast.ai/about/

17. Deep learning-based image segmentation for grape bunch detection. [cited 30 May 2022]. doi:10.3920/978-90-8686-888-9_98

18. Albumentations. [cited 23 May 2022]. Available: https://albumentations.ai/

19. fastai/fastai/vision/models at 87ee5d42003ea09acc9f71095f15cd26b8e42106 · fastai/fastai. In: GitHub [Internet]. [cited 24 May 2022]. Available: https://github.com/fastai/fastai

20. [No title]. [cited 1 Jun 2022]. Available: https://arxiv.org/pdf/1803.09820.pdf

21. SemTorch. In: PyPI [Internet]. [cited 18 May 2022]. Available: https://pypi.org/project/SemTorch/

22. segmentation-models-pytorch. In: PyPI [Internet]. [cited 18 May 2022]. Available: https://pypi.org/project/segmentation-models-pytorch/

23. Hutchinson B, Robinson C. Home. [cited 1 Jun 2022]. Available: https://www.fast.ai/

24. Home. In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/frankkramer-lab/MIScnn

25. GitHub - LikeLy-Journey/SegmenTron: Support PointRend, Fast_SCNN, HRNet, Deeplabv3_plus(xception, resnet, mobilenet), ContextNet, FPENet, DABNet, EdaNet, ENet, Espnetv2, RefineNet, UNet, DANet, HRNet, DFANet, HardNet, LedNet, OCNet, EncNet, DuNet, CGNet, CCNet, BiSeNet, PSPNet, ICNet, FCN, deeplab). In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/LikeLy-Journey/SegmenTron

26. GitHub - qubvel/segmentation_models.pytorch: Segmentation models with pretrained backbones. PyTorch. In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/qubvel/segmentation_models.pytorch

27. GitHub - PaddlePaddle/PaddleSeg: Easy-to-use image segmentation library with awesome pre-trained model zoo, supporting wide-range of practical tasks in Semantic Segmentation, Interactive Segmentation, Panoptic Segmentation, Image Matting, 3D Segmentation, etc. In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/PaddlePaddle/PaddleSeg

28. GitHub - qubvel/segmentation_models: Segmentation models with pretrained backbones. Keras and TensorFlow Keras. In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/qubvel/segmentation_models

29. SemTorch. In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/WaterKnight1998/SemTorch

30. GitHub - open-mmlab/mmsegmentation: OpenMMLab Semantic Segmentation Toolbox and Benchmark. In: GitHub [Internet]. [cited 1 Jun 2022]. Available: https://github.com/open-mmlab/mmsegmentation

31. LikeLy-Journey. GitHub - LikeLy-Journey/SegmenTron: Support PointRend, Fast_SCNN, HRNet, Deeplabv3_plus(xception, resnet, mobilenet), ContextNet, FPENet, DABNet, EdaNet, ENet, Espnetv2, RefineNet, UNet, DANet, HRNet, DFANet, HardNet, LedNet, OCNet, EncNet, DuNet, CGNet, CCNet, BiSeNet, PSPNet, ICNet, FCN, deeplab). [cited 23 Nov 2021]. Available: https://github.com/LikeLy-Journey/SegmenTron

32. 📦 Segmentation Models — Segmentation Models documentation. [cited 23 Nov 2021]. Available: https://smp.readthedocs.io/en/latest/models.html

33. 🏔 Available Encoders — Segmentation Models documentation. [cited 23 Nov 2021]. Available:

https://smp.readthedocs.io/en/latest/encoders.html

34. ImageNet. [cited 2 Jun 2022]. Available: https://www.image-net.org/

35. Instagram. In: Instagram [Internet]. [cited 2 Jun 2022]. Available: https://instagram.com/

36. Welcome to MMSegmenation's documentation! — MMSegmentation 0.19.0 documentation. [cited 23 Nov 2021]. Available: https://mmsegmentation.readthedocs.io/en/latest/

37. OpenMMLab. [cited 23 Nov 2021]. Available: https://github.com/open-mmlab

38. Patrones de diseño. [cited 23 Nov 2021]. Available: https://refactoring.guru/es/design-patterns

39. PyPI · The Python Package Index. In: PyPI [Internet]. [cited 2 Jun 2022]. Available: https://pypi.org/

40. open-mmlab. GitHub - open-mmlab/mmdetection: OpenMMLab Detection Toolbox and Benchmark. In: GitHub [Internet]. [cited 10 Jan 2022]. Available: https://github.com/open-mmlab/mmdetection

41. Lin T-Y, Goyal P, Girshick R, He K, Dollár P. Focal Loss for Dense Object Detection. 2017 [cited 19 May 2022]. doi:10.48550/arXiv.1708.02002

42. openmim. In: PyPI [Internet]. [cited 10 Jan 2022]. Available: https://pypi.org/project/openmim/

43. Google Colaboratory. [cited 8 Jun 2022]. Available: https://colab.research.google.com/github/open-mmlab/mmsegmentation/blob/master/demo/MMSegmentation_Tutorial.ipynb

44. Cross-validation (statistics). Wikimedia Foundation, Inc.; 31 Dec 2003 [cited 8 Jun 2022]. Available: https://en.wikipedia.org/wiki/Cross-validation_(statistics)

45. Aspect-oriented programming. [cited 22 Mar 2022]. Available: https://en.wikipedia.org/wiki/Aspect-oriented_programming

46. Data augmentation. Wikimedia Foundation, Inc.; 28 Aug 2016 [cited 16 Jun 2022]. Available: https://en.wikipedia.org/wiki/Data_augmentation

47. Random-access memory. Wikimedia Foundation, Inc.; 19 Sep 2001 [cited 30 Jun 2022]. Available: https://en.wikipedia.org/wiki/Random-access_memory

48. Here TYN. The AspectJ Project. [cited 24 Mar 2022]. Available: https://www.eclipse.org/aspectj/

49. 4. Aspect Oriented Programming — Spring Python v1.2.1.FINAL documentation. [cited 24 Mar 2022]. Available: https://docs.spring.io/spring-python/1.2.x/sphinx/html/aop.html

50. LemonFramework. GitHub - LemonFramework/python-aop: Python AOP Framework. In: GitHub [Internet]. [cited 24 Mar 2022]. Available: https://github.com/LemonFramework/python-aop

51. PEP 318 – Decorators for Functions and Methods. [cited 21 Jun 2022]. Available: https://peps.python.org/pep-0318/

52. NumPy. [cited 8 Jun 2022]. Available: https://numpy.org/