Deep Learning Project

-

Detecting and classifying malignant lung nodules from CT scans using PyTorch

Pietro Venturini (715166)

A.Y. 2020/2021

Contents

Introduction

Early diagnosis of cancer is a crucial task in medicine since it greatly increases the chances for successful treatment. This project focuses on lung tumors, more specifically on lung nodules. A lung nodule is an abnormal growth of round or oval shape that forms in a lung. Most lung nodules are benign, nevertheless, some of them can turn out to be malignant. In that case it is crucial to detect them as early as possible, in order to limit their growth and treat them specially. They are usually about 5 mm to 30 mm in size. Larger ones (30 mm or larger) are more likely to be cancerous than smaller ones. Some studies [1] showed that the probability of a nodule being malignant is generally less than 5%. CT scans of patients who have undergone an imaging test gets analyzed by pathologists in order to detect malignant nodules.

In this project, which is inspired by a book [2] about deep learning, I present a deep learning model based on neural network architectures using PyTorch [3], which aims at detecting and classifying nodules from CT scans of patients in order to support specialists in the task of identifying cancerous nodules. Currently, that work must be performed by highly trained specialists and it is doomed by cases where no malignant nodules exist. This is a hard task to automate, and it is still an unsolved problem. It is important to emphasize that the final predictions of the model will not be accurate enough to be used clinically. Nevertheless, it can be a starting point for a tool that may support specialists by containing the number of nodules that must be analyzed, saving time spent for each CT scan and allowing to examine a greater number of patients. This is not intended to be a fully automated system in which, once a piece of data is flagged as irrelevant, then it is gone forever, but rather an assistive system designed to augment a human's abilities. We want to present predictions for a human, but at the same time we want to allow him to peel back some of the layers and look for misses as well as annotate the model's findings with a certain degree of confidence.

In Chapter 1 we take a look at the dataset used for this project and we discuss some issues related to the processes of loading and preprocessing the data. In Chapter 2 we walk through the overall pipeline, the models involved and the metrics adopted for each model and their performance. Finally, we conclude our talk with a brief overview of the possible improvements that could be made

to the model.
The file `README.md` describes the content of the project directory. In particular, the Jupyter notebook `Training_models.ipynb` reports some of the scripts that have been used to train and test the models as well as to generate some of the images of this project report. The file `requirements.txt` contains the required dependencies as well as the instructions to download them into a VirtualEnv environment.

Chapter 1

Dataset and data preparation

In this chapter we are going to discuss the type of data we have to work with in order to perform cancer detection, as well as the dataset used. Then, we will go into details about the process of data preparation, i.e., which operations must be performed on the data before being fed into the model.

1.1 What is a CT scan?

A CT scan or computed tomography scan is the result produced by a CT scanner using X-rays. It differs from MRI (Magnetic Resonance Imaging) in many aspects, including cost (CT scans are less expensive), invasiveness (CT scans expose the patients to ionizing radiations) and time required (CT scans are faster to perform). Basically, it is a 3D image measuring radiodensity (a function of the atomic number and mass density of the examined material) of a human body section. For the purposes of this project, we are interested in scans of the thoracic section of the body.

From a technical point of view, a CT scan is represented by a 3D array of single-channel data. We will refer to the 3D equivalent of a 2D pixel as *voxel* (a volumetric pixel). Each voxel's numeric value corresponds to the average mass density of the material contained inside. Note that a voxels are not constrained to be cubic (typically the row and column dimensions have the same voxel size, with the index dimension being a little bit larger). Indeed, the way the distance along the head-to-foot axis is measured is different than the other axes. As stated in the introduction, the probability to find a malignant nodule in a CT scan is quite low. In the case of a malignant tumor being present, the vast majority of voxels still won't be cancer.

1.2 The dataset

The dataset involved in this project is the LUNA2016 dataset from the LUNA (LUng Nodule Analysis) Grand Challenge [4, 5], which is a subset of a larger, publicly available database called LIDC/IDRI [6] (only 888 out of 1018 CT scans taken from 1010 different patients have been considered for the challenge). The LUNA2016 dataset does not contain demographic information about patients. The script `download_dataset.sh` can be used to download the dataset from the Internet.
The dataset weights around 120 GB and among the files it contains, it includes:

- `subset0.zip`, ..., `subset9.zip`: 10 zip files with the CT scans, for a total of 888 scans.
- `annotations.csv` contains information about 1186 candidates that have been flagged as nodules, such as the coordinates and the diameter.
- `candidates.csv` contains information about 551065 lumps that potentially looks like nodules, such as the coordinates and the class (nodule/non-nodule).

Another file of interest is `annotations_with_malignancy.csv` which comes instead from the LIDC/IDRI database [6] and includes annotations about the malignancy of each nodule. This file will be used instead of `annotations.csv`.
The csv files are located under the `data/part2/luna/` folder while the ct scans will be placed under the `data-unversioned/part2/luna/` folder.
Each CT scan is represented as two files: a `.mhd` file containing the metadata header information (such as ID (`series_uid`), origin, spacing, direction...), and a `.raw` file containing the raw bytes of the CT scan. Commonly, CTs are 512 rows by 512 columns, with the index dimension varying from 100 up to 250 total slices, resulting in at least $2^{25} \sim 32M$ data points.
By invoking `showCandidate(series_uid)` from `cls.vis.py` we can visualize the CT scan identified by the `series_uid` provided, as shown in Figure 1.1.

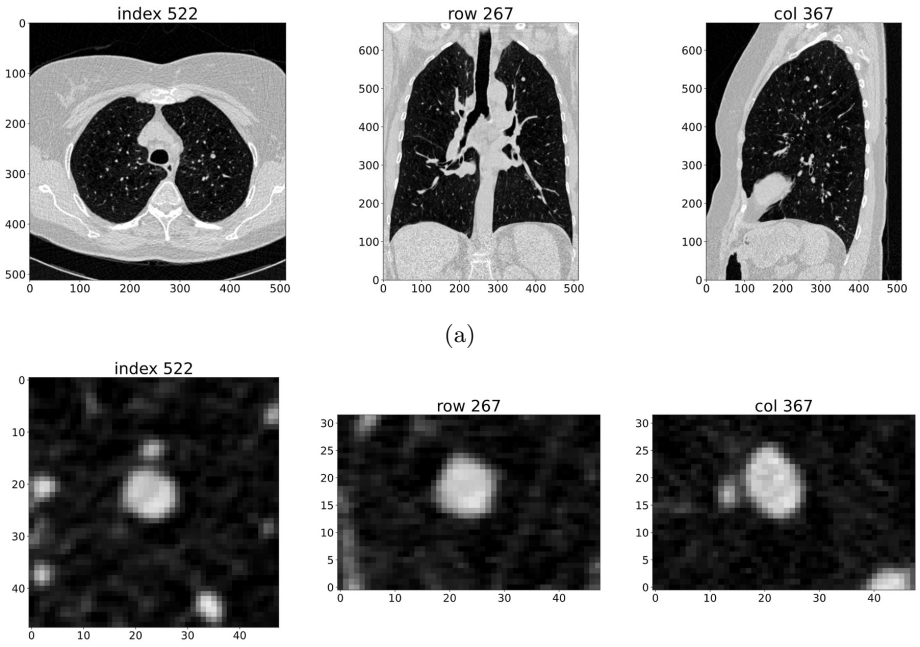


Figure 1.1: (a) Visualization of the CT scan with `series_uid: '1.3.6.1.4.1.14519.5.2.1.6279.6001.126264578931778258890371755354'` from `subset0`, using the `showCandidate` method from `cls.vis.py`. The three slices have been performed along the main axes and around the center of the scan. (b) A crop centered around a nodule from the same scan.

It is also possible to use 3D visualization libraries, such as IPyvolume[1], to visualize the full CT scan in three dimensions. An example of such a visualization is shown in Figure 1.2.

[1] `https://ipyvolume.readthedocs.io`

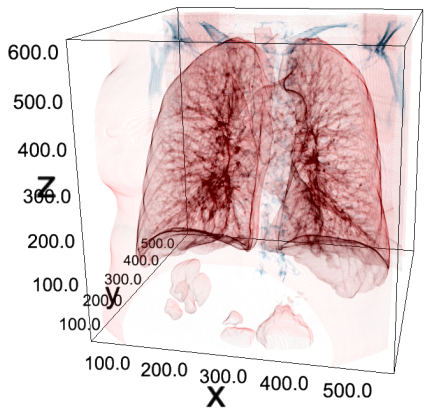


Figure 1.2: 3D visualization of the CT scan with `series_uid: '1.3.6.1.4.1.14519.5.2.1.6279.6001.126264578931778258890371755354'` from `subset0`, using the IPyvolume library. The quality of the plot is questionable since, in order to obtain a clear view of the sections we are interested in, we need to manually tweak the opacity of each channel. Refer to the notebook `Training_models.ipynb` for the code used to draw this plot.

1.3 Data preparation

There are some caveats about data that we must pay attention to. First of all, the CT scans format is MetaImage [7]. We can deal with that kind of format using the `SimpleITK` Python library [8]. We need to deal with the transformation from the patient coordinate system, in which the dimensions are expressed in millimiters (X,Y,Z) and the origin is arbitrarily positioned, to the row-column-index coordinate system, in which the origin corresponds to the origin of the voxel array. Figure 1.3 illustrates this conversion. The two methods `xyz2irc` and `irc2xyz` from the `util.util.py` module perform those transformations using data (origin, voxel size and direction) extracted with `SimpleITK`.

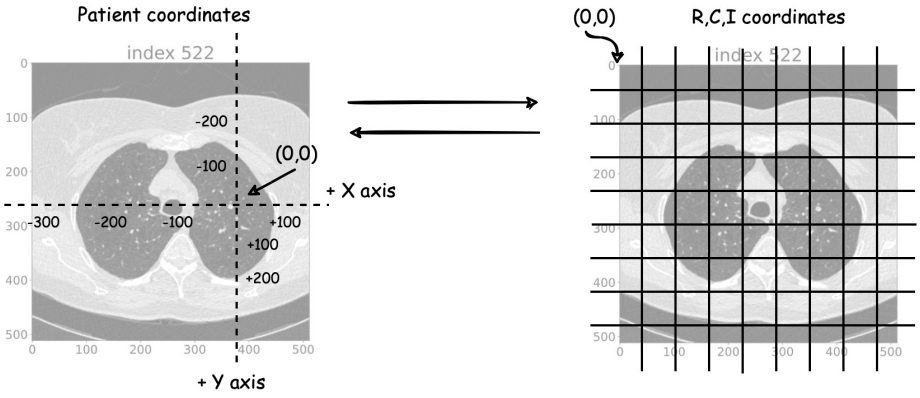


Figure 1.3: Transform from the patient coordinate system (X,Y,Z) to the row-column-index coordinate system.

As we will see in Chapter 2, we are going to build a pipeline consisting of multiple neural networks, each of which solves a specific task. Two of the models involved (the nodule/non-nodule classifier and the malignant/benign classifier) make predictions for individual nodules. Therefore they expect as input small sub-volumes of the CT scan surrounding the nodule in question. In order to do that, the `cls.dsets.LunaDataset` class has been created, inheriting from the abstract class `torch.utils.data.Dataset` and overriding the methods `__len__` and `__getitem__`. This is the wrapper for our dataset that will be used later to instantiate the data loaders during the training and evaluation phases. The method `__getitem__` handles the cropping of individual nodules from the whole CT scans, using information from the files `candidates.csv` and `annotations_with_malignancy.csv`. The code that handles the loading phase and the preprocessing of individual CT scans is in the class `cls.dsets.Ct` instead.
Actually, there are many other things to do before feeding the data to the model, such as balancing the positive and negative samples in the dataset, choosing the train-validation-test split and so on. These aspects will be discussed later in Chapter 2.

1.3.1 Caching CT scans and nodules

In order to avoid bottlenecks loading the data from disk, we can exploit some caching libraries both on disk and in RAM. Both the caching methods we are going to use are Python decorators. Decorators are simply wrappers around a certain function, that, besides executing the function, can perform some arbitrary operations before and after its execution. In particular, we use `lru_cache` from the `functools`[2] module for in-RAM caching and `FanoutCache` from the

[2] `https://docs.python.org/3/library/functools.html`

`diskcache`[3] library for on-disk caching.
More precisely, we want to prevent reading an entire CT scan from disk for every nodule (a single CT typically contains a lot of nodules). Therefore, by wrapping the `cls.dsets.getCt` method with the `functools.lru_cache` decorator, when that method is invoked for the first time, its return value gets cached in RAM and used if that method is invoked again later with the same arguments.
Similarly, since some nodules have to be used multiple times (in particular, in order to balance the positive and negative samples in the dataset, many positive samples will be replicated multiple times), the `cls.dsets.getCtRawCandidate` method have been wrapped with the decorator from the `diskcache` library in order to cache the smaller crops around individual nodules on disk. It is useful to loop over a `LunaDataset` instance before running any training script, in order to prepare the cache and speed up the training that we are going to perform later. The script `cls.prepcache.py` does exactly that.

1.3.2 Augmenting the data

In order to reduce the risk of overfitting the training set, it is recommended to augment the dataset by applying synthetic alterations to the samples, aiming at obtaining new samples that remain representative of the same class as the source ones, but prevent the model to memorize individual images. Augmentation of the training data is handled by the class `SegmentationAugmentation` from the module `seg.model.py` for the segmentation model, and by the method `Ct.getCtAugmentedCandidate` from the module `cls.dsets.py` for the classification models.

The techniques adopted to perform augmentation of images are:

- Mirroring
- Horizontal/vertical shifting (note that convolutions are translation independent but this will make the model more robust with respect to imperfectly centered nodules)
- Scaling in/out
- Rotation around the index axis
- Noise addition

Those augmentation methods have been implemented using suitable affine transformation matrices [9] and PyTorch functions. An example of these transformations is depicted in Figure 1.4.

[3] `http://www.grantjenks.com/docs/diskcache`

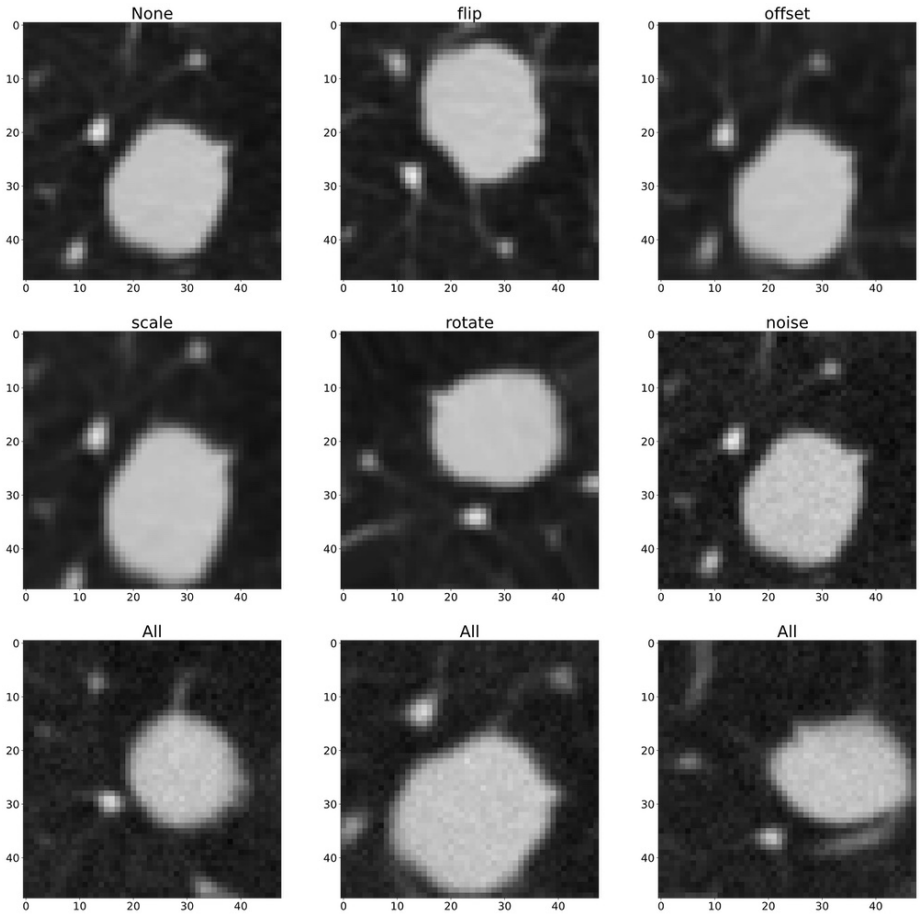


Figure 1.4: Augmentation techniques applied to a CT crop of a candidate nodule from the CT scan having `series_uid = 1.3.6.1.4.1.14519.5.2.1.6279.6001.752756872840730509471096155114`. The bottom row depicts three different results when all the transformations are applied; the reason they differ is that each transformation is applied with some random parameters. Refer to the notebook `Training_models.ipynb` to see how this figure has been generated.

1.3.3 Using Google Cloud services

Because of the large amount of data and the computational resources required to train the models, we had to use a cloud service because our machine wasn't capable to handle all that workload. We chose Google Cloud Platform, which offers configurable virtual machines[4] with the option to install also a GPU and

[4]Google Cloud deep learning VM images: `https://cloud.google.com/deep-learning-vm`

common deep learning frameworks. We rented a virtual machine for about two months with the configuration reported in Table 1.1.

Virtual machine's configuration	
O.S.	Debian 10 (Linux)
Machine type	n1-highmem-2 (2 vCPU, 13 GB RAM)
GPU	1 x NVIDIA Tesla T4
Disk	SSD (400 GB)
Zone	europe-west1-b (St. Ghislain, Belgium, Europe)

Table 1.1: Configuration of the virtual machine rented on Google Cloud Platform.

Chapter 2

The pipeline

Ideally, we want our model to consume CT scans and identify malignant nodules. A single end-to-end model can turn out to be quite difficult to design and train. On the contrary, having many models, each of which is specialized in a particular task, favors a more modular solution, which is both easier to debug and preferable for a clinical environment: having a system that flags suspicious nodules for review is preferable than a system which outputs a single binary prediction (malignant/benign).
Since the Luna dataset already comes with annotations of the nodules, we could have focused on the single task of classifying the nodules, however, a deep learning model of that kind, in order to be used, would require a priori knowledge of the candidate nodules (some specialists should provide the candidates) which is probably the most-time consuming job. A more realistic system, would take an entire CT scan as input, then it would identify candidate nodules, understand which of them are actually nodules and finally predict which are cancerous. Each of those tasks is performed by a single model. That will allow, other than making a final prediction about the malignancy of each of the nodules that have been found, to flag suspicious nodules that, eventually, can be reviewed manually by an expert.
The overall pipeline looks like Figure 2.1. First, CT scans are fed into a segmentation model whose job is to segment the 3D array and produce a mask highlighting the nodules. From that mask we extract a list of coordinates of the detected candidates. Then those nodules are fed individually into a classification model that discriminates between true nodules and false ones. Those that have been flagged as nodules are fed into another model that distinguishes between malignant nodules and benign ones.

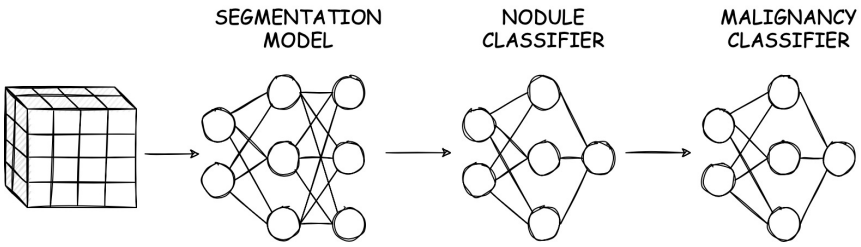


Figure 2.1: Sketch of the overall pipeline that we are going to build. The segmentation model perform segmentation on the CT scan provided as input, and produces a list of candidate nodules that are fed into a nodule classifier. The classifier keeps only the samples that are classified as true nodules, and pass them to a malignancy classifier, which determines if a nodule is cancerous rather than benign.

2.1 Finding nodule candidates with segmentation

The job of the first model of the pipeline is to compute a segmentation mask for a CT scan, grouping together voxels that belong to the same candidate nodule. After that, we need to extract information such as center position and diameter for each candidate nodule. There are many possible approaches to this kind of problem, such as:

- semantic segmentation (labelling voxels belonging to nodules as `nodule` and remaining ones as `background`),
- instance segmentation (labelling voxels belonging to different nodules with different labels, such as `nodule1`, `nodule2`...)
- object detection (outputting a bounding box around each nodule)

Training a segmentation model is easier than an object detection model, furthermore, since nodules are unlikely to overlap, we can proceed with semantic segmentation. A suitable architecture based on neural networks is the so-called U-Net architecture [10]. U-Net consists of a series of convolution and max-pool layers that progressively downsample the input image, followed by a series of upsampling convolutions that restore the image size, making it matching the input size. The PyTorch implementation of the U-Net architecture from the original paper can be found in many repositories on GitHub, such as the one [11] that has been copied in the `util.unet.py` module. Note that the original U-Net is a 2D segmentation model; we could adapt it to use 3D convolutions in order to use information from multiple slices, however, that would make the memory usage considerably larger (each CT scan is about $2^9 \times 2^9 \times 2^7$ with 2^2 bytes per voxel, the first layer of the U-Net architecture is 2^6 channels, resulting

in a memory occupation of $2^{9+9+7+2+6} = 2^{33}$ bytes, or 8 GB, just for the first convolutional layer). Furthermore, the pixel spacing along the Z dimension is larger than along the XY plane, so it is unlikely for a nodule to be present across many slices. For this reason, the model adopted for this project perform 2D segmentation on individual slices of each CT. However, since CTs are single-channel, we provide few neighboring slices (three slices below and three slices above) as the channel dimension. Doing so will cause us to lose the direct spacial relationship between adjacent slices (they will get linearly combined by the convolution kernels with no notions of being one rather than two slices away) but, at least, we still keep into account the local context of each slice.

The full segmentation model based on the U-Net architecture is illustrated in Figure 2.2. In our implementation, we reduced the depth and the number of filters because the capacity of the standard model outstrips the dataset size. Our model considers input images with 7 channels (the current slice together with the 3 slices below and above it). The number of filters applied in the first layer is 2^4 and doubles at each layer in depth. The total depth is 3 and batch normalization is applied after each activation function. For the upsampling we use transposed convolutions.

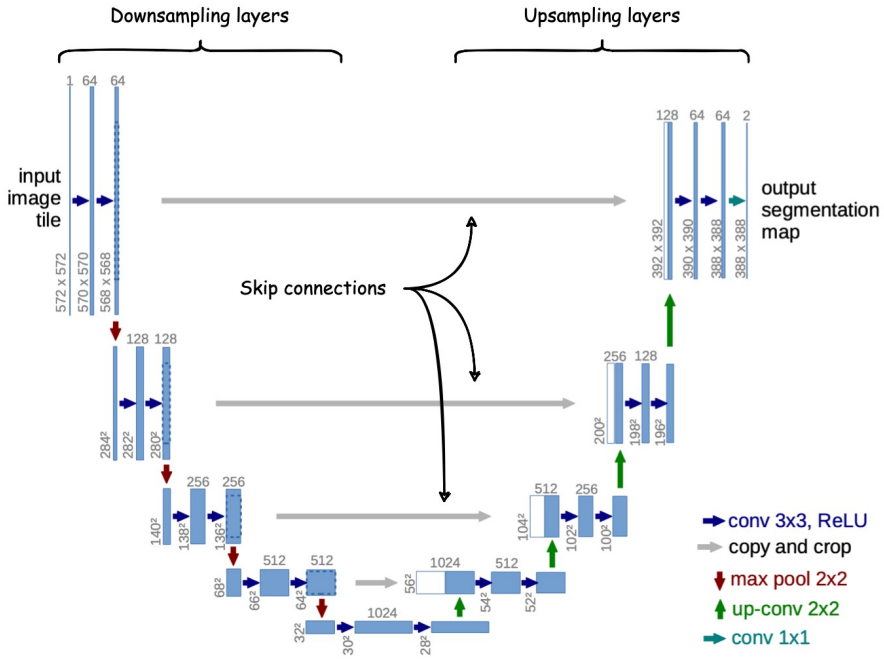


Figure 2.2: Graphical representation of the U-Net architecture from the original paper. In our implementation, instead, the total depth is 3, the input images have 7 channels and the output images have one channel.

2.1.1 Data preparation for the segmentation model

In order to train the U-Net model, we need pairs of input CT slices and the corresponding segmentation mask. We need to create the segmentation masks using information from the `candidates.csv` file. That can be done by a simple algorithm (similar to region growing), that starts from the nodule center and explore the neighboring voxels along the three axis, until some voxel falls under some threshold. The algorithm is performed by the method `buildAnnotationMask` of the class `Ct` from the module `seg.dsets`. The creation of the training and validation sets is handled by the classes `TrainingLuna2dSegmentationDataset` and `Luna2dSegmentationDataset` respectively, which inherit from the `nn.Module` class of the `PyTorch` library.
The train/validation/test split ratio can be controlled by the `val_stride` argument of the `Luna2dSegmentationDataset` class during instantiation. For this project the adopted split is 80%-10%-10%.

2.1.2 Training the model

Once we have the segmentation masks corresponding to the input CT slices we can train the segmentation model. Note that the U-Net architecture is pixel-to-pixel and takes images of arbitrary size. Therefore, training and validation samples can differ in size without any issues. Training the model using full CT slices leads to poor results, due to a class-balancing problem: the number of negative pixels is much larger than the number of positive ones, since nodules are quite small compared to the full CT slice. In order to solve this issue, we will train the model using 64×64 crops centered around each candidate nodule, reducing the negative pixel count. This will also make the training converge more quickly. On the contrary, the validation and test sets won't contain cropped patches of the CT slice but the whole slice (this explains the need to have 2 different `Luna2dSegmentationDataset` classes for training and validation).

In order to train the model we first have to define the optimizer to use and the loss to optimize. We use Adam [12] as the optimizer for our network. Differently from stochastic gradient descent, Adam mantains a separate learning rate for each parameter and automatically update the learning rate as the training progresses. A common loss metric for segmentation tasks is the Sørensen-Dice coefficient [13], which is defined for boolean data as

$$\frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

where TP, FP and FN indicates true positives, false positives and false negatives, respectively. The loss is computed for a batch of samples by the function `computeBatchLoss` from the `seg.training` module. In order to balance the number of positive pixels to the number of negative pixels we cropped our training examples around the each nodule center. However, it's still very important to have high recall. Therefore, we need a weighted loss such that getting the

population of positive pixels right is valued more than getting the population of negative pixels right. In order to account for that, we compute:

- `diceLoss_g`: Dice coefficient for the entire input sample
- `fnLoss_g`: Dice coefficient such that only the false negative pixels will generate loss. This is done by computing the intersection of the predicted tensor and the target tensor and using that as the predicted mask. Doing so will remove any false positive from our original prediction.

Finally, the overall loss is given by linearly combining those two losses, for instance, if we want to consider `fnLoss_g` as 8 times more important than `diceLoss_g`, then the overall loss is given by `diceLoss_g + 8 · fnLoss_g`. Since we're willing to give up a large number of true negative pixels in the pursuit of having better recall, we should expect a large number of false positives in general. Furthermore, since our validation set contains orders of magnitude more negative pixels, the model will have a huge false positive rate during validation.

The model has been trained on the augmented dataset for 15 epochs, and the validation set has been employed to select the epoch that yielded the largest score. The code that runs the training loop is in the `main` function of the `seg.training` module, including the logging utilities that saves metrics and information to be plotted in Tensorboard.

2.1.3 Model performance

Training performance looks good (Figure 2.3): false positives and false negatives are trending down while F1 score and true positives are trending up. However, since we trained our model on 64×64 slices but we are validating it on full 512×512 CT slices, we should expect quite different results. In particular, the false positives count is about 64 times bigger (the full slice is 64 times bigger than the training crop), also resulting both in low precision and low F1 score (as Table 2.1 shows). The problem is with recall: we begin overfitting after few epochs.

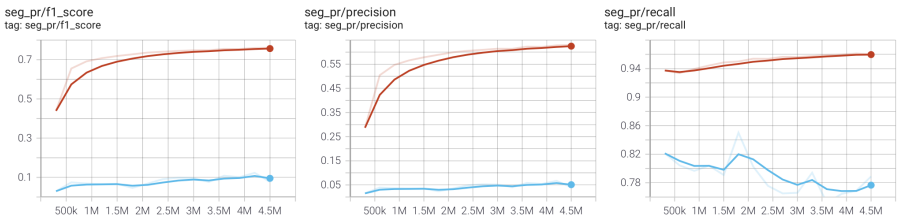


Figure 2.3: Tensorboard plots of F1 score, precision and recall for the segmentation model on both the training (red plot) and validation (blue plot) set for the first 15 epochs. The largest recall on validation set is attained at the 6th epoch.

Segmentation model metrics for the best model (6th epoch)				
	F1 score	Precision	Recall	Loss
Training set	0.7275	0.5894	0.9501	0.2333
Validation set	0.0466	0.0239	0.8502	0.9375
Test set	0.1088	0.0585	0.7745	0.8805

Table 2.1: F1 score, precision, recall and overall loss of the best segmentation model obtained at the 6th epoch. The best model has been selected according to the recall on the validation set.

The best model has been selected according to epoch that yielded the largest recall on the validation set, which is the 6th. Despite the questionable results, issues with low precision will be handled by the nodule classification model, which will take care of filtering out the candidate nodules that are not actual nodules.

2.2 Classifying candidate nodules

The next step in our pipeline is to filter out the candidate nodules by classifying them into nodules and non-nodules. This can be done by a binary classifier that receives as input a crop around a candidate nodule of a CT slice, and produces as output the probability of the nodule in the image being an actual nodule. We need to define what data the next model in the pipeline is going to consume. Since the classifier makes predictions for a single nodule, instead of looking at the entire CT, we provide to the model a smaller crop around a candidate nodule. The size of each crop has been set to 32 voxels along the index dimension and 48 voxels along both the row and the column dimensions, resulting in a volume of size $32 \times 48 \times 48$. If we consider also the batch size (which is set by default to be 48) and the single channel, the dimension of the input tensors is $48 \times 1 \times 32 \times 48 \times 48$. As stated in Section 1.3, the `LunaDataset` class from the `cls.dsets` module handles all the data preparation phase, including data augmentation (see Section 1.3.2) and balancing of positive and negative samples (see Section 2.2.3).
It is important to note that we need to avoid any leaks from the training set to the validation set. Since we trained the segmentation model on full CT scans and we are going to train the nodule classifier on crops around nodules, we must ensure that we don't have any nodule from the segmentation validation set in the classification model's training set (and vice versa).

2.2.1 The architecture of the nodule classifier

There are infinitely many possibilities for the design of a neural network architecture which is capable of detecting nodules. For this project, the model consists of a tail, a backbone and a head as follows:

- **Tail**: The shallower layer performs batch normalization (using `nn.BatchNorm3d` from PyTorch) of $32 \times 48 \times 48$ single-channel samples.
- **Backbone**: Four repeated blocks, each of which consists of two 3×3 3D convolutions, each followed by a ReLU activation, ending with a max-pooling operation. Each block reduces the image size but increases the number of channels, starting from a single channel and ending with 64 channels.
- **Head**: The deeper layers flatten the output of the backbone (which by now consists of $2 \times 3 \times 3$ images that are 64 channels in depth), pass it to a fully connected layer that reduces the dimension from 1152 to 2, which is followed by a `nn.Softmax` layer from PyTorch.

Figure 2.4 illustrates the overall architecture of the model, which is defined by the `LunaModel` class of the `cls.model` module.

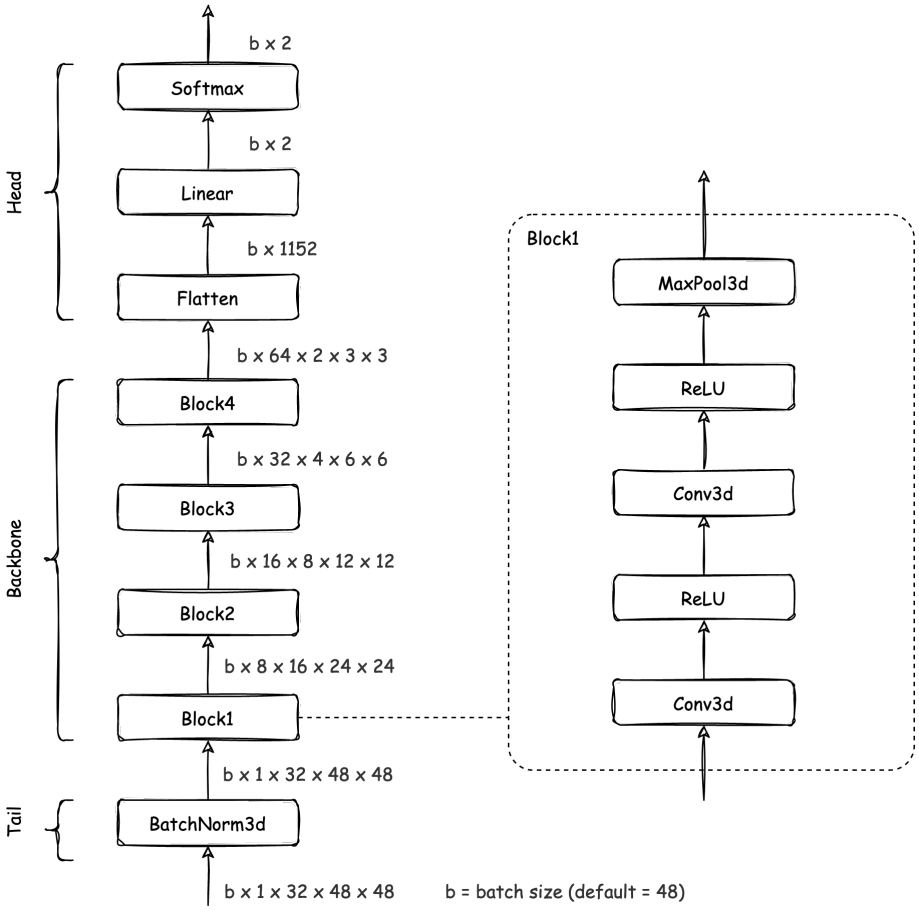


Figure 2.4: Visual representation of the architecture of the nodule classifier. The entire architecture is shown on the left, while the inner structure of each convolution block is shown on the right. The shape of the input to each block is written near the entering arrow to that block. By default, the batch size is set to be 48.

2.2.2 Initialization

In order to prevent the phenomena of either exploding gradients or vanishing gradients, the network is initialized according to the He initialization method [14], which is suitable when using ReLU activation functions. The resulting tensors will have values sampled from $\mathcal{N}(0, \text{std}^2)$, where

$$\text{std} = \sqrt{\frac{2}{\hat{n}_l}}$$

being $\hat{n}_l$ the fan-out of the l-th layer (for linear layers it is the number of output

neurons, while for convolutional layers it is k^2d, where k is the kernel size and d is the number of output channels).

2.2.3 Dataset balancing

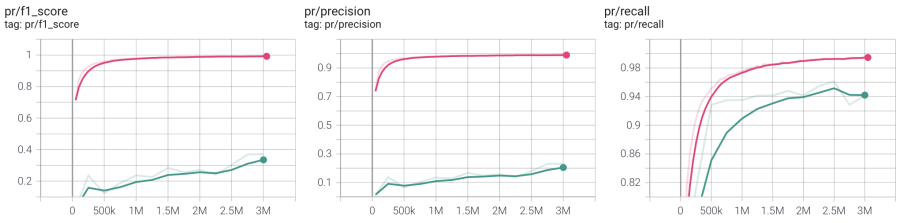
Without paying enough attention to the data we use to train the model, results can be quite unsatisfactory. The main reason is due to the fact that the number of positive samples in our dataset (hence in the training, validation and test sets) is definitively less than the number of negative ones, with a ratio of 1:400 positive to negative samples. Therefore, without balancing the training set, the model would struggle to learn anything, ending up classifying every sample as negative. In order to make the training data more suitable for our purposes (ideally we'd like to have just as many positive samples as negative ones), we'll present the positive samples repeatedly in our training set. Notice that balancing is neither performed on the validation set nor on the test set, because the model needs to function well in the real world, which is imbalanced, and those two sets have to be representative of the real world. This will reduce the number of true negatives in favor of a larger recall. Samples balancing is performed by the `__getitem__` method of the `LunaDataset` class from `cls.dsets.py`.

2.2.4 Model performance

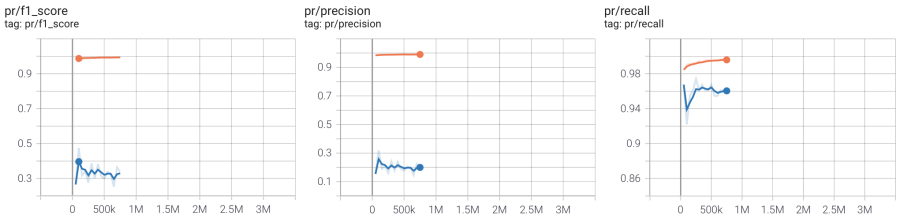
We used stochastic gradient descent with weight decay as the optimizer to train this model. The loss on which backpropagation is performed is the classical cross-entropy between the model predictions and the true labels. The model have been trained initially for 60 epochs on a balanced training set whose samples have been augmented with the data-augmentation techniques described in Section 1.3.2 in order to prevent overfitting. After plotting some performance metrics using Tensorboard (Figure 2.5) we decided to train for some more epochs with a different learning rate, for a total of 95 epochs. The final model has been selected according to the performance on the evaluation set, choosing the 82nd epoch, which is the one in which the model scored the largest F1 score on the validation set. The overall performance is reported in Table 2.2.

Nodule classifier metrics for the best model (82nd epoch)				
	F1 score	Precision	Recall	Detected nodules
Training set	0.9891	0.9877	0.9905	-
Validation set	0.4749	0.3198	0.9221	142 of 154
Test set	0.3340	0.2067	0.8696	80 of 92

Table 2.2: F1 score, precision and recall of the best model obtained at the 82nd epoch. Performance are not excellent: some nodules were not detected by the classifier on both the validation set and the test set. Note that both the validation set and the test set contain a small number of positive examples because they must be representative of the true population.



(a)



(b)

Figure 2.5: (a) F1 score, precision and recall of the nodule classifier on both the training (pink) and validation (green) sets for the first 60 epochs. The best model is selected according to the F1 score achieved on the validation set. Since at the 60th epoch it was still increasing, we trained the model for more epochs. (b) Metrics of the nodule classifier on the training (orange) and validation (blue) sets from the 80th to the 95th epoch. The largest F1 score is obtained at the 82nd epoch and it is equal to 0.4749.

2.2.5 A bridge between the segmentation model and the nodule classifier

The segmentation model outputs masks with same dimension as the CT slices input to it. However, the nodule classifier expects $32 \times 48 \times 48$ crops around candidate nodules. As a consequence of this, we need an intermediate layer which is able to convert the boolean masks computed by the segmentation model into 3D crops around candidate nodules. That job is performed by the `NoduleAnalysisApp.groupSegmentationOutput` method from the `nodule_analysis` module.
Generally speaking, the steps performed are:

1. Threshold the slice-wise predictions to get a binary array
2. Stack all the slices together into a 3D array
3. Apply erosion using `binary_erosion`[1] from `scipy.ndimage.morphology` smoothing out boundaries and causing components smaller than $3 \times 3 \times 3$ voxels to vanish.

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.binary_erosion.html`

4. Group together labeled voxels that share at least an edge. That can be done using the `label`[2] function from `scipy.ndimage.measurements`, which returns a mask in which each cluster of voxels is uniquely labeled.

5. Get the coordinates of the center of each cluster using the function `center_of_mass`[3] from `scipy.ndimage.measurements`.

The result is a list of candidate coordinates for centers of nodules. Using information from that list, we can crop a fixed-size volume around each candidate center from the corresponding CT scan, and feed it into the nodule classifier.

2.3 Find cancer with a malignancy classifier

The nodule classifier helped to filter out candidate nodules that weren't considered as nodules. By removing some data at each step, we reduce the number of samples that a specialist have to examine if we wanted to manually identify malignant tumors. Speaking about the order of magnitude of data removed at each step, we started with a full CT scan of about 33 million ($\sim 2^{25}$) voxels. The segmentation model flagged about a million ($\sim 2^{20}$) voxels as being of interest. With grouping we reduced the number of items to consider, producing around 1000 ($\sim 2^{10}$) candidates. The nodule classifier left us with tens ($\sim 2^5$) of nodules. The job of the final model of our pipeline, which we are going to discuss in this section, is to take tens of nodules and identify the one or two ($\sim 2^1$) that are cancer. Figure 2.6 depicts the overall pipeline with the rough amount of data removed at each step.

We are going to train a malignancy classifier using data from `annotations_with_malignancy.csv`, which we already discussed in Chapter 1. The code that handles the sampling from this dataset is encapsulated in the class `MalignantLunaDataset` from the module `dsets`.

[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.label.html`

[3] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.center_of_mass.html`

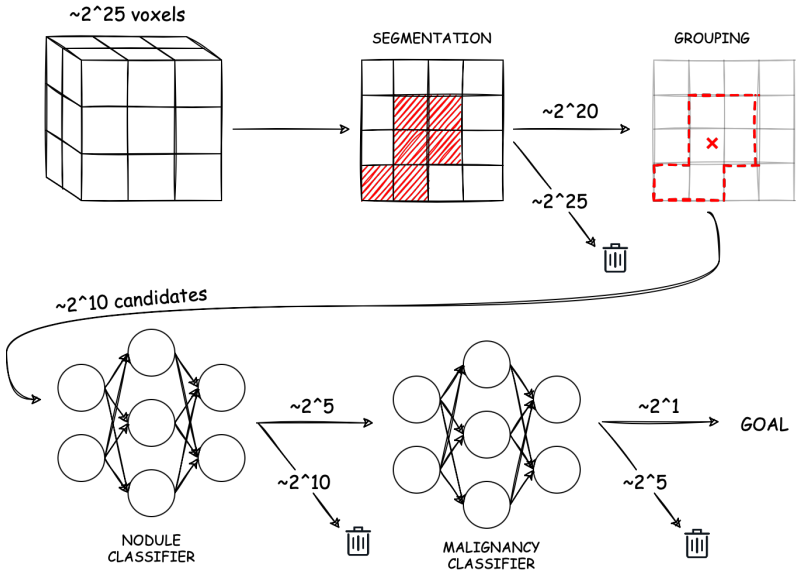


Figure 2.6: Overall pipeline with the rough amount of data removed at each step.

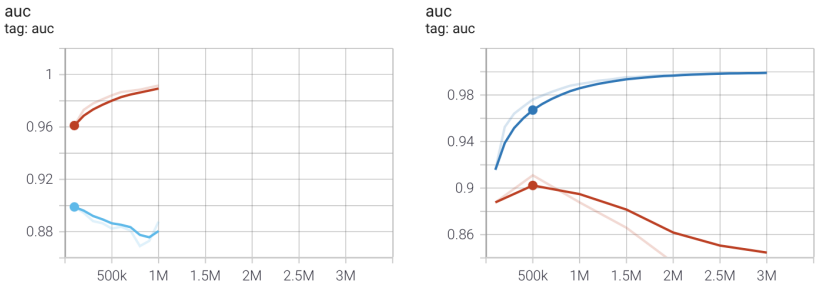
2.3.1 Model architecture

We've already designed and trained a model (the nodule classifier) which receives as input the same kind of data we expect the malignancy classifier to consume, and tries to extract features to make predictions about the input nature (nodule vs non-nodule). We will reuse that architecture (Figure 2.4) also for the malignancy classification task, performing *fine-tuning*: we are going to retrain only the last two layers (a linear layer and a the last convolutional block) of our model. In the first layers we are going to keep the weights we learned when training the nodule classifier fixed, allowing only the weights in the last layers to be updated by backpropagation. The main reason behind that is that we could reasonably believe that the low-level features learned by the nodule classifier, which were useful to predict about nodules, can turn out to be useful also to predict about malignancies.

2.3.2 Model performance

In order to see if our model performance are better than nothing, it is good to have a baseline. A baseline is a simple model that provides a reasonable result on a certain problem. As stated in [1], nodules' size can be used as a predictor for malignancy. Therefore, a trivial model could use the nodule diameter as the only input to predict whether that nodule is malignant or not. Rather than fixing a threshold and comparing the baseline's F1 score with our model's, we can use the receiver operating characteristic (ROC) curve and compute the

area under curve (AUC). ROC curve plots the true positive rate (TPR or recall) against the false positive rate (FPR or $1 - \text{specificity}$). To compute AUC we first generate an array of possible thresholds (which, for the baseline, are nodules diameters), then, for each threshold, we compute the TPR and the FPR, and we use numeric integration by the trapezoidal rule [15] to compute the area under the curve. We do the same for our model, since it outputs probabilities (computed by a softmax function) of nodules being malignant, which we can threshold in order to compute TPR and FPR.
The baseline's AUC turned out to be 0.901. The malignancy classifier has been trained for 10 epochs but, as Figure 2.7a shows, there have been problems with overfitting since the beginning. Unluckily, the fine-tuned malignant model had a lower AUC on the validation set than the baseline, being it equal to 0.8989.
Because of the low performance of the fine-tuned malignancy model, we tried to train a malignancy classifier from scratch using the same architecture of the nodule-nonnodule classifier (Figure 2.4). The model have been trained for 30 epochs, but the largest AUC on the validation set has been reached after 5 epochs. That AUC is equal to 0.911 and it is larger than the baseline's AUC.



(a) (b)

Figure 2.7: (a) AUC computed on the training (red) and validation (light blue) set for the malignancy classifier trained by fine-tuning only the last two layers of the best nodule-nonnodule classifier. We begin immediately to overfit. (b) AUC computed on the training (blue) and validation (red) set for the malignancy classifier trained from scratch using the same architecture adopted for the nodule-nonnodule classifier. At the fifth epoch, AUC on the validation set is 0.911.

Table 2.3 reports metrics for the malignancy classifier trained by fine-tuning the best nodule-nonnodule classifier. That model is able to classify correctly 48 of 60 benign nodules and 28 of 32 malignant nodules from the test set.

Malignancy classifier metrics (trained by fine-tuning)				
	AUC	F1 score	Precision	Recall
Training set	0.9611	0.8998	0.8941	0.9055
Validation set	0.8989	0.7333	0.6471	0.8462
Test set	0.9258	0.7778	0.7000	0.8750

Table 2.3: Performance metrics of the malignancy classifier trained by fine-tuning the last two blocks of the nodule classifier.

If we instead train the malignancy model from scratch, then we can overcome the baseline AUC on the validation set (full metrics on validation and test sets are reported in Table 2.4), being able of classifying correctly 53 of 60 benign nodules and 25 of 32 malignant ones.

Malignancy classifier metrics (trained from scratch)				
	AUC	F1 score	Precision	Recall
Validation set	0.9110	0.7619	0.7581	0.7692
Test set	0.8964	0.7812	0.7812	0.7812

Table 2.4: Performance metrics of the malignancy classifier trained from scratch using the same architecture used for the nodule classifier.

By using AUC on validation set as the criterion to select the best model, we chose the malignancy model trained from scratch as the final model of the overall pipeline, since it scored a larger AUC on the validation set than the fine-tuned model.

2.3.3 End-to-end performance

After having trained and evaluated each model individually, we need to stack them into a single end-to-end model which takes CT scans as input, segments them using the segmentation model in order to produce a list of candidate nodules (procedure is described in Section 2.2.5), filters those candidates using the nodule-nonnodule classifier and, finally, makes predictions about the malignancy of the remaining nodules with the malignancy classifier.
By invoking the `cls.nodule_analysis.py` module we can run the full three-models pipeline and print a summary table reporting performance on the validation set (Table 2.5) and on the test set (Table 2.6). Which dataset to use can be specified by either the `--run-validation` or the `--run-testing` arguments, while the models to use can be specified by providing their path to the `--segmentation-path`, `--classification-path` or `malignancy-path` parameters (the best models are automatically saved during training). Each dataset required about half an hour to be evaluated through the whole pipeline.

End-to-end performance on validation set				
	Missed	Filtered out	Pred. ben	Pred. mal
Non nodules		162825	641	113
Benign	12	6	71	13
Malignant	1	6	8	37

Table 2.5: Performance of the end-to-end model on the validation set. The first column reports the nodules that have been missed by the segmentation model. The second column reports the nodules that have been filtered out by the nodule-nonnodule classifier. The third and the fourth columns report the predictions of the malignancy classifier.

End-to-end performance on test set				
	Missed	Filtered out	Pred. ben	Pred. mal
Non nodules		144659	552	80
Benign	12	3	39	6
Malignant	2	7	7	16

Table 2.6: Performance of the end-to-end model on the test set.

The results are not very satisfactory: in the validation set we detected 129 of 154 nodules (84%) and flagged correctly 37 of 52 malignant nodules (71%), missing 15 malignant nodules. In the test set we detected 68 of 92 nodules (74%) and flagged correctly 16 of 32 malignant nodules (50%), missing 16 malignant nodules.

Chapter 3

Conclusions

3.1 Commenting the final results

The performance of the overall end-to-end model are not very satisfactory. 15 out of 52 (29%) malignant nodules have been missed in the validation set. Performance are even worse in the test set, missing 16 out of 32 (50%) malignant nodules. There are a lot of false positives, but thanks to the segmentation model, we reduced significantly what needs to be looked at. However, the large number of false negatives (speaking about malignant nodules) makes this model not suitable to be used in a real context. Authors of [2] were able to obtain better results since they split the dataset into a training set and a validation set only. Doing like that, allowed them to use more data to train the models. In our case, including also a test set reduces the training set size by a tenth. The main reason behind the choice of considering also a test set is that we found inappropriate to evaluate the model performance on the same dataset that has been used to select the model according to the epoch that yielded the largest score on it. Doing like so introduces a bias. We should expect performance in real-world to be slightly worse than the performance on the validation set.

3.2 Ideas for future developments

Some ideas that could improve the performance of such a system will be proposed in the next paragraphs. First of all we can try to add more data to our dataset by looking for some medical database with annotated CT scans. As stated initially, the LUNA16 dataset was only a subset of the LIDC/IDRI dataset (CT scans with a slice thickness greater than 3mm were discarded). We could use the excluded CT scans to increase our dataset size. Second, medical papers, such as [1], state some predictors that are typically used to predict about nodule malignancy that we currently do not take into account in our model, such as the patient's age, sex, family history of lung cancer, location of the nodule, size, type (solid, part-solid, nonsolid), number of nodules... The

only visual information is probably not enough if our goal is to build a reliable system which does not miss any malignant tumor.

The annotations about malignancy that we used come from a more general categorization by several doctors. Each doctor assigned one of five classes (highly unlikely, moderately unlikely, intermediate, moderately suspicious, highly suspicious) to each candidate nodule rather than a binary malignant/benign classification. Therefore, we could train a model on the cross entropy between the output and the target distribution using vectors of 5 elements for the targets. In that case we would lose the ROC and AUC notions that we have in binary classification. Another way could be to train multiple models, each one trained on annotations from individual radiologists. At the end, when we perform inference, we could ensemble the models predictions by, for instance, averaging their output probabilities.

Another solution is to find alternative ways to cope with overfitting. Examples of that could be augmentation techniques such as elastic deformations [16] of the images, or label smoothing, where we do not use one-hot vectors to represent the labels but we put a small amount of probability also on the wrong classes.

In conclusion, we want to say that even if the final model is far from a system which could be taken seriously into consideration as an assisting system for doctors, this activity allowed to learn many things about approaching a deep learning project: from setting up and use a cloud service, to handle a large amount of data and cope with things such as caching, to stack together different models into a pipeline, to handle issues such as data imbalance and overfitting and many more aspects.

Bibliography

[1] Annette Mcwilliams et al. "Probability of Cancer in Pulmonary Nodules Detected on First Screening CT". In: *The New England journal of medicine* 369 (Sept. 2013), pp. 910–9. DOI: 10.1056/NEJMoa1214726.

[2] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with PyTorch*. 2020. ISBN: 9781617295263. URL: https://www.manning.com/books/deep-learning-with-pytorch.

[3] *PyTorch Deep Learning Framework (Official website)*. https://pytorch.org.

[4] *LUNA 2016 Grand Challenge*. https://luna16.grand-challenge.org/Description/.

[5] Arnaud Arindra Adiyoso Setio et al. "Validation, comparison, and combination of algorithms for automatic detection of pulmonary nodules in computed tomography images: The LUNA16 challenge". In: *Medical Image Analysis* 42 (). ISSN: 1361-8415. DOI: 10.1016/j.media.2017.06.015.

[6] Samuel G 3rd et al. Armato. "The Lung Image Database Consortium (LIDC) and Image Database Resource Initiative (IDRI): a completed reference database of lung nodules on CT scans." In: *Medical physics* 38,2 (2011), pp. 915–31. DOI: 10.1118/1.3528204.

[7] *MetaIO documentation*. https://itk.org/Wiki/ITK/MetaIO/Documentation.

[8] *SimpleITK Python library*. https://simpleitk.readthedocs.io/en/master/gettingStarted.html.

[9] *Affine Transformations (Wikipedia)*. https://en.wikipedia.org/wiki/Transformation_matrix.

[10] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[11] *U-Net implementation with PyTorch (GitHub)*. https://github.com/jvanvugt/pytorch-unet.

[12] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[13] *Sørensen-Dice coefficient (Wikipedia).* https://en.wikipedia.org/wiki/S\OT1\orensen\OT1\textendashDice_coefficient.

[14] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* 2015. arXiv: 1502.01852 [cs.CV].

[15] *Trapezoidal rule (Wikipedia).* https://en.wikipedia.org/wiki/Trapezoidal_rule.

[16] *Elastic Deformations.* https://github.com/deepmind/multidim-image-augmentation/blob/master/doc/elastic_deformation_colab.md.