Recurrent Residual Convolutional Network based on U-Net (R2U-Net) for Medical Image Segmentation

Cognitive Computing and Artificial Intelligence 2020/2021

Roberto Pillitteri (1000012403)

Davide Lo Presti (1000008936)

Semantic Segmentation

- Semantic Segmentation is an application of CNN, and consists of dividing an image into parts and classifying each pixel to understand which category they belong to.
- It is often used on datasets of medical images, like in this paper.

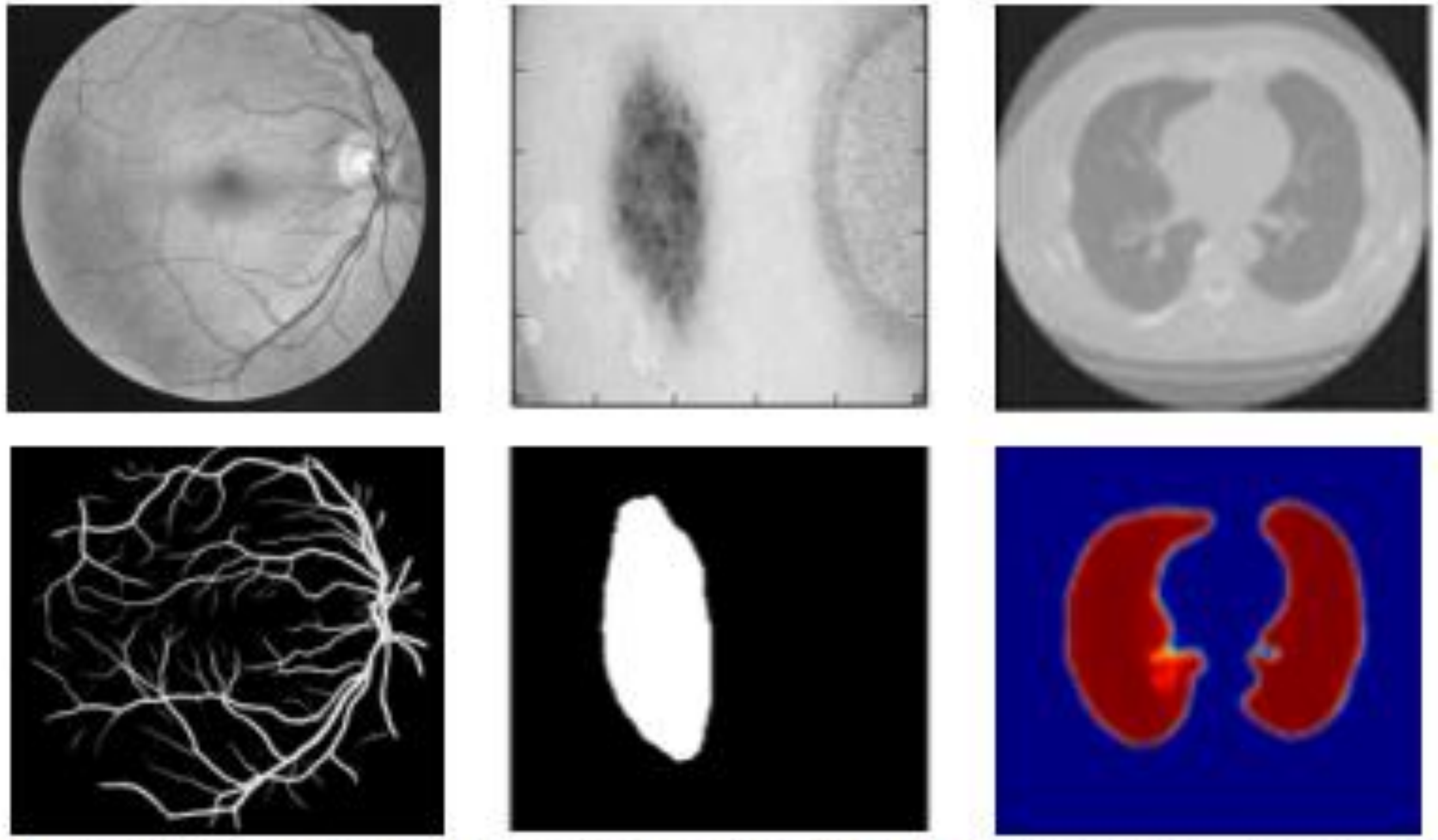


Fig. 1. Medical image segmentation: retina blood vessel segmentation in the left, skin cancer lesion segmentation, and lung segmentation in the right

U-Net

- U-Net is a very common architecture used for image segmentation. It consists of three «paths»:

1. Downsampling path, to reduce image’s size through convolutions;
2. Bottleneck, to process features of images;
3. Upsampling path, to increase image’s size through transposed convolutions.

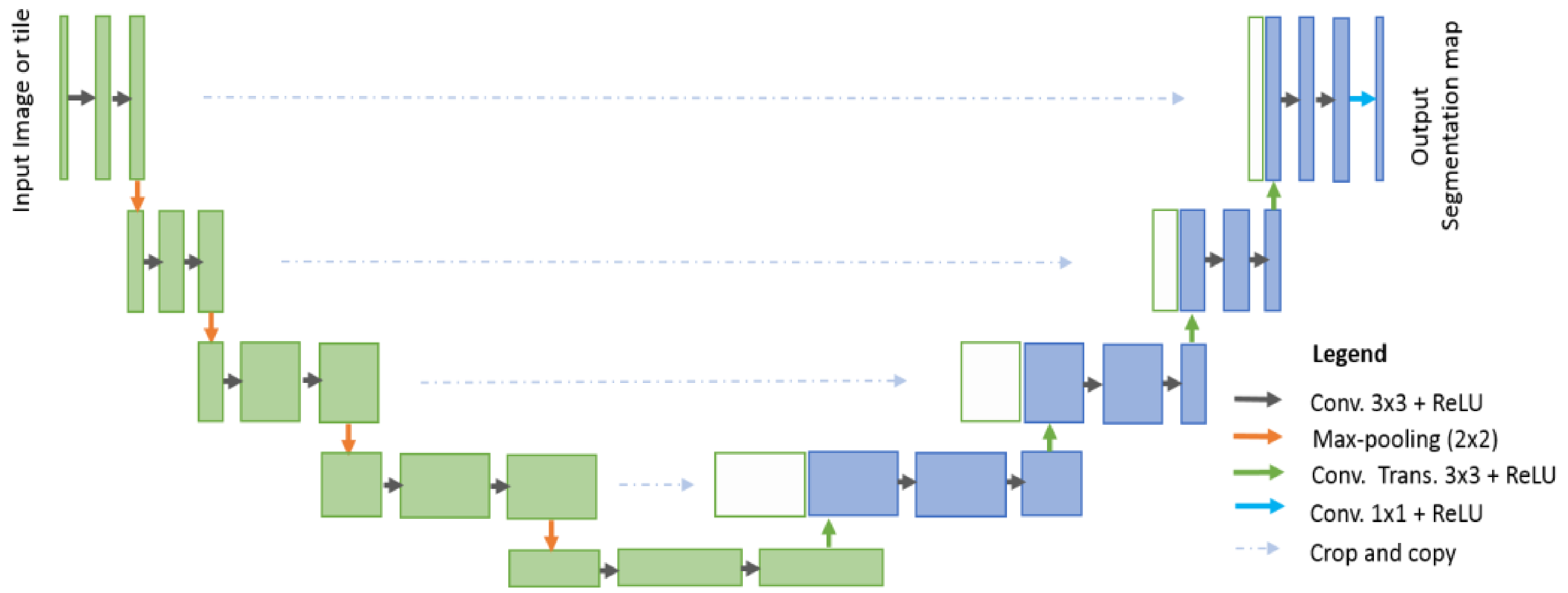


Fig. 2. U-Net architecture consisted with convolutional encoding and decoding units that take image as input and produce the segmentation feature maps with respective pixel classes.

Main Problem and proposed variants

- Vanishing gradient, which can be solved thanks to residual layers: a kind of connection between layers that make it easy to propagate the gradient.
- So, the authors have proposed some variants of the standard U-Net, by adding residual layers and recurrent layers (t=3), which showed better performances.

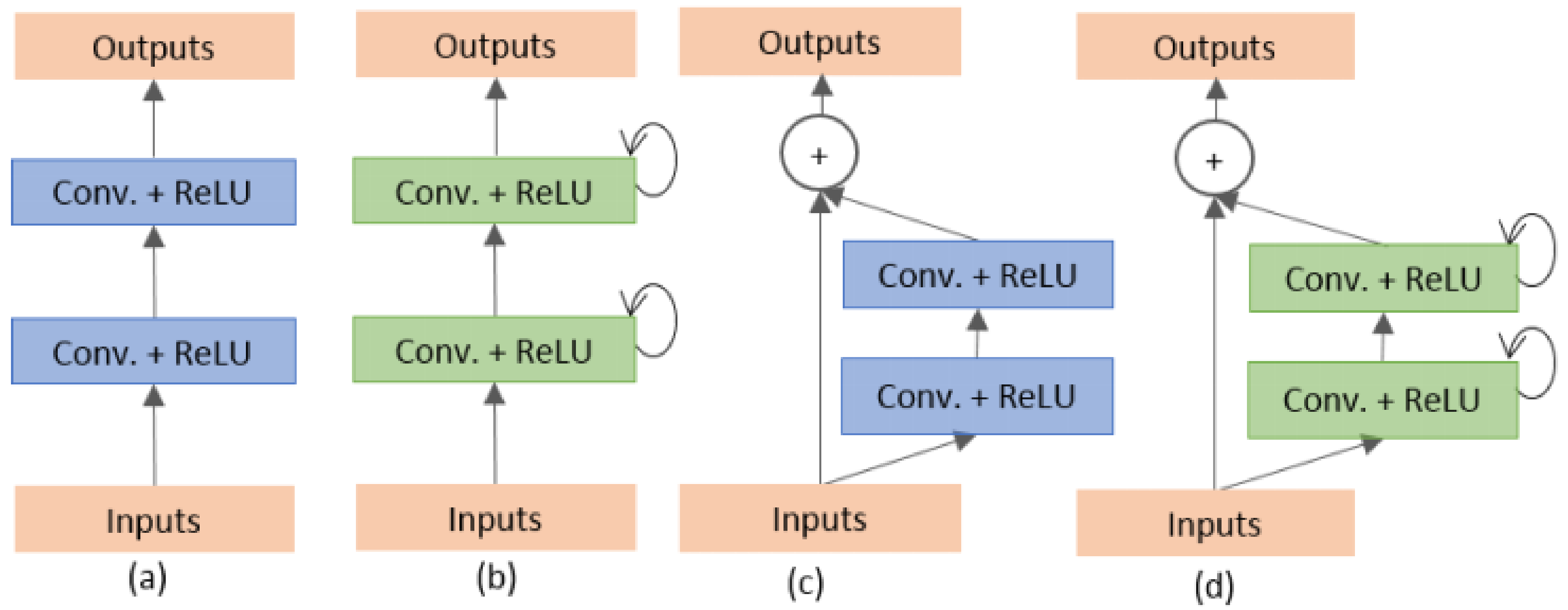


Fig. 4. Different variant of convolutional and recurrent convolutional units (a) Forward convolutional units, (b) Recurrent convolutional block (c) Residual convolutional unit, and (d) Recurrent Residual convolutional units (RRCU).

Datasets

TOPIC	DATASET NAME	DATASET SIZE	IMAGE SIZE	TRAINING-TEST
Blood Vessel	DRIVE	40	565x584	20-20
Blood Vessel	STARE	20	700x605	«leave-one-out»
Blood Vessel	CHASE_DB1	28	999x960	20-8
Skin Cancer	ISIC (2017)	2000	700x900	1600-400
Lung	LUNA16	534	512x512	70%-30%

Quantitative Analysis Approaches

$$AC = \frac{TP+TN}{TP+TN+FP+FN}$$

$$DC = 2\frac{|GT \cap SR|}{|GT|+|SR|}$$

$$SE = \frac{TP}{TP+FN}$$

$$JS = \frac{|GT \cap SR|}{|GT \cup SR|}$$

$$F1 = 2 * \frac{PC * SE}{PC + SE}$$

$$SP = \frac{TN}{TN+FP}$$

$$PC = \frac{TP}{TP+FP}$$

Results of Skin Cancer Segmentation

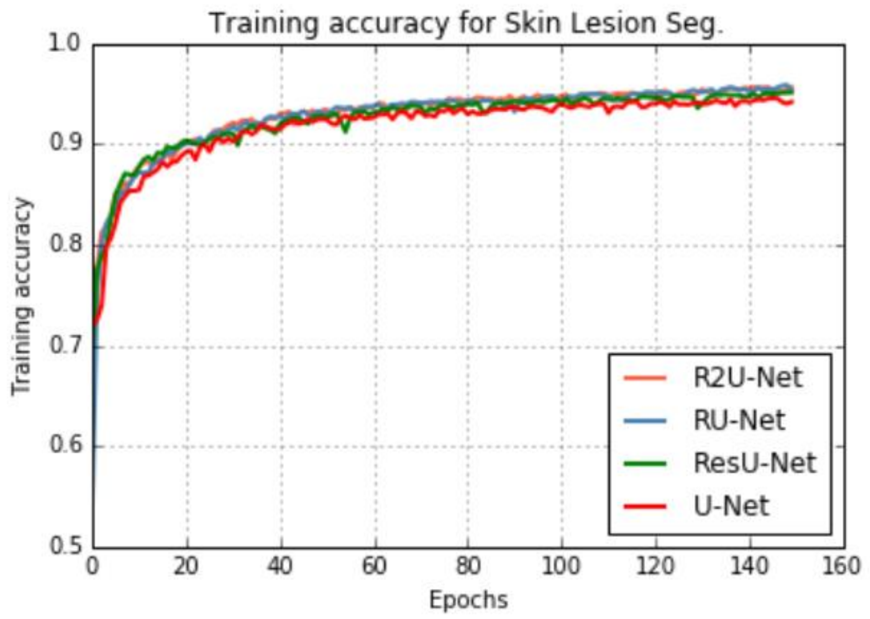


Fig. 16. Training accuracy for skin lesion segmentation.

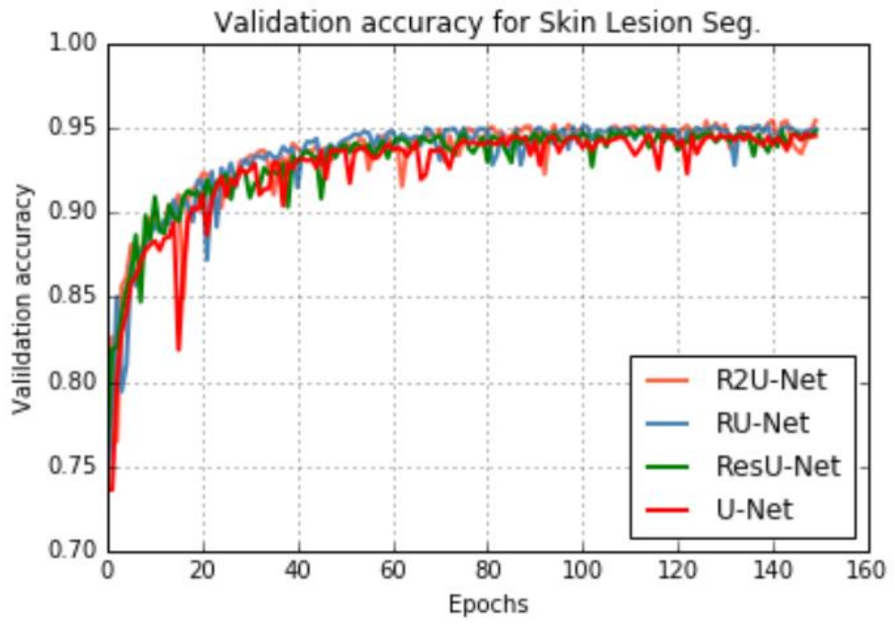
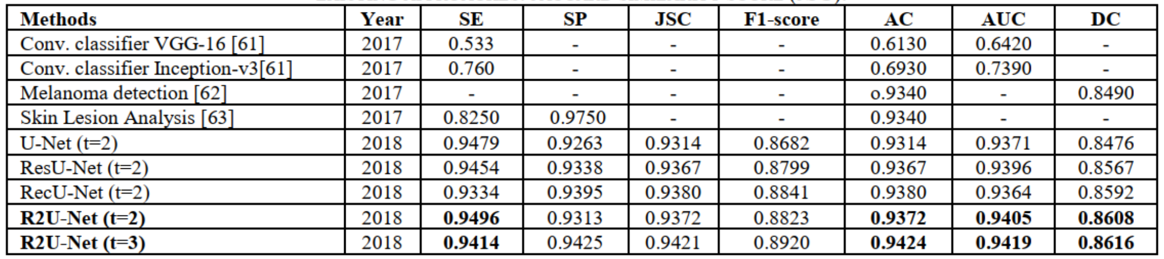


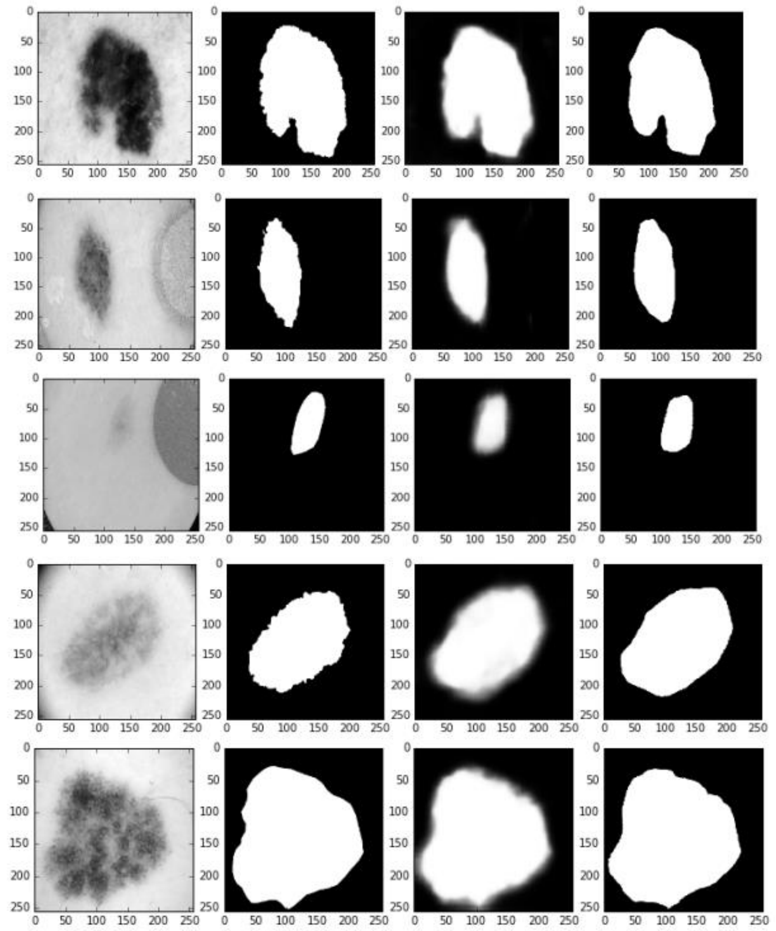
Fig. 17. Validation accuracy for skin lesion segmentation.

Results of Skin Cancer Segmentation

TABLE II. EXPERIMENTAL RESULTS OF PROPOSED APPROACHES FOR SKIN CANCER LESION SEGMENTATION AND COMPARISON AGAINST OTHER EXISTING APPROACHES. JACCARD SIMILARITY SCORE (JSC).



Methods	Year	SE	SP	JSC	F1-score	AC	AUC	DC
Conv. classifier VGG-16 [61]	2017	0.533	-	-	-	0.6130	0.6420	-
Conv. classifier Inception-v3[61]	2017	0.760	-	-	-	0.6930	0.7390	-
Melanoma detection [62]	2017	-	-	-	-	o.9340	-	0.8490
Skin Lesion Analysis [63]	2017	0.8250	0.9750	-	-	0.9340	-	-
U-Net (t=2)	2018	0.9479	0.9263	0.9314	0.8682	0.9314	0.9371	0.8476
ResU-Net (t=2)	2018	0.9454	0.9338	0.9367	0.8799	0.9367	0.9396	0.8567
RecU-Net (t=2)	2018	0.9334	0.9395	0.9380	0.8841	0.9380	0.9364	0.8592
R2U-Net (t=2)	2018	**0.9496**	0.9313	0.9372	0.8823	**0.9372**	**0.9405**	**0.8608**
R2U-Net (t=3)	2018	**0.9414**	0.9425	0.9421	0.8920	**0.9424**	**0.9419**	**0.8616**



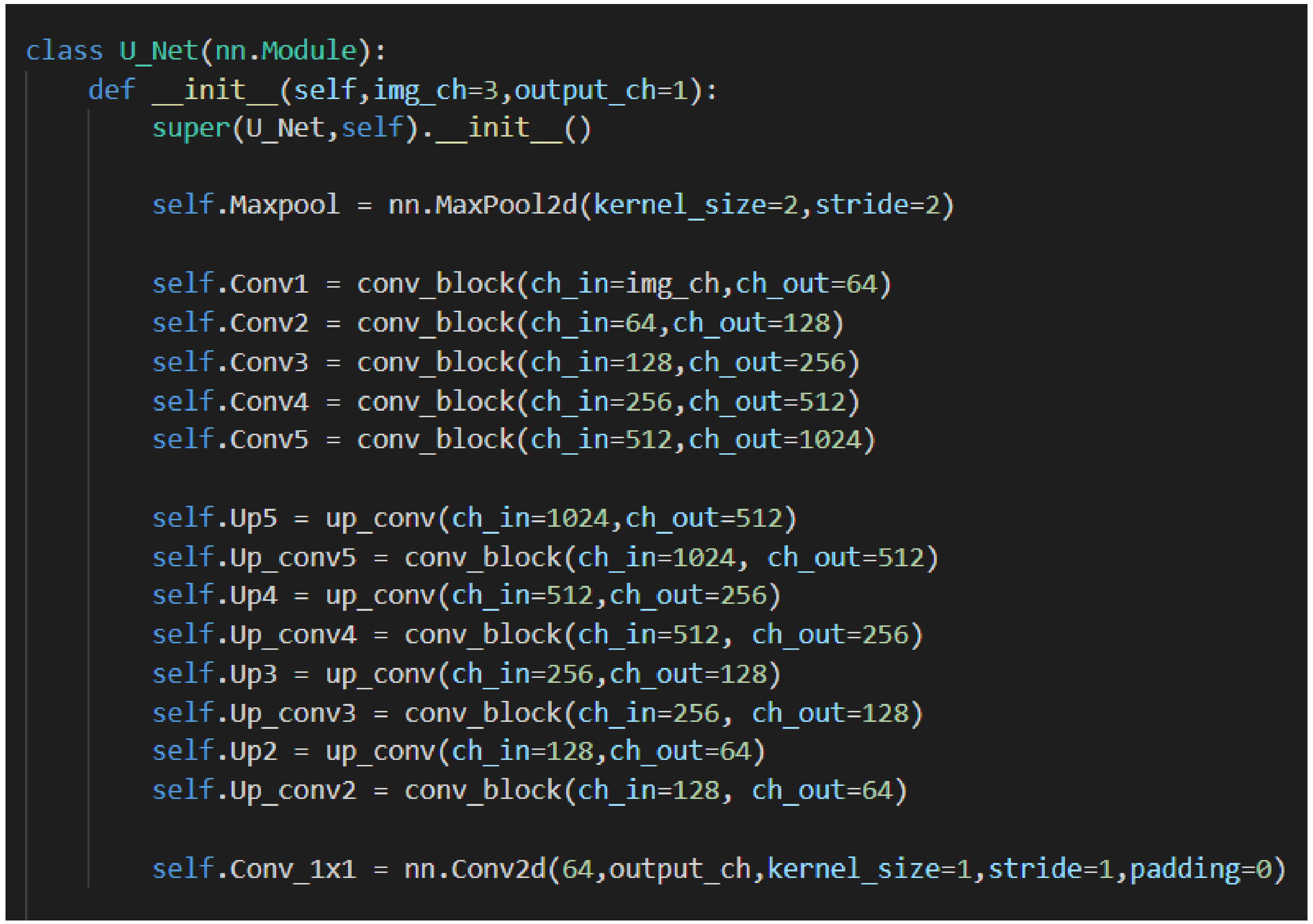
Code

+ network.py: defines the architectures with convolutional, recurrent and residual layers, and creates the classes U_Net and R2U_Net, which use them.
+ data_loader.py: uses the ImageFolder class to organize the dataset and perform resizing and data augmentation, then uses the DataLoader class to perform shuffling and work with mini-batches.
+ evaluation.py: defines all the functions to compute metrics.

Code

+ solver.py: takes the loaders as inputs and performs training on all dataset.
+ main.py: some parameters are configured and calls a Solver, by passing it the loaders.
+ misc.py: implements a progress bar.
+ dataset.py: some parameters are configured and splits dataset into training set, validation set and test set.

Code U-Net



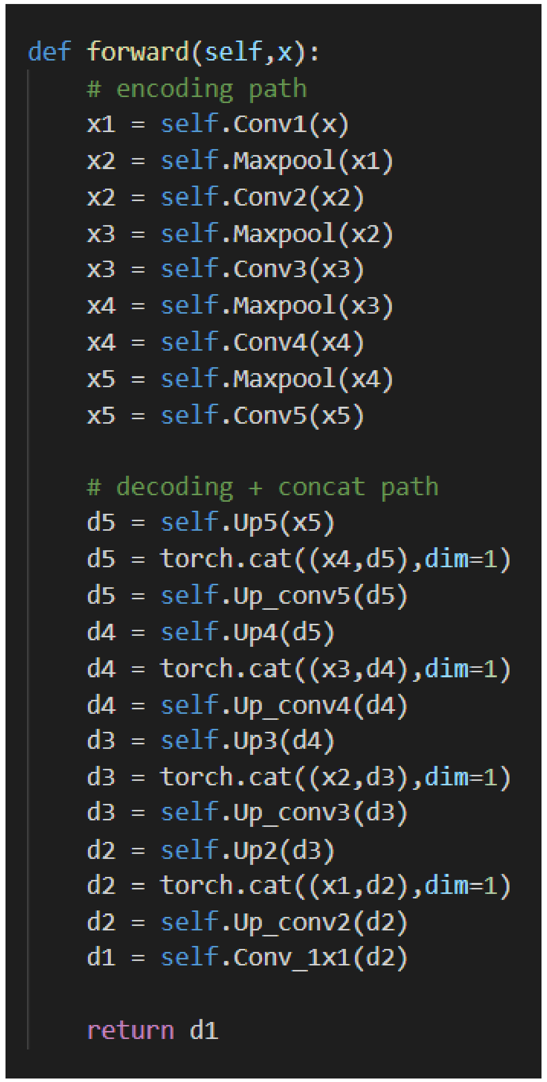
```
class U_Net(nn.Module):
    def __init__(self,img_ch=3,output_ch=1):
        super(U_Net,self).__init__()

        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.Conv1 = conv_block(ch_in=img_ch,ch_out=64)
        self.Conv2 = conv_block(ch_in=64,ch_out=128)
        self.Conv3 = conv_block(ch_in=128,ch_out=256)
        self.Conv4 = conv_block(ch_in=256,ch_out=512)
        self.Conv5 = conv_block(ch_in=512,ch_out=1024)

        self.Up5 = up_conv(ch_in=1024,ch_out=512)
        self.Up_conv5 = conv_block(ch_in=1024, ch_out=512)
        self.Up4 = up_conv(ch_in=512,ch_out=256)
        self.Up_conv4 = conv_block(ch_in=512, ch_out=256)
        self.Up3 = up_conv(ch_in=256,ch_out=128)
        self.Up_conv3 = conv_block(ch_in=256, ch_out=128)
        self.Up2 = up_conv(ch_in=128,ch_out=64)
        self.Up_conv2 = conv_block(ch_in=128, ch_out=64)

        self.Conv_1x1 = nn.Conv2d(64,output_ch,kernel_size=1,stride=1,padding=0)
```

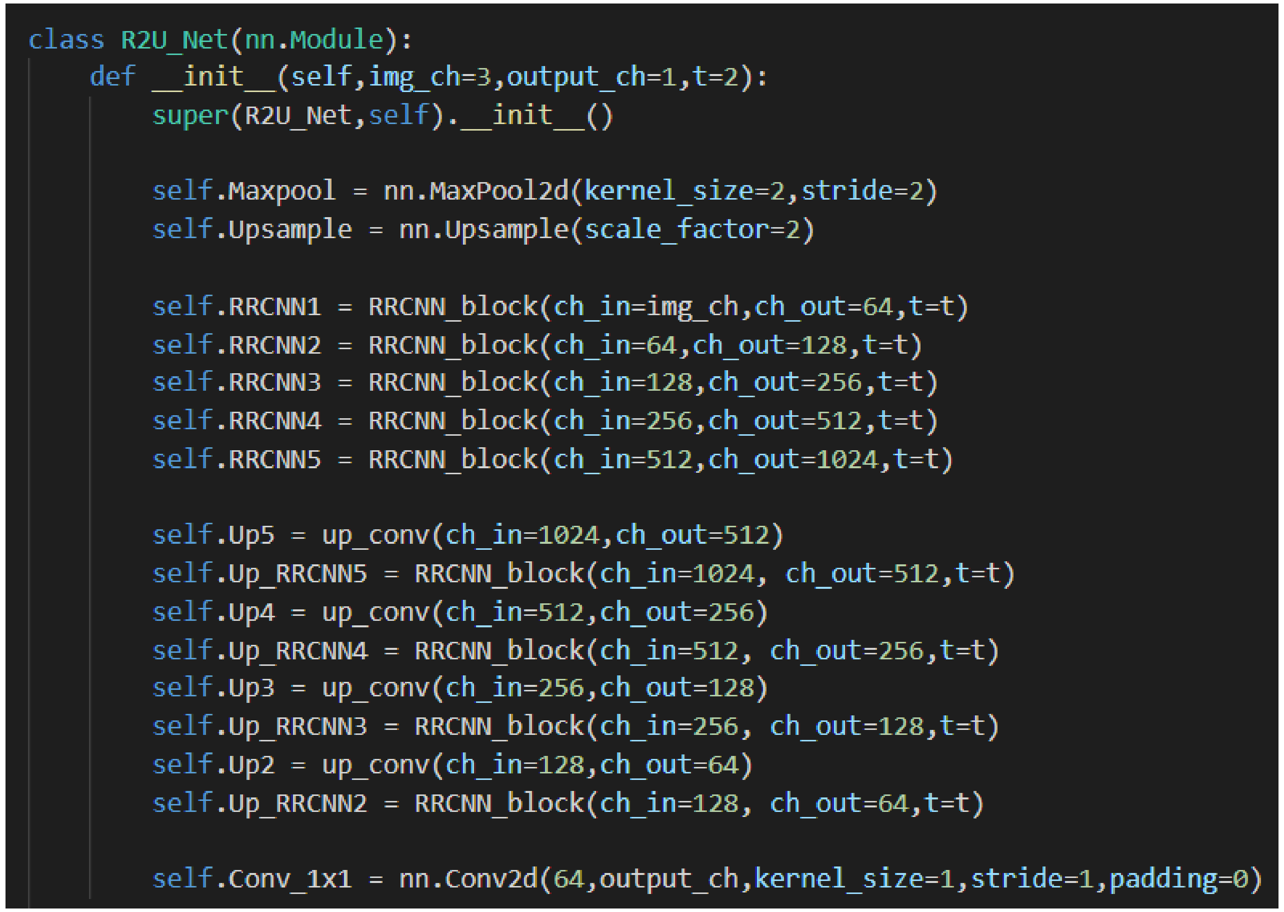


```
def forward(self,x):
    # encoding path
    x1 = self.Conv1(x)
    x2 = self.Maxpool(x1)
    x2 = self.Conv2(x2)
    x3 = self.Maxpool(x2)
    x3 = self.Conv3(x3)
    x4 = self.Maxpool(x3)
    x4 = self.Conv4(x4)
    x5 = self.Maxpool(x4)
    x5 = self.Conv5(x5)

    # decoding + concat path
    d5 = self.Up5(x5)
    d5 = torch.cat((x4,d5),dim=1)
    d5 = self.Up_conv5(d5)
    d4 = self.Up4(d5)
    d4 = torch.cat((x3,d4),dim=1)
    d4 = self.Up_conv4(d4)
    d3 = self.Up3(d4)
    d3 = torch.cat((x2,d3),dim=1)
    d3 = self.Up_conv3(d3)
    d2 = self.Up2(d3)
    d2 = torch.cat((x1,d2),dim=1)
    d2 = self.Up_conv2(d2)
    d1 = self.Conv_1x1(d2)

    return d1
```

Code R2U-Net



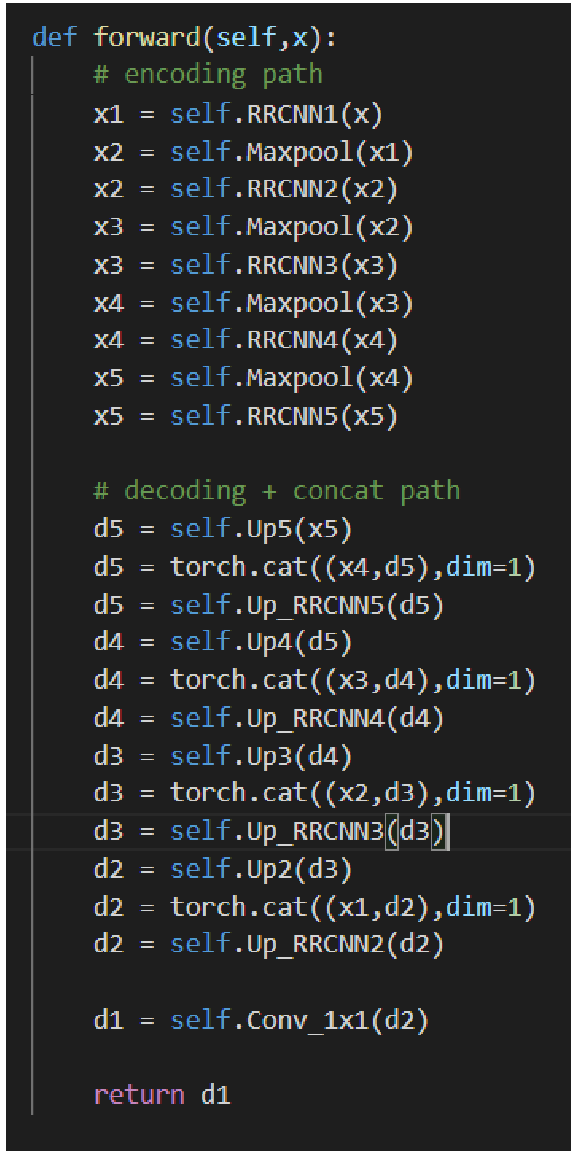
```python
class R2U_Net(nn.Module):
    def __init__(self,img_ch=3,output_ch=1,t=2):
        super(R2U_Net,self).__init__()

        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)
        self.Upsample = nn.Upsample(scale_factor=2)

        self.RRCNN1 = RRCNN_block(ch_in=img_ch,ch_out=64,t=t)
        self.RRCNN2 = RRCNN_block(ch_in=64,ch_out=128,t=t)
        self.RRCNN3 = RRCNN_block(ch_in=128,ch_out=256,t=t)
        self.RRCNN4 = RRCNN_block(ch_in=256,ch_out=512,t=t)
        self.RRCNN5 = RRCNN_block(ch_in=512,ch_out=1024,t=t)

        self.Up5 = up_conv(ch_in=1024,ch_out=512)
        self.Up_RRCNN5 = RRCNN_block(ch_in=1024, ch_out=512,t=t)
        self.Up4 = up_conv(ch_in=512,ch_out=256)
        self.Up_RRCNN4 = RRCNN_block(ch_in=512, ch_out=256,t=t)
        self.Up3 = up_conv(ch_in=256,ch_out=128)
        self.Up_RRCNN3 = RRCNN_block(ch_in=256, ch_out=128,t=t)
        self.Up2 = up_conv(ch_in=128,ch_out=64)
        self.Up_RRCNN2 = RRCNN_block(ch_in=128, ch_out=64,t=t)

        self.Conv_1x1 = nn.Conv2d(64,output_ch,kernel_size=1,stride=1,padding=0)
```



```python
def forward(self,x):
    # encoding path
    x1 = self.RRCNN1(x)
    x2 = self.Maxpool(x1)
    x2 = self.RRCNN2(x2)
    x3 = self.Maxpool(x2)
    x3 = self.RRCNN3(x3)
    x4 = self.Maxpool(x3)
    x4 = self.RRCNN4(x4)
    x5 = self.Maxpool(x4)
    x5 = self.RRCNN5(x5)

    # decoding + concat path
    d5 = self.Up5(x5)
    d5 = torch.cat((x4,d5),dim=1)
    d5 = self.Up_RRCNN5(d5)
    d4 = self.Up4(d5)
    d4 = torch.cat((x3,d4),dim=1)
    d4 = self.Up_RRCNN4(d4)
    d3 = self.Up3(d4)
    d3 = torch.cat((x2,d3),dim=1)
    d3 = self.Up_RRCNN3(d3)
    d2 = self.Up2(d3)
    d2 = torch.cat((x1,d2),dim=1)
    d2 = self.Up_RRCNN2(d2)

    d1 = self.Conv_1x1(d2)

    return d1
```

Strengths and weaknesses

+ Everything treated in the article is supported by a huge number of bibliographic sources that make the argumentations more solid. ✓
+ The language is clear and helps the reader to understand what the topic is talking about. ✓
+ The experimental results demonstrate that the proposed model "R2U-Net" show better performance in segmentation tasks with the same number of network parameters when compared to existing methods including the U-Net. ✓

Strengths and weaknesses

+ There are many inconsistencies in code with regard to the paper. ✕
+ In some parts of the code, it tries to delete some variables without istantiating them first. ✕
+ The evaluation functions were wrong due to a new pytorch version. ✕
+ Some parameters are never taken, and some functions are never called. ✕

Strengths and weaknesses

+ The scripts are not well-organised are written in a confused way, and this makes the reading hard. ✕
+ The code is not documented. ✕
+ Instead of using a script which splits the dataset in three standard subset, they could have used the cross-validation technique. ✕

Our Project assignment

- Due to tecnhical issues, we could not access to all the dataset, so we focused just on skin cancer’s one.
- The task of replicating the results of the paper was very hard, because the authors used a GPU machine with 56Gb of RAM and an NVIDIA GEFORCE GTX-980 Ti, which is out of our league, so we had to reduce the image’s size to 64x64 pixels, the batch size to 1, the number of workers to 0, and the epochs amount to 50.

Our Project assignment

+ To discover the part of the code which take most time, we computed the times of each getitem, each backward and each forward function, for 1 epoch and 1250 samples:

1. Total time forward: 167.51840710639954 sec
2. Average time forward: 0.13401472568511963 sec
3. Total time backward: 304.59891057014465 sec
4. Average time backward: 0.24367912845611572 sec
5. Total time getitem: 88.1659905910492 sec
6. Average time getitem: 0.07053279247283936 sec

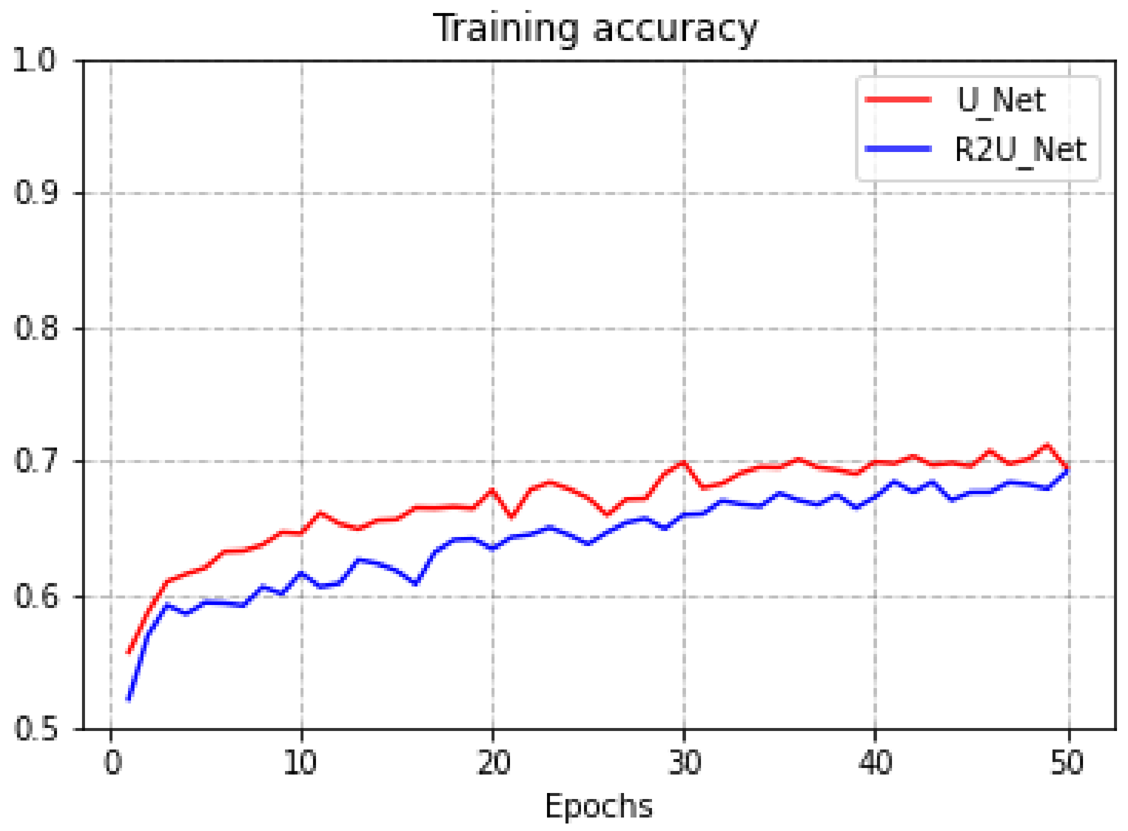
Our Project assignment

+ We experimented with the code and changed something:

1. In the DataLoader function, we shuffle the dataset for each epoch, only if mode is «train»;
2. In the evaluation functions, we fixed the problem of the zero metrics;
3. In the evaluation functions, we tried to decrease the threshold from 0,5 to 0,25.

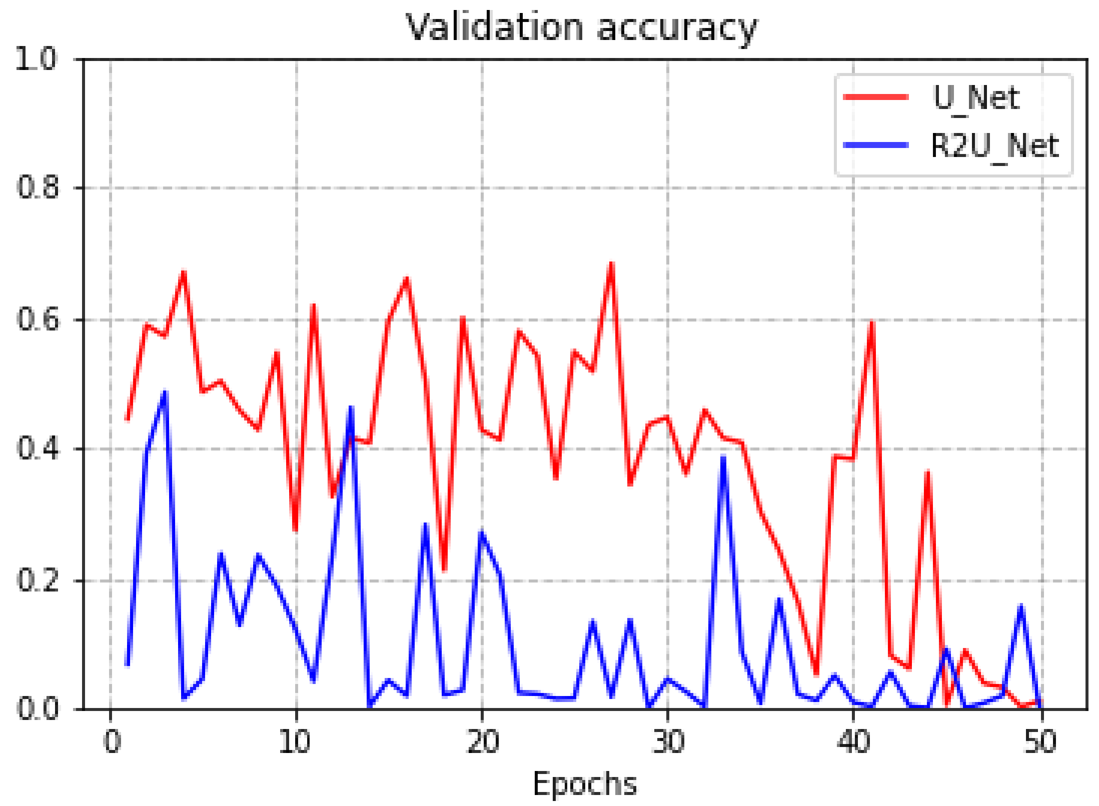
Our results (training accuracy)

Threshold=0.5



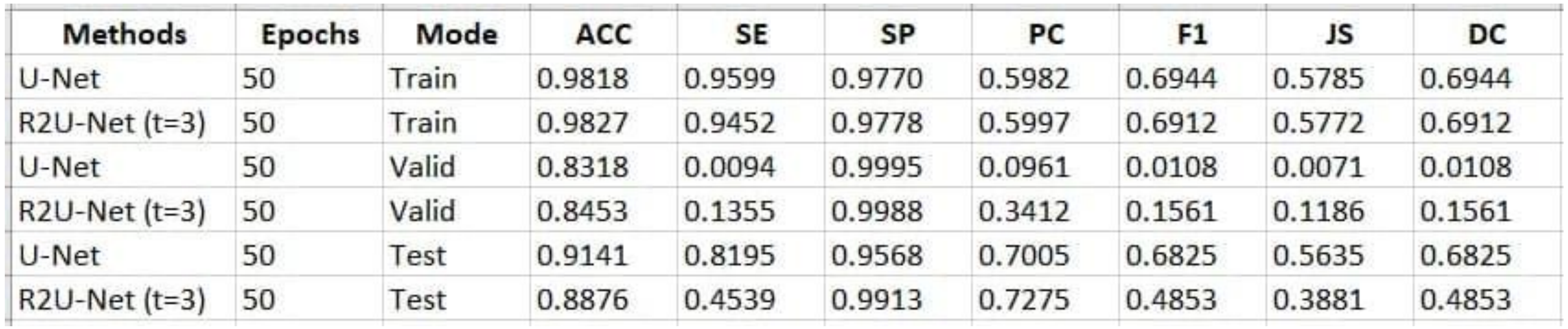
Our results (validation accuracy)

Threshold=0.5



Our results (all metrics at epoch 50)

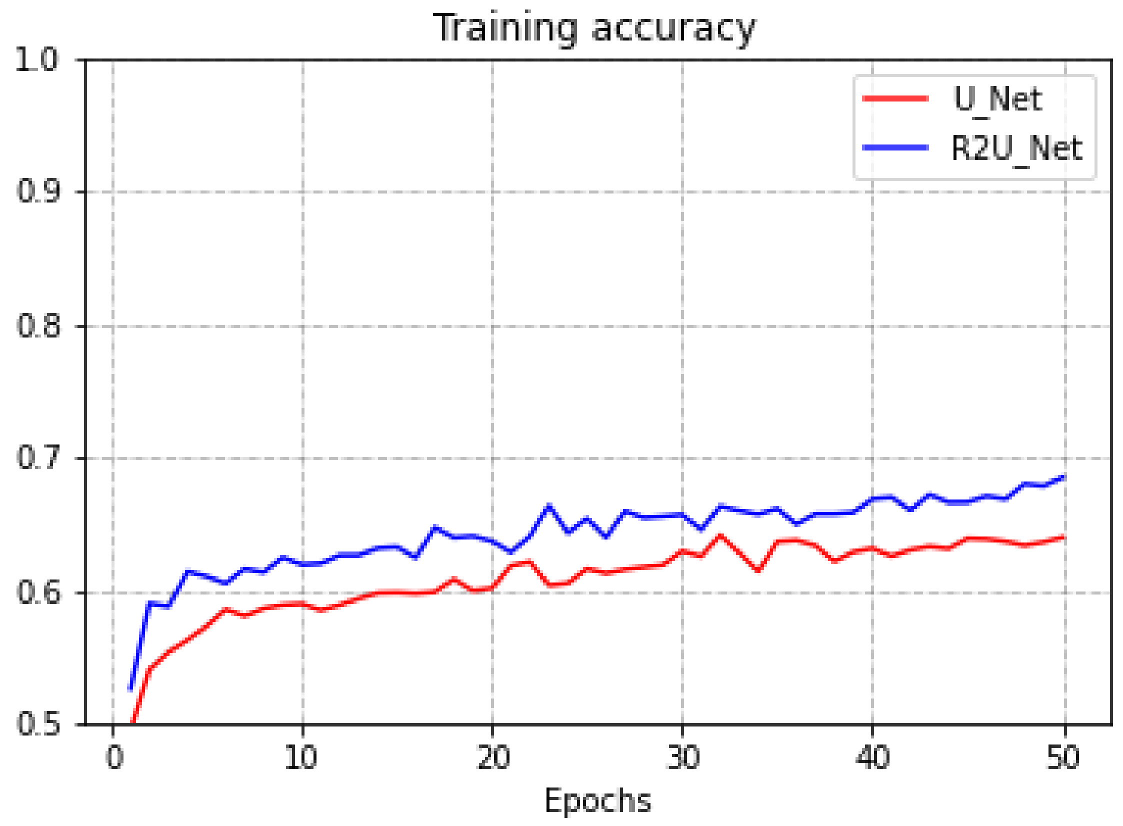
Threshold=0.5



Methods	Epochs	Mode	ACC	SE	SP	PC	F1	JS	DC
U-Net	50	Train	0.9818	0.9599	0.9770	0.5982	0.6944	0.5785	0.6944
R2U-Net (t=3)	50	Train	0.9827	0.9452	0.9778	0.5997	0.6912	0.5772	0.6912
U-Net	50	Valid	0.8318	0.0094	0.9995	0.0961	0.0108	0.0071	0.0108
R2U-Net (t=3)	50	Valid	0.8453	0.1355	0.9988	0.3412	0.1561	0.1186	0.1561
U-Net	50	Test	0.9141	0.8195	0.9568	0.7005	0.6825	0.5635	0.6825
R2U-Net (t=3)	50	Test	0.8876	0.4539	0.9913	0.7275	0.4853	0.3881	0.4853

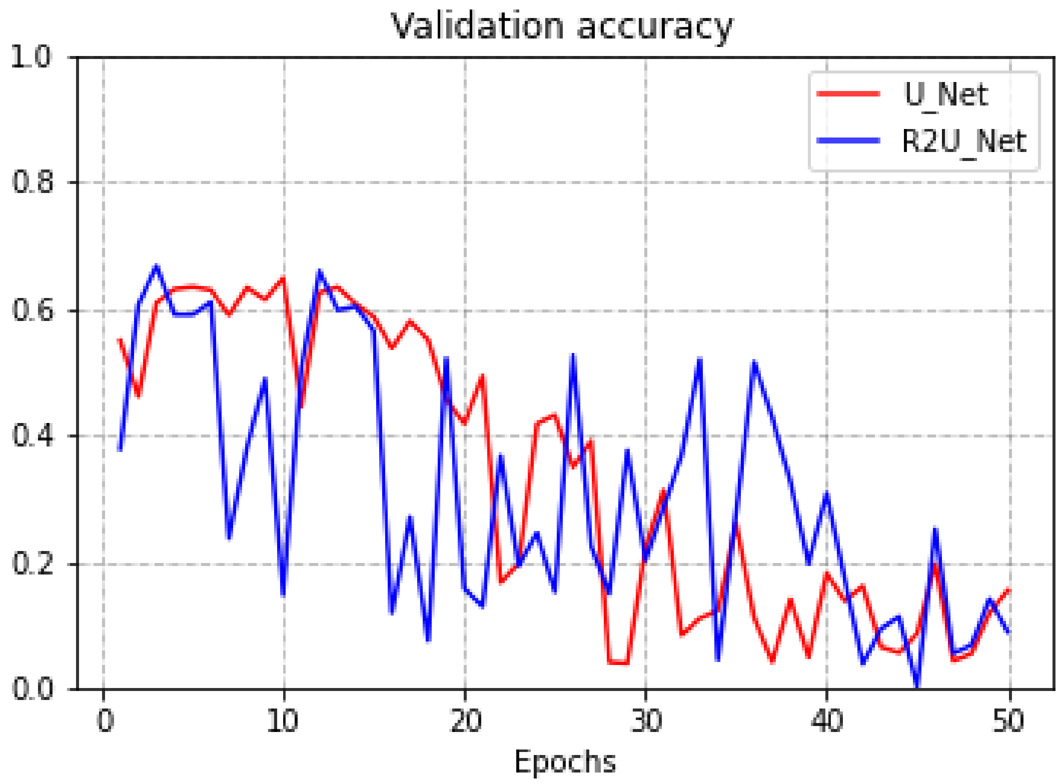
Our results (training accuracy)

Threshold=0.25



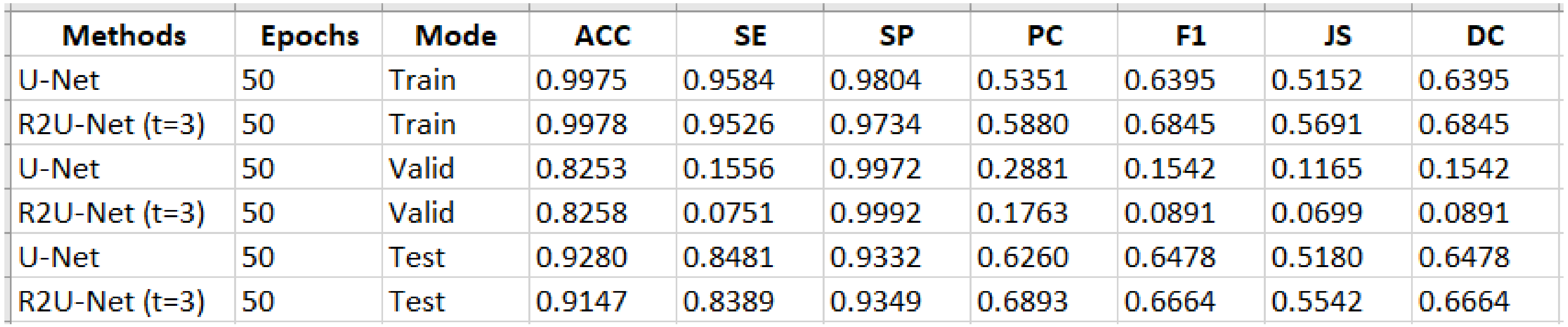
Our results (validation accuracy)

Threshold=0.25



Our results (all metrics at epoch 50)

Threshold=0.25



Methods	Epochs	Mode	ACC	SE	SP	PC	F1	JS	DC
U-Net	50	Train	0.9975	0.9584	0.9804	0.5351	0.6395	0.5152	0.6395
R2U-Net (t=3)	50	Train	0.9978	0.9526	0.9734	0.5880	0.6845	0.5691	0.6845
U-Net	50	Valid	0.8253	0.1556	0.9972	0.2881	0.1542	0.1165	0.1542
R2U-Net (t=3)	50	Valid	0.8258	0.0751	0.9992	0.1763	0.0891	0.0699	0.0891
U-Net	50	Test	0.9280	0.8481	0.9332	0.6260	0.6478	0.5180	0.6478
R2U-Net (t=3)	50	Test	0.9147	0.8389	0.9349	0.6893	0.6664	0.5542	0.6664

Conclusions

- It is noted that R2U_Net model should be better than U_Net model, but it is not our case because we made several simplifications.
- After downgrading the threshold, we can see better performances in validation accuracy (expecially for R2U_Net), but worse performances in training accuracy.

Conclusions

- In conclusion, our results do not match exactly the ones shown in the paper, we would have liked to get as close to them, but with what we had, we did everything we could, at the cost of the quality of images.
- Despite this, we enjoyed working with these topics in a pratical context, which is the field of medicine.