AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering

Automatic segmentation of kidney tumor computed tomography images

SUBJECT - MEDICAL IMAGING TECHNIQUES

Karolina Barczyk
Anna Ceglarz
Justyna Tokarz
Natalia Wasik

Kraków, 22th June 2020

Table of contents

1. Introduction

The goal of the project was automatic segmentation of cancer and kidneys in computed tomography images. Data was taken from https://github.com/neheller/kits19. There were 300 random cases. The imaging, as well as ground truth labels, were provided in a NIFTI format.
Initial approach for handling this problem was using growing region method. After some complications with defining the homogeneity criterion, we decided to focus on convolutional neural network-based U-net.
Our main objective was to create code using U-net which will train cases and show results by loss of major values. In the paper, every step, main methods and significant algorithms that were implemented during the coding process to achieve the target will be presented.

Code can be found at Github: https://github.com/natalie-rgb/teamTOM named:
UNET_PROJECT_TEAM9_TOM.ipynb

2. Stages and methods

In the project Tensor Flow and PyTorch libraries were mainly used. We have loaded data from Github repository - folder kits19. To do this, we used notebook colab because it allows us to fit such large files.

2.1. Pre-processing

We made visualization of three classes - kidney, tumor, and background. For this, we used data from imaging.nii.gz and segmentation.nii.gz. SegmentationPari2D class build 2D segmentation datasets and represent two data volumes, input data (images) and masks. The result is presented in Fig. 1. Secondly, we loaded needed libraries for performing segmentation using deep learning techniques. We have imported classic libraries like NumPy or Matplotlib and also Torch which provides a very wide range of algorithms needed for deep learning.
Next, we have read data paths for processing and getting cases numbers in file_name variable:

```
train_path = "/content/kits19/data/"
cases =  next(os.walk(train_path))
file_name = cases[1]
```

Then we split data for training and validation. We wanted first 240 cases to training and 60 cases to testing:

```
train_data = file_name[:240]
validation_data = file_name[240:]
```

Next step was to create transform function which aims:

a) Change the format from .nii to an array (float type)

b) Normalize the pixel values so that each pixel has value between 0 and 1
In next functions, we also change dimensions of labels and images to one specific 244x244 using the resize function from the NumPy library (without changing this, dimension would be different in every case, so that would disturb our compilation)

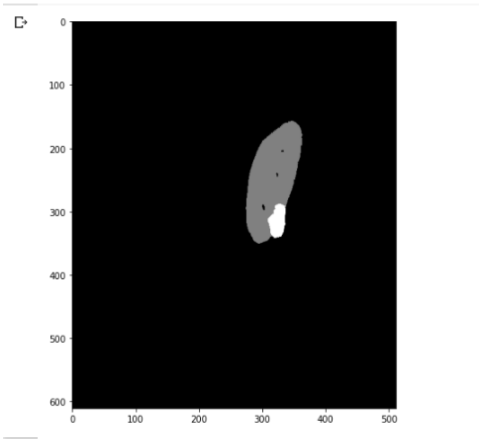


Fig. 1. Visualization of kidney and tumor.

2.2. Modeling

It is implemented from medicaltorch library by code:

```
model = mt_models.Unet(drop_rate=0.4, bn_momentum=0.1
```

U-net is a semantic segmentation technique widely used in medical image processing.
It is mainly based on encoder-decoder architecture. Each encoder is composed of two convolutions layers with 3x3 filter size to extract several feature vector from the image, each followed by ReLU Activation function and a 2x2 map of a max-pooling layer with stride size of 2 for down-sampling to decrease the size of the image and extract feature till pixel level. The purpose of encoder is to decrease the size of the image using the pooling layer. Decoder gives back the original size of the image with pixel-wise feature using the up-sampling layer. U-net is balanced and skips connections between the down-sampling path and the up-sampling path apply an integration operator in lieu of sum (unlike FCN-8). It provides local information to the global information while up-sampling [1].

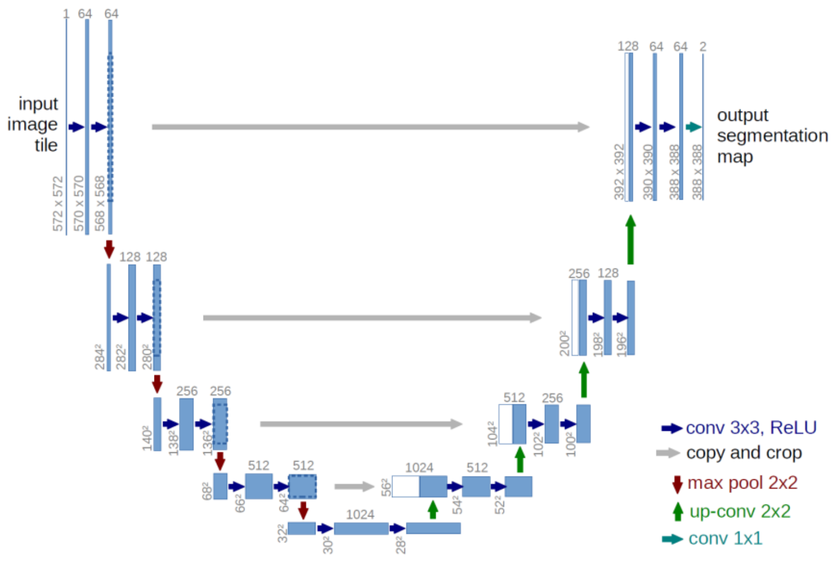


Fig. 2. U-net architecture [1].

2.3. Declaring classes for data loaders

In order to train the model and proof its efficacy downloaded data were divided into two categories - *train_dataset* containing 240 cases and *valid_dataset* containing 60 cases left. As names may suggest, first group was used for training, whereas second for verification of the already trained model. Subsequently classes *TrainDataset* and *ValidDataset* were created, both of which responsible for loading the data (train_dataset and valid_dataset respectively) and consists of three functions. First function, (`__init__(self, df)`) is the initialization function and second (`__len__(self)`) returns the length of the object. The main part of loading data is being held in the third function (`__getitem__(self, idx)`). First, we extract data paths, marking them as `input_filename`, which stands for input image path and `gt_filename`, which stands for ground truth mask path. Next they are paired using module SegmentationPair2D.
Afterwards image and mask are being transformed (specifically cast to float32 and normalized) with function declared earlier. Eventually mask and input image are resized, converted to tensors and added dimension using `torch.from_numpy(img).unsqueeze_(0)` to fit the expectation of used model.

2.4. Preparing for training and important values

For the training algorithm dataset of 300 images, it is important to choose necessary values like batch, epoch, learning rate. Batch size defines the number of samples to work processed before the model is updated. Epochs defines number of times that algorithm will work through all training dataset [2]. In project epoch size was 10, batch size was 20. We also added the number of workers which is a parameter of DataLoader. Instead of loading one batch at a time, workers speed-up batches at a time. Code in which batch and number of workers were implemented:

```
train_loader = DataLoader(train_dataset, batch_size=2, num_workers=0)
```

To improve and speed up our training but with a reduced risk of oscillating, we have mplemented momentum [3]. We implemented this value during creating net:

```
model = mt_models.Unet(drop_rate=0.4, bn_momentum=0.1)
```

During optimization was used learning rate specified as “lr”. This value controls how much each weight and bias is changed in each iteration of training. It has a small value between 0.0 and 1.0. In our project, it has value lr = 0.01. Too small value can cause a long process of training, but too fast training rate could cause instability because it would be learning too fast [2].

Before constructing optimizer we used .cuda() because of GPU in devices. If somebody have CPU it also will work because of code:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

We implemented PyTorch version of weight decay Adam optimizer which has many advantages like it easy to implement, computationally efficient and achieve good result fast.

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

We using also binary cross entropy with logits:

```
criterion = nn.BCEWithLogitsLoss()
```

2.5. Dice loss

The dice loss is a continuous approximation of the well-known dice coefficient. In our code, we define Dice Coefficient Class which contains 2 functions - one for individual examples and one for batches.

Dice loss function is based on the Dice coefficient function. Dice coefficient function is essentially a measure of overlap between two samples. This measure ranges from 0 to 1 where a Dice coefficient of 1 denotes perfect and complete overlap.

Basing on value Dice coefficient we can get the value of Dice Loss, because they are related with the formula:

Dice_Loss = 1 - Dice_coefficient

Our aim is to reach as low as possible value of Dice Loss. Using created class we can use them to visualize the total loss during training and testing processes and loss for the individual case. In our case, we calculated the dice loss for each batch and averaged the results, overall batches. This way, we were able to naturally take into account the class imbalance without adding a class weighting. With these two loss functions, we were able to achieve satisfactory results.

2.6.Training and validation

In machine learning we try to generate a model to prefigure the test data because of tendency to over-fitting the data. It can cause that the model would seem very effective during training stage but in fact it would not cope with new data. To handle it, dataset is divided into two smaller datasets. The training set is implemented to fit the model - the model learns from this data, whereas validation set is to validate the model built - it is used to evaluate a given model.
This part begins with for loop responsible for iterating through epochs, which starts with resetting values of variables such as *train_loss_total, valid_loss_total, tot, and tot_train* (used for storing values of loss in each epoch). Inside of it there are two nested loops, iterating adequately through training and validation data sets. Firstly, images and labels are converted to float type and sent to CPU, outputs are pass to model. Next a backpropagation is being done, which will allow to make a change in the weights of a neural network. After that, losses and their gradients are being calculated and sent to the optimizer. Dice coefficient is calculated as evaluation metrics. In the case of validation data we just passing data to model and we are calculating loss.
We use 0.5 as a threshold (if the output is larger than 0.5 input will be assign to class 1, otherwise to class 0):

```
pred = (outputs > 0.5).float()
```

Received values (train loss value, dice losses for train and validation data) are saved in history and displayed by function *DataFrame* from pandas library.

3. Results and Analysis

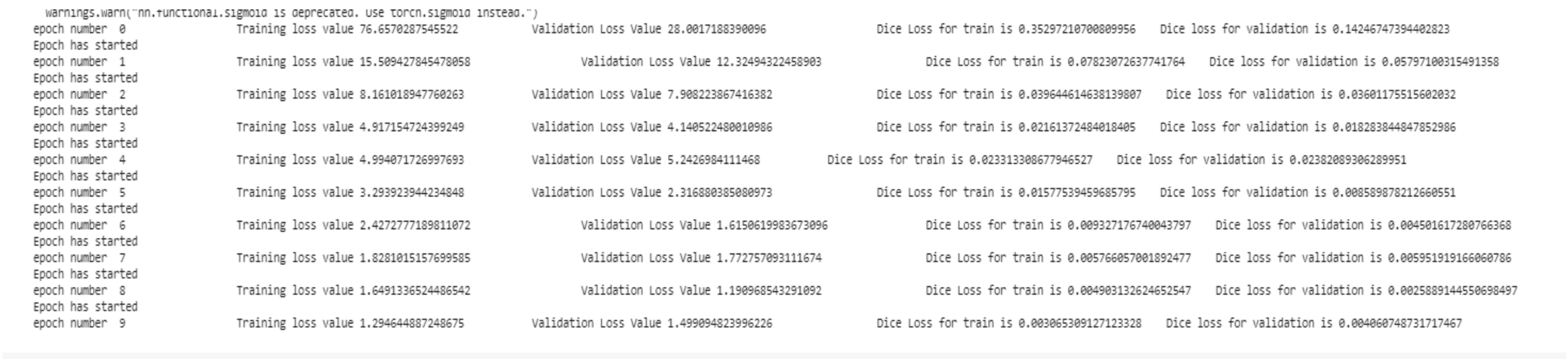
Analyzing our results shown below (Fig. 3.), we can notice that after each epoch started generally, all values of loss seem to decrease, except for a few cases where loss value is slightly increasing. This means that in general training works because each time the differences are smaller so that means our training is more and more accurate.

Viewing our loss values, we can observe that value for epoch number 0 is always the biggest comparing to subsequent epochs numbers, which is the result of successful training. The largest loss value is for Training loss value - for that parameter, the value of loss is decreasing the most significantly. This is a very good sign - if training loss would be lower than validation loss then it would mean the network might be overfitting. For *Dice loss for train and for validation* values of loss are from the beginning very tiny and also decreasing continuously.

The plots (Fig. 4.) we obtained present change training and validation losses and average accuracy depending on number of epoch. We can observe exponentially decrease in the first plot. Training loss value is much bigger than validation value but at epoch 2 they have similar value and they decreasing slowly. This is a good sign of well-trained data.

In the second plot, we can see average accuracy. It was noticed that values are decreasing.
It should measure how accurate our model's prediction compares to true data. Accuracy should be increasing. The reason for this situation could be over-fitting of the model on the training data. It is a good sign that dice loss for valid is lower than dice loss for training.

After visualizing the images we can see some differences between output image and prediction image. The second one has more details in it and probably shows tumor. Ground truth can be blurred because of too few epochs. It is not what we expected. Improving hyperparameters, number of epochs or hours of training could cause better effects.
Below we can see results for 10 epochs:



```
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.")
epoch number  0              Training loss value 76.6570287545522          Validation Loss Value 28.0017188390096              Dice Loss for train is 0.35297210700809956    Dice loss for validation is 0.14246747394402823
Epoch has started
epoch number  1              Training loss value 15.509427845478058              Validation Loss Value 12.32494322458903              Dice Loss for train is 0.07823072637741764    Dice loss for validation is 0.05797100315491358
Epoch has started
epoch number  2              Training loss value 8.161018947760263          Validation Loss Value 7.908223867416382              Dice Loss for train is 0.039644614638139807    Dice loss for validation is 0.03601175515602032
Epoch has started
epoch number  3              Training loss value 4.917154724399249          Validation Loss Value 4.140522480010986              Dice Loss for train is 0.02161372484018405    Dice loss for validation is 0.018283844847852986
Epoch has started
epoch number  4              Training loss value 4.994071726997693          Validation Loss Value 5.2426984111468          Dice Loss for train is 0.023313308677946527    Dice loss for validation is 0.02382089306289951
Epoch has started
epoch number  5              Training loss value 3.293923944234848          Validation Loss Value 2.316880385080973              Dice Loss for train is 0.01577539459685795    Dice loss for validation is 0.00858987821266055l
Epoch has started
epoch number  6              Training loss value 2.4272777189811072              Validation Loss Value 1.6150619983673096              Dice Loss for train is 0.009327176740043797    Dice loss for validation is 0.004501617280766368
Epoch has started
epoch number  7              Training loss value 1.8281015157699585              Validation Loss Value 1.772757093111674              Dice Loss for train is 0.005766057001892477    Dice loss for validation is 0.005951919166060786
Epoch has started
epoch number  8              Training loss value 1.6491336524486542              Validation Loss Value 1.190968543291092              Dice Loss for train is 0.004903132624652547    Dice loss for validation is 0.002588914455069849 7
Epoch has started
epoch number  9              Training loss value 1.294644887248675          Validation Loss Value 1.499094823996226              Dice Loss for train is 0.003065309127123328    Dice loss for validation is 0.004060748731717467
```

Fig. 3. Training results**.**

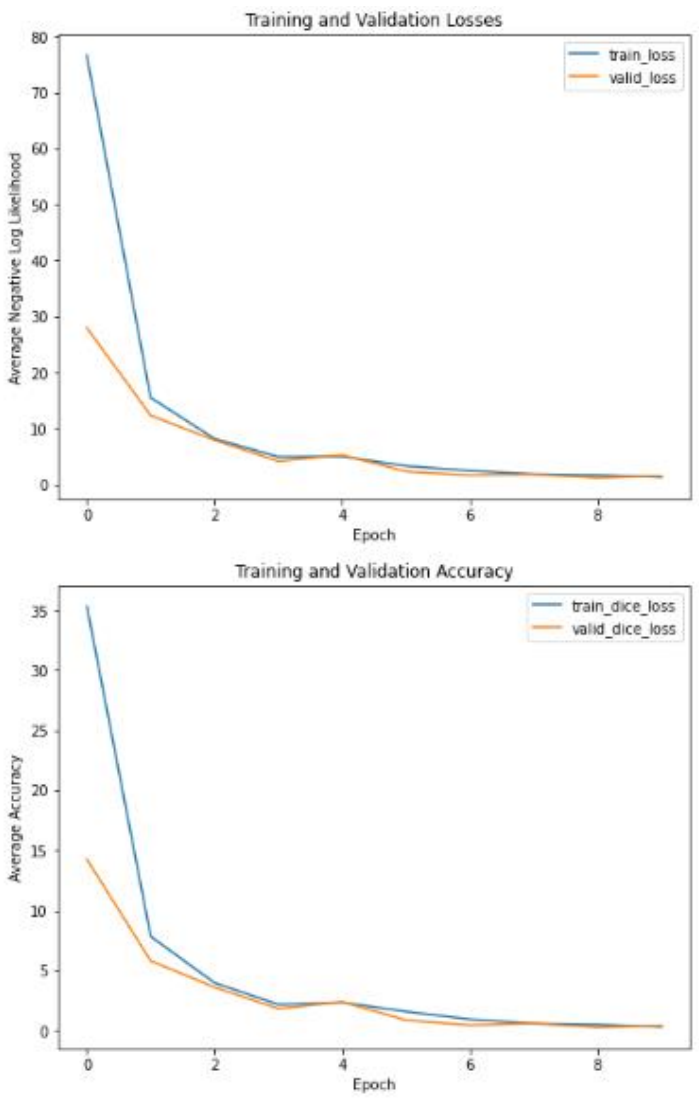


Fig. 4. Plots of Training and Validation losses and accuracy.

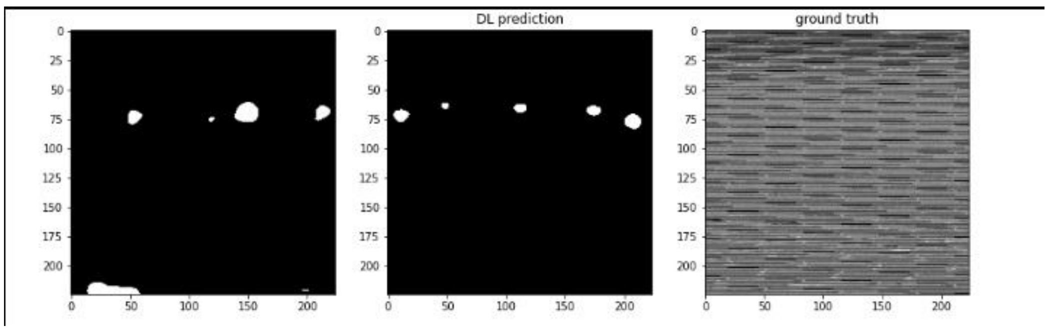


Fig. 5. Images of output, prediction and ground truth.

4. Discussion and Summary

The results are dependent from factors such as number of epochs or value of lr (learning rate). To get well-presented results we should properly fit these values.

The rate of learning should be between 0 and 1 and there is no one, good value for all models. In our project, we set up lr=0,001, but better results were obtained when lr=0,01 so this is the final value for learning rate in our training.

The number of epochs is the number of learning cycles and that parameter is very important in the fitting model well and minimalizing the loss value. The number of epochs should be as high as possible to achieve error rate minimum. Increasing number of epochs after that can cause overfitting. In our project, we tried different values of epochs starting from very low equal to 5 and then continuously increasing to stop at value 10, which gave as good results and low loss values.

To sum up U-net network helped us to perform automatic segmentation kidney tumor however our results are not perfect and there is still some things that could be improved. Maybe our network needs more time to train or our number of epochs is too small, maybe it could be bigger but due to the huge size of data, it was not easy.

Generally except this average accuracy plot and inaccurate images we managed to train data correctly with decreasing values of losses' results.

5. Division of work

It is important to notice that most of the project was done using colab notebook due to the fact that it was easier for us to implement and test different solutions on it. Also, it allowed us to easily share different versions of the project and to stay updated.
Because of that, activity in Github repository does not reflect our real work-sharing.

Barczyk Karolina
part of code: visualizing results (images) and three classes of case
part of the project: Stages (2.6. Training and Testing)

Ceglarz Anna
part of code: Diceloss, Implemented model
part of the project: Stages (2.5. Dice coefficient/dice loss), data analysis, discussion and summary, neatly realization of code and project, articles research

Tokarz Justyna
part of code: classes TrainDataSet/ValidDataSet, loading data(train_loader, valid_loader), criterion, optimizer, handling errors
part of the project: Stages (2.3. Declaring classes for dataloaders), article research

Wasik Natalia
part of code: part of training and testing(instruction for), handling errors, transform function
part of the project: Stages (2.1.Preprocessing, 2.2.Modeling, 2.4.Preparing for training/important values), article research, Introduction, Data analysis

6. Bibliography

[1] https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

[2] https://machinelearningmastery.com

[3] https://jamesmccaffrey.wordpress.com/2017/06/06/neural-network-momentum/

[4] https://pytorch.org/

[5] https://miccai-sb.github.io/materials/U-Net_Demo.html#refs

[6] https://medium.com/dair-ai/medical-imaging-analysis-mri-cnn-pytorch-4877e64e7303

[7] https://discuss.pytorch.org/t/relation-between-num-workers-batch-size-and-epoch-in-dataloader/18201

[8]
https://medicaltorch.readthedocs.io/en/stable/modules.html#medicaltorch.datasets.SegmentationPair2D.get_pair_slice

[9] https://medium.com/ai-salon/understanding-dice-loss-for-crisp-boundary-detection-bb30c2e5f62b

[10] https://dida.do/blog/semantic-segmentation-of-satellite-images

[11] https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/