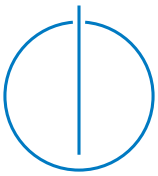
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Interdisciplinary Project Report

Kidney Segmentation in CT scans using QuickNAT

Author: Joana Halili
Supervisor: Prof. Dr. Nassir Navab
Advisor: Dr. Thomas Wendler, Christina Bukas, MSc., Dr. Johannes Oberreuter
Submission Date: July 9, 2020



Abstract

Determining the location and shape of kidneys in CT scans can be extremely useful in many medical procedures. One such procedure is brachytherapy, where the segmentation of the kidneys helps determine the correct amount of radioactivity that can be injected into the spine without damaging it. To estimate the kidneys' location and shape in CT scans we segment the CT scans using QuickNAT. Originally used as a brain segmentation network, QuickNAT has already shown promising results in other medical segmentation tasks, such as thyroid segmentation [3]. In this project, we adapt the network and its hyper-parameters to optimize the network's performance for the task of kidney segmentation in CT scans.

Contents

1 Introduction

Kidney segmentation from dynamic, contrast-enhanced computed tomography (CT) is of immense importance for many computer-assisted procedures, such as pathological tissue localization, radiotherapy planning, and brachytherapy [2]. Brachytherapy is a procedure that involves injecting radioactive material inside the body. For injections in the lower spine, such as the case of patients suffering from spine metastases, it is of paramount importance to accurately estimate the dose of the needed radioactive material. Higher doses of radiation could damage nearby soft tissue organs such as kidneys or the spinal cord. Furthermore, the determination of the location and shape of the kidneys may permit comprehensive radiation dose analysis in a manner that would often be prohibitively time consuming using conventional methods.

In recent years, there have been strong developments of computer-assisted tools to help clinicians in tedious and time-consuming organ segmentation tasks. Nevertheless, organ segmentation copes with challenges emerging from inherent hardware acquisition noise, imaging artefacts, patients and vendor variability, etc. More specifically, kidney segmentation on CT imaging presents challenges due to the similarity of the organs' anatomical structure in the inferior- and superior- most slices with other abdominal organs, in both shape and physical density.

In this project, we tackle the kidney segmentation task by training a fully convolutional network on abdominal CT imaging data. The network architecture used is QuickNAT [10], a network originally developed for brain segmentation on MRI data. QuickNAT has shown promising results in other organ segmentation tasks and other modalities, such as thyroids in 3D ultrasound [3], but to our knowledge, it has not yet been used to segment kidneys in CT scans.

2 Related work

Several methods for kidney segmentation in CT scans have been investigated.

Recently, with the developments in supervised learning, some approaches have tried solutions using algorithms such as random forests. However, random forests approaches heavily depend on the initial selection of the input features [13].

Another attractive solution is that of using convolutional neural networks, as in the approach from [13], which trained two different types of fully convolutional networks, a coarse and a fine network. The two networks differ in that the first one processed the input image directly and after a few convolutional and maxpooling layers produced an output that was 4 times smaller than the original input size. The output then was upscaled through interpolation methods such as nearest neighbor or bilinear. For the fine convolutional network, the authors performed a resolution augmentation as a preprocessing step for the input images. The network, very similar in structure to the coarse

network, produced, as a consequence, outputs with the same size as the original input. These outputs were later post-processed with 3D opening and closing operations. The two biggest connected components in the 3D structure were considered as the kidneys. The accuracy of the models was estimated separately for the right and left kidneys as in Figure 1.

	ConvNet-Coarse		ConvNet-Fine	
	Left	Right	Left	Right
Dice (%)	94.53	93.07	93.62	92.52

Figure 1: Dice score results from [13] where left is the left kidney and right is the right kidney.

Another fully convolutional neural network solution was developed by [6]. The motivation behind the method was the localization of the kidneys through CT scans in order to do a comprehensive radiation dose analysis. The model was trained on non-contrasted CT scans and it detected left and right kidney contours. It used as ground truth manually drawn contours by experts. The estimated contours were assessed for their accuracy using a dice score estimate, mean distance-to-agreement and total segmented volume. The architecture of the network, as described in Figure 2, was similar to that of U-Net [9]. Nevertheless, this model used three dimensional convolutional layers as opposed to the 2D version in the original U-Net. This model achieved a mean dice score of 0.91 and 0.86 for right and left kidneys, respectively.

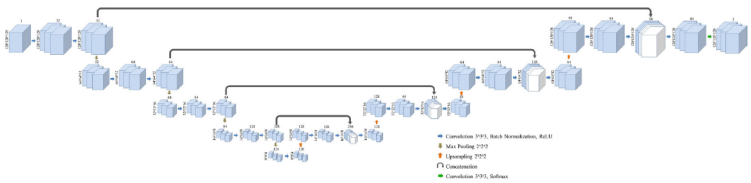


Figure 2: Architecture using 3D-CNNs used by [6] following a U-Net model.

Finally, another model that was also based on the original U-Net architecture wast that developed by [12]. This model used the KiTS19 dataset [4] to train the network. It did not only perform a kidney segmentation task, but also a tumor segmentation. The benefit of using directly the U-Net architecture is that it was designed specifically for biomedical image segmentation [9]. A visualization of the architecture can be seen in Figure 3. The trained network achieved a dice score for kidney of 0.97 and a dice score for tumor of 0.32.

3 Dataset

The dataset used in this project is provided by KiTS19 Grand Challenge [4]. It consists of 300 unique abdominal venous phase enhanced CT scans out of

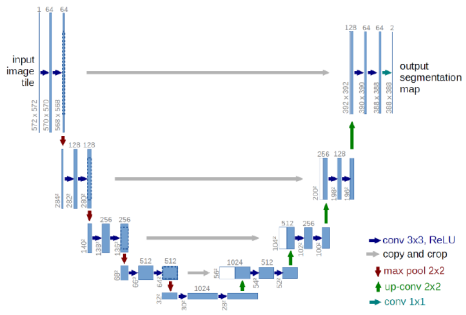


Figure 3: U-Net architecture as described [9] and used by [12].

which 210 have ground truth labels. The remaining 90 CT volumes can be used for evaluation.

KiTS is organized by the University of Minnesota and the University of Melbourne, supported by the Climb 4 Kidney Cancer (C4KC) and National Cancer Institute, of the National Institutes of Health. Ground truth labels have been provided for each patient who underwent partial or radical nephrectomy.

3.1 Data format

Both the imaging and the ground truth labels are provided in NIfTI format. Each NIfTI file consists of a header and the image data array. The image data array has a shape of num_slices x height x width. The number of slices varies per patient and each slice has a width and height of 512. The slices are taken in an axial view and the index of each slice increases from the superior to the inferior part of the body. Slice thicknesses ranges from 1mm to 5mm.

The ground truth label volumes consists of values zero, one and two. Zero represents the background, i.e. neither kidney nor tumor. One represents the kidney and two represents the tumor.

3.2 Data filtering

The cases included in the dataset are of patients that suffer from kidney tumors. The main purpose of the KiTS19 kidney dataset is to assist in the development of an automatic kidney tumor detection pipeline. However, detecting the tumor is out of the scope of this project. For this reason, two preprocessing steps are taken:

- Visualize all volumes and remove large tumor cases from the training dataset. The remaining volumes in the training dataset contain tumors that do not heavily distort the shape of the kidney.
- Reassign all tumor labels to kidney labels.

Figure 4 is a representation of the dataset content and the preprocessing steps.

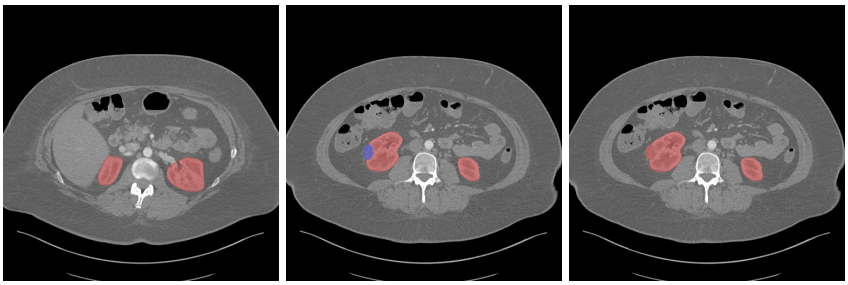


Figure 4: The image on the left is a slice in the CT containing only kidney. The image on the center is a slice with a kidney and small tumor. The slice on the right is the same as the one in the center after the preprocessing steps.

4 QuickNAT

In this project we use QuickNAT (quick segmentation NeuroAnaTomy) as a fully convolutional neural network (F-CNN) model [10]. QuickNAT was originally used for the task of brain segmentation in MRI images where it provided fast and accurate results.

4.1 Architecture

QuickNAT is also similar in structure with the encoder/decoder based U-Net architecture [9]. Similar to the U-Net architecture, it has skip connections between all encoder and decoder blocks that have the same spatial resolution. These skip connections allow features to go from the encoder to the decoder path directly. Moreover, they provide a path of gradient flow from shallower to deeper layers. However, unlike U-Net, the deconvolution stage used as upsampling methods un-pooling layers rather than transpose convolutions [8].

The encoder and decoder paths are composed of dense blocks. Every convolutional layer is preceded by a batch normalization layer and a Rectifier Linear Unit (ReLU) layer. The first two convolutional layers are followed by a concatenation layer that concatenates the input feature map with outputs of the current and previous convolutional blocks. These connections are referred to as dense connections [5]. The dense blocks communicate with each other either by a maxpooling or a un-pooling layer, depending on whether they belong to the encoder or decoder path.

The connection between the encoder and decoder is done via the bottleneck. The bottleneck consists of a convolutional layer and a batch normalization. This block restricts the information flow between the encoder and decoder paths.

Finally, the last block in the network architecture is the classifier block. The classifier block is a convolutional layer with a 1x1 kernel size that maps the input to an N channel feature map, where N is the number of classes (1 in our

case).

QuickNAT's architecture and its components are depicted in Figure 5.

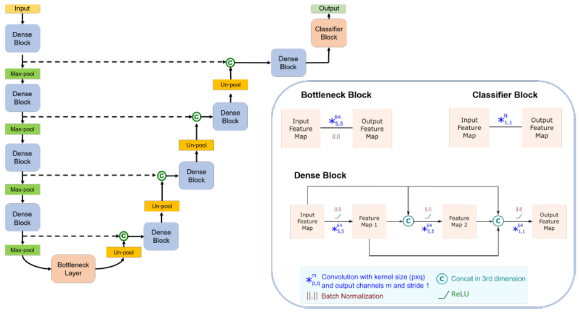


Figure 5: QuickNAT architecture as described by [10].

5 Network optimization pipeline

The network training and optimization process, as depicted in Figure 6, has the following pipeline components:

- Settings and initialization specifications.
- Dataloader and dataset class.
- Logger class.
- Solver class.
- Evaluation methods.

5.1 Settings specification

The settings specifications are handled by two *.ini files that correspond to the local and cluster environments. The *.ini files contain parameters such as:

- Data settings (location, data split, loading configurations, etc..).
- Network settings (kernel size, pool, class number etc..).
- Training settings (learning rate, training and validation batch size, epoch number etc...).
- Evaluation settings (data loading method, model location etc..).
- Common settings (device, logging directory etc..).

These settings are the starting point of the network optimization and are loaded through the settings.py file. The configurations are used and updated throughout the pipeline.

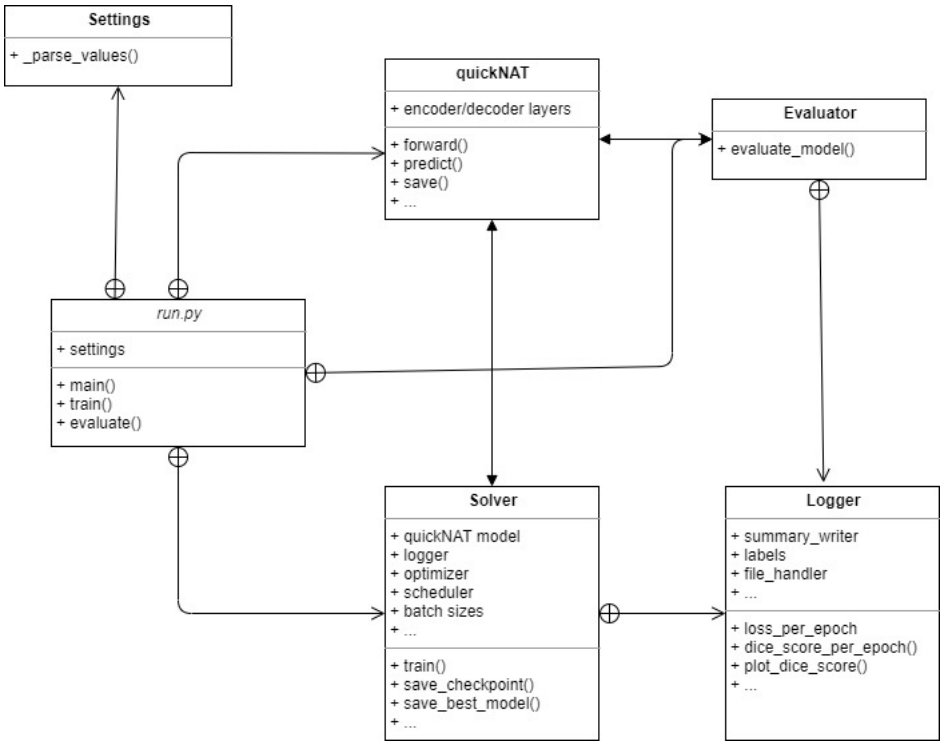


Figure 6: Project pipeline.

5.2 Dataloader

Before running the network optimization tasks, the original data is split into a training and testing set. The names corresponding to the testing volumes are stored into the evaluation_data.json file and are not used until we have achieved what we believe is the best model we can get. The ratio of training and testing data is 80:20.

The testing data is only used once we are convinced that we have achieved a good model and cannot further optimize it. The purpose of the testing data is to do a rough estimation on the performance of the network on real world data.

On the remaining training set, we do another split of ratio 80:20 into the training and validation set. The training set is used throughout the model optimization in order to learn. The validation set is used to keep track of the network that has better chances of performing best in the testing dataset, i.e it helps us understand when the network might be overfitting.

Once the data is split, the volume file paths are passed to a dataset class, which handles the loading and preprocessing of the data.

5.2.1 Loading and preprocessing

Since we are using the pytorch framework, the data loading utility we have used is the *torch.utils.data.DataLoader* class.

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None)
```

The most important argument of the DataLoader constructor is dataset, which indicates a dataset object to load data from. In this implementation, the dataset class implements a pytorch map-style dataset and it is called NiftiData class.

The NiftiData object takes a list of paths pointing the the volumes' NIfTI files and loads them one by one. For each case, it loads the CT volume and the label volume. Once loaded the following preprocessing steps are taken:

1. **Downsampling:** Since each slice in the dataset is a 512x512 resolution image, for an easier and faster training process they are downsampled by a factor of 2 on each dimension.

2. **Black slices removal:** The CT volumes have a relatively large number of slices, with a very small number of them corresponding to the kidney region. In order to reduce the background bias, slices that are more than 20 slices away from the kidney data are removed from the volumes for both training and validation sets. This step also reduces the training time.

3. **Class weights:** The area of the kidney in the CT scan is relatively small compared to the overall CT size. When using pixel-wise losses, such as the cross entropy loss, with such an unbalanced dataset, the training can be dominated by the most prevalent class. In our case, the network will try to reduce all the volumes into background, i.e. no kidney organs.[9] discusses a loss weighting scheme for each pixel, giving some pixels a higher weight at the border of segmented objects. The generated weights have the same size as the original volume and are used in the loss function estimation as described in subsection 5.5.

Any map-style dataset class in torch needs to implement the *__len__()* and *__getitem__(idx)* functions. The first returns the entire length of the dataset, which in our case corresponds to the overall number of slices in the loaded volumes. Meanwhile, *__getitem__(idx)* needs to be able to return one specific slice from all the slices in the list of volumes. In order to avoid loading and preprocessing the volumes each time *__getitem__(idx)* is called, we write the volumes, labels and class weights into temporary numpy files when the NiftiData object is created and keep track of the number of slices in each volume. By doing this, whenever *__getitem__(idx)* is called with a random slice index, we can calculate the volume that slice belongs to. Once the volume is identified, we need only load the specific slice with the corresponding index.

This is possible by using the option mmap_mode with numpy.load(). This option allows reading small segments of large files on disk, without reading the entire file into memory.

```
image = numpy.load(identified_volume, mmap_mode='r+')[idx, :, :]
```

5.3 Solver

The solver class manages the hyper-parameter optimizations of the network. Here we specify the loss function, as defined in subsection 5.5 and the needed scheduler and optimizer.

5.3.1 Optimizer and Scheduler

From pytorch we use the optim package to define a scheduler and optimizer. The best performing scheduler was StepLR, which decays the learning rate of each parameter group by γ every *step_size* number of epochs. The value of γ and *step_size* are initialized in the settings as 50 and 0.5 respectively. The values are chosen after running a small parameter search and monitoring the network's performance.

The optimizer used for this network is Adam [7], also provided by the torch optim package. The optimizer object holds the current state and updates the parameters based on the computed gradients. The Adam optimizer is defined as below:

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t \tag{1}$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2 \tag{2}$$

$$\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)} \tag{3}$$

$$\hat{v}_t = \frac{m_t}{(1 - \beta_2^t)} \tag{4}$$

where m_t and v_t are estimates of the first moment (the mean) and the second moment and g_t is the gradient of the weights. The parameters β_1, β_2 and ϵ are hyper-parameters.

Finally, the weight update is done as follows:

$$\theta_{t+1} = \theta_t - \frac{\mu}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t \tag{5}$$

with θ_t being the weights of the network at time t and μ is the learning rate.

The authors of Adam [7] proposed default values of 0.9 for β_1 and 0.999 for β_2, and 10^{-8} for ϵ. These values also work for this network.

5.3.2 Network parameters

The QuickNAT dense block implementation builds on the implementation of NN Common Modules provided by Shayan Ahmad Siddiqui and Abhijit Guha Roy. For initializing the dense blocks the following parameters are used:

- num_channels = 1
- num_filters = 64
- kernel_h = 5
- kernel_w = 5
- kernel_c = 1
- stride_conv = 1
- pool = 2
- stride_pool = 2
- se_block = "CSSE"
- drop_out = 0.2

where CSSE is a dense block type as defined by [11] and implemented by NN Common Modules.

Aside from changing the number of channels to 1, the rest of the parameters are left to their default values. Changing the other parameters does not provide any improvement in the accuracy of the network.

The number of channels as well as the number of classes for this project is set to one, i.e. only kidney. The network is expected to generate a probability map estimating for each pixel weather it belongs to the kidney class or not. In order to achieve the probability map as an output of the network, a sigmoid layer is added to the original architecture of the network. Any probability values bigger than 0.5 are classified as kidney.

5.3.3 Model Accuracy

To estimate the model's accuracy we use the dice score, which is defined as follows:

$$DC = \frac{2 * TP}{2 * TP + FP + FN} = \frac{2|X \cap Y|}{|X| + |Y|} = \frac{2 * X * Y + 1}{X + Y + 1} \tag{6}$$

where TP are true positives, FP are false positives and FN are false negatives. X represents the output and Y the predictions. The dice score is estimated for each volume. The overall training, validation and testing dice score is the average among all volumes in the respective sets.

5.4 Logging

The logging of the results and images is handled by the Logger class, which builds on tensorboard. It implements three different summary writers i.e. training, validation and evaluation. Each of the summary writers keeps track and logs the average loss and the average dice score per epoch. The final results of the experiment in section 6 are extracted from these tensorboard logs.

5.5 Loss function

In this project we use a weighted combined loss of dice and binary cross entropy. The dice loss is defined as follows:

$$D_L(p, \hat{p}) = 1 - \frac{2p\hat{p} + smooth}{p + \hat{p} + smooth} \tag{7}$$

where $p \in 0, 1$, $0 \leq \hat{p} \leq 1$ and $smooth = 1$. The variable "*smooth*" on both the nominator and denominator of the loss is a smoothing applied to the dice score in order to handle corner cases. One such corner case is $p = 0$ and $\hat{p} > 0$. This is the case when the entire slice is a background slice, i.e. no kidneys present and our network predicts some kidney pixels.

Given that the network tries to predict a binary output, we use a weighted binary cross entropy.

$$BCE_L(p, \hat{p}) = mean(c_w * [\hat{p} * log(p) + (1 - \hat{p}) * log(1 - p)]) \tag{8}$$

where p is the ground truth, $\hat{p}$ is the prediction from the model and c_w are the class weights. From the multiplication we take the mean of the output tensor, resulting in the loss being a scalar.

The overall loss then results in:

$$Loss(p, \hat{p}) = \alpha * D_L(p, \hat{p}) + \beta * BCE_L(p, \hat{p}) \tag{9}$$

the coefficients α and β are estimated as hyper-parameters and add more weight to one loss over the other. The coefficients with the best results are $\alpha = 1$ and $\beta = 0.5$.

5.6 Gradient accumulation

Increasing the training batch size helps create a more generic model and achieves better results in this project. The batch size used here is 16. However, memory usage during training with large batch sizes can become problematic. In order to still be able to train with these batch sizes and more limited memory resources we use gradient accumulation.

Gradient accumulation means running a configured number of steps without updating the model variables while accumulating the gradients of those steps and then using the accumulated gradients to compute the variable updates 7.

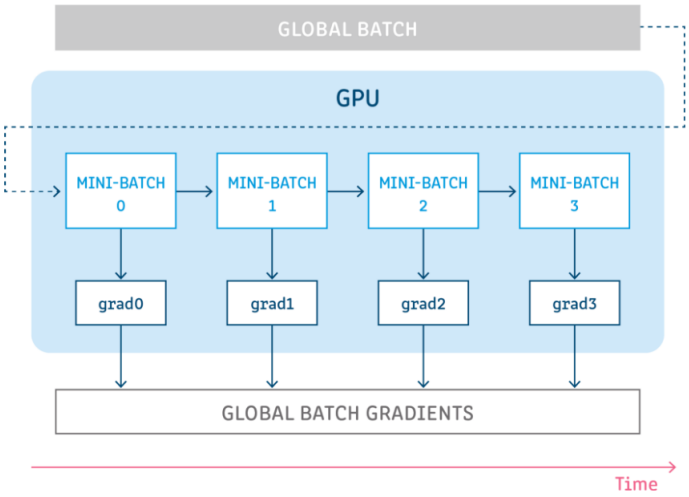


Figure 7: Gradient accumulation running on a gpu [1]

In this project, we load our dataset in smaller batches than the predefined batch size and compute the gradients of the weights for each iteration with the small batch size. However, we only update the weights after n steps, where n is the number of small batches needed to create a larger batch, i.e. $small\_batch * n = large\_batch$. This method increases the network's time to train, but reduces the required memory resources.

$$acc\_grad = \sum_{i}^{n} g_i \tag{10}$$

where g_i is the gradient of batch i and acc_grad is the accumulated gradient through out the n small batches.

The acc_grad is then unsed by the optimizer, as explained in subsubsection 5.3.1.

5.7 Learning Rate

The last elements to define in the solver are the model learning rate and the stopping criteria. After running a parameter search, the learning rate with a meaningful decline in the combined loss function values and better model accuracy was specified as 0.001.

5.8 Stopping criteria

The model is considered to be overfitting when the combined loss value for the validation set was increasing for more than two consequential epochs. This was also used as a stopping criteria for the network training. The overall number

of epochs needed to train the network was 6. The model saved by the solver is that with the higher model accuracy in the validation set.

6 Results

The model's performance is evaluated on the training, validation and testing data set. The dataset used to obtain the best results consists of 87 training volumes, 22 validation volumes and 27 testing volumes. The kidney dice score is evaluated on each volume in the set and the overall dice score is the average among all the volumes. The results are as follows:

Training	**Validation**	**Testing**
0.9467	0.9154	0.9259

The progression of the dice score through the epochs can also be seen in Figure 8. This dice score is obtained by training the network on the full dataset with a learning rate of 0.001 and loss coefficients of $\alpha = 1$ and $\beta = 0.5$ as defined in subsection 5.5. The network parameters used are described in subsubsection 5.3.2. This parameter setup resulted in the best dice score so far for this project.

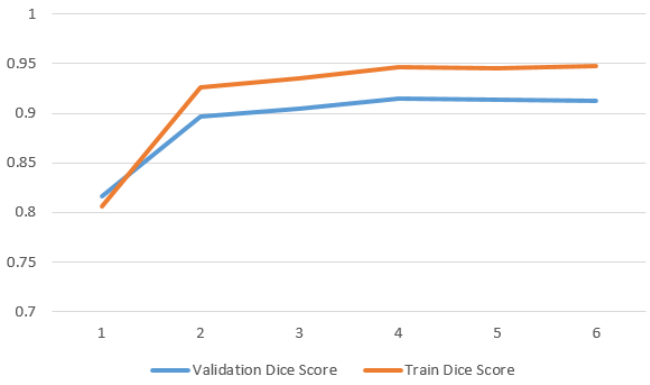


Figure 8: Training and validation dice score

In order to achieve these results the following tweaks and experiments were conducted:

1. **Learning rate adaptation.**
   The choice of the learning rate proved to be a slightly complicated mater. Throughout the training we used a combined loss of dice and binary cross entropy. Both losses were expected to drop in value, with the validation loss staying on average per epoch slightly below the training loss. When the validation loss starts to progressively grow over the training loss, we

can say that our model is overfitting. Experiments showed that the dice loss and cross entropy loss did not converge at the same speed, i.e. the binary cross entropy loss visibly overfit the model faster than the dice loss. For learning rates higher than 0.001, the binary cross entropy would overfit directly after the second epoch, resulting in the best model having a lower dice score than the one presented above. However, choosing a learning rate that was smaller than 0.001, resulted in the loss overfitting at a later epoch, and produced lower dice scores than the presented model.

2. **Weighing combined loss components.** Another solution to the binary cross entropy loss overfitting slightly faster than the dice loss was that of weighing the losses differently in the combined loss. By applying a smaller weight to the binary cross entropy loss the network optimizer will try to minimize more the dice loss. The weights applied that performed best were $\alpha = 1$ and $\beta = 0.5$, where α is a coefficient to the dice loss and β is a coefficient to the binary cross entropy.

3. **Batch size and gradient accumulation.** In order to create a more generic network and avoid overfitting in the earlier epochs, we used a batch size of 16. This presented memory issues at first, which we tackled by using gradient accumulation as explained in subsection 5.6. However, while this implementation choice improved the results, it also resulted in a longer training time for the network.

4. **Dropout.** Another method used to avoid overfitting in the early epochs was dropout. Passing a standard dropout value of 0.2 to the network further improved the results as expected.

5. **Reducing redundant work.** QuickNAT was originally designed for brain segmentation, i.e. the network's output was a classification into 25 different classes. As such, it made sense for its output to be a 25 channel output. However, for the purpose of this project, our output is binary. If the network had tried to predict two classes, i.e. background and kidney, the results would have been complementary. Therefore, we reduced the input and output labels' number of channels into just one, where voxels set to one represented "kidney" and voxels set to zero represent "no kidney". This also led to a simplification of the cross entropy loss to a binary cross entropy and improved the accuracy of the model.

The final loss functions for this network are depicted in Figure 9. We can see the network starts to overfit in the last two epochs, i.e. epoch 5 and 6, where the validation loss starts increasing. Nevertheless, the network is able to detect this and stops training at epoch 6.

The network performs best in batches that are either far away from the kidneys or contain a substantial kidney labeled number of voxels. The performance drops for slices that are slightly above or below the kidneys. In these batches the network predicts a small number of kidney pixels where the labels

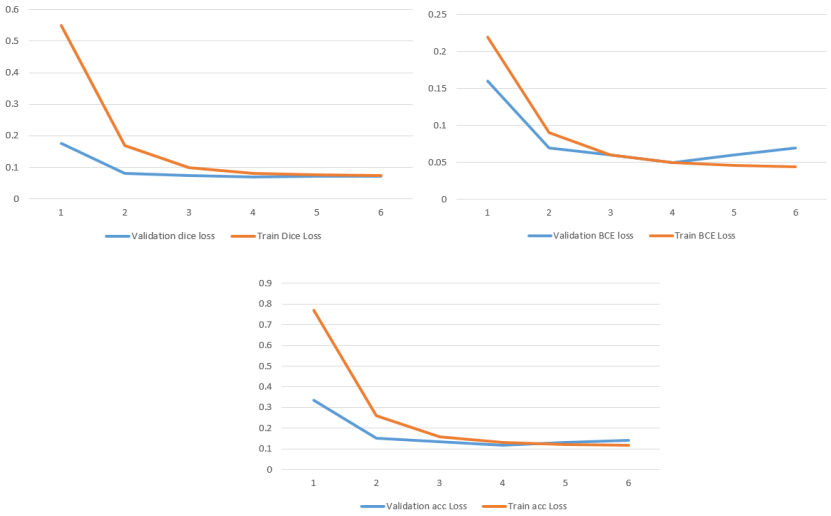


Figure 9: Dice Loss and BCE Loss

indicate there should not be any. The best and worst performing batches are also visualized in Figure 10 and Figure 11 respectively.

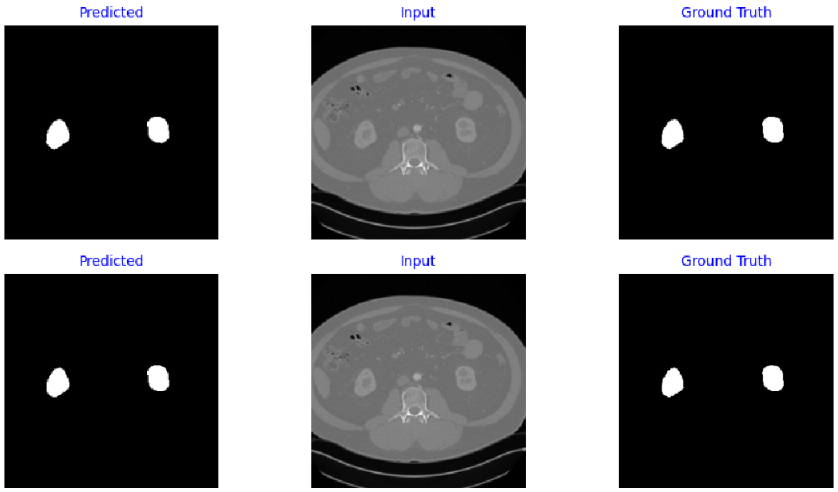


Figure 10: The best performing batches in the test set results.

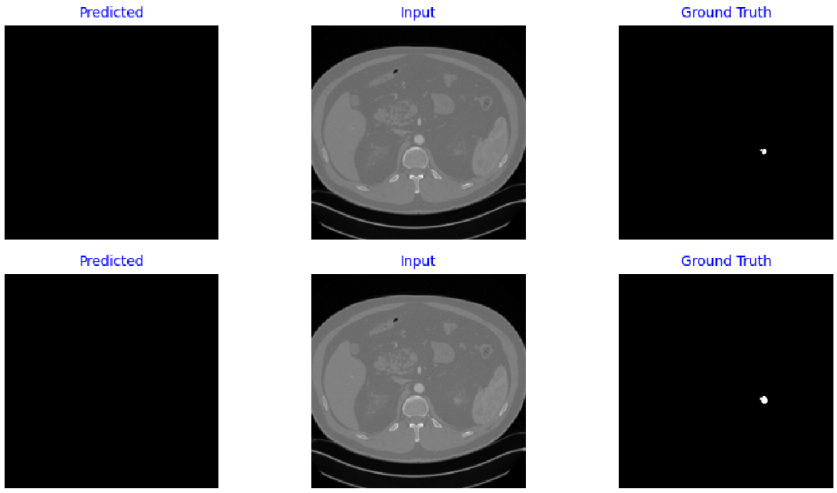


Figure 11: The worst performing batches in the test set results. The slices are located slightly above the kidneys.

7 Conclusion

Overall, the project yielded satisfactory results and was a very good learning experience. Apart from getting familiar with quickNAT, I also got some experience in processing medical data volumes and passing them to a segmentation network. The dataset was rather large, which helped me get familiar with data loading and processing techniques such as gradient accumulation. Furthermore, I learnt to cope with imbalanced classes segmentation tasks and using weighted combined losses.

This project also leaves some room for future improvements steps that might increase the overall accuracy of the model. One such step could be applying post processing steps to the results of the network, such as 3D opening and closing operations. This could lead to an improved outcome in the bottom and top slices of the kidneys, as presented in Figure 11.

Moreover, while KiTS19 provided a large dataset to work with, the kidney labels were all adjusted to include small tumors. We could achieve better results if we used CT scans that do not contain any tumors. However, at the moment, no such dataset was available online.

8 Acknowledgments

I would like to thank everyone who helped and advised me along the way with this project. Thank you to my advisors Thomas Wendler, Christina Bukas and Johannes Oberreuter, as well as, Desislava Dimova for providing valuable advice and feedback throughout this project.

References

[1] What is gradient accumulation in deep learning?

[2] Xinjian Chen, Ronald M. Summers, and Jianhua Yao. Automatic 3D kidney segmentation based on shape constrained GC-OAAM. In *Medical Imaging 2011: Image Processing*, volume 7962. International Society for Optics and Photonics, SPIE, 2011.

[3] D. Dimova. Multimodal thyroid volumetry. *Master's thesis. Arcisstraße 21, 80333 Munich: Technical University of Munich, Dec. 2019.*

[4] Nicholas Heller, Niranjan Sathianathen, Arveen Kalapara, Edward Walczak, Keenan Moore, Heather Kaluzniak, Joel Rosenberg, Paul Blake, Zachary Rengel, Makinna Oestreich, et al. The kits19 challenge data: 300 kidney tumor cases with clinical context, ct semantic segmentations, and surgical outcomes. *arXiv preprint arXiv:1904.00445*, 2019.

[5] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[6] Price Jackson, Nicholas Hardcastle, Noel Dawe, Tomas Kron, Michael S Hofman, and Rodney J Hicks. Deep learning renal segmentation for fully automated radiation dose estimation in unsealed source therapy. *Frontiers in oncology*, 8:215, 2018.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[9] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[10] Abhijit Guha Roy, Sailesh Conjeti, Nassir Navab, and Christian Wachinger. Quicknat: segmenting mri neuroanatomy in 20 seconds. *arXiv preprint arXiv:1801.04161*, 2018.

[11] Abhijit Guha Roy, Nassir Navab, and Christian Wachinger. Concurrent spatial and channel 'squeeze & excitation'in fully convolutional networks. In *International conference on medical image computing and computer-assisted intervention*, pages 421–429. Springer, 2018.

[12] Rochan Sharma, Pallavi Halarnkar, and Kiran Choudhari. Kidney and tumor segmentation using u-net deep learning model. *Available at SSRN 3527410*, 2020.

[13] William Thong, Samuel Kadoury, Nicolas Piché, and Christopher J Pal. Convolutional networks for kidney segmentation in contrast-enhanced ct scans. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 6(3):277–282, 2018.