



Conditional GAN based data augmentation for automated delineation of the parotid gland in CT scans

Caesar Pepijn de Keijzer

2619092

Vrije Universiteit Amsterdam

Supervisor: Wilko Verbakel

Daily Supervisor: Victor Strijbis

External reader: Natalia Silvis-Cividjian

Abstract

In this thesis, an analysis is presented of how generating synthetic computed tomography scans can contribute in the delineation of the parotid gland by a U-Net. Current radiotherapy research often deals with a limited data set of parotid gland images. By applying a conditional Generative Adversarial Networks to generate synthetic data for further data augmentation, the goal is to improve the segmentation accuracy of a U-Net to contour the parotid gland from computed tomography scans. Previous research has shown that extending standard data augmentation techniques with GAN-based data augmentation has yielded higher segmentation scores. The setup in this paper consists of two models where the first is used to generate synthetic parotid masks, and the latter is employed to generate tomography images that correspond with these masks. The applied method shows that although able to generate high resolution, accurate training images, overfitting occurs on the conditional parotid image generation during testing. Further improvements can be made by using larger data sets as well as inserting additional conditional image information and better GAN metrics.

Keywords: deep conditional generative models, generative adversarial networks, deep learning, parotid, computed tomography, radiotherapy, image synthesis, segmentation

Preface

I would like to thank my supervisors Willko Verbakel and Victor Strijbis for all their patience and guidance. This paper required an enduring effort, and it would not have been completed without their help. I am very grateful for their assistance and their patience to help me to the completion of this paper. I would also like to express my gratefulness towards my Natalia Silvis-Cividjian who has proven to be an indispensable help throughout this thesis and the past years. I sincerely hope that any future papers will be less of a personal struggle, as well as be interrupted by less pandemics.

TABLE OF CONTENTS

Abstract	2
Preface	3
1. Introduction	6
1.1 Background	6
1.1 Motivation	10
1.3 Research Question	10
1.4 Report Outline	11
2. Related Work and Literature	12
2.1 Radiotherapy	12
2.2 Deep Learning	12
2.2.1 Segmentation.....	17
2.2.1 Data Augmentation.....	19
2.1.1 Parotid gland	21
2.2.1 U-Net.....	24
2.2.2 GANS.....	28
3. Experimental Framework	34
3.1 Proposed Architectural Design.....	35
3.2 Data Set - Cancer Imaging Archive	35
3.3 Pre-Processing.....	35
3.4 Amazon Web Services (AWS).....	37
3.5 MONAI	38
4. GAN Specific Implementation Details	40
4.1 DCGAN parameters	40
4.2 Pix2Pix parameters.....	43
4.3 Problems addressed.....	44
5. Results	45

5.1 Stage I results	45
5.2 Stage II results	50
6. Discussion	60
6.1 Framework limitations	64
6.1.1 Lack of network information	64
6.1.2 Hardware constraints	65
6.1.3 Higher Resolutions.....	66
6.1.4 Lack of evaluation metrics	67
6.2 Ethical considerations.....	68
6.3 Future Challenges.....	70
6.4 Future Work & Applications.....	70
7. Conclusion.....	73
Bibliography	74
Abbreviations and Names	82
Tables	83
Figures	84
Appendix A: Data Pre-Processing	86
Appendix B: DCGAN on AWS.....	101
Appendix C: Pix2Pix on AWS (adapted from official Pix2Pix)	114

1. Introduction

1.1 Background

With the continuous rise and size of elderly populations around the world, providing affordable healthcare remains an illustrious challenge. Nowhere is this more prevalent in the link between age and the risk of getting cancer. With cancer being second only to heart-disease in the cause of death for the oldest old. For instance, just head-and-neck cancer accounts for about 53.260 new cancer cases and 10.750 new deaths in the United States in 2020 alone [1]. Therapy for said cancers customarily involves ionized X-ray radiation to target the malignant tumor cells in a process called radiotherapy. Radiotherapy concerns methods to use radiation to combat cancerous cells in a patient's body while doing minimal damage to healthy tissue. Medical examination of patients to evaluate radiotherapy procedures usually involve several different imaging devices. These include Magnetic Resonance Imaging (MRI), Electroencephalography (EEG) and Computed Tomography (CT) scans and are a painless and noninvasive procedure to detect abnormalities in a patient's body. By delineating and marking organs at risk (OAR), special, highly trained radiotherapy clinicians ensure that the surrounding organs are minimally affected during treatment. This time-consuming, but essential delineation is an important step to ensure that the corresponding patient treatment plan safeguards the correct dose calculation on the treated patient.

Looking for ways to speed up this time-consuming segmentation process, medical researchers have been examining machine learning (ML) techniques. These deep machine learning techniques require training on large contoured clinical data sets, usually containing several hundreds, if not thousands of patients, providing sufficient variation in the data [2].

Coupled with the growth of digitally available medical data, the medical community has seen an increased use of machine learning algorithms to make sense of all this data allowing examiners to treat more patients in a shorter amount of time. Some of these algorithms currently already outperform clinicians in controlled tasks and much work is done to integrate them further to improve diagnosing and patient care [3], [4]. Machine and deep learning algorithms promise to advance medical care on various fronts. Yet what is often forgotten is how all these models are

only as good as the underlying data that it is representing. With the expansion of these deep machine learning networks, acquiring adequate amounts of labeled medical imaging data has proven to be difficult and expensive but essential.

While there are several publicly available data repositories supplied by various institutions, they sometimes differ in size, scope, and a common framework. As well as there being a nearly insatiable demand for additional images. To collect this labeled data, special radiotherapy technicians are usually employed to contour and mark tumors manually from CT scans, inadvertently causing a new bottleneck.

That is why, in collaboration with the Vrije Universiteit Medisch Centrum (VUmc), this study investigates if a novel deep learning data augmentation technique can be used to expand the amount of available medical data, with a particular focus on the parotid gland in the head and neck region. During treatment for head or neck cancers (HNC), parotid glands are often delineated and marked as organs at risk (OAR) because they are particularly susceptible to radiation during therapy. Damage to the parotid glands could lead to tumors and xerostomia (dry mouth) which reduces the quality of life of the irradiated patient. Current best practices for auto-segmenting the parotid glands are limited to an accuracy of about 90% [5].

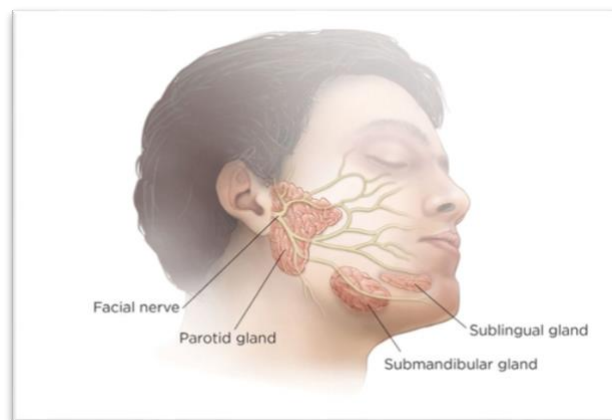


Image 1: Visualization of the Parotid gland

The approach that will be explored in this thesis is a way of generating synthetic data samples to artificially create a varied data set as can be seen in **Error! Reference source not found.** This w

ill be done using generative models which are able to learn the underlying image data, in the system named as DCGAN and Pix2Pix [6]. These novel generative deep learning models have thus far shown their effectiveness in a myriad of applications ranging from generating faces to the technology behind many deepfake videos [7]. In the medical imaging community however, their use for data augmentation is only just recently being explored. The synthetic outcome of the system design shown below will be used to add image variability in the data augmentation phase for parotid automatic (see ML) contouring architectures.

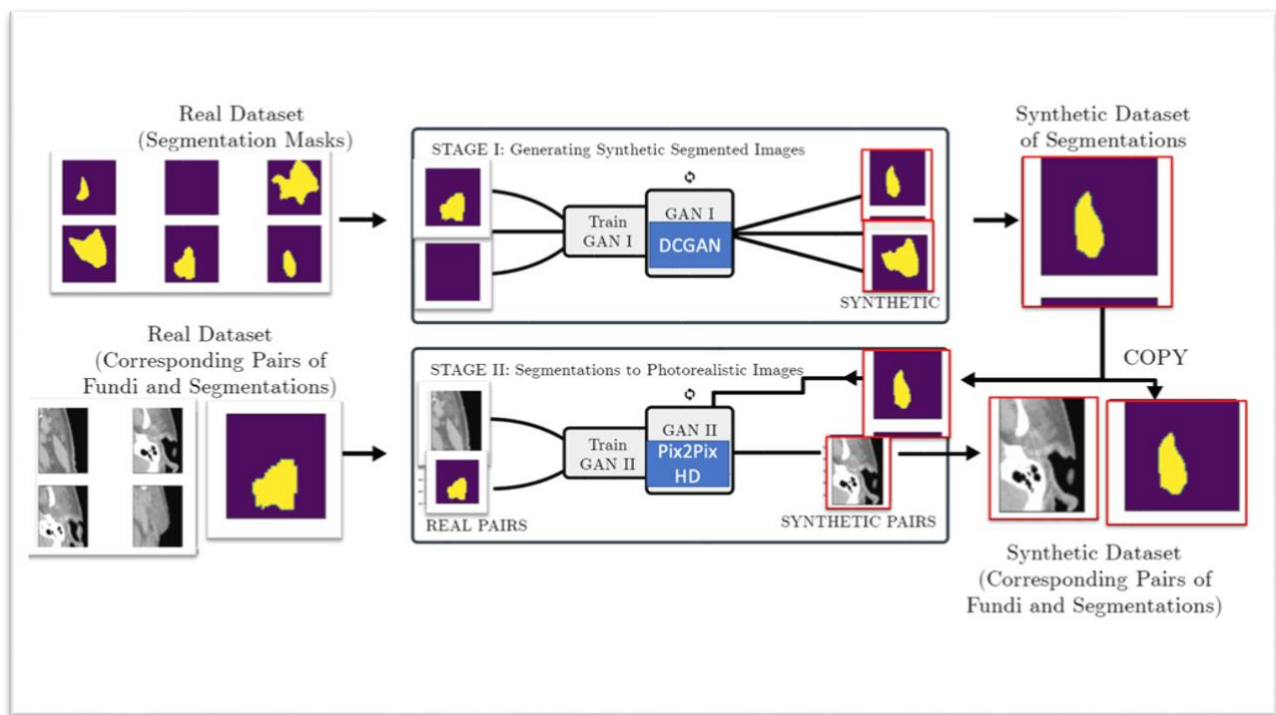


Figure 1: Proposed high level overview of system design

The results of the proposed custom pipeline show that generating high fidelity synthetic images of the parotid gland is certainly possible and should be further explored as a method to supplant traditional augmentation techniques. However, this is not without noting the infancy of the technique and the lack of objective metrics to measure the success of generative models. A more detailed explanation of radiotherapy and the area of deep learning will be given in subsequent chapters.

The proposed augmentation technique shows one way that traditional machine learning pipelines can benefit from artificially generated data.

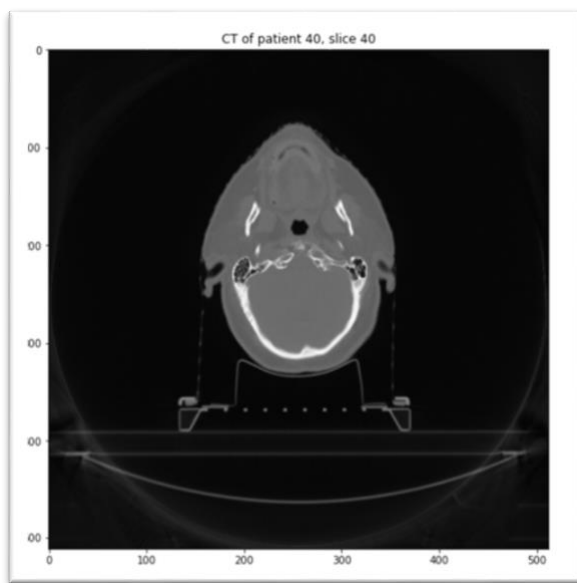


Image 2: Grayscale CT head image from the Cancer Imaging Archive

1.1 Motivation

1.3 Research Question

With the above description of the existing problem of contouring of the parotid glands still in mind, we will next turn to the proposed objectives of this research.

- i) A literature overview of what segmentation entails, as well as an outline of several of the mentioned deep generative models that are employed.
- ii) The implementation of a two-stage GAN framework to create CT images of the parotid gland and its corresponding mask.

Previous research with synthetic data has shown that significant improvements in segmentation and classification tasks can be achieved [8], [9].

It is hypothesized that the generative pipeline for extended augmentation will indeed cause a higher segmentation performance.

1.4 Report Outline

This report consists of several main sections with nested subsections. Chapter 2 discusses related work that is being done in the field as well as a broader overview of the models that are being used in later chapters. The following Chapter (3) delves into the framework and the setup of the experiment. It might be less of interest to the medical community but is of interest if the experiment is to be repeated or expanded. While Chapter 4 expands upon this framework by elaborating the parameters of the used models. Chapter 5 is used to examine the results of the experiment and the last two chapters 6 and 7 discuss and conclude the findings of the original stated hypothesis. Acknowledgments, a list of abbreviations and code snippets are supplied in the appendix.

2. Related Work and Literature

2.1 Radiotherapy

This research is done in accordance with researchers at the radiotherapy department of the VUmc. Radiotherapy concerns methods to use radiation to combat cancerous cells in a patient's body while doing minimal damage to healthy tissue. Head-and-Neck cancer accounts for about 53.260 new cancer cases and 10.750 new deaths in the United States in 2020 [1]. Continuous research is being done to limit the side effects that come with treatment herein sparing organs-at-risk [10]. Ionized X-ray radiation is used to target the tumor and the surrounding organs around the tumor area are commonly delineated. A patient specific treatment plan is subsequently made to limit the radiation dose and damage on the surrounding organs [11]. The delineation of OAR is done by a trained radiotherapy technician and is an expensive and time-consuming process. Since extra caution is necessary for the head and neck region during treatment due to their complexity, significant expertise is required. According to [12], the nonconvex geometrical shapes of radiation targets and the typical overlaps of irradiated targets and organs-at-risk (OARs) require delicate radiation beam modulations for OAR sparing. As mentioned before, delineated contours are usually drawn on individual slices of CT or MRI images. Where each CT scan typically consists of hundreds of slices [13]. To lower the delineation time, cut expenses and reduce the amount of inter-observer variability, auto-contouring techniques are slowly being introduced. From ongoing machine learning research, deep learning techniques have been emerging to address the challenges that come with manual contouring. These techniques are slowly being introduced in the various stages of radiotherapy planning, ranging from automatic outlining of regions of interest (ROI) to patient outcome prediction [14]. While significant regulatory hurdles still exist for full integration in clinical routines, their innovation is widely recognized and further explored in the following sections.

2.2 Deep Learning

Deep learning (DL) is a form of machine learning with several hidden layers, where more layers often lead to more complexity. Due to the large volumes of available, high quality

oncology data, deep learning has been regarded as mature enough to be employed in a wide range of medical imaging tasks [15]. However, before we delve into a narrower application of deep learning models, a more generalized and broader overview of what machine learning entails is in order.

Machine learning allows for computers to make inferences based on heaps of available data. Generally, there exists two different approaches. *Supervised learning* concerns itself with knowing a specific available outcome to predict, such as presence of a tumor, length of survival of a patient, or treatment response. *Unsupervised learning* identifies patterns and subgroups within data where there is no clear outcome to predict [16]. With supervised learning, the aim is to gain an understanding of the underlying data, which is subsequently used to label new, unseen data points. These data points either come labeled, in which case it is called a classification task, or continuous, making it a regression problem. A simple example here could see a model, categorize incoming patients either as having cancer or no cancer. If a new patient matches the features of previous patients, the model classifies (labels) the patient accordingly.

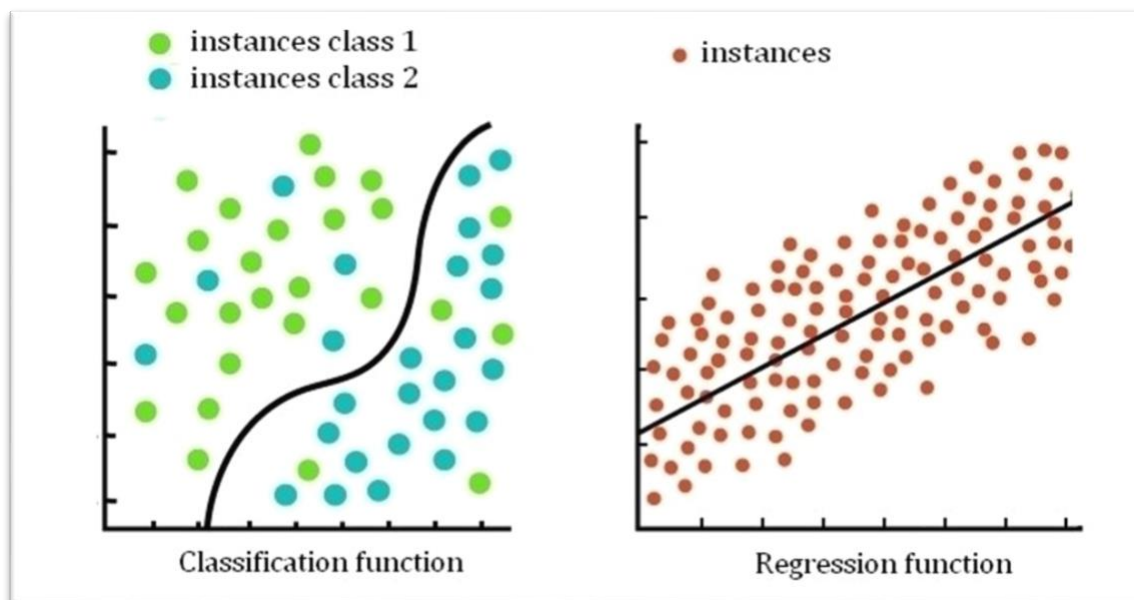


Figure 2: Supervised Learning techniques

In unsupervised learning, no such labels or categories exist. It infers a function that expresses hidden characteristics residing in the input data. The objective here is to cluster certain groups and features together to discover underlying structures or similarities in the data.

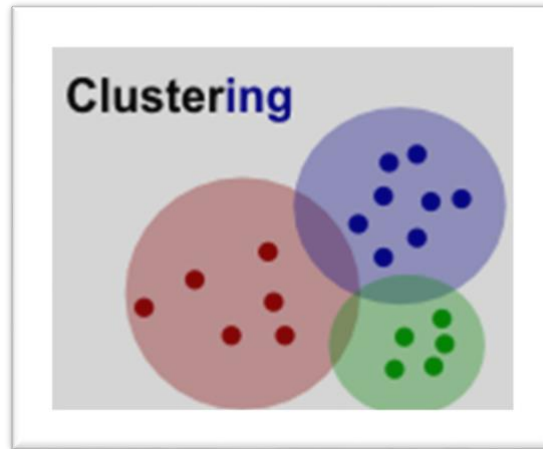


Image 3: Clustering, as an unsupervised learning technique

Semi-supervised learning follows a middle ground method and is used when one has a small amount of labeled data set. Since acquiring the labeled data is generally very difficult or very expensive, semi-supervised learning can be a cost-effective option [17].

Deep learning is a subfield of machine learning that try to mimic human intelligence in the form of creating an artificial brain made up of a connected network of neurons.

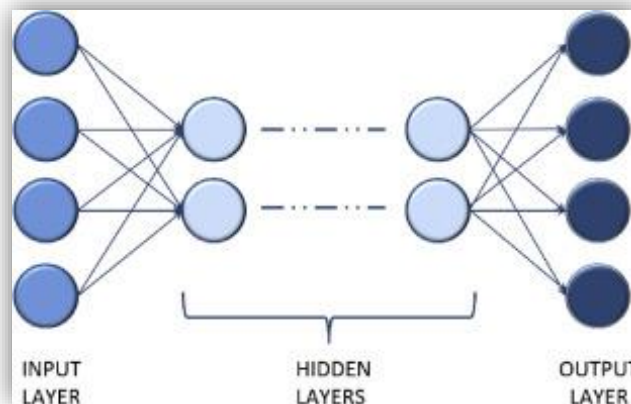


Image 4: A neural network visualized with an unknown amount of hidden layers

In creating these artificial neural networks (ANN), a computer essentially trains and learns to process data iteratively. During this important training phase, the value of each neuron in the layered network is adjusted by parameterizing weights through learning algorithms such as back propagation [18]. Since larger and multilayered networks take an exponential longer time to run and complete, the recent advent of specialized graphics processing unit chips and cloud computing have shortened the computation time and democratized machine learning research. Applying deep learning techniques on medical images is a rather recent innovation but has seen exponential growth in the amount of published research, kick starting in 2012 [17] [19].

One of the most widely used DL networks is the convolutional neural network (CNN), inspired by the primary visual cortex, and mainly applied to image processing. A study by Ibragimov et al. [20] describes for example how CNNs have been used to speed up and improve organs at risk contouring in head and neck cancer patients using 50 CT images. While a high segmentation performance was achieved with regards to the spinal cord, the mandible and optic nerves, lower performance for the submandibular glands and optic chiasm show that difficulties remain. CNN's as the name suggests, has at least one layer that performs a convolution. This layer consists of a two-function operator, the image and a filter/kernel at each voxel. As mentioned in [21], "CNNs are favorable for image processing because the convolution operator is good at extracting latent image features while keeping the computational cost low". By 'striding' over an image with a kernel, it is possible to recreate an entire new image, encoding information of the original image. The role of this transformation is to detect local features in different parts of an image yet preserving the spatial information of the original input image [22]. The resulting, encoded images are known as feature maps. This process can be observed in the image below where filter and kernel refer to one and the same thing and the stride refers to shift of the kernel along the y or x axis.

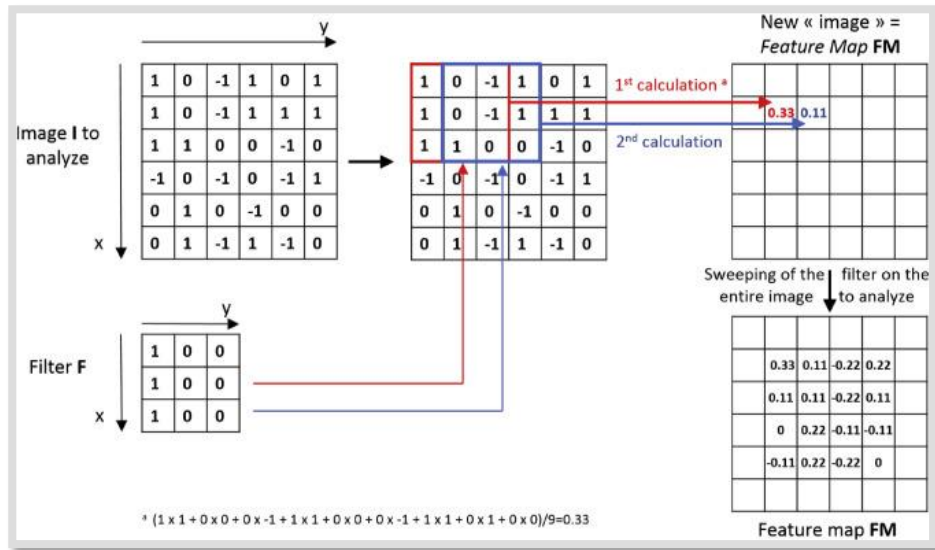


Image 5: A convolution step applied to an image, taken from [22]

Another common layer used in CNN's is a pooling layer. The pooling layer is applied to reduce dimension of the input image by taking the mean or finding the maximum value of different areas. This reduction in the dimension of the image decreases the necessary computational power and aids with the extraction of dominant features. The images below show how pooling layers and convolutional layers are effectively combined to create a convolutional neural network.

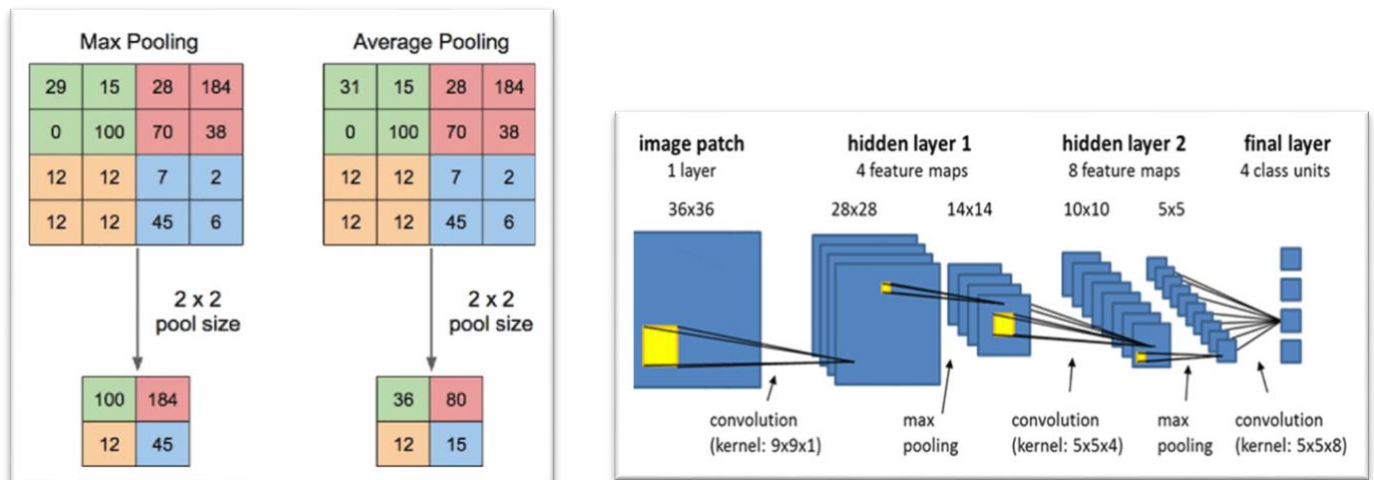


Image 6: (Left) Two common pooling techniques and (right) a simplified CNN schematic

Further advances in computing power will allow researchers to work on the inherent 3D nature of many of the medical images. This would allow for inferences to be made from adjacent images where depth of for example a tumor is also taken into consideration. These networks include 3D kernels and would allow for easier and more efficient analysis since organs are considered in full. For an even more in-depth explanation of CNN's, this author would like to point to the footnotes on the bottom of this page^{1 2}. Overall, it is important to remember that both ML and specifically DL follow a data-driven approach and that the condition of the model is inherently related to the availability of high-quality data. Medical images can be imbalanced, privacy sensitive and time-consuming to label [17].

2.2.1 Segmentation

Automating medical image segmentation is one key area in which researchers are currently investigating [23]. With deep learning techniques, extracting regions of interest from a wide range of image modalities such as CT or MRI can be done through automated processes. Properties such as image gradient, voxel intensities and morphology of anatomical structures are taken into consideration in this process. Radiation therapy dose calculation is primarily based on CT scans however and therefore planning systems generally require anatomical information to be accurately delineated on these images [24]. Segmentation results in the contouring of organs at risk and is usually a crucial but time-consuming step in RT planning. As well as being subjective in the definition of the structure boundaries by the clinician. Great efforts have been spent on creating automated pipelines particularly for segmentation to ensure that this is done effectively and accurately [10]. Currently these pipelines assist doctors in the diagnosing and decision-making process after an image from a patient is taken. A large amount of exploratory work has already been done on segmentation methods and learning the shape and appearance characteristics of

¹ Convolutional Neural Networks Explained by Mayank Mishra on Towards Data Science, 2020: <https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>

² A Comprehensive Guide to Convolutional Neural Networks by Sumit Saha on Towards Data Science, 2018: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

anatomical structures in medical images. Atlas based auto contouring (ABAS) has been a widely popular technique and uses an image with carefully delineated OARs as a reference for delineating new patients. While ABAS has done a good job in reducing this time-consuming process and improving inter-variability, issues such as with small OAR's and selecting appropriate atlases remain [10]. There is a general difficulty in handling anatomy variations among patients because they use a fixed set of atlas images [13]. As mentioned earlier, the growth of GPU computing has allowed for the proliferation of recent machine learning techniques to overcome some of the challenges in atlas contouring. A frequently used metric to evaluate medical image segmentation is the Dice similarity coefficient (DSC). The Dice similarity score is calculated by comparing the automated segmented image with a ground truth image that manually segmented by a clinician. Officially called the Sørensen–Dice index, it is a statistical tool that allows for an evaluation of the spatial overlap between two sets of data and is frequently used in image segmentation. Calculated as

$$DSC = \frac{2 * |A \cap B|}{|A| + |B|}$$

where A and B represent images from two differing data sets.

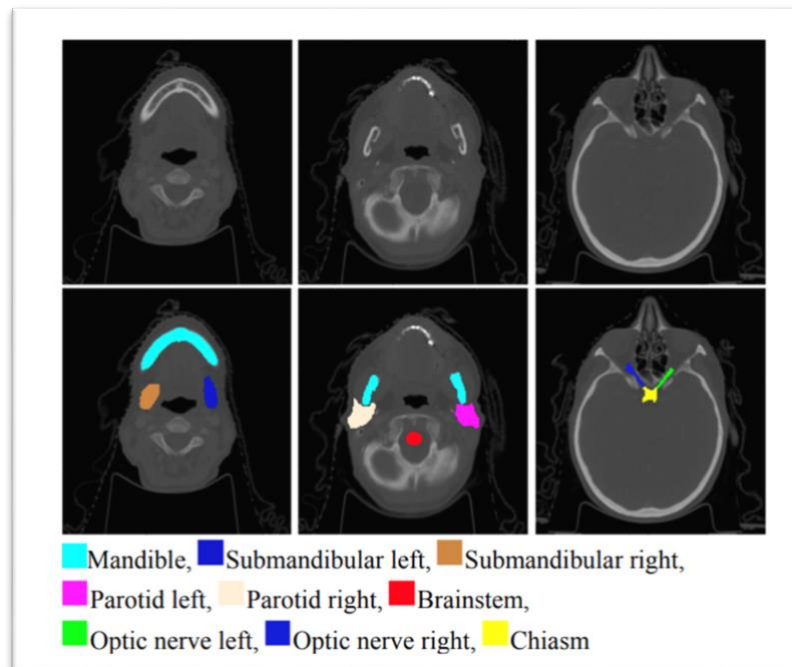


Image 7: Manually annotated OAR's in a patient CT image, taken from Wang et al. [25]

2.2.1 Data Augmentation

However for these ML models to perform well and to ensure that all the important regions are contoured correctly, sufficient amounts of labeled data are necessary to train on. Plentiful data allows for when a specific model is deployed, it correctly generalizes upon patient data that it has never seen before. If the supplied data is limited in variation or amount, deep learning models tend to start 'memorizing' their training data causing overfitting of the network [26]. Collecting a large enough data set to avert this has proven difficult in the past. Aside from being time-consuming, privacy laws prevent patient data from being shared [27]. Since patient data is often deeply private with occurrences where it can be tracked back to a specific patient, researchers and doctors are unable to widely publish their data [3]. The coming into force of the General Data Regulation Protection (GDPR) by the European Union in 2018 has made it even more difficult to allow data regarding health to be distributed [28]. Last of all, the rarity of certain conditions or cancers also ensures that contouring could fail [29], [10]. This due to the network never having seen such a condition before. Certainly, this could also happen by any medical professional, yet the black box nature of some of these networks could allow for incomprehensibility by the patient. This, and the ethical discussion that comes paired with fully automatic contouring, won't be discussed in full in this paper but will be briefly touched upon towards the end.

One way to overcome the limited amount of data that is currently employed, is to apply data augmentation (DA) techniques. These techniques usually come in the form of applying transformations on existing images such as shearing, rotation, flipping or scaling. Other times Gaussian noise is added onto an image. According to [30], DA seems to be vital, not only for the limited data cases but for any size of dataset, even models trained on some of the largest datasets. Data augmentation and manipulation is customarily done in the pre-processing step of any neural network framework. While these are effective up to some extent, they do not inherently change the underlying distribution of the data set. Furthermore, these image data sets can be imbalanced leading to poor performance of the applied deep learning network [31]. A more visual way to describe data augmentation is that an image can be changed in order for a face to look more

stretched or mirrored, but that the network will never get an image with just a larger nose or a missing ear. Any trained model which is already deployed will thus have more difficulty segmenting an input image where an ear is missing. This is the current issue with models that encounter new forms of tumors [8].

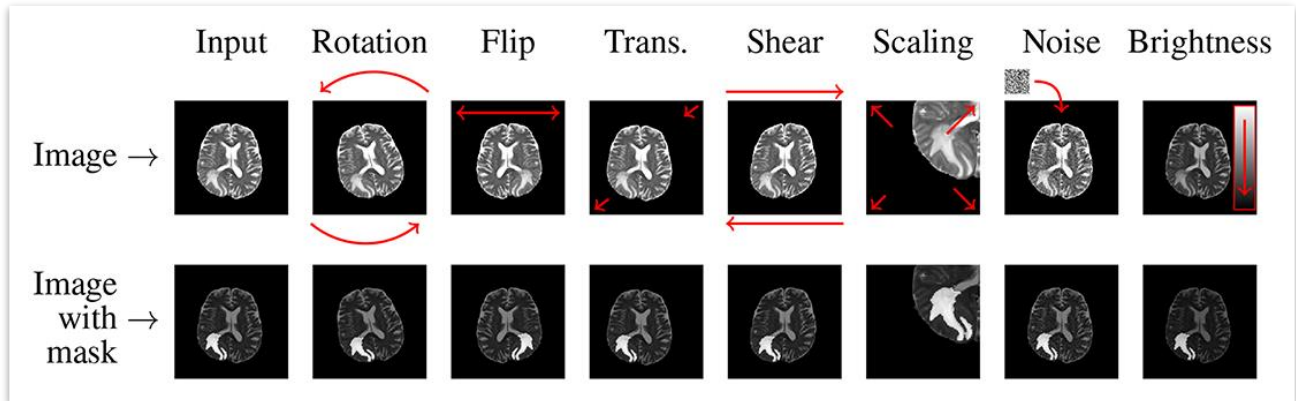


Image 8: Several data augmentation techniques frequently employed in machine learning pipelines

Deep learning models such as segmentation, classification or detection algorithms are routinely judged on their ability to correctly analyze previously unseen examples. Such images commonly face problems such as being “noisy”, slightly “perturbed” or being an outlier as compared to the original data, and therefore more challenging to classify or segment. [32] A perturbation of a CT image can often be caused by a slight movement of the patient. While an outlier could be a child where the algorithm is trained solely on elderly patients. Models trained with enough diversity, should still be able to correctly handle these images [33].

GAN based DA has been attempting to fill the gap left by classical data augmentation techniques. While the intricacies of GAN’s will be detailed in a later section, their application as an augmentation technique has been a recent, novel innovation and the motivation behind this research. Several papers [9] [30] [33] [34] have explored and shown how generated synthetic images can be leveraged as a form of data augmentation to supplement a typical neural network. Shin et al. [33] proposed a generative algorithm to produce synthetic brain tumor MRI images as an additional form for DA and as an effective method of data anonymization. They managed to achieve a higher Dice score with both synthetic and real data combined, as well as a showing that

training on synthetic data only and fine-tuning the model on 10% of the real data can still achieve formidable results. The study done by Bowles et al. from Imperial College London [31] supports these findings. In their results, they state that “*GAN augmentation can provide a modest but significant improvement in segmentation performance in many cases*”. And supplying further evidence for the quality opportunity that synthetic data presents by stating that there are “*no circumstances in which using synthetic [CT] data leads to worse results even when large amounts of real data is available*”. While the SAG-GAN proposed by [34] generates a tumor image from a normal image and vice versa and subsequently showing a boost in accuracy (relative improvement of 25.03%) and area under the curve (relative improvement of 21.91%) compared to traditional data augmentation methods. Costa et al. and Guibas et al. [35] [36] on which this thesis and the corresponding model pipeline is mainly modelled, employs generative models to synthesize new artificial vessel images and the corresponding eye fundus images. Given a dataset of real eye images, the generative models can produce larger amounts of synthetic data to supplement existing datasets, and which are not an image of any real patient, meaning that data produced by these pipelines can also be distributed in the wider medical community. Their research demonstrates that their generative method is “*capable of generating realistic vessel geometries and retinal image texture[s], while keeping the global structure consistent*”.

All these studies demonstrate the opportunity GAN’s can present as a form of synthesizing quality images, and their use for further training as a DA method. Ideally, they would be able to transform the discrete distribution of training medical images into a continuous distribution, while at the same time applying augmentation to each source of variance within the dataset.

2.1.1 Parotid gland

As previously stated, image segmentation of head and neck malignancies is a time-consuming and difficult task in radiotherapy, which hamper the development of appropriate patient care. The presence of large primary or nodal lesions and the effects of surgical procedures can significantly modify the normal anatomy of this site, resulting in the need for laborious manual

segmentation, as auto-segmentation approaches often fail to manage these anatomical variations [14] [24]. Deep learning combined with the GAN augmentation techniques may help in overcoming such difficulties. While this approach could be explored for a wide range of organs in patients, this study is aimed at one specific region of our salivary glands, the parotid glands. These OAR are considered particularly susceptible to radiation and damage leads to a reduced quality of life for patients [37].

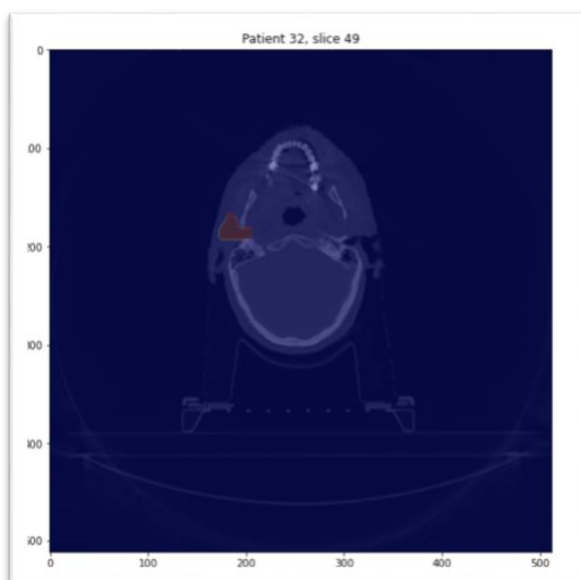


Image 9: Delineated right Parotid Gland CT scan

The parotids aid in the production of saliva which is carried through the Stensen duct to the oral cavity. A benign salivary gland tumor that can develop in the parotid region is adenolymphoma. Parotid gland tumor (PGT) is the most common type of salivary gland tumor. Major PGT types include pleomorphic adenoma (PMA), Warthin tumor (WT), and malignant tumor (MT), according to [38]. Determination of the type of PGT is crucial for clinical diagnosis and subsequent treatment. Damage to the parotid glands could lead to tumors and xerostomia (dry mouth) which reduces the quality of life of the irradiated patient. Current issues with automated contouring is the high degree of variability of the parotid gland in patients as well as their appearance being hard to differentiate from surrounding structures because of low soft-tissue contrast in CT scans [10]. Another common problem is the presence of dental metal artifacts in patients. These artifacts, located in the mouth, cause major distortions in CT scans as can be

observed in the image below.

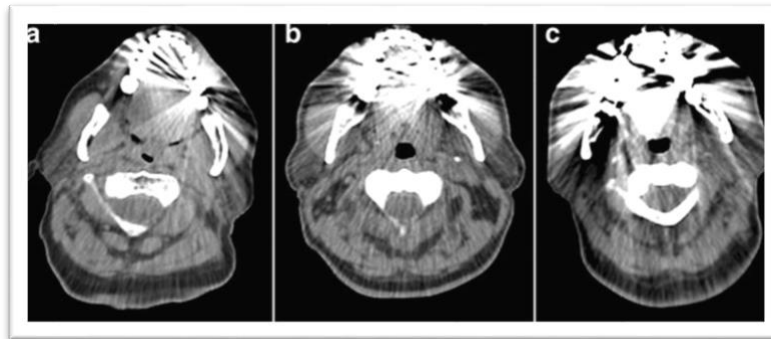


Image 10: Disturbances caused by the presence of metal dental implants in CT images

Deep learning contouring models have shown remarkable results in salivary gland delineation yet have thus far been unable to reach Dice scores higher than 90% [5]. Some of the best scores that were reached on the 2015 MICCAI challenge dataset hover around 0.880 DSC with a 3D U-Net [39]. It is important however to echo the authors of the paper in their discussion of the results in which they mention the difficulty of comparing various DL models. Changing one parameter in these models “directly impacts other network properties” and hence interesting results might not directly correlate with a better model.

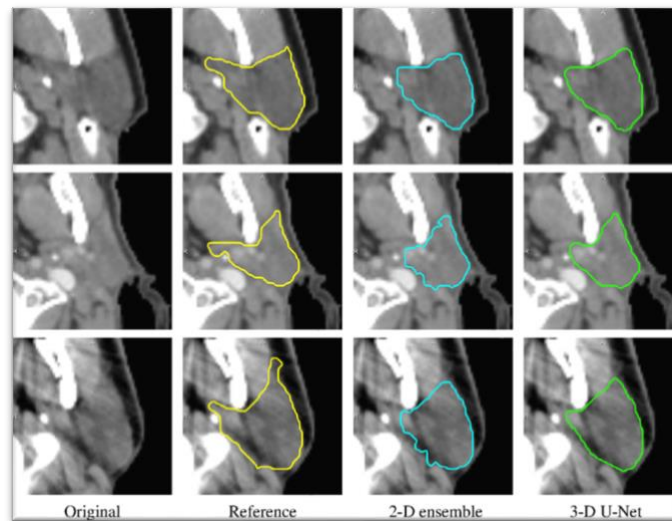


Image 11: Results of automated parotid segmentation on CT scans by 3 different models, from Hänsch et al. [39].

Rooij et al. [5] demonstrated that increasing the amount of available training images as well as data augmentation and patient-specific HU windowing prior to modeling had a positive effect on the results of salivary gland delineation. Improving Dice scores for parotid segmentation from here onwards will become smaller and exceedingly difficult because base-model performance is already quite high. The GAN based DA augmentation for the parotid gland that is being explored in this research will hopefully be one of the many incremental improvement steps taken towards a high accuracy DL contouring model.

2.2.1 U-Net

Current deep learning networks that are employed often consist of several convolutional layers for segmentation. While there is a range of networks that allow for similar segmentation of high-risk areas, one that is frequently coined in the literature, and on which many later models are based, is the U-Net architecture. This architecture was formulated in 2015 to deal with the limited amount of data in the medical domain and yields more precise segmentation with fewer images [40]. The architecture allows for individual pixel or voxel wise segmentation and outputs a label for every pixel of the original image size. A U-Net has received its distinct name because the network foundation consists of a large U. During the contraction part, the feature map of the image is learned and passed to the expansion side of the network. As can be observed in the architecture image, contraction consists of two 3x3 convolution blocks that are downsampled using max pooling. Each convolution block is followed by a rectified linear unit (ReLU) activation function and a batch normalization. Both activation functions and batch normalization will be briefly touched upon before continuing seeing as there are important parts of any DL network, and which will reappear further along.

Activation functions take any real number as input, and outputs a number in a certain range using a non-linear differentiable function. These functions define how the weighted sum of all the input nodes are transformed into an output from a node or nodes in a layer of the network. The non-linearity functions are typically used in the hidden layers of a neural network and allow for a model to learn more complex networks and does not just perform a single layer linear regression. The function being differentiable allows for the network to be able to learn by back-propagation

through having the derivative computed. The most used activation function, which can also be seen in the U-Net image below, is the rectified linear activation function (ReLU). ReLU's preference over other previously popular functions stems from its ability to solve the common vanishing gradient problem when training deep neural networks. It does this by returning zero whenever the incoming value (x) is negative and x otherwise.

$$ReLU = \max(0.0, x)$$

Other activation functions include Sigmoid, where values are normalized between 0 and 1 and TanH, where any real input value is normalized to the output values of -1 and 1.

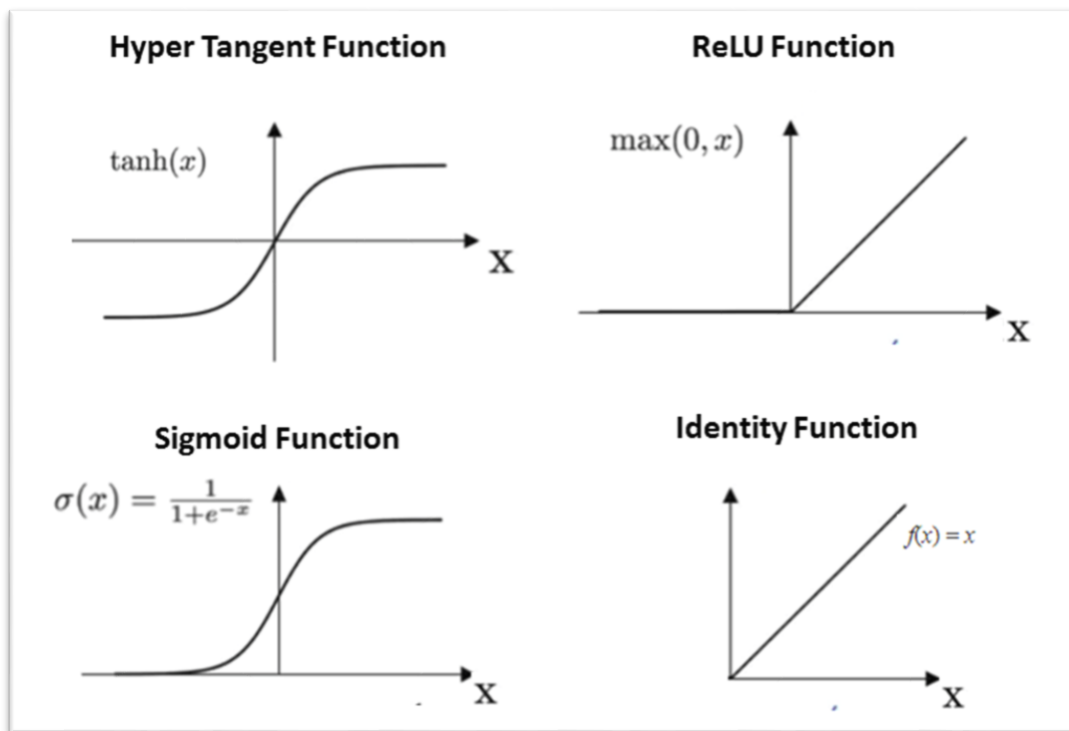


Image 12: Various popular activation functions

Another trick that has been applied by the wider DL community to stabilize and speed up model training is *batch normalization* [41]. In simple terms, batch normalization consists of normalizing input variables using the mean and variance of the current batch of data. This step is applied right before or after the nonlinear activation function to smooth it out and reduce the

(internal) covariate shift. Currently, batch normalization has become a standard strategy to achieve faster and better convergence in any DL model training, ensuring that these unstable models are better trainable. For a comprehensive guide to batch normalization, including current uncertainties, this author would like to direct the reader to the footnote on the bottom of this page³. Back to our original U-Net, the right part of the network allows the up-sampling of the image with 2x2 deconvolutional layers from the feature map of the input. Up-sampling can be described as the layered opposite effect of pooling. It takes an input and upscales this to a higher dimension and thus resolution. Similar to pooling, there are many different ways to achieve up-sampling such as by computing neighboring pixel values and applying bilinear interpolation. The grey arrow connections allow for the concatenation and the copying of high resolution features. The massive success of this original design in 2D segmentation has been extended to allow for 3D images and MRI specific medical volumes which allow for a better preservation of the underlying 3D structure [42].

³ Batch Normalization in 3 levels of understanding by Johann Huber, November 5, 2020
<https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338>

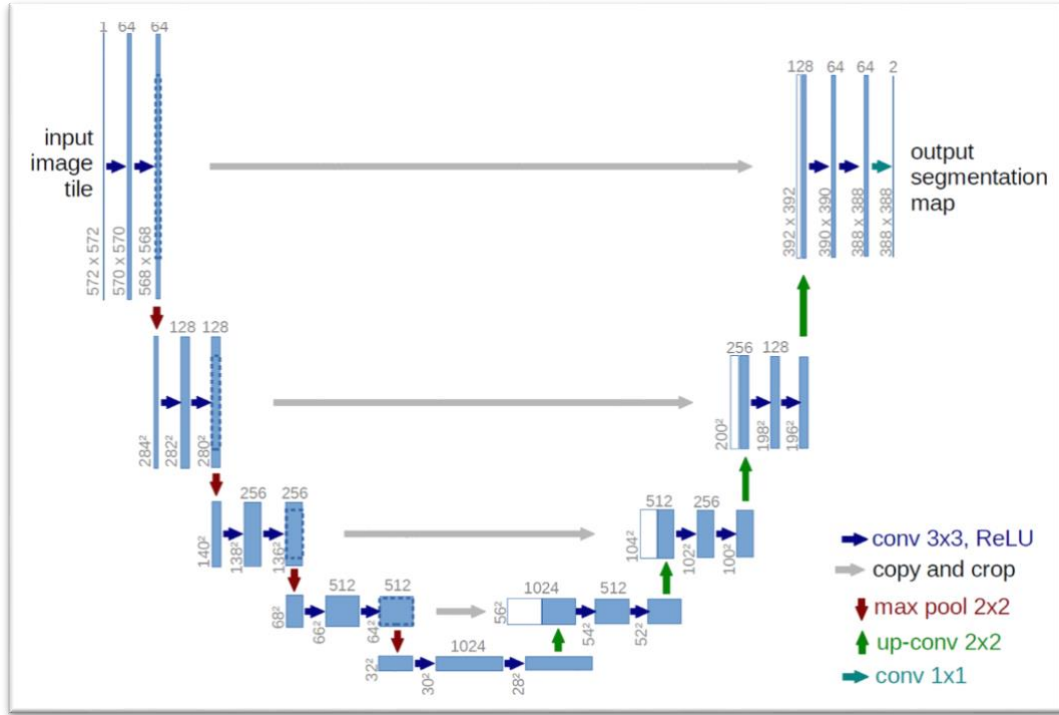


Image 13: Original U-Net architecture as proposed by Ronneberger et al. in 2015

One will be able to test the effectiveness of generated synthetic data by our framework, by first having a standard U-Net trained and deployed on an existing data-set with the usual augmentation techniques. Adjacent to this, a similar model can be trained but this time with the synthetically generated parotid images and masks included in the training process. As hypothesized, this second network will potentially outperform in its delineation task in comparison to the model without the synthetic data. The performances of the two U-Net models can be analyzed by comparing their respective Dice similarity scores.

Current practices put the best mean Dice score for parotid gland segmentation at 0.84, achieved at the 2015 MICCAI challenge [23]. In further research our GAN DA technique can be tested against a standard trained U-Net model on a given image and the results compared to test effectiveness.

2.2.2 GANS

Having touched upon most the prerequisite knowledge previously, this section will dive deeper into the how and what of GAN's. As already stated, a possible solution that will be explored in this paper is to employ Generative Adversarial Networks (GAN's) to create artificial images with parotid gland variability. With the advent of GAN's initially developed by by Ian J. Goodfellow [43] in 2014 and the proliferation thereafter of different variations, more light has been shed on their potential use cases. Their potential applications have also been recognized by the wider medical community. With the recent addition of GAN-based frameworks as an semi-supervised deep learning approach, significant new use cases for rendering labeled data have emerged [2]. With their corresponding *generator* (G) and *discriminator* (D), they have been increasingly used in generating synthetic but realistic images that help in obtaining the much needed, aforementioned, variation in patient data [6] [15] [44]. Generative models are a rather recent innovation but have proven themselves useful in varying fields and applications [45], [46]. Especially in imaging, GANs have outperformed other generative deep learning models such as variational autoencoders(VAE). The first proposed GAN, now often called the Vanilla GAN, consists of two parts [43].The initial component consists of a generator, shown in the figure below, which after training, generates a new image.

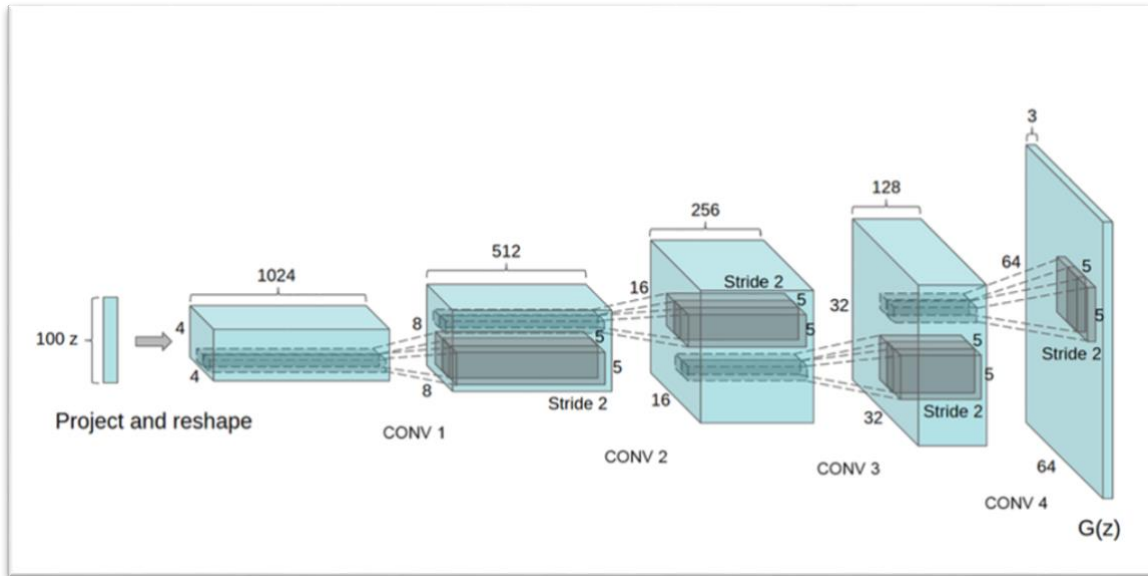


Image 14: Generator Architecture

The discriminator is the second component and classifies the generated data as either real or fake. The goal is to have both networks minimize their cost and to reach a Nash Equilibrium where the generated images are judged as sufficiently real by the discriminator. An intuitive analogy is often made with an art forger and critic. The forger, thought of as the generator, increasingly attempts to create improved fake art. While the critic, see discriminator, progressively gets better at distinguishing the forged art from the real art. The goal for the forger is to create art that is as realistic as possible.

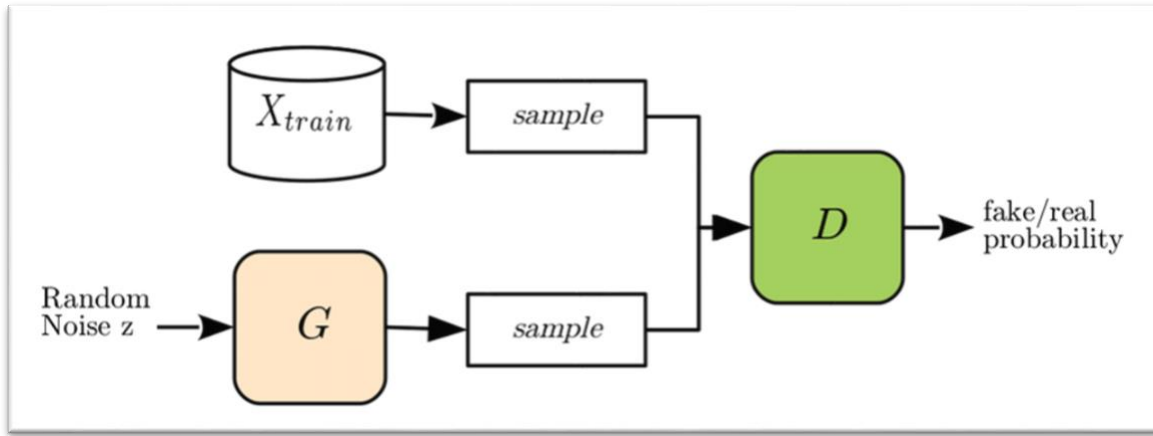


Image 15: Vanilla GAN framework as proposed by Goodfellow et al. [43]

This dual component network has made it particularly strong in generating data over the probability density distribution of the input data [4]. As well as the ability for GANs to discover the high dimensional latent distribution of the data, leading to the successful generation of visually state-of-the-art synthetic images. To update the network, and reach an equilibrium between the discriminator and generator components, a gradient loss function is back propagated and used to update the networks parameters.

The loss function used consists of a Min-Max equation. The discriminator is updated to maximize the probability that the image is correctly labeled as real or fake.

$$Loss(D) = \max[\log_x D + \log(1 - D(G_Z))]$$

While the generator attempts to minimize the probability that its generated images are classified as fake.

$$Loss(G) = \min[\log(1 - D(G_Z))]$$

These two functions can be described as competing with one another by adjusting their parameters depending on the moves of the other ‘player’. Each player has only control over their own parameters yet are dependent on the outcome of the other.

What makes generative models especially powerful is their ability to work with unlabeled data which is often plentiful but not always useful. This is especially true in the biomedical domain. While the optimization is theoretically well supported, in practice it has been found quite hard to reach a stable equilibrium. Convergence of the two components can initially be difficult to achieve and depends on hyper-parameter tuning (on which later more). When the discriminator quickly becomes too strong in discriminating between the real and fake images, no valuable information is passed to the generator anymore [45]. The gradient feedback approaches zero and the model cannot be updated anymore [29]. This problem can initially be observed in the first few cycles of our GAN training. Here the output of the network is still immature and produces poor quality images.

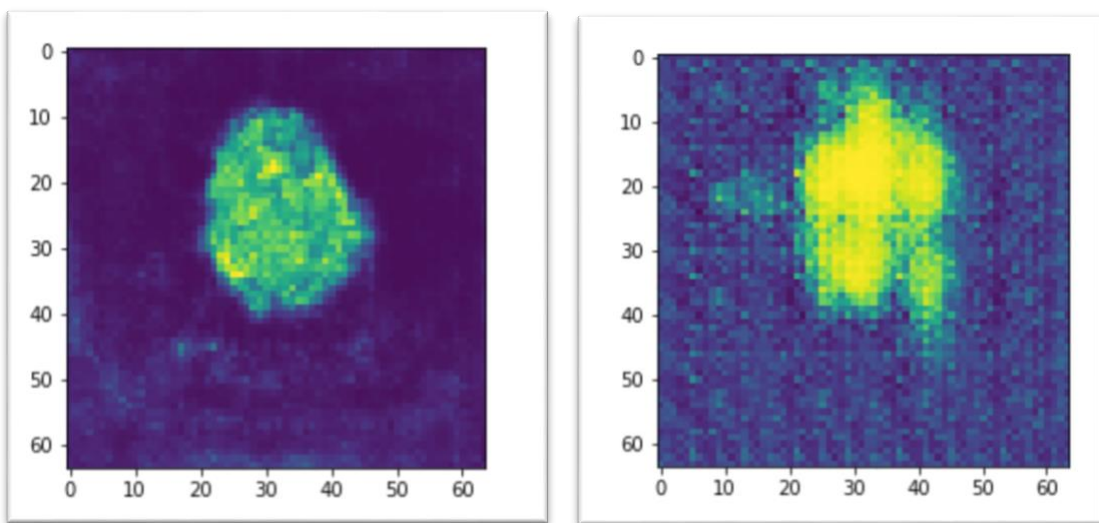


Image 16: Early epoch GAN head(left) and parotid(right) generation results

Another problem often encountered in GAN training is mode collapse. This occurs when the synthetic output samples cover only a very few limited modes of the data distribution [3] [29]. This is a very relevant problem in our framework since we are attempting to have our outcome data distribution as diverse as possible to ensure a wide variety of produced images.

Encountering mode collapse will reduce the overall effectiveness of our hypothesis. Ideally, the outcome probability density distribution should be as spread as is feasible.

To overcome the problems associated with the vanilla GAN, several frameworks extend the architecture to deal with these instabilities and to generate higher resolution images [46]. Which is ideal for our use case since reducing the spatial size of the image to fit the vanilla GAN hurts the performance of the segmentation process by U-Net in a later stadium. Enabling the stable generation of samples with a greater spatial size (eg. 128x128 or 256x256) is thus highly desirable.

One such improved architecture that has proven popular in the literature is the DCGAN [47]. Instead of fully connected layers, deep convolutional up and down layers are utilized. Batch normalization is applied to ensure that the input batch is normalized around the mean with a standard deviation during GAN training. A Leaky ReLU is likewise used instead of a linear activation function. This ensures that individual neurons in the network are maximized on the input. The Leaky addition safeguards from nodes getting stuck on a zero derivative and not being able to update their corresponding weights anymore. Previously, this could prevent those specific nodes of the network from learning [48] [49]. Although mode collapse can occasionally still cause problems, the aforementioned improvements greatly boost training stability. This architecture has already been tested to generate high-quality image samples of lung nodules and liver lesions which easily deceive professionally trained radiologists [6] [50]. And has improved liver lesion classification with synthetic GAN based DA from 78.6% sensitivity and 88.4% specificity to 85.7% sensitivity and 92.4% specificity compared to normal DA methods.

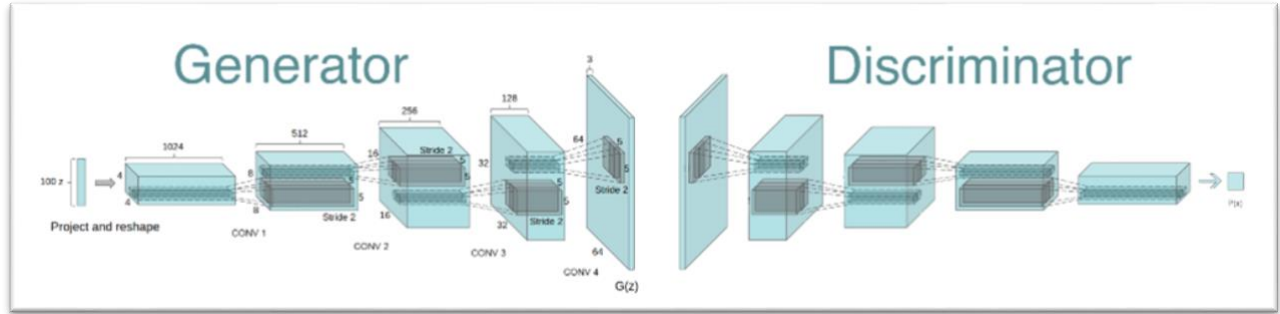


Image 17: DCGAN architecture proposed by Radford et al. [47]

This extra stability, as well as the frequent use of the model in the literature, has led to the decision to use a DCGAN in our framework. Further changes in loss functions and additional methods such as weight clipping have been proven effective in certain conditions as well. However these will not be applied in this paper since there is less literature available, are only theoretical proposals, or are not applicable to our data set. Additionally, comparing GANs has been proven difficult as there currently does not exist a single metric to do so. Performance also depends on hyper parameters, random seed, the data set and computational budget [48].

The second GAN that will play a role in our framework is the conditional GAN (cGAN) [51]. To achieve the goal of controlling the generation of a specific part of the 2D computed tomography scan image, some conditional input has to be applied to specify this focused generation. Instead of the usual random noise provided to the generator, additional prior information is supplied in the form of the conditioned label, c .

$$\min \max V_{(D,G)} = [\log_{(x|c)} D] + [1 - \log D(G_{(z|c)}c)]$$

In this case, the generation that our image is conditioned on is the labeled parotid. Since this concerns the problem of mapping the parotid mask to the corresponding CT slice, an image-to-image conditional framework is used. Specifically, a Pix2Pix conditional generative adversarial network [52]. The target image is a realistically looking but fake CT scan that fits the

corresponding parotid mask. The change that is incorporated in the Pix2Pix cGAN is a modification in the loss function to ensure that the generated image is plausible in the content of the target image, and is a credible translation of the input image [53].

The generator and discriminator in the Pix2Pix architecture consist of a U-Net and PatchGAN respectively [53]. The U-Net is identical to the one discussed above and which could be used for comparison of our results in future research. It allows for the generation of CT scans from our parotid segmentation mask. The synthetic mask gets chained with the original real CT image and used as input for the discriminator. The PatchGAN thereupon classifies patches from the image as either real or fake in a matrix. An additional pixel loss term is implemented to supply more information to the generator about the real CT target image.

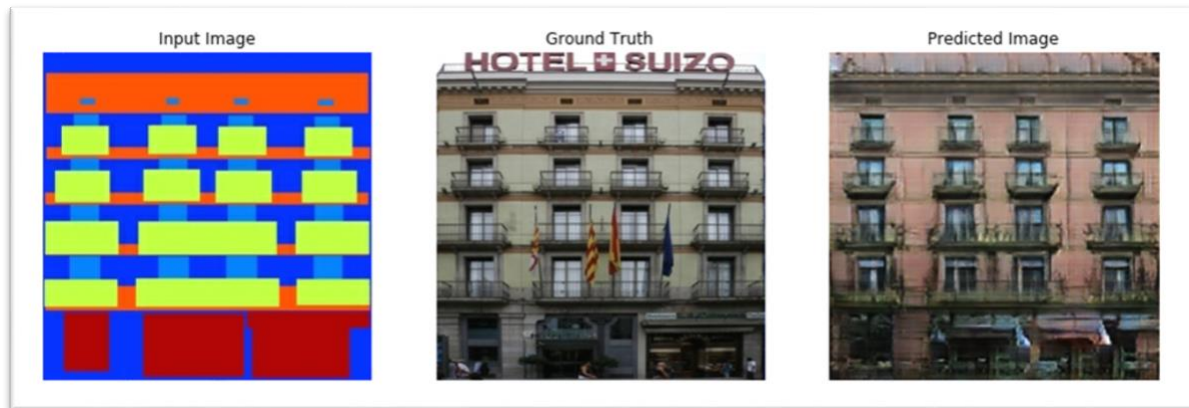


Image 18: An example of a Pix2Pix GAN output

As mentioned, performance of a model depends on a wide range of configurations and dimensions. This can even lead in differing performance by merely changing the random seed or computational time. Hopefully future research will present a more definitive metric to measure outcome of various GAN architectures and losses [48]. New research has also yielded the ability to generate higher resolution images with less training data [54].

3. Experimental Framework

The following chapter will describe the items used in the framework. These include some of the pre-processing steps that have been applied, the external tools and libraries used. Cross references to the implementation in the appendix allow for easier understanding. Furthermore, a justification will be given as to the choice of methods used. Below a full schematic layout is presented with both internal and external systems for this project and the conversion of the data through this pipeline of systems.

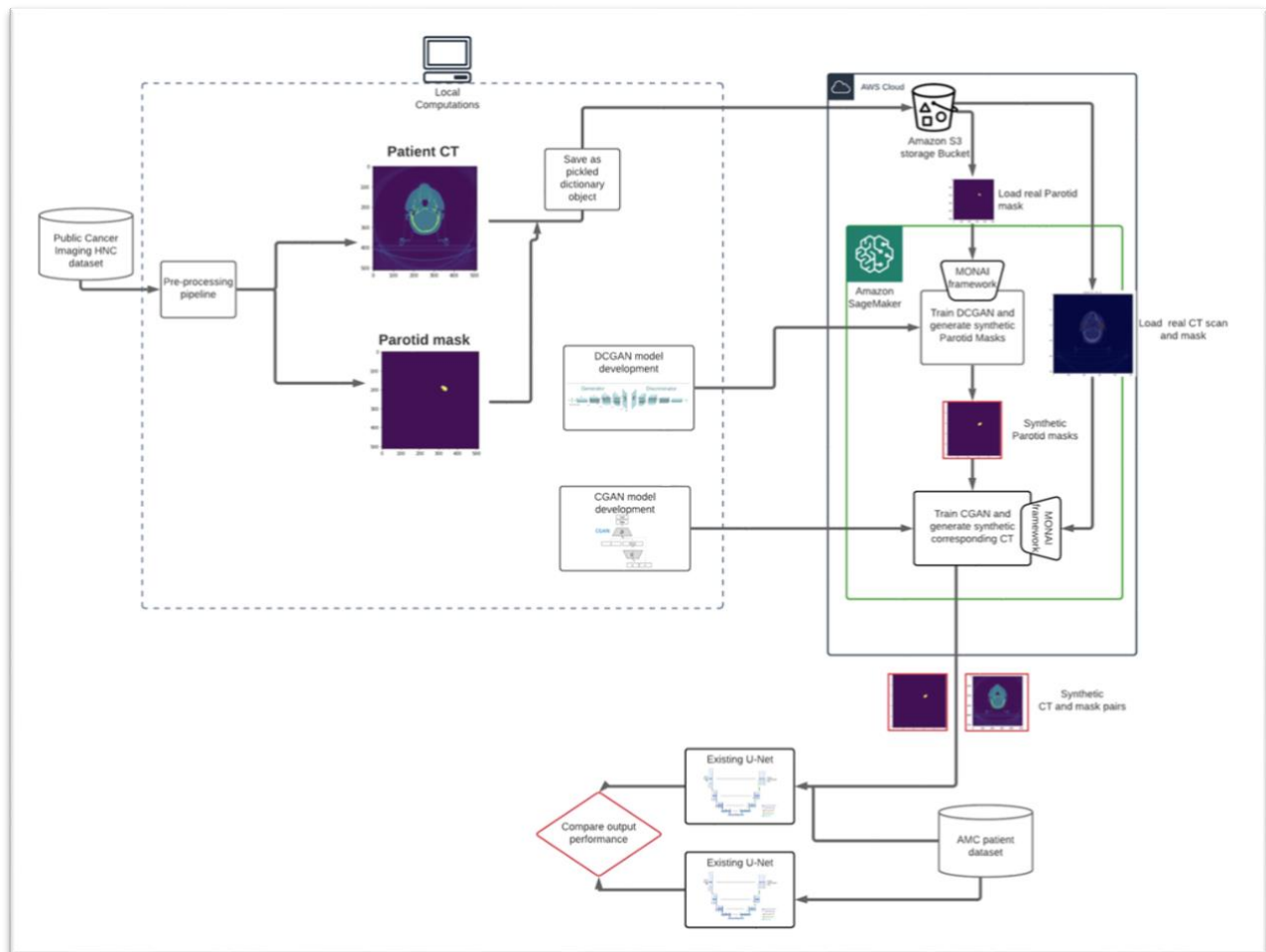


Image 19: Proposed synthetic generation Framework

3.1 Proposed Architectural Design

3.2 Data Set - Cancer Imaging Archive

The data set that was used in this paper originated from the Cancer Imaging Archive, specifically the Head and Neck Cancer CT Atlas set. It included CT, MR, PET and various RT file modalities on 627 patients. The CT scans and the RT structure file were manually selected. The RT Structure Set file defines a set of areas of significance in radiation therapy, such as body contours, tumor volumes, OARs, and other ROIs [55].

The CT files come in the form of 2D slices and can range from circa 100 to 320 slices depending on the utilized imaging device (Philips, Siemens or GE) and whether CT was extended in the taken image to include parts of the chest. The CT files are formatted as Digital Imaging and Communications in Medicine (DICOM), and contain necessary overhead data such as voxel spacing and slice thickness. All other files are discarded. 58 corresponding patient CT and RT files were saved and used as input for the data pre-processing step. [Appendix A: Data Pre-Processing](#) outlines how this data is loaded and the following steps taken to curate these 58 patient files.

3.3 Pre-Processing

The data pre-processing step was done with the help of a pipeline created by PhD students at the Amsterdam Medical Centre and for which this author is very grateful. This ensured that important DICOM meta data was properly conserved. Both the RT and the corresponding CT files were loaded in for all the patients and checked whether the labeled parotid was present. As can be observed in the [appendix](#), individual patient files had to be checked to allow for the variability in the scan data. The dimensionality of most CT files was 512x512 with between 100 to 320 slices. With the spacing between voxels being on an average of 0.95x0.95x3.

Some patients (~7) were already filtered out since either their RT file did not correspond with the CT files or the parotid was not labeled by a physician. Manual inspection had to be made since head slices had differing slice starting points. Image 20: Head and Neck CT scan displays the results of one such black and white loaded slice. One can observe the lower section of the head

along with a vague rendering of the upper part of the left shoulder. Additionally, parts of the CT imaging scanning device can also be seen and allows for an easy understanding of body orientation.

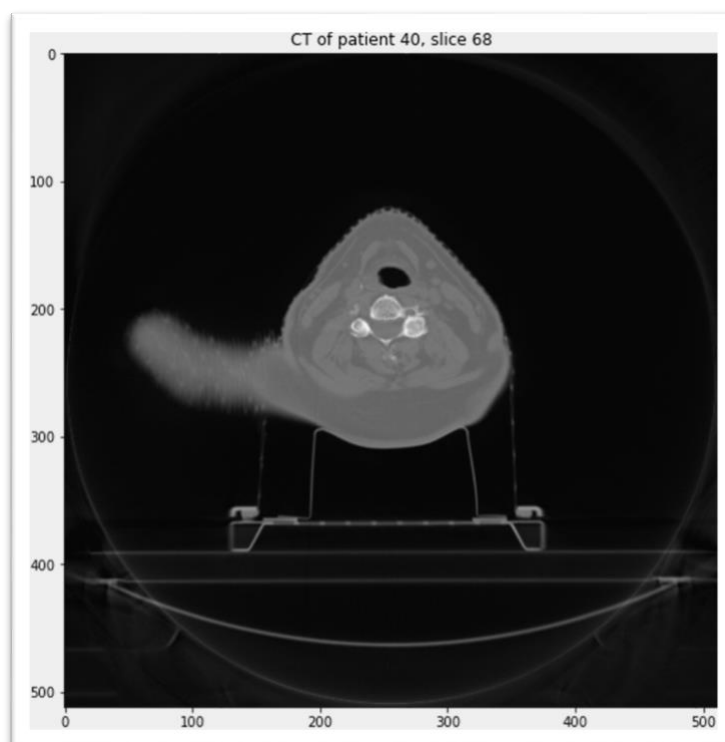


Image 20: Head and Neck CT scan

Next, the RT Structure Set file was loaded and matched with its equivalent CT file. From this structure file, the necessary left and right parotid contours were captured and the various other contours such as the spinal cord, brainstem and larynx, discarded. Some tweaks had to be made to accommodate the differing data. These included but were not limited to, allowing for a larger flexibility in parotid naming since contour naming conventions by physicians are not standardized (e.g. Left parotid vs L Parotid vs Lt parotid). Another confronted problem was a reversal of slice ordering where the resulting contours would be drawn on the ankles. As well as manual inspection exhibiting contours that would fall outside the HNC area, displayed in the image below.

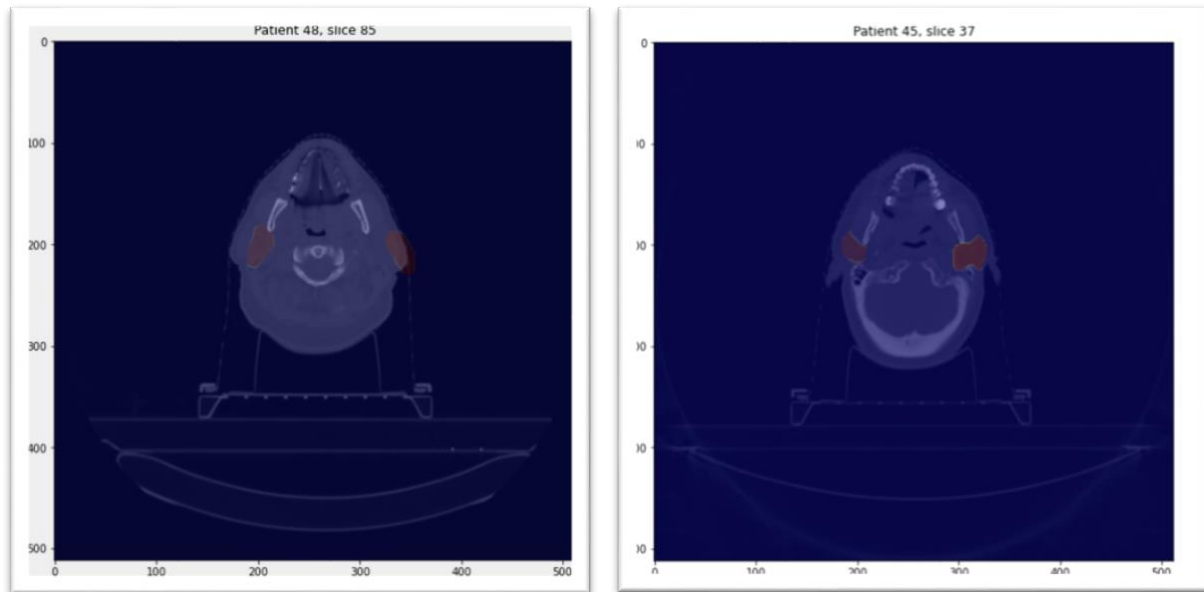


Image 21: Wrongly associated CT head scan(left) and correct CT scan with parotid mask after pre-processing (right)

This culminated in the manual selection of a range of head slices with a parotid mask present which were saved as pickled file objects accordingly. Empty CT slices of the same head and neck area were saved as well to serve as a reference point for elements farther in the model pipeline. The results were 6 patients with correctly drawn in contours, averaging 30 to 40 head and neck slices each, as well as 3 empty patients. Allowing for a dataset of ~250 2D head and neck slice images. Ideally this would be expanded but storage issues prevailed.

3.4 Amazon Web Services (AWS)

To allow for easier access and sharing of the necessary data, and to hedge against any hardware failures, the CT's and masks for the patients was uploaded to an Amazon Web Server S3 data bucket in the cloud.

To deal with the high computational demand usually associated with training the networks, a decision was made to train the duo GAN models with Amazon Sagemaker ⁴. GAN training is GPU hungry and thus was moved to the cloud while model development was done locally.

⁴ Sagemaker can be found on: <https://aws.amazon.com/sagemaker/>

According to Amazon "SageMaker helps data scientists and developers to prepare, build, train, and deploy high-quality machine learning (ML) models quickly by bringing together a broad set of capabilities purpose-built for ML". This allows for easy scaling of the necessary hardware.

Appendix B demonstrates the starting of a Sagemaker session and where to find the bucket in which the list of 2D HN images are stored. Two notebook ml.p2.xlarge instances were requested and granted which allowed for training of the network to occur on Tesla K80 GPU's.

3.5 MONAI

A rather novel developed framework on top of PyTorch for deep learning in healthcare imaging is the Medical Open Network for AI (MONAI), released in April 2020 ⁵. The foundation provides state-of-the-art training workflows and optimized deep learning models. Often, medical images require specific augmentations, pre-processing and specialized input and output modalities. MONAI provides this, as well as improved hardware acceleration. Considering the usual immense demand that deep learning training has on memory hardware, this framework provides a welcome approach in reducing this. One such way, and which is applied in our setup, is through caching data and storing intermediate results when training over numerous epochs.

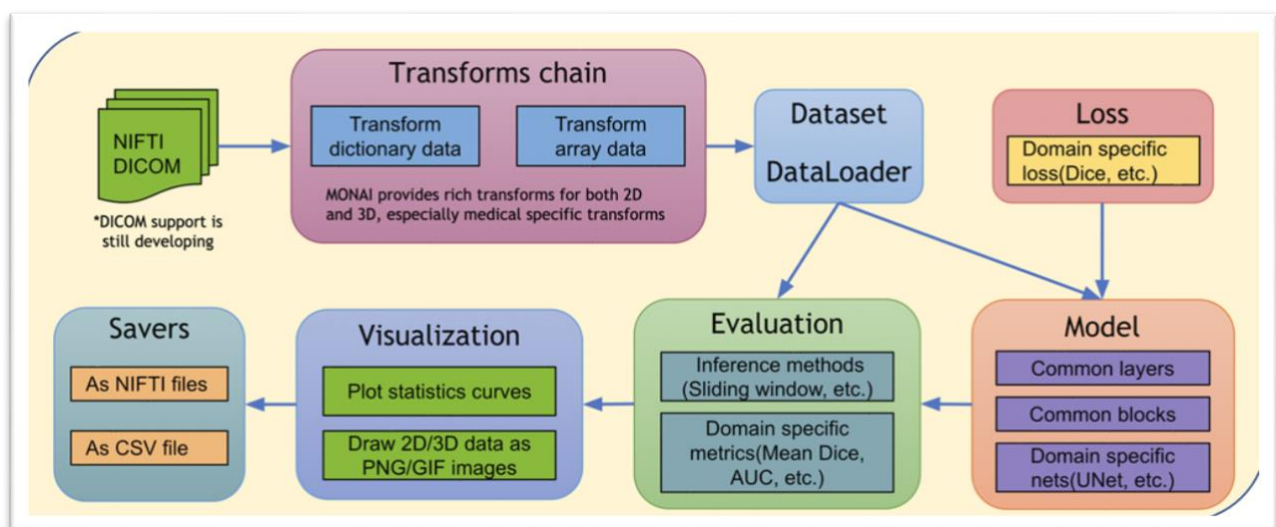


Image 22: MONAI end-to-end workflow

⁵ <https://monai.io/>

Moreover, several frequently used architectures such as an elementary U-Net are already implemented and can be easily build upon. All of this aide in creating an effortless pipeline that limits cloud computational overhead. Details on the implementation of this framework can be found in Appendix B: DCGAN on AWS. The *get_dataloader_sagemaker()* function chains several functionalities in series after parsing in the pickled set of 2D images. A window of 64x64, with a focus on the parotid was extracted and tensorized. The tensor images were cached and batched accordingly. MONAI has been a considerable help in deploying our GAN, of which the implementation details are outlined in the following chapter.

4. GAN Specific Implementation Details

In the next section, the employed dual GAN models will be further elaborated in detail. Figure 1, depicted previously in the introduction, paints an accurate high-level overview of the location of our dual stage GAN design in the overall process.

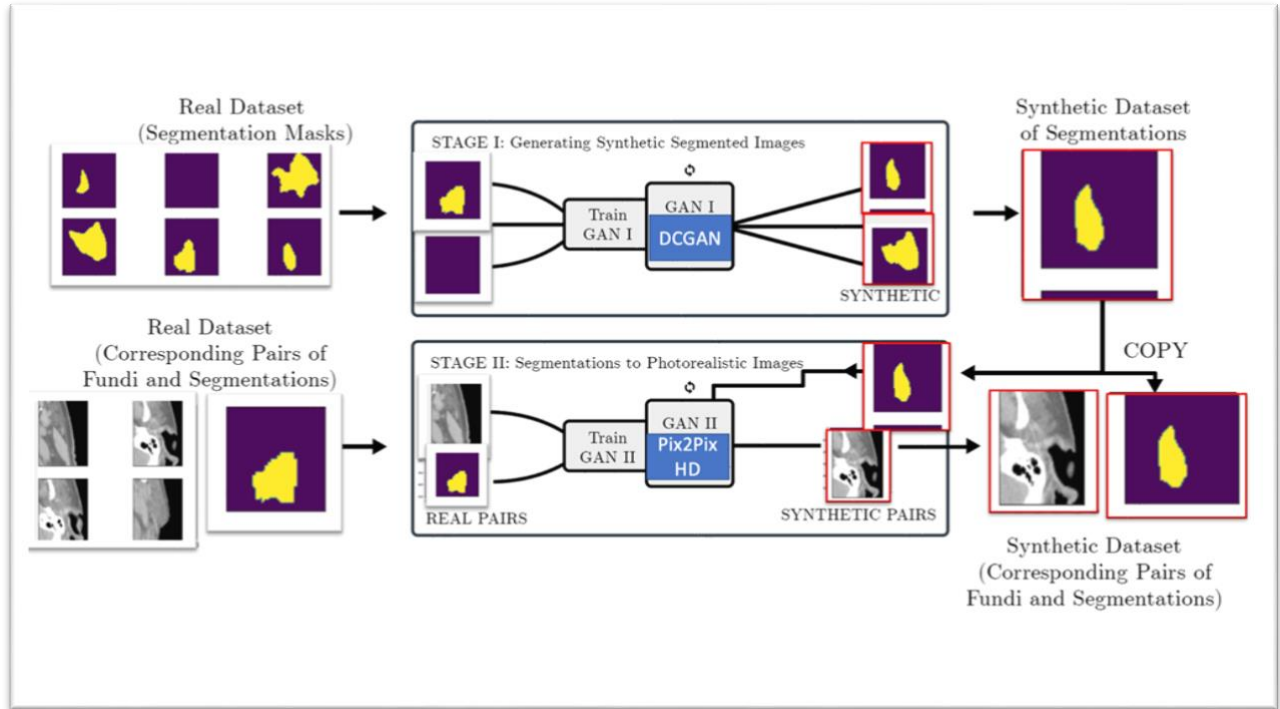


Figure 3: Proposed high level overview of system design

4.1 DCGAN parameters

As mentioned before, a DCGAN is employed for the parotid mask generation task. The input consists of 300 512x512 ground truth masks that are spatially cropped around the parotid to be a feature map of 64x64. Image 23: Input parotid mask visualizes the input segmented mask of the parotid.

The generator class starts with four convolutional up-sampling layers (Transposed convolutions) and with a generated uniform noise distribution vector (z_{noise}) of 100. The channel for the input image is set to 1 since the mask is in a binary black and white, instead of 3 for RGB images. The feature maps, starting at 64, ($conv_dim$) are reduced with every downstream layer.

Following the optimization recommendations for DCGAN's over the original Vanilla GAN, batch normalization and the ReLU activation function are applied at every layer. The last layer concludes with a *Tanh()* activation function to normalize the return of the input from the generator to be between the range of -1 to 1.

Hereafter, the discriminator class is called upon to ensure that the generated fake images are accurately classified and to provide feedback to the generator. Here the number of feature maps equal the feature maps used in our generator. A similar layout is used, however with a Leaky ReLU to all layers except the last one where the final probability (of the image being labeled real or fake) is passed through a single *sigmoid()* output. Batch normalization is not applied to the last layer since [47] has found that this caused 'sample oscillation and model instability'.

Together with a Leaky ReLU, both aid in a robust gradient flow.

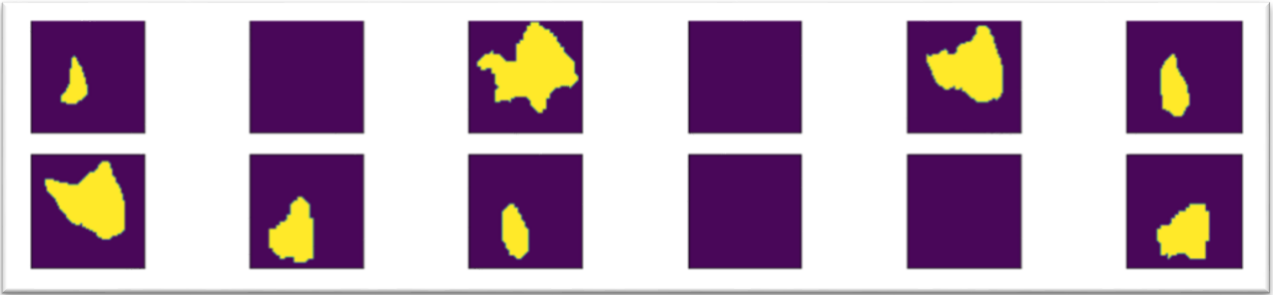


Image 23: Input parotid mask

A learning rate of 0.0002 was found to be ideal. The number of channels for the training image was changed from three to one to accommodate for the masks being black and white and not RGB. Following the official DCGAN literature, the Adam optimizer is used with β_1 and β_2 parameters set to 0.5 and 0.999 respectively to further combat 'oscillation and instability'.

A Binary Cross Entropy(BCE) loss function is used since the images are classified as either true or false, represented as the target value y (0 or 1). Whereas p represents our prediction coming from the sigmoid layer in our network (also between 0 and 1).

$$BCE(y, p) = -(y \times \log(p) + (1 - y) \times \log(1 - p))$$

The BCE loss function increases exponentially with a larger difference between the target and the prediction, allowing for predictions that are very wrong, to be punished significantly more. Making this loss ideal for problems concerning binary classification outcomes.

Following the criteria set out by the Vanilla GAN paper [43], the real label is defined as being 1, while the fake as 0.

After defining all individual GAN components, training was commenced with the *train()* function on the provided cloud GPU. The number of epochs was capped at 500 after which no further visual change in the quality of the output was observed. Each individual epoch iteration includes both a forward and backward pass over all the batch training data. Some early epoch outcomes are visualized in the results section (Chapter 5). Fine tuning of the epoch count was done by manual inspection of intermediate outcomes as well as observing whether mode collapse occurred. Recommendations, such as the normalization between -1 and 1 of the input images, from [32], [48] and *ganhacks*⁶ further aided in the stabilization of image generation. Training time varied per experiment, but generally higher batch sizes led to increased training time with an unchanged epoch count. Several experiments were also performed with differing batch sizes (6, 12, 16, 32) with the generator model and several late epoch visualizations saved accordingly. Testing of the DCGAN was also done on training images with original CT scans instead of RT mask images. This was simply realized by replacing RT to CT in the *get_dataloader_sagemaker()* function. These experimental outcomes can also be observed in the following Chapter.

The outcome synthetic image masks were saved and passed into the second part of this synthetic DA generative framework, the Pix2Pix model.

⁶ <https://github.com/soumith/ganhacks#authors>

4.2 Pix2Pix parameters

For the task of mask to image generation, a Pytorch implementation of the official Pix2Pix image-to-image network was used. The input to this model are two corresponding ground truth images as previously stated. Making the generator learn the mapping relationship between the real data to the fake data by encoding and decoding. Among them synthesized two images (abnormal and normal) into abnormal one, which ensures the legitimacy of the structure but the diversity of it is limited.

The generation, from the mask of the parotid (A) to the parallel ground truth computed tomography image (B), combined with a randomized normally distributed noise vector. The next figure displays the input ground truth CT images that are 128x128 cropped with a focus on the location of the right parotid. The images are clipped between 700 and 1200 Hounsfield units (HU) to allow for greater contrast in the resulting image.

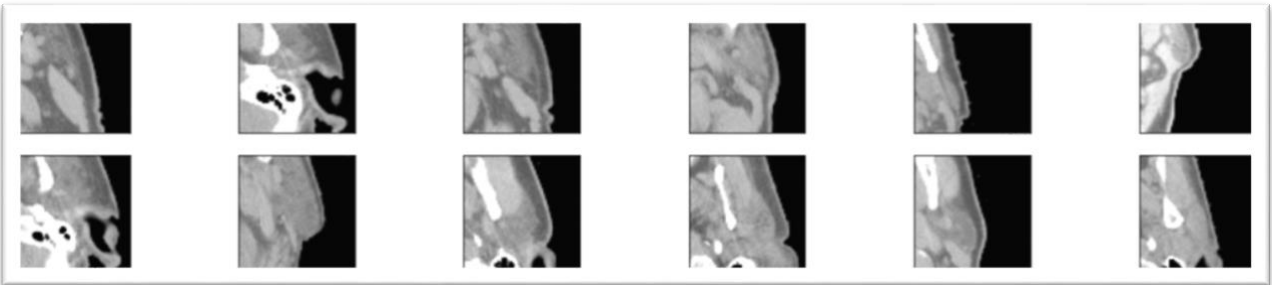


Image 24: Ground truth parotid images

The learning rate was equal to those of the DCGAN, yet a smaller batch size of 4 was utilized. A linear decay rate is used with 100 iterations to zero. Visual results were collected after training for 300 epochs. Training was performed on one Tesla K80 GPU. For both the mask as well as the matching training image, the input channel was changed from the default 3 to 1 for black and white.

The loss associated with Pix2Pix model is a combination of the U-Net generator and the PatchGAN discriminator together with an L1 loss to produce images nearer to the ground truth.

$$G = \min_g \max_D \mathcal{L}_{CGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

With the \mathcal{L}_{CGAN} section detailing the discriminator and generator components as being:

$$\mathcal{L}_{CGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[1 - \log D(x, G(x, z))]$$

Where $D(x, y)$ is in early iterations trained to be close to 1 and $D(x, G(x, z))$ represent the fake images or 0. During the generation training, these labels are reversed to ensure the learning of the original distribution.

4.3 Problems addressed

Several problems were addressed specifically to the implementation which a future replicator might take into consideration when attempting the use this framework. First, to be able to ensure a that all the images are accurately processed in, the data loader was kept equal for both models. Some data augmentation was performed on the Stage I parotid mask images while this was not the case for the Stage II mask-image pairs.

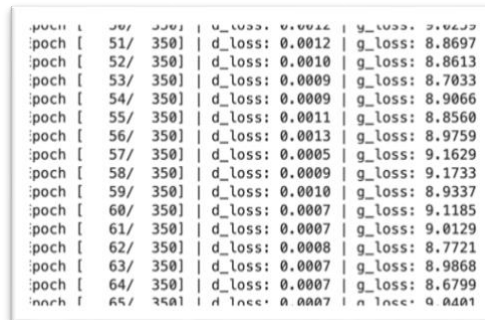
Windowing took a significant amount of time for both mask images as well as parotid area in the original CT images. There is a lot of manual visual inspection taking place to ensure the right area is visible. During the pre-processing this also proved to be a bottleneck and should be simplified throughout the dual Stage network in the future.

5. Results

In the following section, the results from the hypothesis testing are detailed. First the creation of artificial masks is discussed, after which the image-to-image translation is elaborated on. The results discussed are of a qualitative nature. Further quantitative analysis by comparing traditional data augmentation (TA) with our synthetic data augmentation (SA) in for example a VGG-UNet would add additional support for our described architecture.

5.1 Stage I results

Initial tests with a larger batch size of 64 images resulted in convergence failure. While both losses frequently start erratic, Image 25: Batch size 64, discriminator collapse shows the discriminator performing very poorly and not improving with increased epoch count. Here the generator and discriminator fail to find a stable equilibrium. The discriminator continuously hovers zero and is unable to recover. The resulting 64x64 parotid synthetic mask images are displayed in Image 26: Batch size 64, synthetic mask output



epoch	batch	d_loss	g_loss
epoch [51/ 350]		d_loss: 0.0012	g_loss: 8.8697
epoch [52/ 350]		d_loss: 0.0010	g_loss: 8.8613
epoch [53/ 350]		d_loss: 0.0009	g_loss: 8.7033
epoch [54/ 350]		d_loss: 0.0009	g_loss: 8.9066
epoch [55/ 350]		d_loss: 0.0011	g_loss: 8.8560
epoch [56/ 350]		d_loss: 0.0013	g_loss: 8.9759
epoch [57/ 350]		d_loss: 0.0005	g_loss: 9.1629
epoch [58/ 350]		d_loss: 0.0009	g_loss: 9.1733
epoch [59/ 350]		d_loss: 0.0010	g_loss: 8.9337
epoch [60/ 350]		d_loss: 0.0007	g_loss: 9.1185
epoch [61/ 350]		d_loss: 0.0007	g_loss: 9.0129
epoch [62/ 350]		d_loss: 0.0008	g_loss: 8.7721
epoch [63/ 350]		d_loss: 0.0007	g_loss: 8.9868
epoch [64/ 350]		d_loss: 0.0007	g_loss: 8.6799
epoch [65/ 350]		d_loss: 0.0007	g_loss: 9.0401

Image 25: Batch size 64, discriminator collapse

While it is certainly possible for the DCGAN to recover from a convergence failure at higher epoch counts, this was found not be the case, after which the training process for this specific batch size was halted.

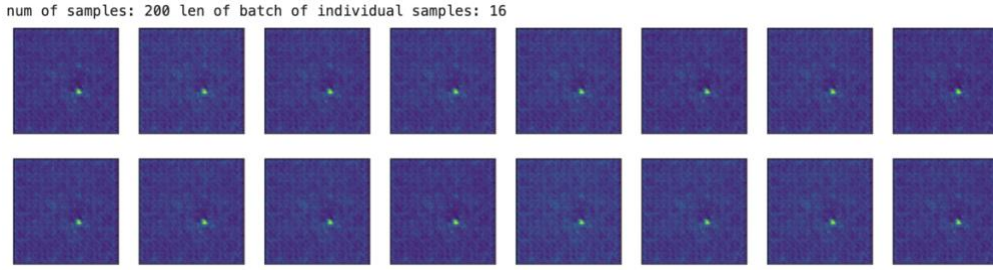


















Image 26: Batch size 64, synthetic mask output

Several tweaks such as lowering the aggressiveness of our Adam optimizer and changing our beta values were made but failed to achieve the desired results. These mask generation results are visualized in Table 1: Generated mask results with various epoch counts 1 below. The bottom row displays a set of reference masks that are used for the input.

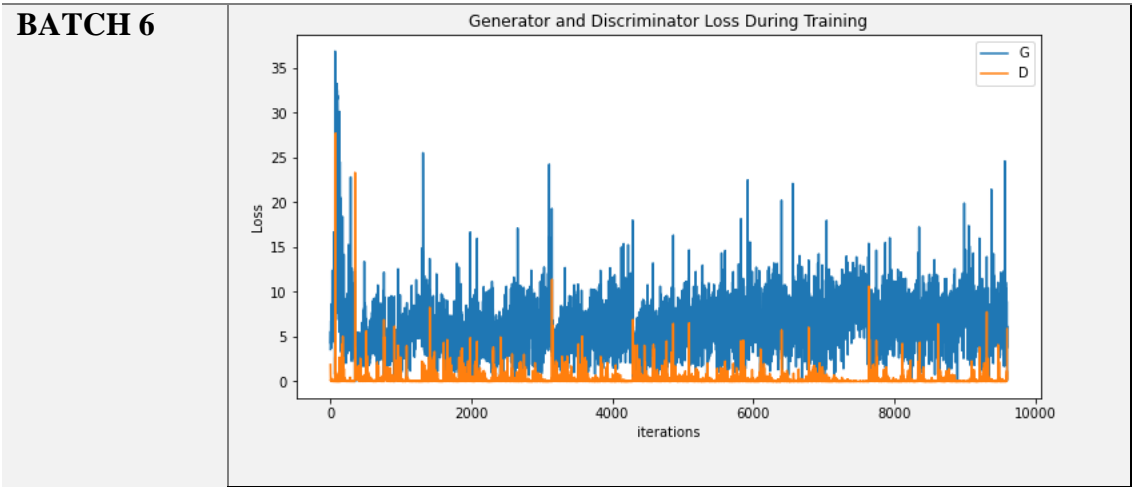
Table 1: Generated mask results with various epoch counts

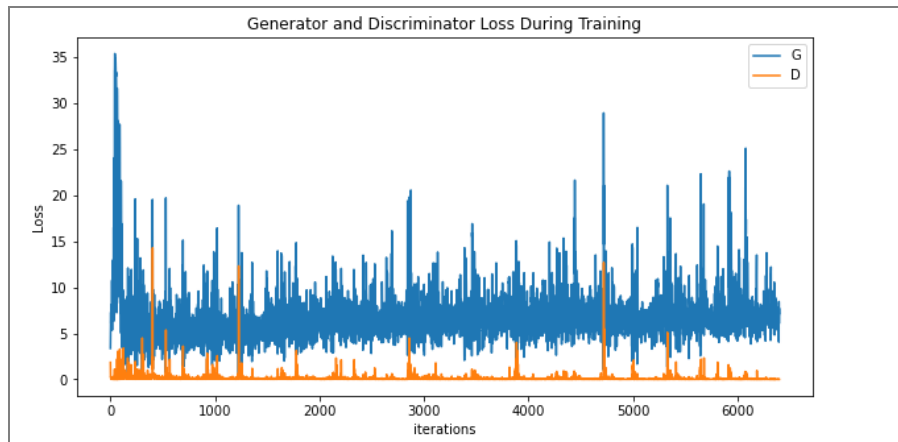
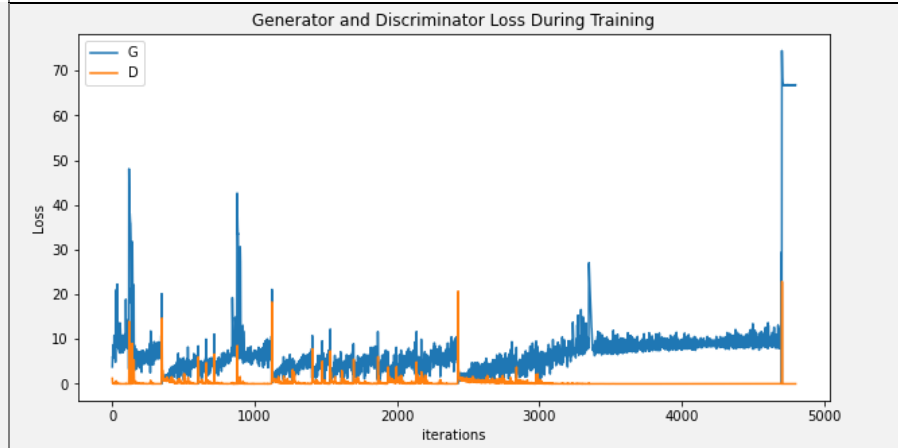
	BATCH SIZE 6	BATCH SIZE 12	BATCH SIZE 16
EPOCH 30-50			
EPOCH 100			
EPOCH 300-500			

REFERENCE MASKS								
								
								

The batch size was significantly reduced following some hyperparameter tuning. The first 50 iterations resulted in no significant mask creation (see **Error! Reference source not found.** despite the clear creation of artifacts, of which a more detailed explanation can be found in Chapter 6. However later epoch iterations saw some convergence starting around epoch 30 to 50, with a clear form of parotid mask images starting to form around epoch 100. It is important to note the existence of the empty images since some input images also lacked a parotid and were used as a reference. The generator thus also adequately learned to generate empty images.

Table 2: Respective DCGAN losses



BATCH 12**BATCH 16**

As can be observed by their respective discriminator and generator losses, a higher batch size resulted in a model collapse, resulting in the images below. While early 64x64 mask generation epoch iterations were successful for a larger batch size of 16, as can be observed in Table 2, anything upward from epoch number 350 saw the DCGAN model collapse. Failing to generate any useful parotid masks.

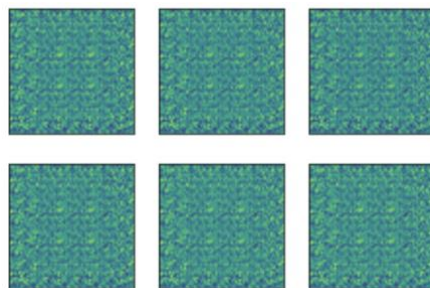
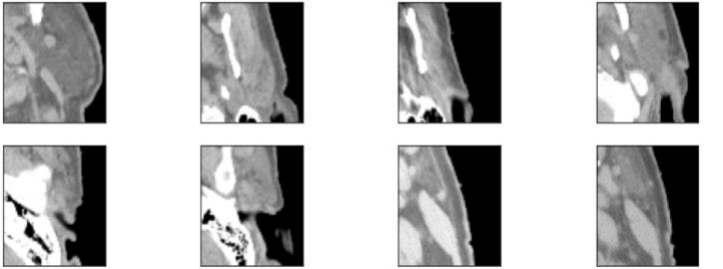
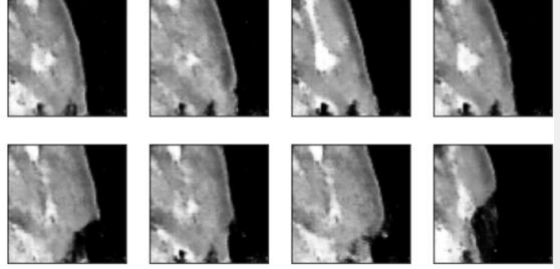

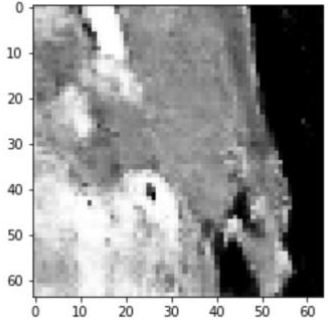


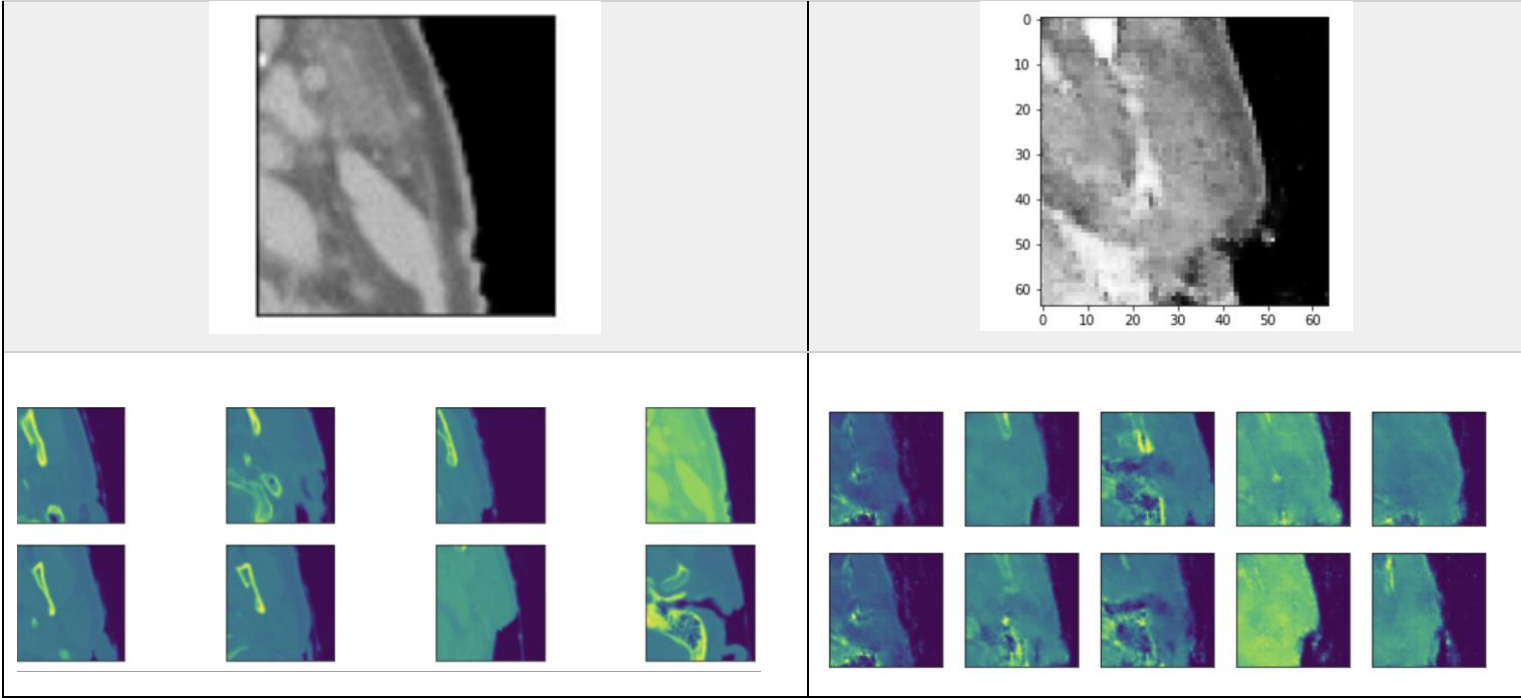
Image 27: Batch size 16, model collapse

After the generation of several masks, an attempt was also made to directly generate a 64x64 window of the parotid CT images. Here we unfortunately cannot influence the condition of the generated results, as is attempted with our dual stage framework, yet it was an interesting experiment to visually test the performance of our DCGAN network. As mentioned before, this was simply achieved by changing the input *dataloader()* function to load the respective head CT images instead of the RS masks.

To ensure that our model during this testing focused on the most important structures in our CT parotid images, some contrast enhancement was applied by defining a narrower window level. Based on histogram analysis of the CT images, a window was clipped from 700 to 1200 Hounsfield Units (HU). All synthetically generated images are made with a batch size of 12, with all other (hyper)parameters being similar to the ones applied to our RS input images. Some comparison results are displayed below.

Table 3: Synthetically generated parotid without mask conditioning

Reference input images	Synthetically generated images (DCGAN)
	
	



The synthetically generated, windowed CT images, show the possibility of generating a resembling right parotid, but similarly leaves much left to imagination due to the various distortions. Recent research into progressive growing GANS [56] have shown remarkable results into progressively generating higher resolution images from 256x256 up to 1024x1024.

Stage II of our proposed synthetic data augmentation framework deals to an extent with these shortcomings. Conditioning is done on the synthetically generated masks and the general architecture of the Pix2Pix network allows for the generation of higher resolution images. The results of this process can be found below.

5.2 Stage II results

First attempts at training images showed the necessity for proper windowing. Image 28: Wrong windowing (left) Epoch 200 on parotid head area, (right) epoch 20 on full CT image visualizes two early attempts of this process. While not being the ultimate goal, it helped to display the capabilities of this conditional GAN and the progress, or lack thereof, over early iterations.

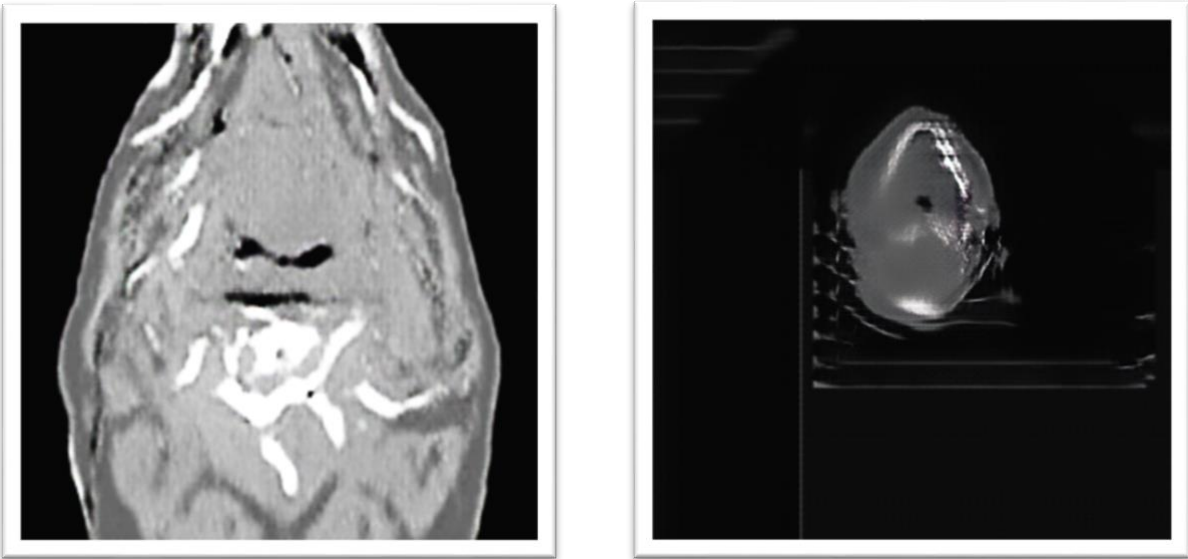
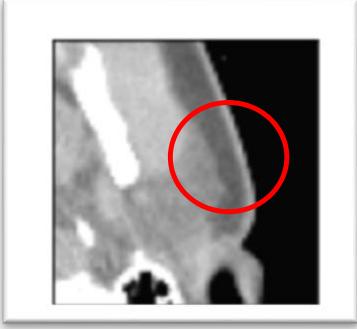


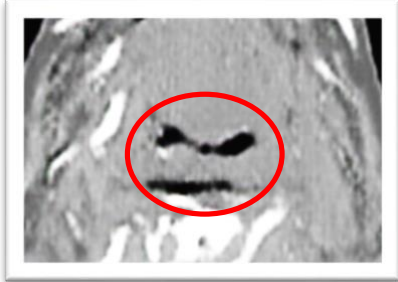

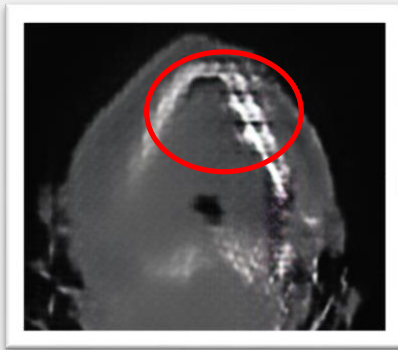




Image 28: Wrong windowing (left) Epoch 200 on parotid head area, (right) epoch 20 on full CT image

The left image emanated from incorrectly resampling the image from 184x184 to 256x256 pixels. Because the models' convolutional layers grow by a factor of 4, as well as the PatchGAN discriminator using a fixed kernels size of 4x4, output images are consistently refactored to be any factor of 2. Resulting in a horizontally stretched head CT image. It does, however, succeed in generating differing structure densities as well as a general accurate outline. One can deduce the correct creation of a cranium - brain boundary. As well as some understanding of the presence of the sphenoid sinus. The right image shows a very early iteration (epoch 20) with a full 512x512 input head CT image with no initial resizing. Although the image being of no use in any practical setting, it illustrates the focus of the network on particular areas such as the mandibular bone structure and curiously, the CT bed scanner outline.

Table 4: Wrongly synthesized CT head images during training

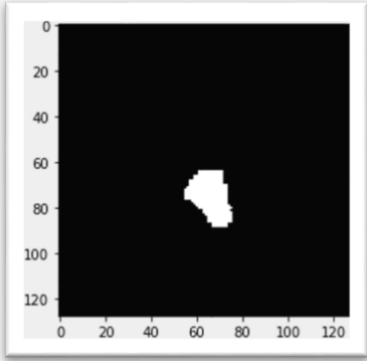
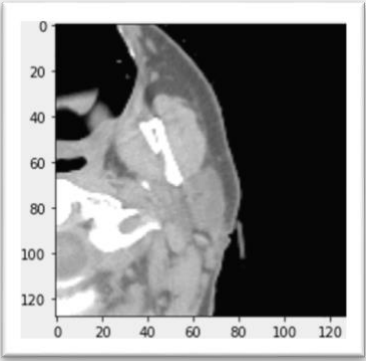
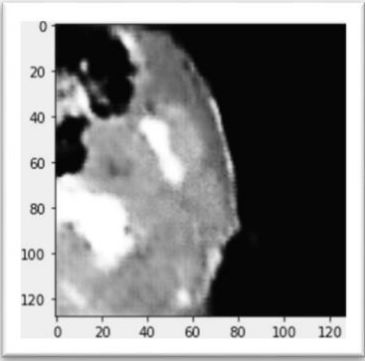
Reference images	Synthetically generated images (Pix2Pix)
	 Cranium - brain boundary creation
	 Sphenoid sinus creation
	 Mandible creation
	 CT scanning bed creation

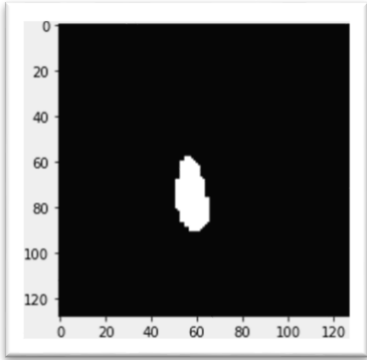
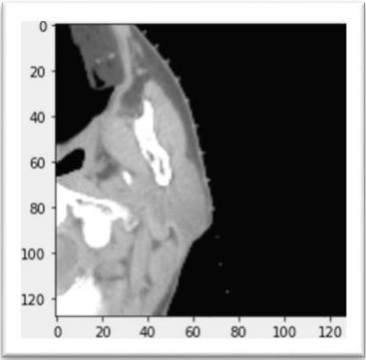
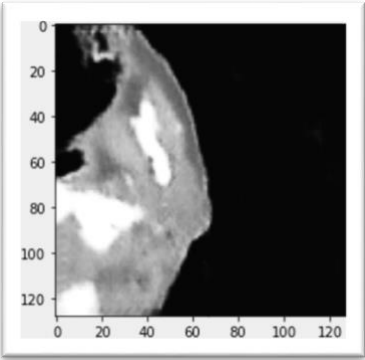
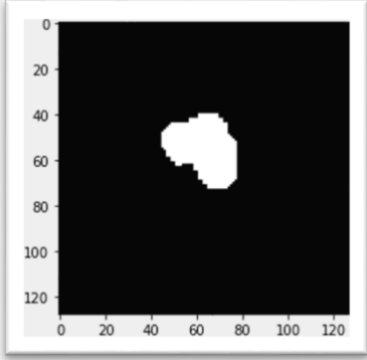
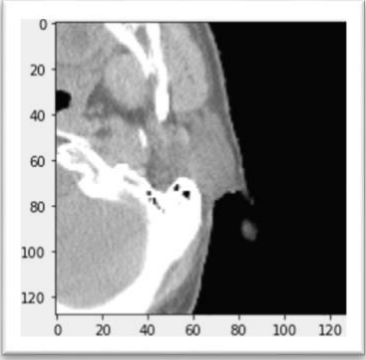
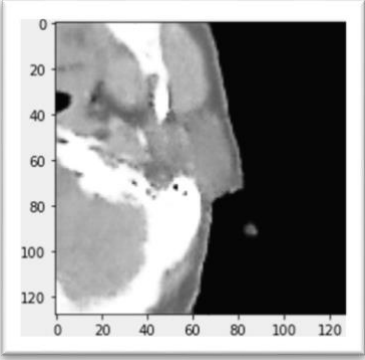
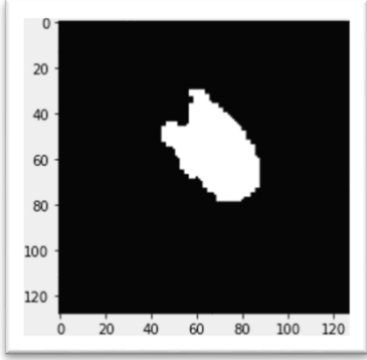
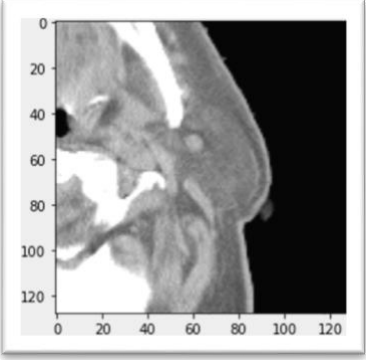
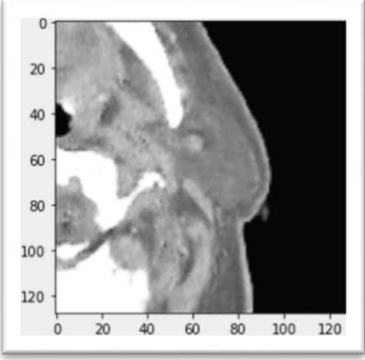
Several tweaks to the network resulted in improved image generation on the right parotid location. The learning rate starts dropping at epoch 199 in lockstep with each count onwards. Starting from 0.0002000 between epochs 200 and 300, finishing at 0.0. The data loader was adapted to enhance HU values and focus the creation area on a 128x128 window of the right parotid. Training commenced according to the hyperparameters outlined in the previous chapter for 300 epochs. Visual inspection of intermediate generations are displayed below. Discriminator and generator losses were monitored as well to watch for any inconsistencies such as dropping to 0. A typical epoch output would be in the form seen below in Figure 4: Output after a successful epoch. The low data load times are due to optimizations already happening within the adapted MONAI data loader.

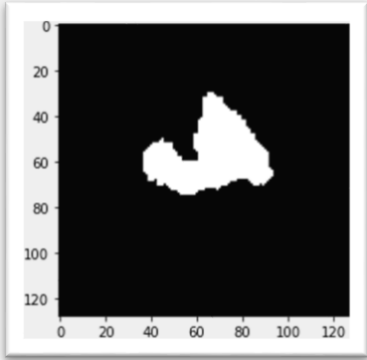
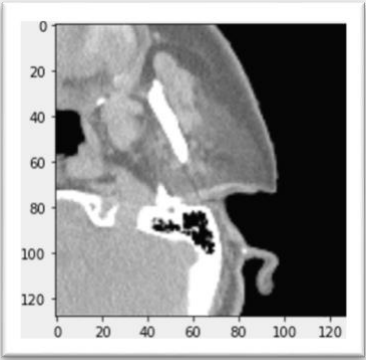
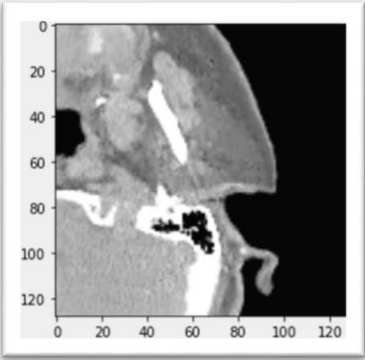
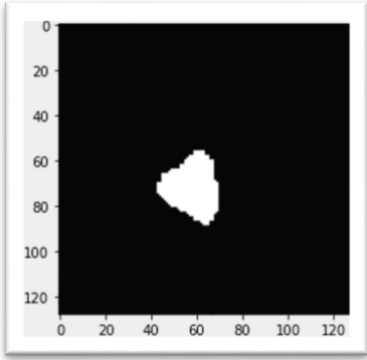
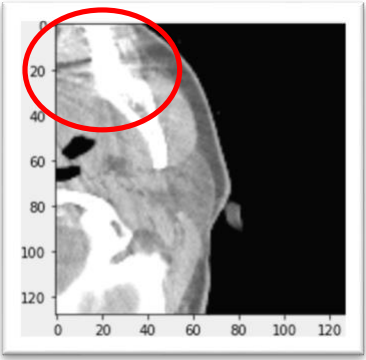
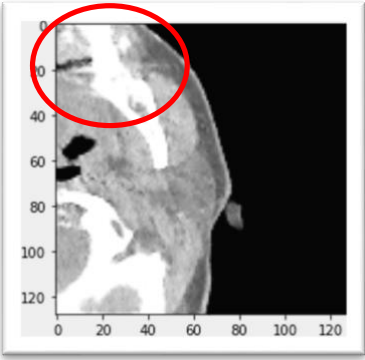
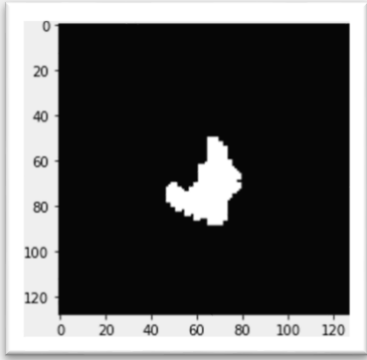
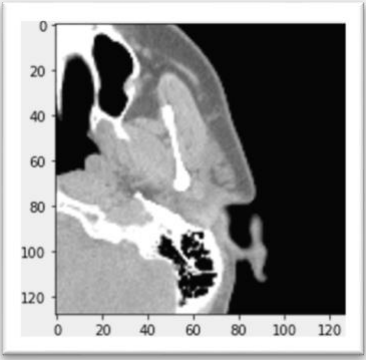
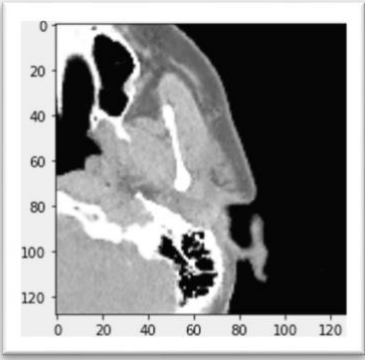
```
(epoch: 166, iters: 80, time: 0.044, data: 0.002) G_GAN: 1.899 G_L1: 3.566 D_real: 0.258 D_fake: 1.311
(epoch: 166, iters: 180, time: 0.116, data: 0.004) G_GAN: 1.602 G_L1: 8.096 D_real: 0.370 D_fake: 0.469
End of epoch 166 / 300          Time Taken: 7 sec
learning rate 0.0002000 -> 0.0002000
```

Figure 4: Output after a successful epoch

Table 5: Pix2pix training visual inspection results

Epochs	Input label	Ground Truth	Predicted image
Count: 009 Iterations: 1880			

Count: 024 Iterations: 4700			
Count: 052 Iterations: 9700			
Count: 126 Iterations: 23500			

Count: 228 Iterations: 42300			
Count: 245 Iterations: 46060			
Count: 290 Iterations: 54280			

After training the network, the generator model was saved corresponding to 41.823 million parameters. Next, testing of the generator commenced, which according to our dual stage network, was done on the synthetic outcome masks of Stage I. Consisting of a single dataset of five synthetic parotid masks, with all the hyperparameters set to mirror the training phase. Generation was performed in a single direction, from the synthetic masks to parotid CT image and running the provided model test script. The options of which are outlined below. Note that

these are similar to the ones implemented in the training phase. The options of this process are

Figure 5: Testing options on conditional Parotid generation

```


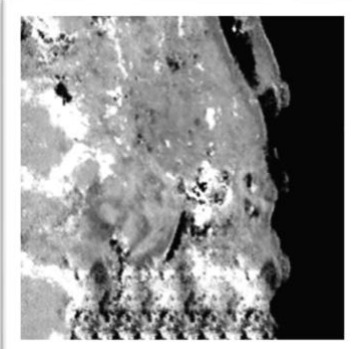

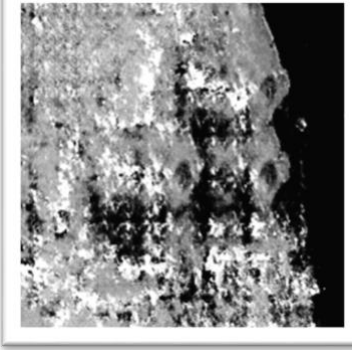
-----
    aspect_ratio: 1.0
    batch_size: 1
    checkpoints_dir: ./checkpoints
    crop_size: 256
    dataroot: ./data/B/test [default: None]
    dataset_mode: single [default: unaligned]
    direction: BtoA [default: AtoB]
    display_winsize: 128 [default: 256]
    epoch: latest
    eval: False
    gpu_ids: 0
    init_gain: 0.02
    init_type: normal
    input_nc: 1 [default: 3]
    isTrain: False [default: None]
    load_iter: 0 [default: 0]
    load_size: 256
    max_dataset_size: inf
    model: test [default: pix2pix]
    model_suffix:
    n_layers_D: 3
        name: train [default: experiment_name]
        ndf: 64
        netD: basic
        netG: unet_128 [default: resnet_9blocks]
        ngf: 64
    no_dropout: True [default: False]
    no_flip: False
    norm: batch [default: instance]
    num_test: 50
    num_threads: 4
    output_nc: 1 [default: 3]
    phase: test
    preprocess: resize_and_crop
    results_dir: ./results/
    serial_batches: False
    suffix:
    verbose: False
----- End -----
dataset [SingleDataset] was created
initialize network with normal
model [TestModel] was created
loading the model from ./checkpoints/train/latest_net_G.pth
----- Networks initialized -----
[Network G] Total number of parameters : 41.823 M
-----
creating web directory ./results/train/test_latest


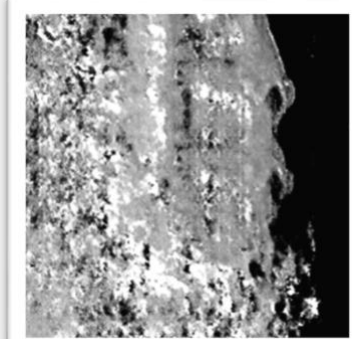

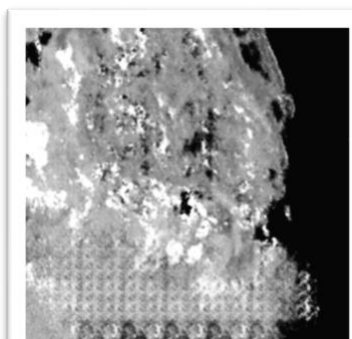
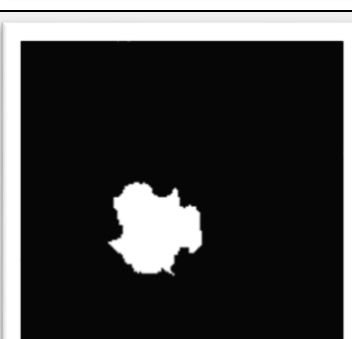
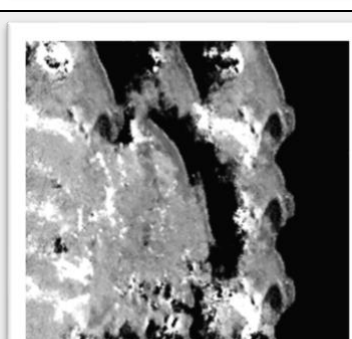
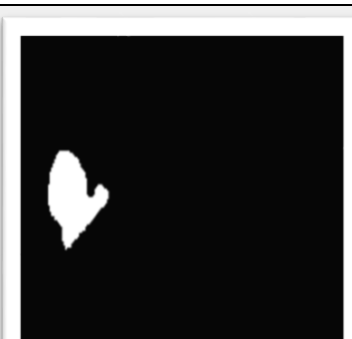
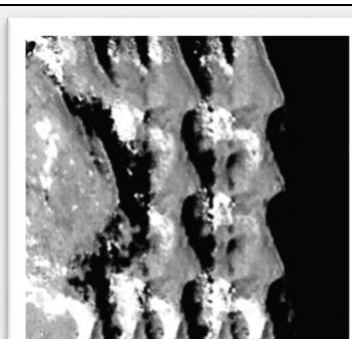
```


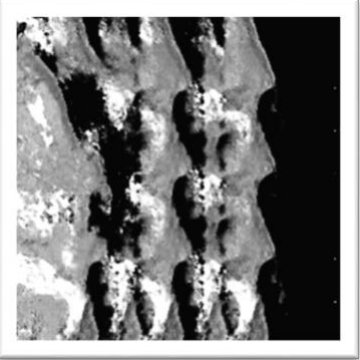

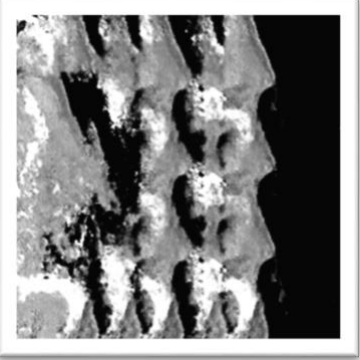

presented in Figure 5: Testing options on conditional Parotid generation, while the resulting image itself are displayed in Table 6: Conditional parotid image generation for DA.

Further analysis is provided in the Discussion chapter of this paper regarding the results. The trained network was tested on 20 images, 15 of which were synthetic mask images, and 5 were pre-seen real parotid mask images.

Table 6: Conditional parotid image generation for DA

	Synthetic parotid mask	Synthetic predicted image
Outcome 1.1: Synthetic Mask image		
Outcome 1.2: Synthetic Mask image		

Outcome 1.3: Synthetic Mask image		
1.4: Synthetic Mask image		
1.5: Synthetic Mask image		
Outcome 2.1: Real Test Mask image		

2.2: Real Test Mask image		
2.3: Real Test Mask image		

Observing the final results with conditioning on the parotid mask, one can notice the outcome of overfitting occurring. The model managed to reflect the training data to a high extent, but when operating on unseen masks, has difficulty extrapolating to generating new synthetic parotid images. The results for both Stages are further analyzed in the Discussion section of this paper.

6. Discussion

The previous chapter demonstrates the possibility for synthetically (GAN) creating parotid CT images, to be used as data augmentation in future automated delineation pipelines. While many hurdles remain, which will be discussed in the framework limitations section, the proposed dual stage method is a definite first step towards artificially acquiring additional CT medical imaging data. As can be observed in various results from Stage I, a smaller batch size led to the best image quality of synthesized masks. This is confirmed by the loss graphs during the training process, where the model with batch size 6 saw both the discriminator and generator moving in tandem. The failure to converge for higher batch size models (64 and 128) could be explained by both the small size of the dataset used for training, as well as the limited memory space. Where higher batch sizes customarily tend to require more memory space. [57] has also observed that in practice “when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize”, as is evident in our experimentation. Running the Stage I DCGAN for a higher epoch count also saw no improved image quality and eventually saw overfitting occurring or repeatedly hitting a limit on memory space. Visual inspection of the resulting mask images confirmed the observations made from the respective loss graphs. The masks with batch size 6 show clear mask boundary creation with only a single created mask area. Higher batch size counts instead exhibited the formation of multiple disparate areas and a lesser defined mask boundary. Evidently, the resulting generated synthetic masks also show the creation of checkerboard artifacts typically caused by the up-sampling layer in transposed convolutions. This was mainly noticeable in the generation of the empty reference mask images. While this could be an issue, their lack of appearance in the images of the generated masks shows that our network is able to overcome artifact creation at higher epoch counts. Artifact formation is also lacking in the DCGAN synthetically created parotid images, seen in Table 3: Synthetically generated parotid without mask conditioning. Possibilities to further solve artifact creation in future renditions should include spectral normalization [58].

These DCGAN generated images beautifully demonstrate the problem plaguing the difficulties earlier GAN models have with stable training at higher resolutions. While these test images certainly provide a boundary of the right head region, differing structure densities are hard to

differentiate. The pixelated creations more closely resemble a sharply zoomed in picture than an employable medical image. It would be considered improper for them to be used in a hospital setting.

Stage II was implemented to mitigate the limitations of the DCGAN and to allow for conditioning on the parotid area generation. Several trials were performed to ensure the target area was properly windowed and trained upon. Table 4: Wrongly synthesized CT head images during training outlines the results of this early training process. The model quickly understands the orientation and boundary of the skull. After 200 epochs the space between the cranium and brain, the parietal, is visible. Early iterations also clearly show the creation of the sphenoid sinus, possibly due to the clear contrast with the dark background. Whereas generating detailed structure shadings is challenging, the dark contrasting sphenoid sinus was present in almost every training visualization. The mandible saw a similar occurrence happening. Since the bone has a high voxel density, and occurs in many of the dataset training images, the network prioritizes the structures creation and general location. The final image in Table 4: Wrongly synthesized CT head images during training can be more regarded as a curiosity to observe the target focus of the GAN network on the generation of the CT scanning bench. Since most of the non-windowed original images had this bench present, iterations early on learned to mimic this feature.

Renewed training ended with some viable intermediate images displayed in Table 5: Pix2pix training visual inspection results. Initial epoch predicted figures appear to be of similar visual quality as the synthetically generated parotid without mask conditioning in Table 3: Synthetically generated parotid without mask conditioning. Various structures are wary, voxel density range is limited to bone structure and background, and jaw outline is incorrectly generated. After approximately 10.000 iterations, image conditions improved significantly enough to deduce grey variation features that were similar to the ground truth images. Although inside strong voxel value areas(bones), details were generally harder to create, as can be observed in intermediate image of epoch 052. Epoch 228 manages to succeed in this and accurately captures the external auditory canal, the mandible as well as the temporal bone. The red circular shape in epoch 245 outlines a disturbed ground truth image by the presence of a metal dental artifact. Interestingly,

the synthetically generated image has little presence of this distortion and excludes the usual streak artifacts. Our GAN could thus pose a potential solution for metal artifact correction in CT HNC images. More on this is outlined in the subparagraphs below such as research done by Nakao et al [59].

The last couple of training iterations manage to closely resemble the ground truth images. Structures such as the masticator and submandibular space are visible as well as fine details in the mastoid and the parotid gland. The creation of artifacts is minimal and the resolution adequate for the image to be used.

Table 6 outlines the results from testing the conditional generation on the parotid with synthetic mask images, the goal of this framework for synthetic DA. As can be seen in the synthetic parotid mask images, after converting them to black and white, several still had minor holes in them, as well as sharper edges compared to the real mask images. Furthermore, importantly the framework did not succeed in creating realistic looking synthetic CT images. There is a large occurrence of artifacts in the images with the synthetic parotid mask. While some general form of the right base of the skull can be determined, the distortions are too large to make out clear structures. Outcome 1.1 displays some differing skull densities relating to the mandible as well as the skull base but fails in aligning the outer head accurately. Curiously, all generated synthetic CT images (1.1 – 1.5) have major problems in the lower image area. With image outcome 1.4 being the most visible representation of this common failure.



Image 29: Bottom artifact creation in Outcome 1.4

No clear explanation for this can be noted, although a lack of spatial information of the parotid mask inside the brain scan might be a reasonable assumption. Many artifacts are also present in this image, presumably due to the limited amount of training images. Image 1.5 displays the clearest visible presentation of a synthetic CT scan that has some resemblance to the desired

outcome. There appears a formation of the auditory canal and what could be part of the external auditory canal. Yet here is also most evident the problem occurring with all further outcomes (2.1 – 2.3). Multiple repetitions on the vertical axis of the outer brain boundary seem to occur. Whereas the area most to the left simulates the desired target head area in all real test mask image outcomes, the two to three layered vertical axis are clearly undesired disturbances. With all three real test mask image vertical disturbances looking very much alike. Suspected is that this results from overfitting during the training phase. Seeing as the vertical row errors are highly comparable, the test mask could have appeared to resemble a mask previously seen very closely during training and thus attempted to produce a pre-seen ground truth CT image. More research would have to be done to determine the final cause of these disturbances. Relating to methods to prevent it, two opportunities jump out at first glance.

Stage I could be considered successful in generating parotid masks but to prevent overfitting in Stage II, more mask-image pairs are required. The training phase of the Pix2Pix GAN outputs high quality head CT images, yet when tested upon new parotid masks, fails to materialize concrete structures and features. With more diversity as well thousands of images instead of hundreds, less learning of the training data and less distorted images could be achieved during the testing phase. The Cityscapes dataset from which the Pix2Pix images are sourced consists of around 2000 mask-image pairs for example. An interesting approach for future research could focus on a minimal threshold number of images required to generate adequate quality medical images with our Stage II Pix2Pix GAN. As well as how the correlation increases between image quality and distribution, and number of images. Another possible solution to the bottom distortions is supplying the Stage II GAN with additional network information, as further outlined below. Instead of a single parotid mask, supplementary structure masks could for example be supplied to the Pix2Pix network to aid in the generation process. Where aside from a parotid mask, also the mandible and the submandibular masks are delineated. A visualization of such detailed delineated map can be seen in the image below.



Image 30: Detailed segmentation map

However, this could in return hinder the outcomes of the Stage I DCGAN since more mask details would have to be generated, upping the potential for failure because convergence is harder to reach. This is a trade-off worth exploring in future multi-stage GAN frameworks.

6.1 Framework limitations

Four main limitations were observed from the resulting synthetic image generation, and which are areas in which significant gains can be made in future research.

6.1.1 Lack of network information

The first constraint arises from not supplying enough detailed information to the network on the exact area of interest. For our network to focus on accurately generating parotid images, the synthetic masks are supplied to Stage II. However, recent research has outlined methods of using texture specific loss functions, extra side information to the discriminator, to allow for added focused generation [56]. Image specific losses instead of general network losses would aid in the conservation of texture and structure specific details. Furthermore, it would ensure a higher confidence in the quality of the generated area. Research done by Apple [9] has for example used a local adversarial loss, “limit[ing] the discriminator’s receptive field to local regions” to preserve annotations in the synthetic generation of eye images. Another similar improvement has been the use of bounding boxes on the ROI. By removing the structure of interest from the larger image, and subsequently attempting to use a network to learn the underlying relationship between the larger image and the removed ‘boxed’ area. The larger image aids the generator with extra information on the color, texture, and edge of the to be generated box. Implementing a

similar bounding box in our Stage II Pix2Pix GAN will aid in generating realistic images with a complete global CT structure overview while maintaining enough information to create local details. As well as potentially limiting the bottom area distortions that can be seen in Table 6: Conditional parotid image generation for DA.

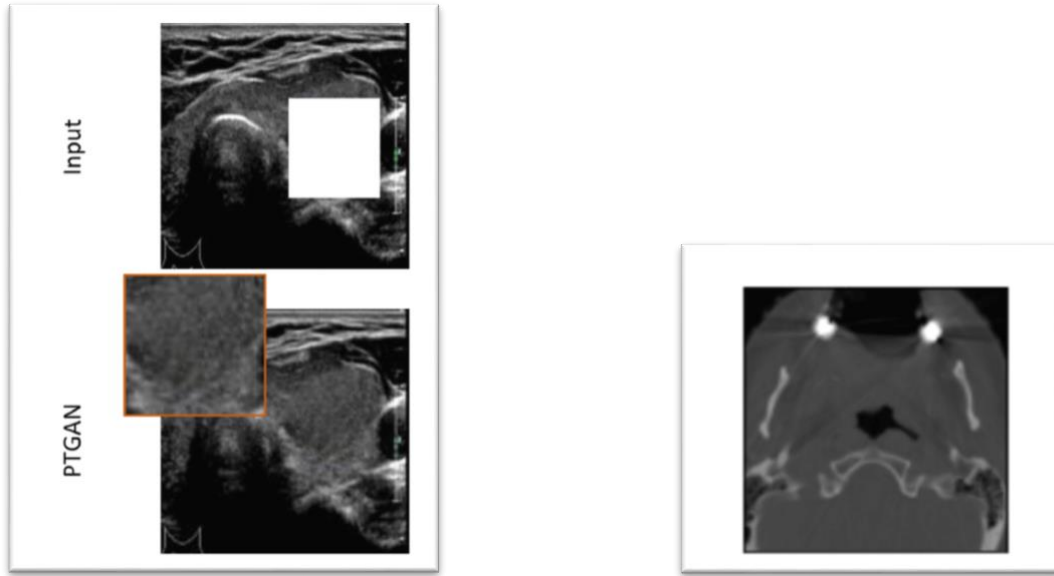


Image 31: (Left) Box inpainting on lesions, taken from [56]. (Right) Dental implant distortions

As mentioned before, supplying the Stage II network with a more detailed segmentation map of various structure masks would also provide the network additional feature information to translate the input mask map to the CT head output image. Together with the aforementioned improvements, this would be the most promising solution to generating more accurate images with the current framework.

6.1.2 Hardware constraints

A second limitation which was imposed on our framework development is a general problem encountered in the building of neural networks. As commented previously in this paper, hardware limitations often impose a limit on network training times. Depending on batch size, epoch count and image size, training times can range from 2 hours to multiple days. Because our chosen framework consists of two stages, Stage II could only be tested after Stage I was done

executing. And since the development and run time of Stage I was more time consuming, less testing and hardware space was available for Stage II. This was partially solved by integrating the open-source MONAI framework as well as requesting additional Tesla GPU cloud memory. Yet scaling this further to speed up development would have required extra financial costs to be made but would be possible if done as part of a larger research project. Together with a finite amount of input data, including distorted images, made for difficulties during pre-processing and caused a skewed dataset. The main cause for input data failure was due to metallic dental implants in patients distorting the HU values in the CT scans. While potential was shown in the Stage II GAN in smoothening these distortions, it was not successful in fully repairing the cross-image streaks. Interestingly, several other adversarial networks have been proposed and developed to counter metal artifact deformities. [59] applies a cGAN to remove artifacts in 74 ear images. While [60] goes even further and demonstrates an “outstanding capacity to reduce strong artifacts and to recover underlying missing voxels, while preserving the anatomical features of soft tissues and tooth structures from the original images” by using a 3D CycleGAN. A regularized loss function is instead applied however, because the cycle consistency loss that is typically applied fails to safeguard the structural anatomical features that are present in head CT scans. It would be interesting to observe the future role of multistage GAN frameworks in both pre-processing as well as DA to counter disturbed as well as limited datasets. Alleviating hardware limitations can either be offset by an increased cloud budget or network optimizations such as automatic mixed precision (AMP) and distributed training.

6.1.3 Higher Resolutions

Observing the Stage I results to synthetically generate parotids with our DCGAN, exemplifies the limitations that early GAN networks commonly experienced. And is also, as stated earlier, why the tested framework consists of two Stages. This constraint is in the inability of (DC)GAN’s to generate at higher resolution images. Images go through multiple up and down-sampling convolutional layers to map the feature dimensions. Unfortunately, working with higher resolution images is not as elementary as solely increasing the number of network layers. Training time significantly increases, and network stabilization drops. Achieving a convergence

between the generator and discriminator becomes unattainable. Generating in larger resolutions also requires smaller minibatches due to further memory limitations and makes it easier for the discriminator to distinguish between training and the artificial images. Significantly increasing the risk of a vanishing gradient problem occurring. This shortcoming is extra noticeable in our testing of creating parotid CT images versus our intended mask creation. While the general right head outline is clearly visible, as well as the bone structures, the lower resolution makes it hard to distinguish the differing structure densities. Some images also display loose structures or holes that do not correspond with any of the training images. Although the artificial synthesis of parotid CT images here could be considered successful, the lack of structure details would render them impassable for use by practitioners.

Here substantial progress has been achieved in recent years. A new training method called Progressive Growing GAN's (PGGAN) [61] combines several insights and modifications to deal with high resolution image generation. A Wasserstein loss function is applied to aid in the evaluation of the quality of the synthetic images. The discriminator and generator are progressively trained from a low spatial resolution, say 4×4 , to images with a resolution of 1024×1024 . Since most training iterations happen at smaller resolutions, training completion of the network is significantly faster and more stable. [56] has applied this exact technique to propose a DA method to overcome data imbalances and “on the repair of lesions and the synthesis of new lesions”. And Chan et al [62] successfully employs a PGGAN to generate realistic 256×256 MRI brain images as well as showing that when combined with classical DA, these images aid in better tumor detection tasks. Demonstrating the success of our hypothesis, the potential of GAN based DA, in other medical imaging domains. Gui et al [46] provides a rather updated overview of the various GAN's and their applications but also outlines the difficulty of replicating even higher resolution networks, such as the novel BigGAN, with sparse training data.

6.1.4 Lack of evaluation metrics

A common and persistent problem plaguing GAN models is the lack of evaluation techniques to distinguish the difference in the applied methods [2] [46]. No standard specification exists

regarding evaluation. These metrics help to quantify occurring issues such as convergence, mode collapse or training instability. As well as aiding in objectively assessing the resulting images versus the natural patient images by computing a similarity distance. Inception Score (IS) and Frechet Inception Distance (FID) are often cited GAN metrics yet are pretrained on the non-medical ImageNet dataset, lacking a biomedical images class and thus ill-suited for this specific use case. Another major deficiency of FID is its inability to detect overfitting [48]. That is why, quality assessment usually falls back onto the use of medical experts in the form of a visual Turing test. This would, however, defeat the entire purpose of this research as valuable physician time would be spent curating images. Furthermore, judging the images based on similarity is inaccurate since images are generated over a wide distribution landscape. Biases from the physical examiners could also be reintroduced. Similar to network loss functions, GAN metrics will have to be suited towards their respective use case. In medical images, this will require prior structure and pathological information as well as knowledge about the type of medical image used. Quantitatively evaluating MRI synthetic images will differ from their corresponding CT outcomes. Besides this, computational budget will also influence hyperparameter fine tuning and thus GAN evaluation. As deduced by Lucic et al, “a ‘bad’ model trained on a large budget can outperform a ‘good’ model trained on a small budget”. This inadequacy of properly evaluating and comparing GAN models remains a very current limitation and has not been uniformly resolved as of today. Future work in this area, specifically concerning biomedical images, will be a major motivation for ensuring GAN DA adoption, safety and legal applicability, as mentioned in the sections below.

6.2 Ethical considerations

A major ethical obstacle to using GAN images by practitioners and radiotherapy clinicians is their obvious synthetic nature. Medical images require extensive data quality assurances and specifications to be met before their use in practice [28]. Adequate resolution, minimal disturbances and proper documentation are some of the minimal requirements. Regulations regarding the synthetic images remain vague causing uncertainty as to whether these images are classified as new or original. The latter of which would require patient consent. Uncertainty abounds as to whether these artificial images meet all the requirements set upon real patient

images. Furthermore, training a neural network with GAN based DA techniques and synthesized images could introduce artifacts and features that are undetectable by human reviewers. An automated segmentation model, such as a U-net, used further down the diagnosis pipeline might be affected in an unknown manner. Harming both a diagnostician, as well as a patient's confidence in the outcome of neural network segmentation models. More research would have to be done to ensure the safety of synthesized images from GAN models. A standardized evaluation metric will go some way to safeguard image quality and will allow regulators to outline best practices for GAN model design in a medical setting.

Preserving a patient's privacy is crucial in medical research and outlined in various EU regulations and laws. Informed patient consent is an ethical requirement to make data sharing between clinical researchers possible. That's why of late, synthetically created medical images have been considered as a conceivable solution for data sharing, educational purposes, and research innovation. By for example artificially generating surrounding patient body parts in MRI or CT images that are unnecessary for reviewing a specific tumor. Complimenting incomplete public datasets. Or generating differing image modalities which cannot be traced back to the original patient. Instead of applying classical anonymization techniques, adversarial networks create new data points which mirror real data. For GAN anonymization to be considered ethical, the resemblance of the generated data should be sufficiently close to the real data, be significantly different from training samples to not be able to deduce the original patient (no overfitting) and preserve some measure of utility [27]. Again, better metrics will expectedly drive adoption and measures such as the proposed nearest neighbor Adversarial Accuracy by Yale et al [27] reporting the "resemblance and privacy of synthetic data" shows a promising way forward. Synthetic DA would not have to strictly adhere to these three points yet will instill confidence of patients in data handling and will put researchers on the ethical safe side if done accordingly. Ensuring the network does not overfit already alleviates some of these risks.

The last ethical consideration to be made is in regard to propagating existing biases to the generated data. GAN based DA has the opportunity to fill the image distribution which might have not been covered by the original dataset. Synthesizing and filling gaps in real data helps to

overcome possible selection biases made by medical researchers. This author visualizes a pipeline to generate a considerable number of medical images of patients from minority groups. Giving researchers and educators a larger database to test hypotheses concerning underrepresented groups, improving diagnosis as whole. Yet these same models can also enhance and amplify these biases in the data used for training. By duplicating systemic biases, problems later in a segmentation pipeline are only enlarged. Leaving out specific data, say dental implant CT images or child head scans, will prevent a GAN from ever generating images over an equivalent distribution. Thus, never aiding an automatic segmentation network in correctly or accurately classifying such images.

6.3 Future Challenges

An abundance of challenges remains ahead for full scale clinical GAN adoption. Several of which have been encountered in this research and were outlined in the previous sections. A major drawback holding back acceptance and endures as a challenge is the trust in AI networks and the reputability of the synthesized data [2]. Explainable AI [63] is one potential solution that will aid in gaining clinicians confidence. To drive collaboration between researchers and ML networks, understanding the decision-making process of an AI system is called-for. Yet the process of reaching convergence between the discriminator and generator in GANs can be notoriously hard to scrutinize. The possibility of nefarious adversarial attacks to tamper with patient medical images also drives this innovation. Research has shown how GANs can be exploited to insert diseases and perturbations in medical images [64]. Tracking the choices made by GANs will hopefully shine some light on these notorious black box networks. Outlining the decision-making progress will likewise ease hospital acceptance of ML segmentation and diagnosis.

6.4 Future Work & Applications

The future for artificial data generation is currently being made. Progress is rapid and new models are released on a monthly basis. While certainly the means to measure the success of a network versus another is lacking, the potential for GANs in DA and medical imaging in general,

are bright. New potential applications are being actively researched. The author of this thesis believes several of these applications are worth mentioning.

The first one is concerning the capability of 3D image generation. Medical images from CT and MRI scans are regularly used to create a three-dimensional profile of a patient. Current methods generate successive 2D slices which are subsequently combined. Tumors and other infections cover multiple slice ranges and interpretation is complicated by the discontinuity of the individual slices. The complexity in 3D medical image generation arises from the larger number of voxels that need to be managed, usually done by lowering the filter map depths [65]. Different metrics to assess the resemblance of the resulting image on all three axes are also necessitated. As simply expanding 2D metrics to 3D will fail to capture the increased image information. Future GAN architectures will be better optimized to capture and generate these complex structures and hopefully lessen the high-memory requirements usually associated with synthesizing three-dimensional medical images for augmentation.

The second opportunity for future GANs is in the context of medical education. Research from NVIDIA [66] has introduced the creation of photorealistic images using only an input semantic layout. Plainly speaking, this GauGAN⁷ manages to synthesize realistic looking images from simple drawings or text sentences. Allowing students to draw segmentation maps of the head and neck area and afterwards converting these to photorealistic CT or MRI images could drive engagement, understanding and exploration of medical edge cases. Drawn images could be compared to actual patient images. Tumors, if trained correctly, could be added to these segmentation masks, and will allow students early on to work with synthesized images and their surrounding shortcomings. Depending on the future capabilities of this type of GANs and the available educational data, this type of learning can be

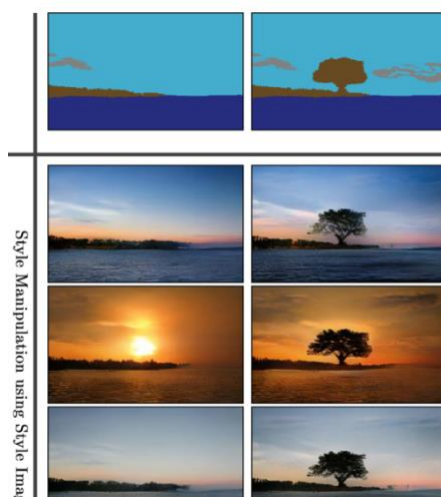


Image 32: GauGAN semantic image synthesis, taken from [66]

⁷ <http://gaugan.org/gaugan2/>

extended to all patient body parts. As well as being used to allow quick (synthetic) visualization of medical text snippets without incurring privacy objections. Pertaining to this specific thesis, this also grants the opportunity for drawing segmentation maps of the parotid and surrounding OAR regions. By educating practitioners in sketching the parotid, better understanding is embedded if distortions are discovered in real images. Potential edge cases, rare nefarious tumors or underrepresented minority patient images may be visualized, though this is limited to the distribution landscape of the GauGAN model and the style control capabilities.

7. Conclusion

Because of the difficulties associated with parotid gland delineation, consistent, high-quality automated segmentation remains an elusive goal. To ensure wider adoption by hospitals and patients alike, trust and accuracy of ML networks are key. Privacy sensitive data augmentation can help with both objectives. This paper explored such a framework for synthesizing computed tomography images as an additional data augmentation tool for parotid gland delineation. Earlier chapters in this paper helped to introduce some of the common models that are discussed in this paper as well as in the wider medical imaging literature. Generative adversarial networks can be a hard concept to grasp, and images and descriptions should have helped in the visualization aspect of the intended goal. Since different source data repositories contain differing data requirements, adequate processing of this data was certainly a necessity. The use of cloud computing and advanced medical imaging tools such as MONAI eased typical network hardware constraints and simplified development.

The proposed hypothesis that this augmentation increases the available sample size and thus will cause a higher segmentation performance has not been fully tested and must be explored in further research. What has been successfully displayed in this study is the training of synthetic CT parotid images and masks with generative adversarial networks. Parameter testing showed the importance of small batch sizes and the necessity of physical inspections during the training stages. This framework also shows what can be achieved with only a limited dataset and where further inroads can be made. The dual stage framework manages to create both synthetic masks as well as higher resolution right parotid CT images during training yet fails to generalize this to testing on synthetic parotid masks. While this thesis should be more seen as an exploration in head and neck artificial image creation, it can certainly serve as a basis for improved delineation model making and DA testing as the dual Stage network structure has been proven effective in past research. Additional investigations into GAN metrics and evaluations will allow for better judgement on the resulting images and the comparability of adversarial networks. Adoption and acceptance of GAN use in biomedical settings continues to grow, including higher resolution synthetic data augmentation. Hopefully the ideas presented in this

paper pave a way forward towards improved automated delineation practices and ultimately, better patient care.

Bibliography

- [1] "Cancer Statistics," *A Cancer Journal for Clinicians*, vol. 70, p. 7–30, 2020.
- [2] S. Kazeminia, C. Baur, A. Kuijper, B. v. Ginneken, N. Navab, S. Albarqouni and A. Mukhopadhyay, "GANs for Medical Image Analysis," *Artificial Intelligence in Medicine*, vol. 109, p. 101938, 2020.
- [3] L. Lan, L. You, Z. Zhang, Z. Fan, W. Zhao, N. Zeng, Y. Chen and X. Zhou, "Generative Adversarial Networks and Its Applications in Biomedical Informatics.," *Frontiers in Public Health*, vol. 8, pp. 164-164, 2020.
- [4] X. Yi, E. Walia and P. S. Babyn, "Generative adversarial network in medical imaging: A review.," *Medical Image Analysis*, vol. 58, p. 101552, 2019.
- [5] W. v. Rooij, M. Dahele, H. Nijhuis, B. J. Slotman and W. F. Verbakel, "Strategies to improve deep learning-based salivary gland segmentation," *Radiation Oncology*, vol. 15, no. 1, pp. 272-272, 2020.
- [6] P. Sedigh, R. Sadeghian and M. T. Masouleh, "Generating Synthetic Medical Images by Using GAN to Improve CNN Performance in Skin Cancer Classification," in *2019 7th International Conference on Robotics and Mechatronics (ICRoM)*, 2019.
- [7] R. L. J. B. Z. L. Tianxiang Shen, "'Deep Fakes' using Generative Adversarial Networks (GAN)".
- [8] C. Han, K. Murao, S. Satoh and H. Nakayama, "Learning More with Less: GAN-based Medical Image Augmentation," *Medical imaging technology*, vol. 37, no. 3, pp. 137-142, 2019.
- [9] T. P. O. T. J. S. W. W. R. W. A. I. Ashish Shrivastava, "Learning from Simulated and Unsupervised Images through Adversarial Training," 19 Jul 2017.

- [10] L. V. Van Dijk, L. V. D. Bosch, P. Aljabar and D. Peresutti, "Improving automatic delineation for head and neck organs at risk by Deep Learning Contouring 52 citations*," *RADIOTHERAPY AND ONCOLOGY*, vol. 142, pp. 115-123, 2020.
- [11] H. Z. K. S. R. e. a. Vorwerk, "Protection of quality and innovation in radiation oncology: The prospective multicenter trial the German Society of Radiation Oncology (DEGRO-QUIRO study).," *Strahlenther Onkol*, vol. 190, no. <https://doi.org/10.1007/s00066-014-0634-0>, p. 433–443, 2014.
- [12] X. W. C. S. Y. Z. J. W. W. Y. F.-F. W. Q. W. Q. a. G. Y. Li, " An artificial intelligence-driven agent for real-time head-and-neck IMRT plan generation using conditional generative adversarial network (cGAN)," *Medical Physics*, vol. 48, no. 6, pp. 2714-2723, 2021.
- [13] W. Zhu, Y. Huang, L. Zeng, X. Chen, Y. Liu, Z. Qian, N. Du, W. Fan and X. Xie, "AnatomyNet: Deep learning for fast and fully automated whole-volume segmentation of head and neck anatomy," *Medical Physics*, vol. 46, no. 2, pp. 576-589, 2019.
- [14] B. J.-E. M. C. S. Y. B. M.-I. Boldrini Luca, "Deep Learning: A Review for the Radiation Oncologist," *Frontiers in Oncology*, vol. 9, no. 2234-943X, p. 977, 2019.
- [15] Y. Jiang, H. Chen, M. Loew and H. Ko, "COVID-19 CT Image Synthesis With a Conditional Generative Adversarial Network," *IEEE Journal of Biomedical and Health Informatics*, vol. 25, no. 2, pp. 441-452, 2021.
- [16] D. B. a. H. Wiberg, "Machine Learning in Oncology: Methods, Applications, and Challenges," *JCO Clinical Cancer Informatics*, vol. 4, pp. 885-894, 2020.
- [17] Y. J. C. Y. S. K. J. R. B. H. K. N. Kim M, "Deep Learning in Medical Imaging," *Neurospine*, vol. 16, no. 4, pp. 657-668, 2019.
- [18] L. M.-J. M. Z.-D. Andrés Anaya-Isaza, "An overview of deep learning in medical imaging," *Informatics in Medicine Unlocked*, vol. 26, no. 100723, 2021.
- [19] D. a. W. C. a. D. F. a. B. M. a. A.-P. J. a. L. B. a. Y. G.-Z. Ravi, "Deep Learning for Health Informatics," *IEEE Journal of Biomedical and Health Informatics*, vol. 21, no. 1, pp. 4-21, 2017.

- [20] B. a. L. X. Ibragimov, "Segmentation of organs-at-risks in head and neck CT images using convolutional neural networks.," *Medical physics*, vol. 44, no. 2, pp. 547-557, 2017.
- [21] M. D. H. R. B. A. R. D. B. J. S. W. F. V. Ward van Rooij, "Deep Learning-Based Delineation of Head and Neck Organs at Risk: Geometric and Dosimetric Evaluation," *International Journal of Radiation Oncology*Biology*Physics*, vol. 104, no. 3, pp. 677-684, 2019.
- [22] V. N. C. M. A. L. Philippe Meyer, "Survey on deep learning for radiotherapy," *Computers in Biology and Medicine*, vol. 98, no. 1, pp. 126-146, 2018.
- [23] A. Hänsch, M. Schwier, T. Gass, T. Morgas, B. Haas, J. Klein and H. K. Hahn, "Comparison of different deep learning approaches for parotid gland segmentation from CT images," in *Medical Imaging 2018: Computer-Aided Diagnosis*, 2018.
- [24] F. K. P. V. Sharp G, "Vision 20/20: perspectives on automated image segmentation for radiotherapy," *Medical Physics*, vol. 41, no. 5, p. 050902, 2014.
- [25] X. Ding, Y. Wang, Z. Xu, W. J. Welch and Z. J. Wang, "CcGAN: Continuous Conditional Generative Adversarial Networks for Image Generation," in *ICLR 2021: The Ninth International Conference on Learning Representations*, 2021.
- [26] J. Nalepa, M. Marcinkiewicz and M. Kawulok, "Data Augmentation for Brain-Tumor Segmentation: A Review.," *Frontiers in Computational Neuroscience*, vol. 13, pp. 83-83, 2019.
- [27] A. Yale, S. Dash, R. Dutta, I. Guyon, A. Pavao and K. P. Bennett, "Privacy Preserving Synthetic Health Data," in *FI000Research*, 2019.
- [28] R. Viorescu, "2018 REFORM OF EU DATA PROTECTION RULES," *European Journal of Law and Public Administration*, vol. 4, no. 2, pp. 27-39, 2017.
- [29] T. Karras, M. Aittala, J. Hellsten, S. Laine, J. Lehtinen and T. Aila, "Training Generative Adversarial Networks with Limited Data," in *Advances in Neural Information Processing Systems*, 2020.
- [30] A. Antoniou, A. J. Storkey and H. Edwards, "Data Augmentation Generative Adversarial Networks," *arXiv preprint arXiv:1711.04340*, 2017.

- [31] C. Bowles, L. Chen, R. Guerrero, P. Bentley, R. N. Gunn, A. Hammers, D. A. Dickie, M. d. C. V. Hernández, J. M. Wardlaw and D. Rueckert, "GAN Augmentation: Augmenting Training Data using Generative Adversarial Networks," *arXiv: Computer Vision and Pattern Recognition*, 2018.
- [32] N.-T. Tran, V.-H. Tran, N.-B. Nguyen, T.-K. Nguyen and N.-M. Cheung, "On Data Augmentation for GAN Training," *IEEE Transactions on Image Processing*, vol. 30, pp. 1882-1897, 2021.
- [33] H. C. Shin, N. A. Tenenholtz, J. K. Rogers, C. G. Schwarz, M. L. Senjem, J. L. Gunter, K. P. Andriole and M. Michalski, "Medical Image Synthesis for Data Augmentation and Anonymization Using Generative Adversarial Networks," in *3rd International Workshop on Simulation and Synthesis in Medical Imaging, SASHIMI 2018 Held in Conjunction with 21st International Conference on Medical Image Computing and Computer-Assisted Intervention, MICCAI 2018*, 2018.
- [34] C. Qi, J. Chen, G. Xu, Z. Xu, T. Lukasiewicz and Y. Liu, "SAG-GAN: Semi-Supervised Attention-Guided GANs for Data Augmentation on Medical Images.," *arXiv preprint arXiv:2011.07534*, 2020.
- [35] P. a. G. A. a. M. M. I. a. N. M. a. A. M. a. M. A. M. a. C. A. Costa, "End-to-End Adversarial Retinal Image Synthesis," *IEEE Transactions on Medical Imaging*, vol. 37, no. 3, pp. 781-791, 2018.
- [36] J. T. G. a. T. S. V. a. P. S. Li, "Synthetic Medical Images from Dual Generative Adversarial Networks," *arXiv*, vol. 1709.01872, 2018.
- [37] V. M. L. M. K. C. C. J. N. A. E. Joseph O. Deasy, "Radiotherapy Dose–Volume Effects on Salivary Gland Function," *International Journal of Radiation Oncology*Biography*Physics*, vol. 76, no. 3, pp. S58-S63, 2010.
- [38] Y.-J. e. a. Chang, "Classification of Parotid Gland Tumors by Using Multimodal MRI and Deep Learning," *NMR in Biomedicine*, vol. 34, no. 1, 2021.
- [39] A. Hänsch, M. Schwier, T. Gass, T. Morgas, B. Haas, V. Dicken, H. Meine, J. Klein and H. K. Hahn, "Evaluation of deep learning methods for parotid gland segmentation from CT images.," *Journal of medical imaging*, vol. 6, no. 1, p. 11005, 2018.

- [40] O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2015.
- [41] S. I. a. C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv*, p. 1502.03167, 2015.
- [42] F. Milletari, N. Navab and S.-A. Ahmadi, "V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation," in *2016 Fourth International Conference on 3D Vision (3DV)*, 2016.
- [43] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems* 27, 2014.
- [44] P. M. Burlina, N. Joshi, K. D. Pacheco, T. Y. A. Liu and N. M. Bressler, "Assessment of Deep Generative Models for High-Resolution Synthetic Retinal Image Generation of Age-Related Macular Degeneration," *JAMA Ophthalmology*, vol. 137, no. 3, pp. 258-264, 2019.
- [45] T. Kaneko, "Generative adversarial networks: Foundations and applications," *Acoustical Science and Technology*, vol. 39, no. 3, pp. 189-197, 2018.
- [46] J. Gui, Z. Sun, Y. Wen, D. Tao and J. Ye, "A Review on Generative Adversarial Networks: Algorithms, Theory, and Applications," *arXiv preprint arXiv:2001.06937*, 2020.
- [47] A. Radford, L. Metz and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in *ICLR 2016 : International Conference on Learning Representations 2016*, 2016.
- [48] M. Lucic, K. Kurach, M. Michalski, S. Gelly and O. Bousquet, "Are GANs Created Equal? A Large-Scale Study," in *Advances in Neural Information Processing Systems*, 2018.
- [49] M. A. S. C. T. G. Calimeri F., "Biomedical Data Augmentation Using Generative Adversarial Neural Networks.," *Artificial Neural Networks and Machine Learning – ICANN*, vol. 10614, 2017.

- [50] M. F.-A. a. I. D. a. E. K. a. M. A. a. J. G. a. H. Greenspan, "GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification," *Neurocomputing*, vol. 321, p. 331, 2018.
- [51] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv preprint arXiv:1411.1784*, 2014.
- [52] J.-Y. Zhu, T. Park, P. Isola and A. A. Efros, "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017.
- [53] Z. Zhang, L. Yang and Y. Zheng, "Translating and Segmenting Multimodal Medical Volumes with Cycle- and Shape-Consistency Generative Adversarial Network," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.
- [54] T.-C. W. a. M.-Y. L. a. J.-Y. Z. a. A. T. a. J. K. a. B. Catanzaro, "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs," *arXiv*, vol. 1711.11585, 2018.
- [55] M. Y. Y. L. a. B. Liu, "DICOM-RT and Its Utilization in Radiation Therapy," *Informatics in Radiology*, vol. 29, no. 3, pp. 655-667, 2009.
- [56] R. L. W. W. X. Z. J. J. H. L. Z. G. J. L. X. Y. J. Y. M. & Y. R. Zhang, "A Progressive Generative Adversarial Method for Structurally Inadequate Medical Image Data Augmentation," *IEEE journal of biomedical and health informatics*, vol. 26, no. 1, p. 7–16, 2022.
- [57] N. S. K. a. D. M. a. J. N. a. M. S. a. P. T. P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," in *International Conference on Learning Representations*, Virtual, 2017.
- [58] T. K. T. K. M. & Y. Y. Miyato, "Spectral normalization for generative adversarial networks," in *ICLR 2018*, 2018.
- [59] J. W. a. Y. Z. a. J. H. N. a. B. M. D. c, "Conditional Generative Adversarial Networks for Metal Artifact Reduction in CT Images of the Ear," *Medical image computing and computer-assisted intervention : MICCAI .International Conference on Medical Image Computing and Computer-Assisted Intervention*, vol. 11070, pp. 3-11, 2018.

- [60] K. I. N. U. Y. I. T. K. a. T. M. Megumi Nakao, "Regularized Three-dimensional Generative Adversarial Nets for Unsupervised Metal Artifact Reduction in Head and Neck CT Images," *IEEE Access*, vol. IV, p. 99, 2020.
- [61] T. A. S. L. J. L. Tero Karras, "PROGRESSIVE GROWING OF GANS FOR IMPROVED QUALITY, STABILITY, AND VARIATION," in *ICLR*, 2018.
- [62] C. R. L. A. R. F. Y. M. G. N. H. & H. Han, "Infinite Brain MR Images: PGGAN-based Data Augmentation for Tumor Detection}," in *Smart Innovation, Systems and Technologies*, Germany, Springer Science and Business Media Deutschland GmbH, 2019, pp. 291-303.
- [63] O. B. A. T. U. K. D. C. F.-M. Pascal Bourdon, "Explainable AI for Medical Imaging: Knowledge Matters," in *Multi-faceted Deep Learning*, Springer International Publishing, 2021, pp. 267-292.
- [64] C. G.-G. S. C. W. F. D. I. K. L. H. B. L. B. v. G. J. P. P. M. V. C. I. S. M. d. B. Gerda Bortsova, "Adversarial attack vulnerability of medical image analysis systems: Unexplored factors,," *Medical Image Analysis*, vol. 73, no. 102141, 2021.
- [65] S. a. M. R. a. D. A. V. a. B. A. K. a. B. M. a. R. N. S. a. G. P. Hong, "3D-StyleGAN: A Style-Based Generative Adversarial Network for Generative Modeling of Three-Dimensional Medical Images," *Deep Generative Models, and Data Augmentation, Labelling, and Imperfections*, vol. 13003, 2021.
- [66] T. a. L. M.-Y. a. W. T.-C. a. Z. J.-Y. {Park, "Semantic Image Synthesis with Spatially-Adaptive Normalization," arXiv, 2019.
- [67] Y.-X. Wang, R. Girshick, M. Hebert and B. Hariharan, "Low-Shot Learning from Imaginary Data," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.
- [68] V. Sandfort, K. Yan, P. J. Pickhardt and R. M. Summers, "Data augmentation using generative adversarial networks (CycleGAN) to improve generalizability in CT segmentation tasks,," *Scientific Reports*, vol. 9, no. 1, p. 16884, 2019.

- [69] X. Li, C. Wang, Y. Sheng, J. Zhang, W. Wang, F.-F. Yin, Q. Wu, Q. J. Wu and Y. Ge, "An artificial intelligence-driven agent for real-time head-and-neck IMRT plan generation using conditional generative adversarial network (cGAN).," *Medical Physics*, vol. 48, no. 6, pp. 2714-2723, 2021.
- [70] H. Alqahtani, M. Kavakli-Thorne and G. Kumar, "Applications of Generative Adversarial Networks (GANs): An Updated Review," *Archives of Computational Methods in Engineering*, vol. 28, no. 2, pp. 525-552, 2021.
- [71] T. P. O. T. J. S. W. W. R. W. Ashish Shrivastava, "Learning from Simulated and Unsupervised Images through Adversarial Training," arXiv:1612.07828v2, Apple, 2016..

Abbreviations and Names

ABAS:	Atlas based auto contouring
ANN:	Artificial Neural Networks
AUC:	Area under the curve
AMP:	Automatic Mixed Precision
CNN:	Convolutional Neural Network
CT:	Computed Tomography
DA:	Data Augmentation (techniques)
DL:	Deep Learning
DSC:	Dice similarity coefficient
DICOM:	Digital Imaging and Communications in Medicine
EEG:	Electroencephalography
EU:	European Union
FID:	Frechet Inception Distance
GAN:	Generative Adversarial Network
cGAN:	conditional Generative Adversarial Network
GPU:	Graphical Processing Unit
HNC:	Head and Neck cancers
HU:	Hounsfield Unit
IS:	Inception Score
PGGAN:	Progressively Growing Generative Adversarial Network
RT:	Radiotherapy
RT Structure Set:	Radiotherapy object file extension
U-NET:	A Convolutional Network for Biomedical Image Segmentation
MRI:	Magnetic Resonance Imaging
OAR:	Organs at risk
ROI:	Region of interest
MONAI:	Medical Open Network for AI
VUmc:	Vrije Universiteit Medisch Centrum

Tables

Table 1: Generated mask results with various epoch counts	46
Table 2: Respective DCGAN losses	47
Table 3: Synthetically generated parotid without mask conditioning	49
Table 4: Wrongly synthesized CT head images during training	52
Table 5: Pix2pix training visual inspection results	53
Table 6: Conditional parotid image generation for DA	57

Figures

Image 1: Visualization of the Parotid gland	7
Image 2: Grayscale CT head image from the Cancer Imaging Archive	9
Image 3: Clustering, as an unsupervised learning technique	14
Image 4: A neural network visualized with an unknown amount of hidden layers	14
Image 5: A convolution step applied to an image, taken from [22]	16
Image 6: (Left) Two common pooling techniques and (right) a simplified CNN schematic	16
Image 7: Manually annotated OAR's in a patient CT image, taken from Wang et al. [25]	18
Image 8: Several data augmentation techniques frequently employed in machine learning pipelines	20
Image 9: Delineated right Parotid Gland CT scan	22
Image 10: Disturbances caused by the presence of metal dental implants in CT images	23
Image 11: Results of automated parotid segmentation on CT scans by 3 different models, from Hänsch et al. [39].	23
Image 12: Various popular activation functions	25
Image 13: Original U-Net architecture as proposed by Ronneberger et al. in 2015	27
Image 14: Generator Architecture	28
Image 15: Vanilla GAN framework as proposed by Goodfellow et al. [43]	29
Image 16: Early epoch GAN head(left) and parotid(right) generation results	30
Image 17: DCGAN architecture proposed by Radford et al. [47]	32
Image 18: An example of a Pix2Pix GAN output	33
Image 19: Proposed synthetic generation Framework	34

Image 20: Head and Neck CT scan.....	36
Image 21: Wrongly associated CT head scan(left) and correct CT scan with parotid mask after pre-processing (right).....	37
Image 22: MONAI end-to-end workflow	38
Image 23: Input parotid mask	41
Image 24: Ground truth parotid images	43
Image 25: Batch size 64, discriminator collapse	45
Image 26: Batch size 64, synthetic mask output.....	46
Image 27: Batch size 16, model collapse.....	48
Image 28: Wrong windowing (left) Epoch 200 on parotid head area, (right) epoch 20 on full CT image.....	51
Image 29: Bottom artifact creation in Outcome 1.4	62
Image 30: Detailed segmentation map.....	64
Image 31: (Left) Box inpainting on lesions, taken from [56]. (Right) Dental implant distortions	65
Image 32: GauGAN semantic image synthesis, taken from [66]	71

Appendix A: Data Pre-Processing

```

1. path = '/Users/vercingetorix/Desktop/University/Thesis/dicom_files/HNSCC/'
2. npat = 43
3.
4.
5. ###
6.
7. import os
8. import pydicom
9. import glob
10. import matplotlib.pyplot as plt
11. import numpy as np
12. from PIL import Image, ImageDraw
13. import pandas as pd
14.

```

```

1. def read_data(path_fold,npat):
2.     "Identifying paths of CT DICOM slices and RS DICOM for given patient"
3.
4.
5.     lowindex_str = ""
6.     if npat < 10:
7.         lowindex_str = '0'
8.
9.
10.    # path_fold = '/Users/vercingetorix/Desktop/University/Thesis/dicom_files/HNSCC/' # " VUL HIER HET PAD
    NAAR JE FOLDER MET PATIENTEN IN "
11.    path = path_fold + lowindex_str + str(npat) + '/'
12.
13.    print(path_fold)
14.    print(path)
15.    folders = os.listdir(path)
16.    print(folders)
17.
18.
19.    #for name in folders:
20.    #    if name.endswith(("DS_Store")):
21.    #        os.remove(name)
22.
23.    print(folders)
24.
25.    assert len(folders) == 2, "Number of dicom folders does not equal 2 for patient {}".format(npat)
26.
27.    ct_folder = folders[1] if folders[1].endswith("[CT]") else folders[0]
28.    rs_folder = folders[0] if folders[0].endswith("[CT]") else folders[1]
29.
30.    print(ct_folder)
31.    print(rs_folder)
32.    " Als de CT dicoms niet op CT eindigt dan regels hierboven even veranderen "
33.
34.
35.    ct_path = os.path.join(path,ct_folder)
36.    rs_path = os.path.join(path,rs_folder)
37.
38.    ct_files = [i for i in os.listdir(ct_path)] #loop over files in CT folder
39.
40.    #for s in ct_files:
41.    #    ct_ids = int(s.split('.')[0][2:])

```

```

42.     # print(ct_ids)
43.     ct_ids = [int(s.split('.')[0][3:]) for s in ct_files]
44.     sorted_zip = sorted(list(zip(ct_files, ct_ids)), key = lambda t: t[1])
45.     sorted_ct = list(zip(*sorted_zip))[0]
46.     sorted_ct = [str(ct_path + "/" + i) for i in sorted_ct]
47.
48.     rs_files = []
49.     rs_files_tmp = os.listdir(rs_path)
50.     rs_files_tmp = [rs_folder + "/" + rs_file for rs_file in rs_files_tmp]
51.     rs_files.append(os.path.join(path,rs_files_tmp[0]))
52.     sorted_rs = rs_files
53.
54.
55.     ct_files = []
56.     for fname in sorted_ct:
57.         ct_files.append(pydicom.dcmread(fname))
58.
59.     rs_files = []
60.     for fname in sorted_rs:
61.         rs_files.append(pydicom.dcmread(fname))
62.
63.     del lowindex_str; del ct_ids; del sorted_zip; del sorted_ct; del fname
64.
65.     return ct_files, rs_files
66.
67. CT_files, RS_files = read_data(path, npat)
68. print(len(CT_files))
69. print(len(RS_files))
70.

```

```

1. slice_nr = 0
2.
3. #print(RS_files[slice_nr].keys)
4.

```

```

1. nrows = CT_files[slice_nr].Rows
2. ncols = CT_files[slice_nr].Columns
3.
4. dims = (nrows,ncols,len(CT_files))
5. print("")
6. print("Dimensionality: " + str(dims))
7.
8. " Pixel spacings are found by : "
9.
10. ps = CT_files[0].PixelSpacing
11. st = CT_files[0].SliceThickness
12.
13. if (st is None):
14.     #CT_files[<slice_nr>].ImageOrientationPatient
15.     list_data = []
16.     for ii in range(len(CT_files)):
17.         list_data.append(CT_files[ii].SliceLocation)
18.     list_data = sorted(list_data)
19.     new_slice_thickness = float(list_data[5]) - float(list_data[4])
20.     print(new_slice_thickness)
21.     voxelspacing_original = (ps[0], ps[1], float(new_slice_thickness))
22.
23. else:
24.     print('test2')
25.     voxelspacing_original = (ps[0], ps[1], float(st))

```

```

26.
27. #removed float in front because: float() argument must be a string or a number, not 'NoneType'
28.
29. print("Voxel spacing: " + str(voxelspacing_original))
30.

```

```

Dimensionality: (512, 512, 130)
Voxel spacing: ("0.9375", "0.9375", 3.0)

```

```

1. """
2. With the find_patients function you will get a list of all patients recognized within your path
3. """
4. def find_patients(path):
5.     pats_data = glob.glob(path + '.*')
6.
7.     pats1 = []
8.     for ii in np.arange(len(pats_data)):
9.         pats1.append(int(pats_data[ii][-2:]))
10.
11.     pats1 = sorted(pats1)
12.
13.     return pats1
14.
15. " Get a list of all available patients in your data folder "
16.
17. patlist = find_patients(path)
18.
19. print("")
20. print("Patlist: " + str(patlist))
21.

```

```

Patlist: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 19, 20, 2
1, 22, 23, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42,
43, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 56, 57,

```

```

1. """
2. If any patients should be removed, do so by filling them in this list. Otherwise leave the list empty
3. """
4.
5. patient_exceptions = [2,5,6,19,26,28,49] # Either no RTstruct or RTstruct didn't contain parotid level annotations
6.
7. patient_list = [x for x in patlist if x not in patient_exceptions]
8. print(patient_list)
9.

```

```

[1, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 20, 21, 22, 23, 27, 29, 30,
31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42, 43, 45, 46, 47, 48, 50, 51, 5
3, 54, 55, 56, 57, 58]

```

```

1. " Reading CT image volume "
2.
3. def load_scan(ct_files, rs_files):
4.     "Constructing a 3D python numpy array of the given patient's CT scan"
5.
6.     slices = []

```



```

7. skipcount = 0
8. for f in ct_files:
9.     if hasattr(f, 'SliceLocation'):
10.         slices.append(f)
11.     else:
12.         skipcount = skipcount + 1
13.
14. # print("Skipped, no SliceLocation: {}".format(skipcount))
15.
16. slices = sorted(slices, key=lambda s: s.SliceLocation)
17.
18. # ps = slices[0].PixelSpacing
19. # ss = slices[0].SliceThickness
20. # ax_aspect = ps[1]/ps[0]
21. # sag_aspect = ps[1]/ss
22. # cor_aspect = ss/ps[0]
23.
24. sloc = []
25. for i, s in enumerate(slices):
26.     sloc.append(float(s.SliceLocation))
27.
28. slocmat = np.zeros((len(sloc),2))
29. slocmat[:,0] = np.arange(len(sloc))
30. slocmat[:,1] = sloc
31. slocmat = np.unique(slocmat[:,1], return_index=True)
32.
33. img_shape = list(slices[0].pixel_array.shape)
34. img_shape.append(len(slocmat[0]))
35. CT_volume = np.zeros(img_shape)
36.
37. tmp = 0
38. for i in slocmat[1]:
39.     img2d = slices[i].pixel_array
40.     CT_volume[:,tmp] = img2d
41.
42.     tmp += 1
43.
44. # =====
45. # sloc = np.unique(sloc)
46. # for i in np.arange(len(sloc)):
47. #     img2d = slices[i].pixel_array
48. #     CT_volume[:,i] = img2d
49. # =====
50.
51. del sloc; del slices; del skipcount; del f; del img_shape; del i; del s;
52. del img2d; del slocmat; del tmp
53. # CT_volume_Z_corrected = (CT_volume - CT_volume.mean())/CT_volume.std()
54.
55. CT_volume = CT_volume*ct_files[0].RescaleSlope + ct_files[0].RescaleIntercept
56.
57. return CT_volume
58.
59.
60. CT_volume = load_scan(CT_files,RS_files)
61. print("CT_volume shape: " + str(CT_volume.shape))
62.

```

CT_volume shape: (512, 512, 130)

```

1.  " You can visualize a slice in python with : "
2.
3.  slice_nr = 68
4.
5.  plt.figure(figsize = (9,9))
6.  plt.title("CT of patient " + str(npat) + ", slice " + str(slice_nr))
7.  plt.imshow(CT_volume[:, :, slice_nr], cmap='gray')
8.

```

```

1.  """
2.  This function reads which structures are available in RTStruct for a given patient.
3.  It also gives a unique label value. Don't pay attention to the warnings
4.  """
5.
6.
7.  def get_available_structures(rs_files):
8.
9.      "Function that identifies the structures available in given"
10.     "patient's DICOM"
11.
12.     available_structure_list_tmp = [rs_files[0].StructureSetROISequence[q].ROIName for q in
np.arange(len(rs_files[0].StructureSetROISequence))]
13.
14.     available_structure_list_tmp = {available_structure_list_tmp[jj]: jj for jj in
np.arange(len(available_structure_list_tmp))}
15.     available_structure_list_inv = {v: k for k, v in available_structure_list_tmp.items()}
16.
17.     available_structures = {}; notfound_ind = []; tmp = 1; #tmp2 = 0
18.     for jj in np.arange(len(available_structure_list_inv)):
19.         structure_name = available_structure_list_inv[jj]
20.
21.         # print(tmp2)
22.         # tmp2 += 1
23.         try:
24.             rs_files[0].ROIContourSequence[jj].ContourSequence
25.
26.             # print("Structure name: " + structure_name)
27.
28.             if len(rs_files[0].ROIContourSequence[jj].ContourSequence[0].ContourData) < 4:
29.                 notfound_ind.append(tmp)
30.                 print(structure_name + ' has faulty contour data (1 voxel)')
31.
32.         except:
33.             print(structure_name + ' ContourSequence has not been found')
34.             notfound_ind.append(tmp)
35.
36.
37.         # available_structures.append(structure_name)
38.         available_structures[structure_name] = tmp
39.         tmp += 1
40.
41.     if len(notfound_ind) > 0:
42.         grammarstr1 = [' ','s']
43.         grammarstr2 = ['does', 'do']
44.         grammarstr3 = ['has no', 'have no']
45.         if len(notfound_ind) == 1:
46.             qq = 0
47.         else:
48.             qq = 1
49.         print("")

```

```

1.     print("WARNING: " + str(len(notfound_ind)) + " structure" + grammarstr1[qq] + " " + grammarstr2[qq] + " not
      have proper contour data or " + grammarstr3[qq] + " attribute ContourSequence!")
2.

```

```

50. #     time.sleep(1)
51.
52.
53.     for ii in range(len(notfound_ind)-1,-1,-1):
54.         print(available_structures[available_structure_list_inv[notfound_ind[ii]]]) #####Something goes wrong
      here for patients 5 and 6
55.         del available_structures[available_structure_list_inv[notfound_ind[ii]]]
56.
57.     # LL = [l for l, p in available_structures.items()]
58.
59.     # available_structures_removeditems = {}
60.     # for ii in np.arange(len(available_structures)):
61.         # available_structures_removeditems[LL[ii]] = ii
62.
63.     # del available_structure_list_tmp;
64.     del available_structure_list_inv;
65.     del notfound_ind; del tmp; del jj; del structure_name;
66.     # del available_structures
67.
68.     return available_structures#, available_structure_list_tmp
69.
70.
71. available_structures = get_available_structures(RS_files)
72. print("")
73. print(available_structures)
74.

```

```

1.     """
2.     We should first get an idea of all structures that are available in all of our RTstruct files,
3.     as it may be too many to visualise clearly. Let us load all available structure lists in a list
4.     by looping over all patients:
5.     """
6.
7.     available_structure_list = []
8.     for npat in patient_list:
9.         print("Loading .dcm data of patient " + str(npat))
10.    #     npat = patlist[ii]
11.
12.    CT_files, RS_files = read_data(path, npat)
13.    CT_volume = load_scan(CT_files, RS_files)
14.
15.    available_structure_list.append(get_available_structures(RS_files))
16.
17.    print(available_structure_list)
18.

```

```

1.     """
2.     This function tells you how many of each structure are available in the set of all patients
3.     You can use this to select a priori which structures may be interesting to look at.
4.
5.     """
6.     def find_occurrences(available_structure_list):
7.
8.         struct_occurrence = {}
9.

```

```

10.
11.     for jj in np.arange(len(available_structure_list)):
12.         print(patient_list[jj])
13.         #patlist[jj] = npat
14.         print("Finding structures in patient " + str(patient_list[jj]))
15.
16.         A = available_structure_list[jj]
17.         print(A)
18.         print(len(A))
19.         B = [r for r, i in A.items()]
20.         for ii in np.arange(len(B)):
21.             struct = B[ii]
22.             print(struct)
23.             if B[ii] in struct_occurrence:
24.                 struct_occurrence[struct] += 1
25.             else:
26.                 struct_occurrence[struct] = 1
27.
28.         return struct_occurrence
29.
30. struct_occurrence = find_occurrences(available_structure_list)
31.
32. """
33. Please create a 'default' list of the 4-6 most occurring structures. For example:
34.
35. default_list = ['Struct_1','Struct_2','Struct_3','Struct_4','Struct_5','Struct_6']
36. """
37. default_list = ['Ring','Brainstem','R parotid','L parotid','R Parotid','L Parotid','Rt Parotid','Lt Parotid','Parotid
38. R','Parotid L','RT Parotid','LT Parotid','Rt parotid','Lt parotid','Left parotid',
39. 'Right parotid','Left Parotid','Right Parotid','rt parotid','lt parotid']#, 'CricoPharynx','Esophagus','Thyroid',\
40. # 'Trachea','U_EsophSphinc','OralCavity','PharynxConst_I','PharynxConst_M',\
41. # 'PharynxConst_S','SpinalCord','Bone_Mandible','PTV_High']
42.
43. """
44. Let us now construct a matrix that tells us which patient contains which structure in our default list.
45. 1 means the column structure is in the row patient where as a 0 indicates the contrary. The row indices
46. are patient numbers from patient_list
47. """
48.
49. assert len(patient_list) == len(available_structure_list)
50.
51. structures = np.zeros((len(available_structure_list),len(default_list)))
52. for ii in np.arange(len(available_structure_list)):
53.     for jj in np.arange(len(default_list)):
54.
55.         if default_list[jj] in available_structure_list[ii]:
56.             structures[ii,jj] = 1
57.         else:
58.             structures[ii,jj] = 0
59.
60. d_struct = {default_list[i]: structures[:,i].astype(int) for i in np.arange(len(default_list))}
61. df_struct = pd.DataFrame(data=d_struct)
62. df_struct.index = patient_list
63.
64. print(df_struct)
65.
66. """
67. Then select the first structure as your query structure. You can select any (multiple) structure(s) among the default
68. list
69. """

```

```

70.
71. query_structures = ['R parotid','L parotid','R Parotid','L Parotid','Rt Parotid','Lt Parotid','Parotid R','Parotid L', 'RT
72.   Parotid', 'LT Parotid', 'Rt parotid', 'Lt parotid', 'Left parotid',
73.   'Right parotid', 'Left Parotid', 'Right Parotid','rt parotid','lt parotid']
74. print(struct_occurrence)
75. print(default_list)
76. print(query_structures)
77.

```

```

1.  """
2.  Necessary for later to create label values for the default set of structures
3.  """
4.
5.  values_dict = {}
6.  for ii in np.arange(len(default_list)):
7.      values_dict[default_list[ii]] = ii + 1
8.
9.
10. print(values_dict)
11.

```

```

1.  """
2.  This function will confirm which of your (non-lymph) query structures are available in the RTstruct of a given patient
3.  """
4.
5.  npat = 48
6.
7.  """ The following lines should be run again before continuing """
8.  print("Loading .dcm data of patient " + str(npat))
9.  CT_files, RS_files = read_data(path,npat)
10. CT_volume = load_scan(CT_files,RS_files)
11.
12. if (CT_files[0].SliceThickness is None):
13.     #CT_files[<slice_nr>].ImageOrientationPatien
14.     list_data = []
15.     for ii in range(len(CT_files)):
16.         list_data.append(CT_files[ii].SliceLocation)
17.
18.     list_data = sorted(list_data)
19.     print(float(list_data[5]), float(list_data[4]))
20.     new_slice_thickness = float(list_data[5]) - float(list_data[4])
21.     print(new_slice_thickness)
22.     original_spacing = (float(CT_files[0].PixelSpacing[0]), float(CT_files[0].PixelSpacing[1]), new_slice_thickness)
23.     print(original_spacing)
24. else:
25.     print('test2')
26.     original_spacing = (float(CT_files[0].PixelSpacing[0]), float(CT_files[0].PixelSpacing[1]),
27.         float(CT_files[0].SliceThickness))
28.     #original_spacing = (float(CT_files[0].PixelSpacing[0]), float(CT_files[0].PixelSpacing[1]),
29.         float(CT_files[0].SliceThickness))
30.     available_structures = get_available_structures(RS_files)
31.
32. def select_structures_non_lymph(query_nlststructure_list, available_structure_list, default_list):
33.     default_NL_list = default_list
34.
35.     print(query_nlststructure_list,available_structures,default_NL_list)
36.

```

```

37.
38. if abs(len(np.unique(query_nlststructure_list)) - len(query_nlststructure_list)) > 0:
39.     print("Multiple identical entries were given")
40.
41. for ii in np.arange(len(query_nlststructure_list)):
42.     if query_nlststructure_list[ii] == None:
43.         query_nlststructure_list[ii] = ""
44.
45. if len(query_nlststructure_list[0]) != 0:
46.     if query_nlststructure_list[0] == 'all' or query_nlststructure_list == 'all' or query_nlststructure_list[0] == 'All' or
query_nlststructure_list == 'All':
47.         structure_list = default_NL_list
48.     else:
49.         structure_list = []
50.         for query_structure in query_nlststructure_list:
51.             if "arotid" in query_structure:
52.                 structure_list.append(query_structure)
53.
54.             #####new code #####
55.             #matching_parotid = [s for s in query_nlststructure_list if "Parotid" in s]
56.             #print(matching_parotid)
57.             #structure_list = matching_parotid
58.
59.
60.             #####end new code#####
61.             #if query_structure == 'GlnD_Submand' or query_structure == 'Parotid':
62.             # intermediate1 = query_structure + '_L'
63.             # intermediate2 = query_structure + '_R'
64.             # structure_list.append(intermediate1)
65.             # structure_list.append(intermediate2)
66.             # else:
67.             # pass
68.         else:
69.             structure_list.append(query_structure)
70.
71. print('Struct_list:', structure_list)
72.
73. #structure_list.append(query_structure)
74.
75. remove_items = []
76. for ii in np.arange(len(structure_list)):
77.     if structure_list[ii] == 'Cricopharyn':
78.         structure_list[ii].append('x')
79.     if structure_list[ii] in available_structure_list:
80.         print('Structure ' + structure_list[ii] + ' was identified in reference RTStruct')
81.         assert structure_list[ii] in available_structure_list
82.     elif structure_list[ii] not in available_structure_list:
83.         tmp_lr = structure_list[ii][-2:]
84.         if structure_list[ii].startswith("GlnD_Submand") and "Submandibular" + tmp_lr in available_structure_list:
85.             print('Structure ' + structure_list[ii] + ' was recognized as "Submandibular" + structure_list[ii][-2:] + "')
86.             structure_list[ii] = "Submandibular" + tmp_lr
87.         elif structure_list[ii] == 'Esophagus' and 'oesofagus' in available_structure_list:
88.             print('Structure ' + structure_list[ii] + ' was recognized as "oesofagus"')
89.             structure_list[ii] = 'oesofagus'
90.         elif structure_list[ii] in default_NL_list:
91.             print('Structure ' + structure_list[ii] + ' is unavailable in reference RTStruct: continuing without queried
structure')
92.             remove_items.append(ii)
93.
94.     else:
95.         print('Structure ' + structure_list[ii] + ' has not been identified in any RTStruct')
96.         assert structure_list[ii] in available_structure_list

```

```

97.
98.     if len(remove_items) > 0:
99.         qq_list = np.arange(len(remove_items)-1,-1,-1)
100.     else:
101.         qq_list = []
102.
103.     for qq in qq_list:
104.         structure_list.pop(remove_items[qq])
105.
106.     del qq_list; del ii; del remove_items; del default_NL_list
107. else:
108.     structure_list = []
109.
110. return structure_list
111.
112.
113. identified_structures = select_structures_non_lymph(query_structures,available_structures,default_list)
114.
115.
116. print('identified_structures', identified_structures)
117.

```

```

1.  """
2.  This function will extract contour data of the structures identified for a given patient
3.  """
4.
5.  def get_contour_data(CT_data, RS_data, selected_list, available_structure_list):
6.
7.      CTs = CT_data; RS = RS_data[0]
8.      CTs.sort(key=lambda x: int(x.InstanceNumber))
9.
10.     negative_Z_coordinate = 1
11.     if CTs[0].ImagePositionPatient[2] < 0:
12.         negative_Z_coordinate = -1
13.
14.     slices = [float(CT_data[iii].SliceLocation)*negative_Z_coordinate for ii in np.arange(len(CT_data))]
15.     #for CT in CTs:
16.         #slices.append(CT.ImagePositionPatient[2])
17.
18.     contour_list = []
19.
20.     for structure in selected_list:
21.
22.         structure_ind = available_structure_list[structure]-1
23.
24.         #print(RS.__dict__)
25.         contour_seq = RS.ROIContourSequence[structure_ind].ContourSequence
26.         contour = {i: [] for i in slices}
27.         for contour_slice in contour_seq:
28.             try:
29.                 contour[contour_slice.ContourData[2]].append(contour_slice.ContourData)
30.             except:
31.                 val = min(contour, key=lambda x: abs(float(x)-float(contour_slice.ContourData[2])))
32.                 contour[val].append(contour_slice.ContourData)
33.         contour_list.append(contour)
34.
35.     # del CTs; del RS; del slices; del structure;
36.     # del contour_seq; del contour; del contour_slice;
37.
38.     return contour_list
39.

```

```

40. contour_list = get_contour_data(CT_files, RS_files, identified_structures, available_structures)
41.

```

```

1. print(identified_structures)

```

```

['R parotid', 'L parotid']

```

```

1. def get_rs_image(CT_data, RS_data, query_structure_list, contour_data, values_dict, patient_CT,
  originals_spacing):
2.
3.     CTs = CT_data; RS = RS_data[0]
4.
5.
6.     origins = np.around(((CTs[0].ImagePositionPatient[0], CTs[0].ImagePositionPatient[1])), decimals=1)
7.     C0_x = origins[0]; C0_y = origins[1]; CL_x = abs(2*origins[0]); CL_y = abs(2*origins[1])
8.     IL_x = CTs[0].Rows; IL_y = CTs[0].Columns; spacing_x = CL_x/IL_x; spacing_y = CL_y/IL_y
9.     spacing_x = originals_spacing[0]; spacing_y = originals_spacing[1]
10.
11.
12.     negative_Z_coordinate = 1
13.     if CTs[0].ImagePositionPatient[2] < 0:
14.         negative_Z_coordinate = -1
15.
16.     SliceLocations = [float(CT_data[iii].SliceLocation)*negative_Z_coordinate for iii in np.arange(len(CT_data))]
17.
18.     #print(SliceLocations)
19.     #print(contour_data[0])
20.
21.     PP = [v for j, v in values_dict.items()]
22.
23.     vol_label_list = np.zeros((CTs[0].Rows, CTs[0].Columns, patient_CT.shape[-1], np.max(PP)))
24.     # tmp = 0
25.     for qq in np.arange(len(contour_data)):
26.
27.         # print('test:', query_structure_list[qq])
28.         # print('Query Structures:', values_dict)
29.
30.         #####added code#####
31.
32.         labelvalue = values_dict[query_structure_list[qq]]-1
33.
34.         #####
35.
36.         #labelvalue = values_dict[query_structure_list[qq]]-1
37.         #
38.         # print(tmp)
39.         # tmp += 1
40.         # print(contour_data)
41.
42.     # vol_label = np.zeros((CTs[0].Rows, CTs[0].Columns, patient_CT.shape[-1]))
43.     print('Patient shape:', patient_CT.shape[-1])
44.     for ii in np.arange(patient_CT.shape[-1]):
45.         #print('\n', 'it:', qq, 'it 2de loop:', ii, 'contour:', contour_data[qq], 'slicelocation:', SliceLocations[ii+1])
46.
47.
48.         # print('Contour data type=', type(contour_data[0]), 'Contour data len=', len(contour_data), 'Contour data
  index len=', len(contour_data[0]))
49.         # print('sliceloc data type=', type(SliceLocations), 'sliceloc data len=', len(SliceLocations), 'sliceloc a index=',
  [SliceLocations[ii]])
50.         #print('\nsliceloc in contour_data', contour_data[0][-1178:31]) #have to convert to negative?

```



```

51.         # print('\nsliloc in contour_data', contour_data[0][0]) #this gives an error
52.         #for k, v in contour_data[0].items():
53.             # print(k,v)
54.         #print(pd.DataFrame(contour_data))
55.         # print('Contour data qq', contour_data[-1])
56.         #####added code#####
57.         #print('keys in contour data = ', contour_data[0].keys())
58.         #for keys,values in contour_data[qq].items():
59.             #print(keys, values)
60.             # if values == []:
61.                 # print('check empty')
62.             #else :
63.                 # contour_slice = values
64.
65.         #print(len(contour_slice))
66.         #print(contour_slice)
67.         #####
68.         if len(contour_data[qq][SliceLocations[ii]]) < 1: #what happens here? #converted to negative values
69.             contour_slice = contour_data[qq][SliceLocations[ii]]
70.         else:
71.             contour_slice = contour_data[qq][SliceLocations[ii]][0]
72.
73.         if len(contour_slice) > 0:
74.             img = Image.new("1", (IL_x, IL_y))
75.             draw = ImageDraw.Draw(img)
76.
77.             x = [int(np.round((contour_slice[i] - C0_x)/spacing_x)) for i in np.arange(0,len(contour_slice),3)]
78.             y = [int(np.round((contour_slice[i] - C0_y)/spacing_y)) for i in np.arange(1,len(contour_slice),3)]
79.             poly = list(zip(x,y))
80.
81.             draw.polygon(poly, fill=1, outline=1)
82.
83.             vol_label_list[:,ii,labelvalue] = np.array(img)
84.
85.             # vol_label_list[:,ii,labelvalue] = vol_label
86.
87.         if len(contour_data) > 0:
88.             del CTs; del origins; del C0_x; del C0_y;
89.             del IL_x; del IL_y; del spacing_x; del spacing_y;
90.             del SliceLocations; del qq; del ii
91.
92.         return vol_label_list
93.
94.
95. RS_volume = get_rs_image(CT_files, RS_files, identified_structures, contour_list, values_dict, CT_volume,
96. voxelspacing_original)
97. print("RS_volume shape: " + str(RS_volume.shape))
98.
99. """
100. The RS_volume shape is the same as CT_volume, but with an extra dimension of size (len(default_list)).
101. This is so that every (:,:,i) matrix corresponds to an image for a structure in the default list. Not
102. all patients may have all structures
103. """
104.

```

Patient shape: 145

RS_volume shape: (512, 512, 145, 20)

```

1. """
2. Code to check which way the patient is loaded in the CT scan based on volume of the bottom or top of the scan
3. """
4.
5.
6. print(CT_files[2].ImageOrientationPatient)
7. print(npat)
8. print(CT_volume[-1])
9. top_CT = np.sum(CT_volume[:,0])
10. bottom_CT = np.sum(CT_volume[:, -1])
11. print(top_CT, bottom_CT)
12. print(top_CT - bottom_CT)
13. if ((top_CT - bottom_CT) > 0):
14.     print("body first")
15. else:
16.     print("head first")
17.
18. print(RS_volume.sum())

```

```

1. values_dict

```

```

{'Ring': 1, 'Brainstem': 2, 'R parotid': 3, 'L parotid': 4, 'R Parotid': 5,
'L Parotid': 6, 'Rt Parotid': 7, 'Lt Parotid': 8, 'Parotid R': 9, 'Parotid
L': 10, 'RT Parotid': 11, 'LT Parotid': 12, 'Rt parotid': 13, 'Lt parotid'
: 14, 'Left parotid': 15, 'Right parotid': 16, 'Left Parotid': 17, 'Right P
arotid': 18, 'rt parotid': 19, 'lt parotid': 20}

```

```

1. def save_CT_for_GAN(CT, RS, slice_nr):
2.
3.     print("saved slice", slice_nr)
4.     GAN_input_data_folder_name = "/path/Desktop/University/Thesis/dicom_files/GAN_input"
5.
6.     new_path = os.path.join(GAN_input_data_folder_name, str(npat))
7.     print(new_path)
8.
9.     save_CT = CT_volume[:, slice_nr]
10.
11.     for ii in np.arange(len(identified_structures)):
12.         label_ind = values_dict[identified_structures[ii]]-1
13.         save_RS = (RS_volume[:, slice_nr, label_ind])
14.
15.     if not os.path.exists(new_path):
16.         os.mkdir(new_path)
17.
18.     route = os.chdir(GAN_input_data_folder_name + '/' + str(npat))
19.
20.     saved_data = {"CT": save_CT, "RS": save_RS}
21.     file = open('Pat' + "(" + str(npat) + ")" + "Slice" + "(" + str(slice_nr) + ")" + ".pkl", 'w+b')
22.     pickle.dump(saved_data, file)
23.     file.close()
24.
25.     return
26.

```

```

1. def save_emptyCT_for_GAN(CT, slice_nr):
2.
3.     print("saved slice", slice_nr)

```

```

4.     GAN_input_data_folder_name =
       "/Users/vercingetorix/Desktop/University/Thesis/dicom_files/GAN_input/empty_ct"
5.
6.     new_path= os.path.join(GAN_input_data_folder_name, str(npat))
7.     print(new_path)
8.
9.     save_CT = CT_volume[:, :, slice_nr]
10.
11.    #   for ii in np.arange(len(identified_structures)):
12.    #       label_ind = values_dict[identified_structures[ii]]-1
13.    #       save_RS = (RS_volume[:, :, slice_nr, label_ind])
14.
15.    if not os.path.exists(new_path):
16.        os.mkdir(new_path)
17.
18.    route = os.chdir(GAN_input_data_folder_name + '/' + str(npat))
19.
20.    saved_data = {"CT": save_CT}
21.    file = open('Pat' + "(" + str(npat) + ")" + "Slice" + "(" + str(slice_nr) + ")" + ".pkl", 'w+b')
22.    pickle.dump(saved_data, file)
23.    file.close()
24.
25.    return
26.

```

```

"""
Code to check whether the saved images are satisfiable or wrong
"""
1.
2.     index = 85
3.     plt.figure(figsize=(9,9))
4.     plt.title("Patient " + str(npat) + ", slice " + str(index))
5.
6.     transposed_CT = CT_volume[:, :, index]
7.     #transposed_CT = transposed_CT.transpose((1,2,0))
8.     #plt.imshow(CT_volume[:, 0, :, -1], cmap='gray')
9.     #CT_volume[:, 48, :, -1]
10.
11.    plt.imshow(transposed_CT, cmap='gray')
12.
13.
14.
15.    for ii in np.arange(len(identified_structures)):
16.        new_index = index *-1
17.        label_ind = values_dict[identified_structures[ii]]-1
18.        plt.imshow(RS_volume[:, :, new_index, label_ind], cmap='jet', alpha=0.2)
19.

```

```

1.     print(RS_volume.sum())
2.
3.     """
4.     This visualises all structures in your query_structures list that have been found in your queried
5.     RTStruct file. The color map is rather blue because of the way it is visualised. Another to visualise
6.     without compromising the black/white of the CT image would be desirable
7.     """
8.
9.
10.    top_CT = np.sum(CT_volume[:, :, 0])
11.    bottom_CT = np.sum(CT_volume[:, :, -1])

```

```

12. if ((top_CT - bottom_CT) > 0):
13.     print("body first", original_spacing)
14.
15. for slice_ind in np.arange(CT_volume.shape[-1]):
16.
17.     plt.figure(figsize=(9,9))
18.     plt.title("Patient " + str(npat) + ", slice " + str(slice_ind))
19.     plt.imshow(CT_volume[:, :, slice_ind], cmap='gray')
20.
21.     #plt.imshow(CT_volume[:, :, slice_ind], cmap='gray')
22.     for ii in np.arange(len(identified_structures)):
23.         label_ind = values_dict[identified_structures[ii]]-1
24.         new_index = slice_ind * -1
25.         if ((top_CT - bottom_CT) > 0):
26.             plt.imshow(RS_volume[:, :, new_index, label_ind], cmap='jet', alpha=0.3)
27.         else:
28.             plt.imshow(RS_volume[:, :, slice_ind, label_ind], cmap='jet', alpha=0.3)

```

```

1.  """
2.  Best to manually enter these values
3.  """
4.
5.  start_slice = 64
6.  end_slice = 84
7.
8.  RS_or_empty = input("Please decide whether you want to save the slices empty or with the RS file:")
9.
10. print(str(RS_or_empty))
11.
12. if "empty" in RS_or_empty:
13.     print("saving empty slices", start_slice, "to" , end_slice , "for patient:", npat)
14.     # for ii in range(start_slice, end_slice + 1):
15.     #     save_emptyCT_for_GAN(CT_volume, ii)
16. elif "RS" in RS_or_empty:
17.     print("saving CT and RS slices", start_slice, "to" , end_slice , "for patient:", npat)
18.     for ii in range(start_slice, end_slice + 1):
19.         save_CT_for_GAN(CT_volume, RS_volume, ii)
20. else:
21.     print("Please either save empty or RS filed slices 'empty' or 'RS'")
22.

```

Appendix B: DCGAN on AWS

```

1. from __future__ import print_function
2. %matplotlib inline
3.
4. import glob
5. import argparse
6. import json
7. import os
8. import pickle
9. import sys
10. import pandas as pd
11. import torch
12. import torch.optim as optim
13. import torch.utils.data
14. import matplotlib.pyplot as plt
15. import pandas as pd
16. from torch.utils.data import Dataset, DataLoader
17. import pandas as pd
18. import numpy as np
19. import torch.nn as nn
20. import torch.nn.functional as F
21. from sklearn.preprocessing import MinMaxScaler
22. from torch.utils.data import TensorDataset, DataLoader
23. import torch.optim as optim
24. import pickle as pkl
25. from torchvision import transforms
26. from torchvision import datasets
27.
28. pip install monai
29.
30. import sagemaker
31. import boto3
32. import warnings
33. warnings.filterwarnings("ignore") # remove some scikit-image warnings
34.
35. # from monai.utils import progress_bar, set_determinism
36. from monai.transforms import (
37.     AddChannel,
38.     AsChannelFirst,
39.     Compose,
40.     RandFlip,
41.     RandRotate,
42.     SpatialCrop,
43.     RandZoom,
44.     ScaleIntensity,
45.     Resize,
46.     ToTensor,
47.     Transform,
48. )
49. from monai.networks import normal_init
50. from monai.data import CacheDataset, DataLoader, Dataset
51. from monai.config import print_config
52. from monai.apps import download_and_extract
53.

```

```

1. sagemaker_session = sagemaker.Session(boto3.session.Session(
2.     aws_access_key_id='*****',
3.     aws_secret_access_key='*****',
4.     region_name='us-east-1'))
5.
6. from sagemaker import get_execution_role
7. role = get_execution_role()
8.
9. bucket = sagemaker_session.default_bucket()
10. prefix = 'sagemaker/dcgan'
11.

```

```

1. #This is the Discriminator model used for DCGAN. Adapted from the official pytorch DCGAN tutorial
2. # conv_dim is the size of the feature map
3.
4. class Discriminator(nn.Module):
5.     def __init__(self, conv_dim):
6.         super(Discriminator, self).__init__()
7.
8.         self.conv_dim = conv_dim
9.
10.        self.main = nn.Sequential(
11.
12.            # nn.Conv2d(1, conv_dim, 4, stride=2, padding=1, bias=False),
13.            # nn.LeakyReLU(0.2, inplace=True),
14.
15.            # nn.Conv2d(conv_dim, conv_dim * 2, 4, 2, 1, bias=False),
16.            # nn.BatchNorm2d(conv_dim * 2),
17.            # nn.LeakyReLU(0.2, inplace=True),
18.
19.            # nn.Conv2d(conv_dim * 2, conv_dim * 4, 4, 2, 1, bias=False),
20.            # nn.BatchNorm2d(conv_dim * 4),
21.            # nn.LeakyReLU(0.2, inplace=True),
22.
23.            # nn.Conv2d(conv_dim * 4, conv_dim * 8, 4, 2, 1, bias=False),
24.            # nn.BatchNorm2d(conv_dim * 8),
25.            # nn.LeakyReLU(0.2, inplace=True),
26.
27.            # nn.Conv2d(conv_dim * 8, conv_dim * 16, 4, stride=2, padding=1, bias=False),
28.            # nn.BatchNorm2d(conv_dim * 16),
29.            # nn.LeakyReLU(0.2, inplace=True),
30.
31.            # nn.Conv2d(conv_dim * 16, 1, 4, stride=1, padding=0, bias=False),
32.            # nn.Sigmoid()
33.
34.            nn.Conv2d(1, conv_dim, 4, 2, 1, bias=False),
35.            nn.LeakyReLU(0.2, inplace=True),
36.
37.            nn.Conv2d(conv_dim, conv_dim * 2, 4, 2, 1, bias=False),
38.            nn.BatchNorm2d(conv_dim * 2),
39.            nn.LeakyReLU(0.2, inplace=True),
40.
41.            nn.Conv2d(conv_dim * 2, conv_dim * 4, 4, 2, 1, bias=False),
42.            nn.BatchNorm2d(conv_dim * 4),
43.            nn.LeakyReLU(0.2, inplace=True),
44.
45.            nn.Conv2d(conv_dim * 4, conv_dim * 8, 4, 2, 1, bias=False),
46.            nn.BatchNorm2d(conv_dim * 8),
47.            nn.LeakyReLU(0.2, inplace=True),
48.
49.            nn.Conv2d(conv_dim * 8, 1, 4, 1, 0, bias=False),

```

```

50.     nn.Sigmoid()
51. )
52.
53.
54. def forward(self, input):
55.     # input=input.float()
56.     return self.main(input)
57.

```

```

1.  # This is the Generator model used for DCGAN. Adapted from the official pytorch DCGAN tutorial
2.  # z_size is the latent vector parsed in
3.  # conv_dim is the size of the feature map
4.
5.  class Generator(nn.Module):
6.      def __init__(self, z_size, conv_dim):
7.          super(Generator, self).__init__()
8.          self.z_size = z_size
9.          self.conv_dim = conv_dim
10.         self.main = nn.Sequential(
11.
12.             # nn.ConvTranspose2d(z_size, conv_dim * 16, 4, 1, 0, bias=False),
13.             # nn.BatchNorm2d(conv_dim * 16),
14.             # nn.ReLU(True),
15.
16.             # nn.ConvTranspose2d(conv_dim * 16, conv_dim * 8, 4, 2, 1, bias=False),
17.             # nn.BatchNorm2d(conv_dim * 8),
18.             # nn.ReLU(True),
19.
20.             # nn.ConvTranspose2d(conv_dim * 8, conv_dim * 4, 4, 2, 1, bias=False),
21.             # nn.BatchNorm2d(conv_dim * 4),
22.             # nn.ReLU(True),
23.
24.             # nn.ConvTranspose2d(conv_dim * 4, conv_dim * 2, 4, 2, 1, bias=False),
25.             # nn.BatchNorm2d(conv_dim * 2),
26.             # nn.ReLU(True),
27.
28.             # nn.ConvTranspose2d(conv_dim * 2, conv_dim, 4, 2, 1, bias=False),
29.             # nn.BatchNorm2d(conv_dim),
30.             # nn.ReLU(True),
31.
32.             # nn.ConvTranspose2d(conv_dim, 1, 4, 2, 1, bias=False), #first on here stands for the number of channels
33.             # nn.Tanh()
34.
35.             nn.ConvTranspose2d(z_size, conv_dim * 8, 4, 1, 0, bias=False),
36.             nn.BatchNorm2d(conv_dim * 8),
37.             nn.ReLU(True),
38.
39.             nn.ConvTranspose2d(conv_dim * 8, conv_dim * 4, 4, 2, 1, bias=False),
40.             nn.BatchNorm2d(conv_dim * 4),
41.             nn.ReLU(True),
42.
43.             nn.ConvTranspose2d(conv_dim * 4, conv_dim * 2, 4, 2, 1, bias=False),
44.             nn.BatchNorm2d(conv_dim * 2),
45.             nn.ReLU(True),
46.
47.             nn.ConvTranspose2d(conv_dim * 2, conv_dim, 4, 2, 1, bias=False),
48.             nn.BatchNorm2d(conv_dim),
49.             nn.ReLU(True),
50.

```

```

51.         nn.ConvTranspose2d( conv_dim, 1, 4, 2, 1, bias=False), #first on here stands for the number of channels
    (1 since B&W)
52.         nn.Tanh()
53.
54.     )
55.
56.     def forward(self, input):
57.         # input=input.float()
58.         return self.main(input)
59.

```

```

1.     #initiate both loss functions and set convention for real and fake labels during training (0 and 1)
2.     #Currently use the BCE loss function
3.
4.     def real_loss(D_out,train_on_gpu, smooth=False):
5.
6.         batch_size = D_out.size(0)
7.
8.         # label smoothing
9.         if smooth:
10.            # smooth, real labels = 0.9
11.            labels = torch.ones(batch_size)*0.9
12.        else:
13.            labels = torch.ones(batch_size) # real labels = 1
14.
15.        # move labels to GPU if available for speed up
16.        if train_on_gpu:
17.            labels = labels.cuda()
18.        # binary cross entropy with logits loss
19.        criterion = nn.BCELoss()
20.
21.        # calculate loss
22.        loss = criterion(D_out.squeeze(), labels.squeeze())
23.        return loss
24.

```

```

1.     def fake_loss(D_out, train_on_gpu):
2.         batch_size = D_out.size(0)
3.         labels = torch.zeros(batch_size)
4.         if train_on_gpu:
5.             labels = labels.cuda()
6.         criterion = nn.BCELoss()
7.         # calculate loss
8.         loss = criterion(D_out.squeeze(), labels.squeeze())
9.         return loss
10.

```

```

1.     #scaling follows GAN hacks for optimization
2.
3.     def scale(x, feature_range=(-1, 1)):
4.         """ Scale takes in an image x and returns that image, scaled
5.             with a feature_range of pixel values from -1 to 1.
6.             This function assumes that the input x is already scaled from 0-1."""
7.         # assume x is scaled to (0, 1)
8.         # scale to feature_range and return scaled x
9.
10.        min, max = feature_range
11.        x = x * (max - min) + min
12.

```



```

13.     return x
14.

```

```

1.  def get_dataloader(batch_size, image_size, data_dir): #monai implementation to efficiently load our custom
dataset into our GAN
2.      """
3.      files = glob.glob(os.path.join(data_dir, '**/*.pkl'), recursive = True) #list all the files that are saved and end
with .pkl.
4.      #This can be changed to include only specific files or objects
5.
6.      class LoadPickleD(Transform):# nested class limited to function
7.          def __call__(self, data):
8.              with open(data, "rb") as file: #better to open with "with" to prevent IObuffer from interrupting loading
9.                  list_of_dict = pickle.load(file)
10.                 return list_of_dict["RS"] #change to RS if you want to load the parotid mask
11.
12.     transforming = Compose([
13.         LoadPickleD(),
14.         AddChannel(),
15.         ScaleIntensity(),
16.         RandRotate(range_x=15, prob=0.5, keep_size=True),
17.         RandFlip(spatial_axis=0, prob=0.5),
18.         RandZoom(min_zoom=0.9, max_zoom=1.1, prob=0.5, keep_size=True),
19.         Resize(spatial_size=[image_size, image_size]),
20.         ToTensor()
21.     ])
22.     #Apply several data transformations and ensure the images are channel-first according to the official Pytorch
documentation.
23.     #Last of all, change the image to a tensor
24.
25.     train_dataset = CacheDataset(data=files, transform=transforming)
26.     #Catch and load the dataset in batches to limit concurrent memory usage
27.
28.
29.     train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
30.     #load the dataset in determined batch sizes
31.
32.     return train_loader
33.

```

```

1.  def clip_CT(inp):
2.      mask1 = (inp < 700)
3.      mask2 = (inp > 1200)
4.      inp[mask1] = 700
5.      inp[mask2] = 1200
6.
7.      return inp
8.

```

```

1.  def get_dataloader_sagemaker(batch_size, image_size, data_dir, bucket):
2.      #monai implementation to efficiently load our custom dataset into our GAN
3.
4.      Batch the neural network data using DataLoader
5.      :param batch_size: The size of each batch; the number of images in a batch
6.      :param img_size: The square size of the image data (x, y)
7.      :param data_dir: Directory where image data is located
8.      :return: DataLoader with batched data
9.      """
10.

```

```

11.
12.
13. conn = boto3.client('s3')
14. contents = conn.list_objects(Bucket=bucket, Prefix='sagemaker/dcgan')['Contents']
15. sagemaker_files = []
16.
17. for f in contents:
18.     sagemaker_files.append(f['Key'])
19.
20.
21. class LoadPickleD(Transform):
22.
23.
24.     def __call__(self, data):
25.         response = conn.get_object(Bucket=bucket, Key=data)
26.         body = response['Body'].read()
27.         list_of_dict = pickle.loads(body)
28.         CT = list_of_dict["RS"]
29.         # CT = clip_CT(CT)
30.         return CT
31.
32. transforming = Compose([
33.     LoadPickleD(),
34.     AddChannel(),
35.     ScaleIntensity(),
36.     RandRotate(range_x=15, prob=0.5, keep_size=True),
37.     RandFlip(spatial_axis=0, prob=0.5),
38.     RandZoom(min_zoom=0.9, max_zoom=1.1, prob=0.5, keep_size=True),
39.     Resize(spatial_size=[64, 64]),
40.     SpatialCrop(roi_center=[199,320],roi_size=[image_size,image_size]),
41.     # Resize(spatial_size=[64, 64]),
42.     ToTensor()
43. ])
44.
45.
46. train_dataset = CacheDataset(data=sagemaker_files, transform=transforming)
47.
48.
49. train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
50.
51. return train_loader
52.

```

```

1. # custom weights initialization called on Generator and Discriminator
2. def weights_init_normal(m):
3.     classname = m.__class__.__name__
4.     if classname.find('Conv') != -1:
5.         nn.init.normal_(m.weight.data, 0.0, 0.02)
6.     elif classname.find('BatchNorm') != -1:
7.         nn.init.normal_(m.weight.data, 1.0, 0.02)
8.         nn.init.constant_(m.bias.data, 0)
9.

```

```

1. #initialize both networks and apply their custom weights
2.
3. def build_network(d_conv_dim, g_conv_dim, z_size):
4.     # define discriminator and generator
5.     D = Discriminator(d_conv_dim)
6.     G = Generator(z_size=z_size, conv_dim=g_conv_dim)
7.

```

```

8.     # initialize model weights
9.     D.apply(weights_init_normal)
10.    G.apply(weights_init_normal)
11.
12.    print(D)
13.    print()
14.    print(G)
15.
16.    return D, G
17.

```

```

1.  def model_fn(model_dir):
2.      """Load the PyTorch model from the `model_dir` directory."""
3.      print("Loading model.")
4.
5.      # Determine the device and construct the model.
6.      device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
7.
8.
9.      #D = DataDiscriminator(64)
10.     G = Generator(z_size=100, conv_dim=64)
11.
12.     model_info = {}
13.     # model_info_path = os.path.join(model_dir, 'generator_model.pt')
14.     model_info_path = model_dir
15.
16.     with open(model_info_path, 'rb') as f:
17.         G.load_state_dict(torch.load(f))
18.
19.     G.to(device).eval()
20.
21.     print("Done loading model.")
22.     return G
23.
24.
25.
26. def train(D, G, z_size, train_loader, epochs, d_optimizer, g_optimizer, train_on_gpu):
27.     """
28.     This is the training method that is called by the PyTorch training script. The parameters
29.     passed are as follows:
30.     model      - The PyTorch model that we wish to train.
31.     train_loader - The PyTorch DataLoader that should be used during training.
32.     epochs     - The total number of epochs to train for.
33.     optimizer  - The optimizer to use during training.
34.     loss_fn    - The loss function used for training.
35.     device     - Where the model and data should be loaded (gpu or cpu).
36.     """
37.
38.     # Get some fixed data for sampling. These are images that are held
39.     # constant throughout training, and allow us to inspect the model's performance
40.
41.     print_every=50
42.     losses = []
43.
44.     if train_on_gpu:
45.         D.cuda()
46.         G.cuda()
47.     device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
48.     samples = []
49.     sample_size=16
50.     # fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))

```

```

51. # fixed_z = torch.from_numpy(fixed_z).float()
52. fixed_noise = torch.randn(sample_size, z_size, 1, 1, device=device)
53.
54. # train the network
55. for epoch in range(epochs):
56.
57.     #y = enumerate(train_loader)
58.
59.     for batch_i, real_images in enumerate(train_loader):
60.
61.         batch_size = real_images.size()[0]
62.
63.         real_images = scale(real_images)
64.
65.         # =====
66.         # TRAIN THE DISCRIMINATOR
67.         # =====
68.
69.         D.train()
70.         G.train()
71.
72.         d_optimizer.zero_grad()
73.
74.         # 1. Train with real images
75.
76.         # Compute the discriminator losses on real images
77.         if train_on_gpu:
78.             real_images = real_images.cuda()
79.
80.         D_real = D(real_images)
81.
82.         d_real_loss = real_loss(D_real, train_on_gpu)
83.
84.         # 2. Train with fake images
85.
86.         # Generate fake images
87.         # z = np.random.uniform(-1, 1, size=(batch_size, z_size))
88.         # z = torch.from_numpy(z).float()
89.         noise = torch.randn(batch_size, z_size, 1, 1, device=device)
90.         # move x to GPU, if available
91.         if train_on_gpu:
92.             noise = noise.cuda()
93.         fake_data = G(noise)
94.
95.         # Compute the discriminator losses on fake images
96.         D_fake = D(fake_data)
97.         d_fake_loss = fake_loss(D_fake, train_on_gpu)
98.
99.         # add up loss and perform backprop
100.        d_loss = d_real_loss + d_fake_loss
101.        d_loss.backward()
102.        d_optimizer.step()
103.
104.
105.        # =====
106.        # TRAIN THE GENERATOR
107.        # =====
108.        g_optimizer.zero_grad()
109.
110.        # 1. Train with fake images and flipped labels
111.
112.        # Generate fake images

```

```

113. #         z = np.random.uniform(-1, 1, size=(batch_size, z_size))
114. #         z = torch.from_numpy(z).float()
115.         noise = torch.randn(batch_size, z_size, 1, 1, device=device)
116.         if train_on_gpu:
117.             noise = noise.cuda()
118.
119.         fake_images = G(noise)
120.
121.         # Compute the discriminator losses on fake images
122.         # using flipped labels!
123.         D_fake = D(fake_images)
124.         g_loss = real_loss(D_fake, train_on_gpu) # use real loss to flip labels
125.
126.         # perform backprop
127.         g_loss.backward()
128.         g_optimizer.step()
129.
130.         # Print some loss stats
131.         if batch_i % print_every == 0:
132.             # append discriminator loss and generator loss
133.             losses.append((d_loss.item(), g_loss.item()))
134.             # print discriminator and generator loss
135.             print('Epoch [{:5d}/{:5d}] | d_loss: {:.4f} | g_loss: {:.4f}'.format(
136.                 epoch+1, epochs, d_loss.item(), g_loss.item()))
137.
138.
139.         ## AFTER EACH EPOCH##
140.         # generate and save sample, fake images
141.         G.eval() # for generating samples
142.         if train_on_gpu:
143.             fixed_noise = fixed_noise.cuda()
144.
145.         samples_z = G(fixed_noise)
146.         samples.append(samples_z)
147.         G.train() # back to training mode
148.
149. #     _ = visualize_images(samples_z)
150.
151.     print(samples_z)
152.
153.     with open('generator_model_RS_6.pt', 'wb') as f:
154.         torch.save(G.state_dict(), f)
155.
156.     # Save training generator samples
157.     with open('train_samples_2.pkl', 'wb') as f:
158.         pkl.dump(samples, f)
159.
160.     return G
161.

```

```

1. if __name__ == '__main__':
2.     # All of the model parameters and training parameters are sent as arguments when the script
3.     # is executed. Here we set up an argument parser to easily access the parameters.
4.
5.     parser = argparse.ArgumentParser()
6.
7.     # Training Parameters
8.     parser.add_argument('--batch-size', type=int, default=128, metavar='N',
9.                         help='input batch size for training (default: 64)')
10.    parser.add_argument('--epochs', type=int, default=10, metavar='N',
11.                        help='number of epochs to train (default: 10)')

```

```

12. parser.add_argument('--seed', type=int, default=1, metavar='S',
13.     help='random seed (default: 1)')
14. parser.add_argument('--z_size', type=int, default=100, metavar='N',
15.     help='input z-size for training (default: 100)')
16.
17. # Model Parameters
18. parser.add_argument('--conv_dim', type=int, default=64, metavar='N',
19.     help='size of the convolution dim (default: 64)')
20. parser.add_argument('--lr', type=float, default=0.0002, metavar='N',
21.     help='Learning rate default 0.0002')
22. parser.add_argument('--beta1', type=float, default=0.5, metavar='N',
23.     help='beta1 default value 0.5')
24. parser.add_argument('--beta2', type=float, default=0.999, metavar='N',
25.     help='beta2 default value 0.999')
26. parser.add_argument('--img_size', type=int, default=32, metavar='N',
27.     help='Image size default value 32')
28.
29.
30. # SageMaker Parameters / Container environment
31. parser.add_argument('--hosts', type=list, default=json.loads(os.environ['SM_HOSTS']))
32. parser.add_argument('--current-host', type=str, default=os.environ['SM_CURRENT_HOST'])
33. parser.add_argument('--model-dir', type=str, default=os.environ['SM_MODEL_DIR'])
34. parser.add_argument('--data-dir', type=str, default=os.environ['SM_CHANNEL_TRAINING'])
35. parser.add_argument('--num-gpus', type=int, default=os.environ['SM_NUM_GPUS'])
36.
37. args = parser.parse_args()
38.
39. device = torch.device(("cuda:0" if (torch.cuda.is_available()) else "cpu"))
40. #Use the cpu if GPU hardware is not available
41.
42. print("Using device {}".format(device))
43.
44. torch.manual_seed(args.seed) #change seed to manual 0 to ensure stable outcome
45.
46.
47. # Load the training data.
48. train_loader = get_dataloader(args.batch_size, args.img_size, args.data_dir)
49.
50.
51. # Build the model.
52.
53. D, G = build_network(args.conv_dim, args.conv_dim, z_size=args.z_size)
54.
55.
56.
57. # Create optimizers for the discriminator and generator
58. d_optimizer = optim.Adam(D.parameters(), args.lr, [args.beta1, args.beta2])
59. g_optimizer = optim.Adam(G.parameters(), args.lr, [args.beta1, args.beta2])
60.
61. G = train(D, G, args.z_size, train_loader, args.epochs, d_optimizer, g_optimizer, device)
62.
63. # Save the model parameters ou must save the model defined by environment variable SM_MODEL_DIR in
    your training script
64. G_path = os.path.join(args.model_dir, 'generator_model_main.pt')
65. with open(G_path, 'wb') as f:
66.     torch.save(G.cpu().state_dict(), f)
67.

```

```

1. import torch.optim as optim
2.
3. data_dir = "/Users/vercingetorix/Desktop/University/Thesis/dicom_files/GAN_input/"

```

```

4.
5. # Load the training data.
6. train_loader = get_dataloader_sagemaker(6, 64, data_dir, bucket)
7. z_size = 100
8. conv_dim = 64
9.
10. lr = 0.0002
11. beta1 = 0.5
12. beta2 = 0.999
13. epochs = 500
14.
15. device = torch.cuda.is_available()
16. torch.set_default_dtype(torch.float64)
17. # Build the model.
18.
19. D, G = build_network(conv_dim, conv_dim, z_size)
20.

```

```

Loading dataset: 100% |██████████| 187/187 [00:11<00:00, 16.67it/s]

```

```

Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)

Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)

```

```

1. # Create optimizers for the discriminator and generator
2. d_optimizer = optim.Adam(D.parameters(), lr, [beta1, beta2])
3. g_optimizer = optim.Adam(G.parameters(), lr, [beta1, beta2])
4.
5. train(D, G, z_size, train_loader, epochs, d_optimizer, g_optimizer, device)
6.
7. # Save the model parameters
8. # G_path = os.path.join(data_dir, 'generator_model.pt')
9. with open('generator_model_RS_6.pt', 'wb') as f:
10.     torch.save(G.cpu().state_dict(), f)
11.

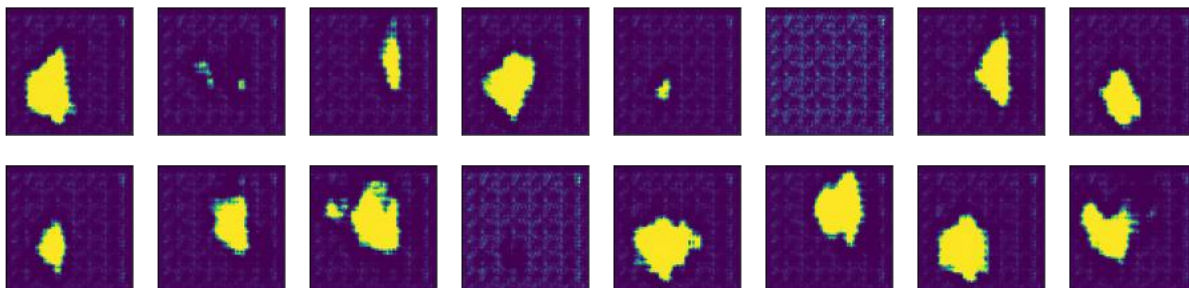
```

```

1. #Function to visualize the saved set of generated images
2. def visualize_images(samples):
3.     fig, axes = plt.subplots(figsize=(16,4), nrows=2, ncols=8, sharey=True, sharex=True)
4.     for ax, img in zip(axes.flatten(), samples[-1]):
5.         img = img.detach().cpu().numpy()
6.         img = np.transpose(img, (1, 2, 0))
7.         img = ((img + 1)*255 / (2)).astype(np.uint8)
8.         ax.xaxis.set_visible(False)
9.         ax.yaxis.set_visible(False)
10.        im = ax.imshow(img.reshape((64,64,1)), cmap='viridis')
11.
12. # Load samples from generator, taken while training
13. # with open('CT_Generated_test.pkl', 'rb') as images:
14. #     samples = pickle.load(images)
15.
16. with open('train_samples.pkl', 'rb') as images:
17.     samples = pickle.load(images)
18.
19. print("num of samples:", len(samples), "len of batch of individual samples:", len(samples[15]), )
20. results = visualize_images(samples)
21.

```

num of samples: 150 len of batch of individual samples: 16



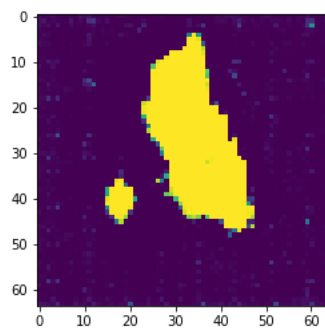
```

1. #load the saved model
2.
3. path = "generator_model_batch_12.pt"
4.
5. G.load_state_dict(torch.load(path))
6. G.eval()
7.
8. generator = model_fn("generator_model_batch_12.pt")
9.
10. generator.eval()
11. latent_size = 100
12. device = torch.cuda.is_available()
13.
14. device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
15. sample_size = 1
16. # fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
17. # fixed_z = torch.from_numpy(fixed_z).float()
18. noise = torch.randn(sample_size, latent_size, 1, 1, device=device)
19.
20. prediction = generator(noise)
21. prediction.size()
22. cpu_pred = prediction.cpu()

```



```
23. result = cpu_pred.data.numpy()
24. array_res = np.reshape(result, (64,64))
25. plt.imshow(array_res)
26.
```



Appendix C: Pix2Pix on AWS (adapted from official Pix2Pix)

Only changes and/or additions are shown below. For the full Pix2Pix version, head to the official Pytorch version page at <https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix>

C.1) base_dataset.py

```

1.  """This module implements an abstract base class (ABC) 'BaseDataset' for datasets.
2.
3.  It also includes common transformation functions (e.g., get_transform, __scale_width), which can be later used in
4.  subclasses.
5.  """
6.  import random
7.  import numpy as np
8.  import torch.utils.data as data
9.  from PIL import Image
10. import torchvision.transforms as transforms
11. from abc import ABC, abstractmethod
12.
13. class BaseDataset(data.Dataset, ABC):
14.     """This class is an abstract base class (ABC) for datasets.
15.
16.     To create a subclass, you need to implement the following four functions:
17.     -- <__init__>:      initialize the class, first call BaseDataset.__init__(self, opt).
18.     -- <__len__>:       return the size of dataset.
19.     -- <__getitem__>:    get a data point.
20.     -- <modify_commandline_options>: (optionally) add dataset-specific options and set default options.
21.     """
22.
23.     def __init__(self, opt):
24.         """Initialize the class; save the options in the class
25.
26.         Parameters:
27.         opt (Option class)-- stores all the experiment flags; needs to be a subclass of BaseOptions
28.         """
29.         self.opt = opt
30.         self.root = opt.dataroot
31.
32.     @staticmethod
33.     def modify_commandline_options(parser, is_train):
34.         """Add new dataset-specific options, and rewrite default values for existing options.
35.
36.         Parameters:
37.         parser -- original option parser
38.         is_train (bool) -- whether training phase or test phase. You can use this flag to add training-specific or test-
39.         specific options.
40.
41.         Returns:
42.         the modified parser.
43.         """
44.         return parser
45.
46.     @abstractmethod
47.     def __len__(self):
48.         """Return the total number of images in the dataset."""
49.         return 0
50.
51.     @abstractmethod

```

```

51. def __getitem__(self, index):
52.     """Return a data point and its metadata information.
53.
54.     Parameters:
55.         index - a random integer for data indexing
56.
57.     Returns:
58.         a dictionary of data with their names. It usually contains the data itself and its metadata information.
59.     """
60.     pass
61.
62.
63. def get_params(opt, size):
64.     w, h = size
65.     new_h = h
66.     new_w = w
67.     if opt.preprocess == 'resize_and_crop':
68.         new_h = new_w = opt.load_size
69.     elif opt.preprocess == 'scale_width_and_crop':
70.         new_w = opt.load_size
71.         new_h = opt.load_size * h // w
72.
73.     x = random.randint(0, np.maximum(0, new_w - opt.crop_size))
74.     y = random.randint(0, np.maximum(0, new_h - opt.crop_size))
75.
76.     flip = random.random() > 0.5
77.
78.     return {'crop_pos': (x, y), 'flip': flip}
79.
80.
81. def get_transform(opt, params=None, grayscale=False, method=Image.BICUBIC, convert=True):
82.     transform_list = []
83.     if grayscale:
84.         transform_list.append(transforms.Grayscale(1))
85.     if 'resize' in opt.preprocess:
86.         osize = [opt.load_size, opt.load_size]
87.         transform_list.append(transforms.Resize(osize, method))
88.     elif 'scale_width' in opt.preprocess:
89.         transform_list.append(transforms.Lambda(lambda img: __scale_width(img, opt.load_size, opt.crop_size,
method)))
90.
91.     if 'crop' in opt.preprocess:
92.         if params is None:
93.             transform_list.append(transforms.RandomCrop(opt.crop_size))
94.         else:
95.             transform_list.append(transforms.Lambda(lambda img: __crop(img, params['crop_pos'], opt.crop_size)))
96.
97.     if opt.preprocess == 'none':
98.         transform_list.append(transforms.Lambda(lambda img: __make_power_2(img, base=4, method=method)))
99.
100.    if not opt.no_flip:
101.        if params is None:
102.            transform_list.append(transforms.RandomHorizontalFlip())
103.        elif params['flip']:
104.            transform_list.append(transforms.Lambda(lambda img: __flip(img, params['flip'])))
105.
106.    if convert:
107.        transform_list += [transforms.ToTensor()]
108.        if grayscale:
109.            transform_list += [transforms.Normalize((0.5,), (0.5,))]
110.        else:
111.            transform_list += [transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]

```

```

112.     return transforms.Compose(transform_list)
113.
114.
115. def __make_power_2(img, base, method=Image.BICUBIC):
116.     ow, oh = img.size
117.     h = int(round(oh / base) * base)
118.     w = int(round(ow / base) * base)
119.     if h == oh and w == ow:
120.         return img
121.
122.     __print_size_warning(ow, oh, w, h)
123.     return img.resize((w, h), method)
124.
125.
126. def __scale_width(img, target_size, crop_size, method=Image.BICUBIC):
127.     ow, oh = img.size
128.     if ow == target_size and oh >= crop_size:
129.         return img
130.     w = target_size
131.     h = int(max(target_size * oh / ow, crop_size))
132.     return img.resize((w, h), method)
133.
134.
135. def __crop(img, pos, size):
136.     ow, oh = img.size
137.     x1, y1 = pos
138.     tw = th = size
139.     if (ow > tw or oh > th):
140.         return img.crop((x1, y1, x1 + tw, y1 + th))
141.     return img
142.
143.
144. def __flip(img, flip):
145.     if flip:
146.         return img.transpose(Image.FLIP_LEFT_RIGHT)
147.     return img
148.
149. def clip_CT(inp):
150.     mask1 = (inp < 700)
151.     mask2 = (inp > 1200)
152.     inp[mask1] = 700
153.     inp[mask2] = 1200
154.     return inp
155.
156. def imshow(img):
157.     npimg = img.numpy()
158.     plt.imshow(np.transpose(npimg, (1, 2, 0)))
159.
160.
161.
162. def __print_size_warning(ow, oh, w, h):
163.     """Print warning information about image size(only print once)"""
164.     if not hasattr(__print_size_warning, 'has_printed'):
165.         print("The image size needs to be a multiple of 4. "
166.               "The loaded image size was (%d, %d), so it was adjusted to "
167.               "('%d, %d'). This adjustment will be done to all images "
168.               "whose sizes are not multiples of 4" % (ow, oh, w, h))
169.     __print_size_warning.has_printed = True
170.

```

C.2) aligned_dataset.py

```

1. import os
2. from data.base_dataset import BaseDataset, get_params, get_transform, clip_CT, imshow
3. from data.image_folder import make_dataset
4. from PIL import Image
5. import matplotlib.pyplot as plt
6.
7. import torch
8.
9. import sagemaker
10. import pickle
11. import boto3
12. import monai
13. from monai.transforms import (
14.     AddChannel,
15.     AsChannelFirst,
16.     Compose,
17.     RandFlip,
18.     RandRotate,
19.     SpatialCrop,
20.     RandZoom,
21.     ScaleIntensity,
22.     Resize,
23.     ToTensor,
24.     Transform,
25. )
26. from monai.networks import normal_init
27. from monai.data import CacheDataset, DataLoader, Dataset
28. from monai.config import print_config
29. from monai.apps import download_and_extract
30. from sagemaker import get_execution_role
31.
32.
33. class AlignedDataset(BaseDataset):
34.     """A dataset class for paired image dataset.
35.
36.     It assumes that the directory '/path/to/data/train' contains image pairs in the form of {A,B}.
37.     During test time, you need to prepare a directory '/path/to/data/test'.
38.     """
39.
40.     def __init__(self, opt):
41.         """Initialize this dataset class.
42.
43.         Parameters:
44.             opt (Option class) -- stores all the experiment flags; needs to be a subclass of BaseOptions
45.         """
46.
47.         sagemaker_session = sagemaker.Session(boto3.session.Session(
48.             aws_access_key_id='A*****',
49.             aws_secret_access_key='*****',
50.             region_name='us-east-1'))
51.
52.
53.
54.         role = get_execution_role()
55.
56.         self.bucket = sagemaker_session.default_bucket()
57.         prefix = 'sagemaker/dcgan'

```

```

58.
59.     conn = boto3.client('s3')
60.     contents = conn.list_objects(Bucket=self.bucket, Prefix='sagemaker/dcgan')['Contents']
61.     sagemaker_files = []
62.
63.     for f in contents:
64.         sagemaker_files.append(f['Key'])
65.
66.     self.sagemaker_files = sagemaker_files
67.
68.     BaseDataset.__init__(self, opt)
69. #     self.dir_AB = os.path.join(opt.dataroot, opt.phase) # get the image directory
70.     self.dir_AB = contents
71. #     self.AB_paths = sorted(make_dataset(self.dir_AB, opt.max_dataset_size)) # get image paths
72.     self.AB_paths = self.sagemaker_files
73.     assert(self.opt.load_size >= self.opt.crop_size) # crop_size should be smaller than the size of loaded image
74.     self.input_nc = self.opt.output_nc if self.opt.direction == 'BtoA' else self.opt.input_nc
75.     self.output_nc = self.opt.input_nc if self.opt.direction == 'BtoA' else self.opt.output_nc
76.
77.
78.     def __getitem__(self, index):
79.         """Return a data point and its metadata information.
80.
81.         Parameters:
82.             index - - a random integer for data indexing
83.
84.         Returns a dictionary that contains A, B, A_paths and B_paths
85.             A (tensor) - - an image in the input domain
86.             B (tensor) - - its corresponding image in the target domain
87.             A_paths (str) - - image paths
88.             B_paths (str) - - image paths (same as A_paths)
89.
90.
91.
92.         """
93. #         read a image given a random integer index
94. #         AB_path = self.AB_paths[index]
95. #         AB = Image.open(AB_path).convert('RGB')
96.
97.
98. #         w, h = AB.size
99. #         w2 = int(w / 2)
100. #         A = AB.crop((0, 0, w2, h))
101. #         B = AB.crop((w2, 0, w, h))
102.
103. #         # apply the same transform to both A and B
104. #         transform_params = get_params(self.opt, AB.size)
105. #         A_transform = get_transform(self.opt, transform_params, grayscale=(self.input_nc == 1))
106. #         B_transform = get_transform(self.opt, transform_params, grayscale=(self.output_nc == 1))
107.
108.
109. #         A = A_transform(A)
110. #         B = B_transform(B)
111.
112. #         print(A.size())
113. #         print(B.size())
114. #         print(AB_path)
115. #         type(A)
116.
117.
118.     AB_path = self.sagemaker_files[index]
119. #     print(AB_path)

```

```

120.
121.     conn = boto3.client('s3')
122.     response = conn.get_object(Bucket=self.bucket, Key=AB_path)
123.     body = response['Body'].read()
124.     list_of_dict = pickle.loads(body)
125.
126.     RS = list_of_dict["RS"]
127.     CT = list_of_dict["CT"]
128. #     print(CT.size,RS.size)
129. #     split AB image into A and B
130.     CT = clip_CT(CT) #clip the CT for the desired range
131.
132. #     mask1 = (inp < 700)
133. #     mask2 = (inp > 1200)
134. #     inp[mask1] = 700
135. #     inp[mask2] = 1200
136.
137.     transforming = Compose([
138.         AddChannel(),
139.         ScaleIntensity(),
140. #         RandRotate(range_x=15, prob=0.5, keep_size=True),
141. #         RandFlip(spatial_axis=0, prob=0.5),
142. #         RandZoom(min_zoom=0.9, max_zoom=1.1, prob=0.5, keep_size=True),
143. #         Resize(spatial_size=[64, 64]),
144.         SpatialCrop(roi_center=[199,320],roi_size=[128,128]),
145. #         Resize(spatial_size=[64, 64]),
146.         ToTensor()
147.     ])
148.
149.     A = transforming(RS)
150.     B = transforming(CT)
151.
152. #     print("tensor A size:", A.size())
153. #     print(B.size())
154. #     print("AB path is:", AB_path)
155. #     type(A)
156.
157.     norm = transforms.Normalize((0.5,), (0.5,))
158.     A = norm(A)
159.     B = norm(B)
160.
161.     A=A.float()
162.     B=B.float()
163.
164. #     tensor1 = B.permute(1, 2, 0)
165. #     tensor1 = tensor1.cpu()
166. #     tensor1 = tensor1.detach().numpy()
167. #     plt.imshow(tensor1)
168.
169.
170.
171.     return {'A': A, 'B': B, 'A_paths': AB_path, 'B_paths': AB_path}
172.
173. def __len__(self):
174.     """Return the total number of images in the dataset."""
175.     return len(self.AB_paths)
176.
177. # def imshow(img):
178. #     npimg = img.numpy()
179. #     plt.imshow(np.transpose(npimg, (1, 2, 0)))
180.

```

C.3) main

```

1. import torch
2. torch.cuda.get_device_name(0)
3. import os
4.
5. torch.cuda.get_device_name(0)
6.
7. os.environ['MKL_THREADING_LAYER'] = 'GNU'
8.
9. cd pytorch-CycleGAN-and-pix2pix/
10. pip install dominate
11. pip install monai
12. !pip install -r requirements.txt
13.
14. !python datasets/combine_A_and_B.py --fold_A data/A --fold_B data/B --fold_AB data
15.

```

```

[fold_A] = data/A
[fold_B] = data/B
[fold_AB] = data
[num_imgs] = 1000000
[use_AB] = False
[no_multiprocessing] = False
split = test, use 2/2 images
split = test, number of images = 2
split = .ipynb_checkpoints, use 0/0 images
split = .ipynb_checkpoints, number of images = 0
split = train, use 188/188 images
split = train, number of images = 188

```

```

1. #!python train.py --dataroot ./data --name train --model pix2pix --netG unet_128 --direction AtoB --batch_size 4 --
   n_epochs 300 --dataset_mode aligned --norm batch --pool_size 0 --display_winsize 128 #--preprocess None
2.
3.
4. !python train.py --dataroot ./data --name train --model pix2pix --direction AtoB --batch_size 4 --n_epochs 200 --
   dataset_mode aligned --netG unet_128 --display_winsize 128 --preprocess None --input_nc 1 --output_nc 1

```

```

learning rate 0.0002000 -> 0.0002000
(epoch: 1, iters: 100, time: 0.054, data: 0.627) G_GAN: 2.263 G_L1: 14.514
D_real: 0.181 D_fake: 0.172
End of epoch 1 / 300 Time Taken: 10 sec
learning rate 0.0002000 -> 0.0002000
(epoch: 2, iters: 12, time: 0.059, data: 0.000) G_GAN: 1.255 G_L1: 13.671
D_real: 0.934 D_fake: 0.451
(epoch: 2, iters: 112, time: 0.036, data: 0.000) G_GAN: 1.151 G_L1: 13.769
D_real: 0.561 D_fake: 0.461
End of epoch 2 / 300 Time Taken: 7 sec
learning rate 0.0002000 -> 0.0002000

```



```

(epoch: 3, iters: 24, time: 0.050, data: 0.000) G_GAN: 1.610 G_L1: 9.520 D
_real: 0.535 D_fake: 0.654
(epoch: 3, iters: 124, time: 0.040, data: 0.000) G_GAN: 1.951 G_L1: 8.652
D_real: 0.244 D_fake: 0.736
End of epoch 3 / 300   Time Taken: 7 sec
learning rate 0.0002000 -> 0.0002000
(epoch: 4, iters: 36, time: 0.047, data: 0.002) G_GAN: 1.231 G_L1: 10.714
D_real: 0.203 D_fake: 0.665
(epoch: 4, iters: 136, time: 0.060, data: 0.000) G_GAN: 2.210 G_L1: 11.326
D_real: 0.229 D_fake: 0.167
End of epoch 4 / 300   Time Taken: 8 sec
learning rate 0.0002000 -> 0.0002000

```

```

1. import matplotlib.pyplot as plt
2.
3. img = plt.imread('/home/ec2-user/SageMaker/pytorch-CycleGAN-and-
pix2pix/checkpoints/train/web/images/epoch111_fake_B.png')
4. plt.imshow(img, cmap='viridis')
5.

```

