CHAPTER 3

Methodology

In this chapter we describe our methodology, following the used pipeline, as shown in Figure 3.1. First, we preprocessed the data to create a dataset consisting of images of Drusen. Then we trained a generative model on this dataset. After training the generative model we trained a model to find interpretable directions in its input space which then were visualized. As this has not been done before, we explored different techniques and encountered challenges. We discuss those as part of the motivation for the decisions we made and show what is interesting to investigate in future work. The implementation can be found under https://github.com/ChristianMuth5/IntDrusen.

3.1 Data

The raw data consists of multiple equally spaced B-scans per patient that stem from SD-OCT. An example of such a B-scan can be seen in Figure 3.2. From these scans the segmentation of the drusen was done with the model trained by Morelle et al. [MWFS23]. We investigated two different datasets, called *Duke* and *Bonn*. With Duke, we refer to the dataset by Farsiu et al. [FCO+14], which is publicly available. It contains 269 scans of patients with Drusen. The data is anonymized. Each scan consists of 100 B-scans, with a resolution of 512 by 1000 pixels. The other one, Bonn, is not publicly available and was provided by Thomas Schultz and Olivier Morelle, who work for the Universität Bonn. It contains 112 scans of patients with Drusen. Each scan consists of around 100 B-scans, with a resolution of 496 by 512 pixels. A visual comparison between those two can be seen in Figure 3.3, showing that they are quite similar. However, Duke is more noisy, which is why we decided to focus on Bonn.

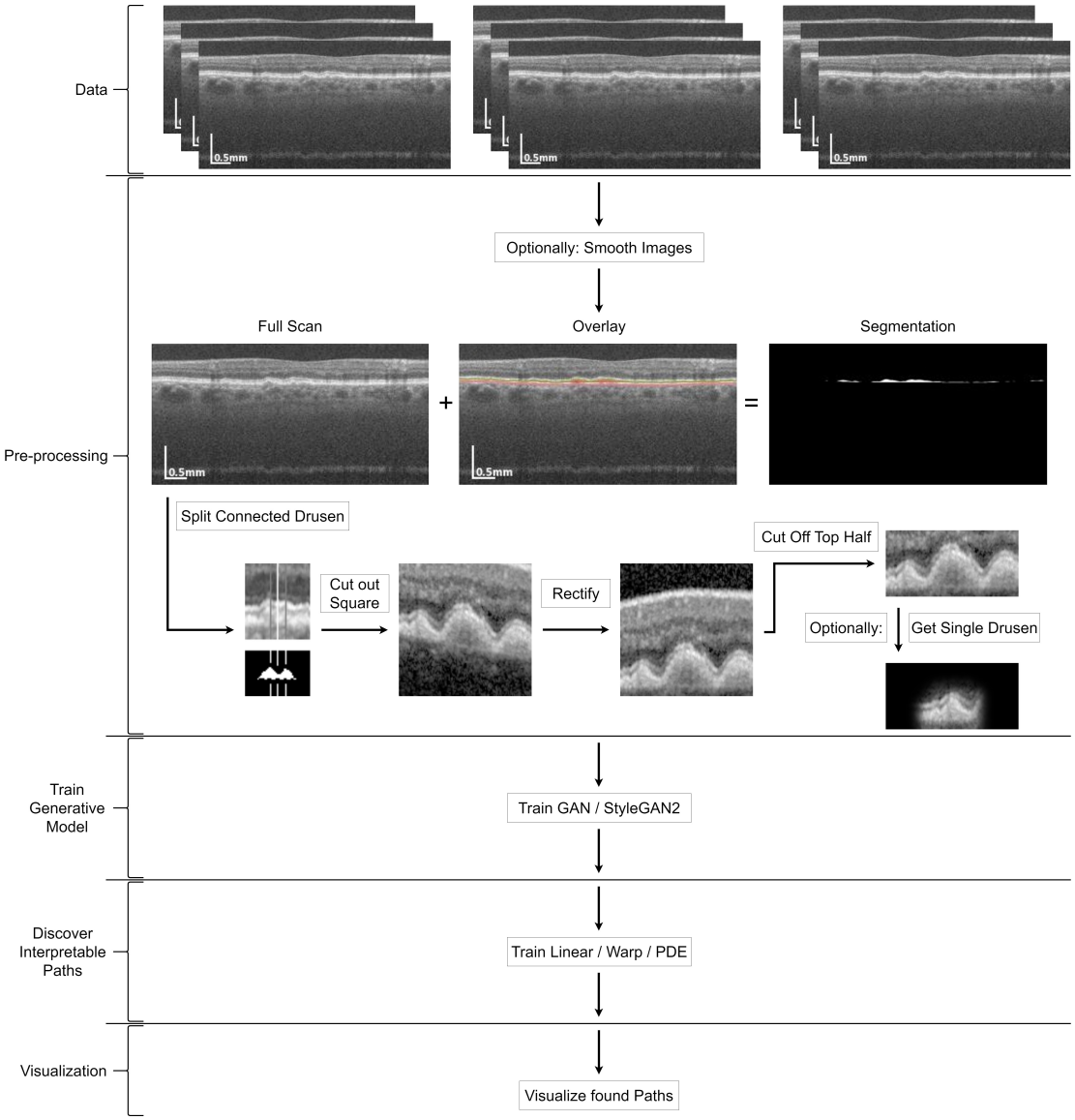


Figure 3.1: Schematic Overview of our proposed Pipeline. The data consists of multiple patients, each with multiple B-scans. These B-scans are optionally smoothed. Combining the full scans with the layer annotations yields the segmentation. The Drusen are segmented and split into individual Drusen. Each Drusen is rectified and the top is removed. At the end, either an image of multiple Drusen or of a single Drusen with a blurred surrounding has been created. The generated dataset is used for the training of a generative model. A path model is trained in order to discover interpretable paths in the input space of the generative model. The found paths are visualized as images and GIFs.

3.2 Pre-processing

The drusen, which we are interested in, are between the BM (Bruch's membrane) line, and the RPE (retinal pigment epithelium) line, allowing us to create a segmentation mask for said structures. In Figure 3.3, one can see that drusen are often not isolated, but rather connected, forming a structure, comparable to a mountain range. This leads to difficulties separating them, such as detecting where one ends and another one starts. Another problem is that for multiple connected Drusen, cutting out a single Drusen in the middle creates an image of two almost parallel lines, the RPE and the BM.

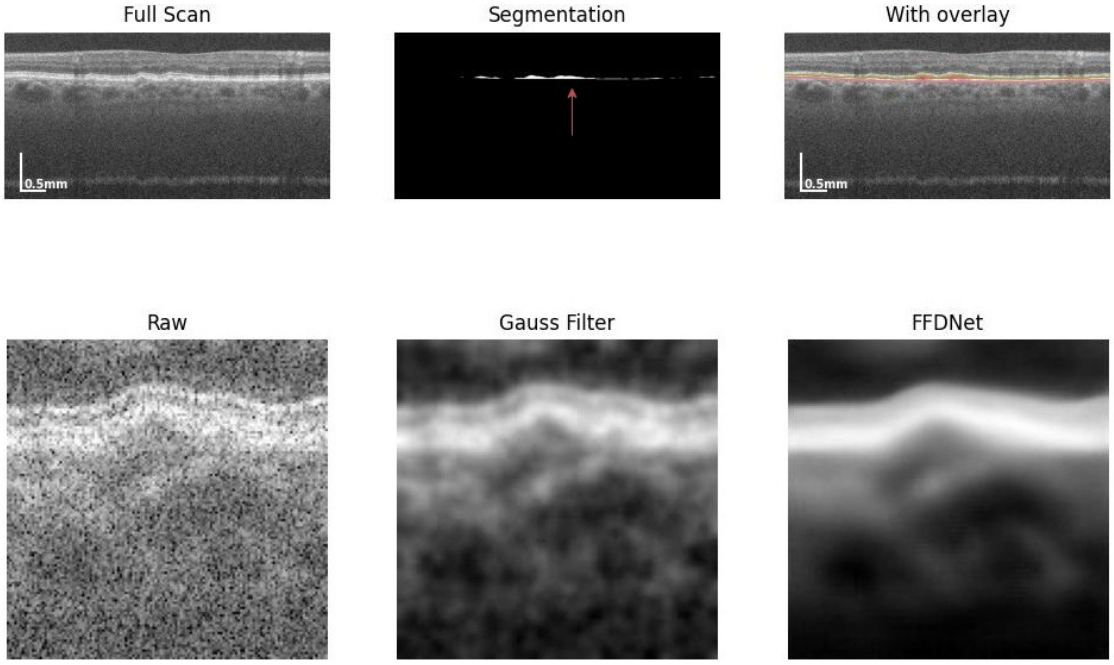


Figure 3.2: Overview of the data, with a full scan from the publicly available Duke dataset from Farsiu et al. [FCO+14]. The segmentation has been done with the model trained by Morelle et al. [MWFS23]. The red arrow in the segmentation shows a connected component of multiple Drusen. We extracted the drusen, which can be seen as the raw image. For noise reduction, we tested Gauss filtering and an FFDNet [ZZZ18, TDV19].

With the Duke data being noisy, the generative models struggled to learn the generation of synthetic drusen images, so we investigated different noise removal techniques. The result of using Gaussian filtering can be seen in Figure 3.2. Similar results were observed with median-filtered images, or bilateral filtering followed by a median filter. Visually the most appealing result comes from using FFDNet, introduced by Zhang et al. [ZZZ18]. We took the pre-trained model from Tassano et al. [TDV19]. When using such noise reduction methods there is information loss, we were not able to determine which of those

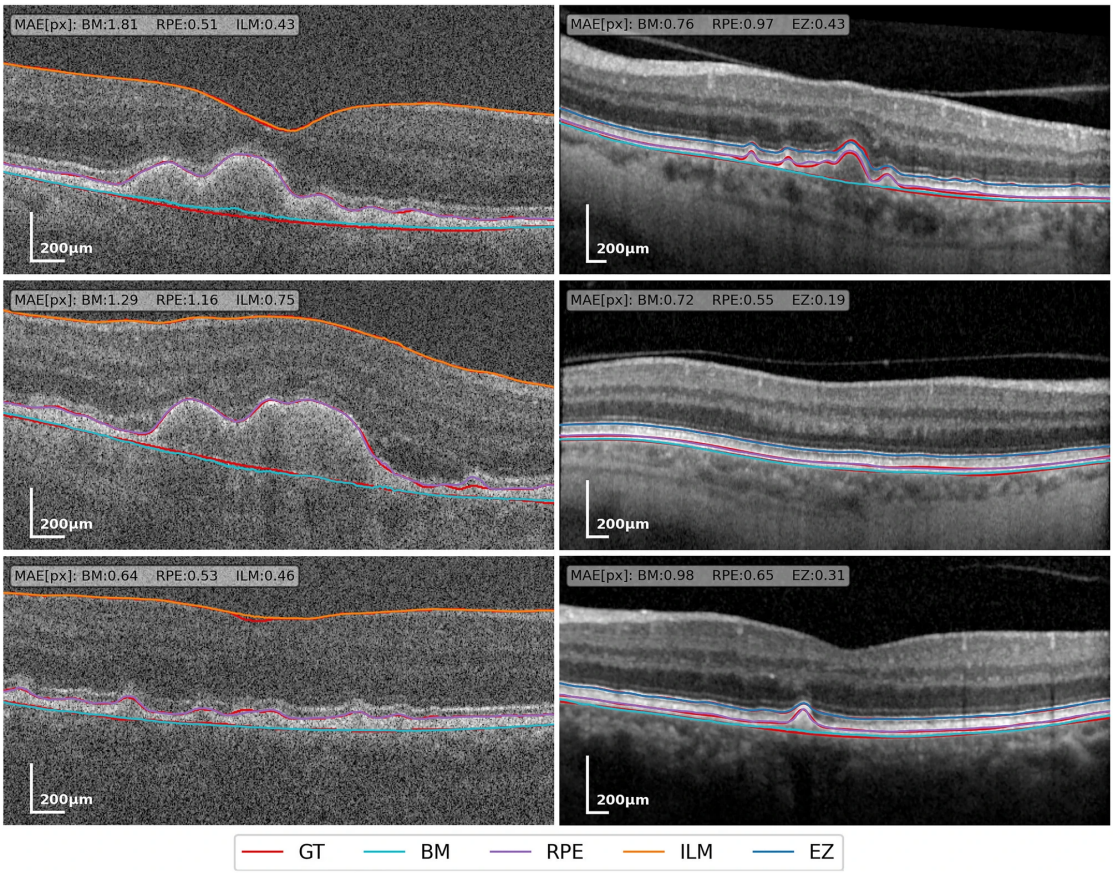


Figure 3.3: A visual comparison between Duke (on the left) and Bonn (on the right). The data was acquired using different machines, Bioptigen on the left and Spectralis on the right. It shows the Bruch's membrane (BM), retinal pigment epithelium (RPE), internal limiting membrane (ILM), and the ellipsoid zone (EZ). These four layer annotations are predicted by Morelle et al. [MWFS23], with the ground truth (GT) for each layer shown in red. The BM and RPE layers aid in the segmentation of the Drusen, which lies between those two. Image taken from Morelle et al. [MWFS23].

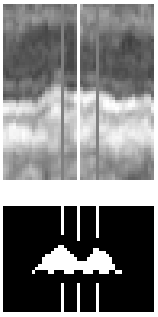
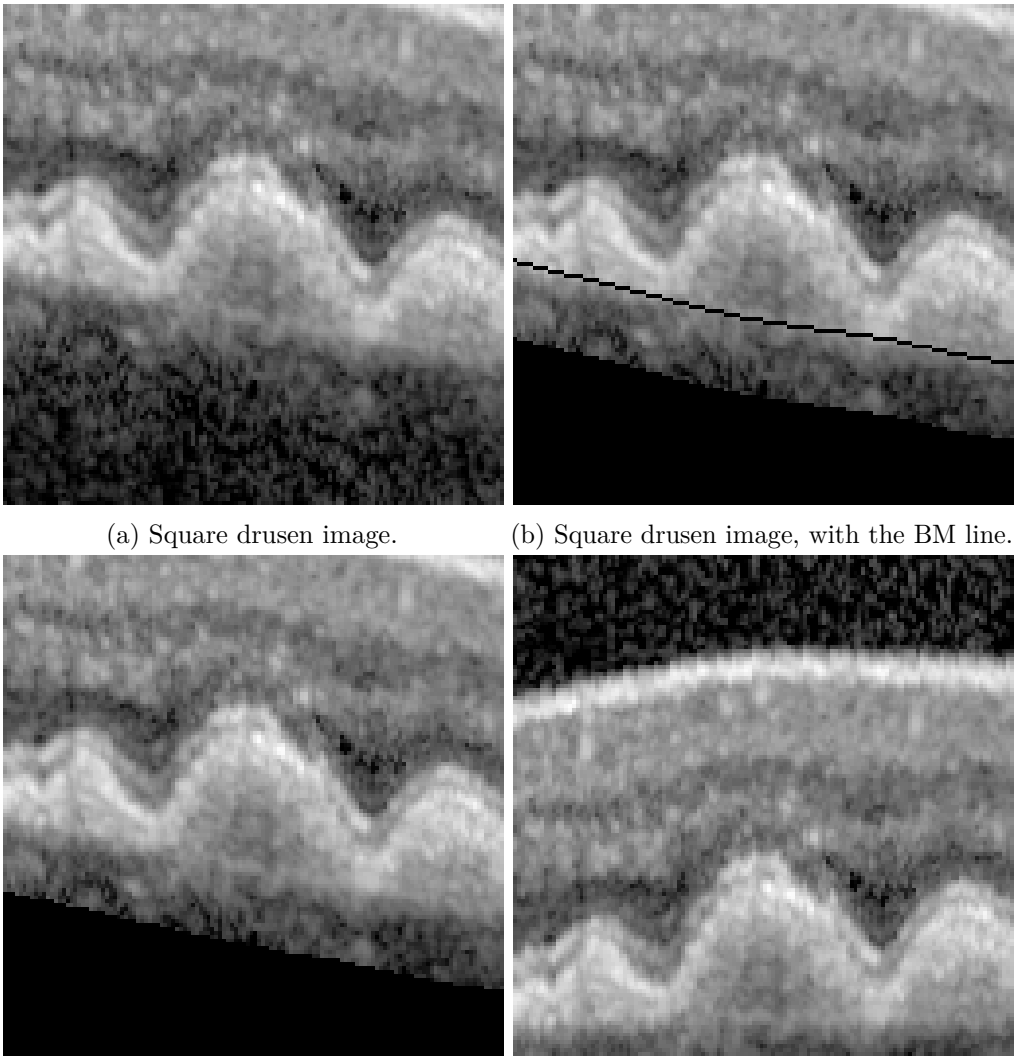


Figure 3.4: Image of two connected drusen, with the two found peaks, the valley in between them, and where the segmentation is cut.

produced the most realistic results, as this requires expert knowledge. As far as we could find out, this might not be possible due to a lack of comparative material containing the ground truth, i.e. microscopic imaging, and SD-OCT imaging.

We decided to move forward with using Bonn, the higher-quality dataset, with less noise. As noise removal was not necessary for it, we did not need to further process the information present in the patch containing the Drusen. For training a model, and finding paths related to features of Drusen we need images of Drusen. Looking at the segmentation in Figure 3.2, we can see in the center that multiple Drusen form a connected component. Thus simply taking its center is not sufficient, as the peak of the Drusen is not always in the center along the y-axis. This creates problems later on in the pipeline where many discovered directions of the Drusen correspond to the Drusen moving along the y-axis. As this is not what we want, we created a robust method for segmenting a single Drusen, which can then be translated along the y-axis to the center. For this purpose, we first tried a binary erosion on the segmentation mask. Figure 3.4 shows an example of two Drusen that are connected with a valley in between them. The thickness of these valleys ranges from a few pixels to dozens. Thus a simple binary erosion did not work. To solve this problem we used signal processing. For each column of the segmentation, we count how many pixels are part of it, creating a sequence of numbers, a signal. The numbers of this signal represent the height of Drusen, allowing us to find peaks. For us to decide whether peaks are Drusen they need to have a minimum height and distance to each other. For two peaks that are next to each other, we find the deepest point of the valley between them. If this valley is deep enough, compared to the peaks, we set the corresponding column in the segmentation to zero. In this way, we were able to split it into separate components.

These components were further analyzed to get the segmentation masks for the Drusen. By calculating the number of pixels in a given component we were able to exclude components that are smaller than a given threshold. For the threshold, we tested several different values to find one that is large enough to remove those that are most likely not Drusen while keeping the ones that are. For the remaining components, we calculate



(a) Square drusen image.

(b) Square drusen image, with the BM line.

(c) Square drusen image, where pixels that are 10 pixels below the BM line are removed.

(d) Square drusen image with columns shifted down for a straightened BM line.

Figure 3.5: Steps in data pre-processing to get square images of drusen.

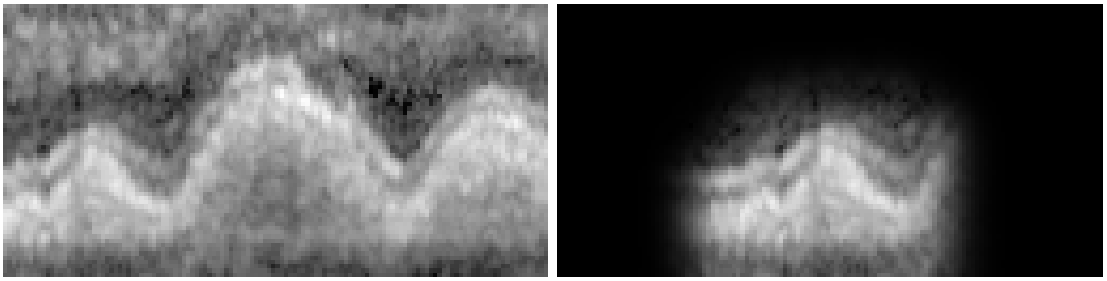
the center of mass. By cutting out a square from the B-scan that is centered on the center of mass of the drusen, we create images similar to Figure 3.5a. As we can see in Figures 3.3, and 3.5 the BM line, which is below the drusen is not straight. Because of this, we observed that most discovered paths revolved around transforming this line, which is not a feature we were interested in. This leads to us needing to straighten the scan so that the BM line becomes straight. As the pixel spacing is not the same at every position and in each direction, we do not want to deform the drusen by rotation. Instead, we shift down the columns of the image to the lowest point of the entire BM line. The prediction for the BM line, created by Morelle et al. [MWFS23], contains jumps, which is

why before using it for line straightening, we smooth it with a one-dimensional Gaussian filter. The result can be seen in Figure 3.5d. These square images contain not only the Drusen but also a lot of additional information such as the shapes of layers which we are not interested in. To not find features related to uninteresting information we decided to remove as much of it as possible. We investigated and found that less than 0.01% of images contain Drusen that are big enough to overlap into the top half of the image. As such we decided to cut off the upper half of the image to limit the amount of uninteresting features present in our data, as shown in Figure 3.6a.

For training models on images of multiple Drusen, this is good enough. Now with the Drusen segmented, we can also create images containing a single Drusen. For this, the Drusen is cut out using the segmentation. Using just the segmentation would result in sharp edges and fine details in the border region missing. As the segmentations are not perfect, these fine details might contain interesting information about our Drusen. Furthermore, the sharp edges are found later on in our pipeline as features that we do not want. As such, we use dilation, followed by Gaussian filtering for blurring the border region. Then the original segmentation is set back to 1, when multiplying the image by the segmentation this results in the original segmentation not being blurred. This approach leads to border cases, for example, the Drusen can be on the left or right edge of the B-scan. To solve these cases, we check how good it is salvageable by moving it to the center, it then should not create a sharp border. Cutting out Drusen from a patch where multiple have grown together can sometimes create an image that looks like a line that is just on the top part of the image. We try to filter those out as well as possible by looking at the height of the line on the left and right edges of the image. If one of those is in the top half of the image, we discard this Drusen. Additionally, a dictionary is created, containing height, width, and volume information for each drusen image. These were used for testing conditional generative models and may be used in future work. In our experiments, we found out that removing these special cases is necessary for discovering interpretable features of images containing a single Drusen. Also, only about 10% of Drusen are lost in this way. The final results of our data pre-processing pipeline can be seen in Figure 3.6. It shows the two datasets we created. The first one contains one or more Drusen per image. The second one has one Drusen per image. This allows us to analyze features of multiple Drusen, and single Drusen.

3.3 Generative Models

We created synthetic images of drusen with two models, DCGAN [RMC15], and StyleGAN2 [KLA+20], as these models were best usable in the chosen papers for finding interpretable paths. Both follow the Generative Adversarial Net (GAN) approach described by Goodfellow et al. [GPAM+14], which can be seen in Figure 3.7. The discriminator D receives as input either images randomly sampled from the dataset, or created by the generator G. The generator receives vectors as input. These so-called *latent vectors* z are randomly sampled from the *latent space* $\mathcal{Z}$. This describes an adversarial game



(a) An example of a Drusen image containing multiple Drusen. This image is centered around the middle Drusen.

(b) An example of a Drusen image modified to contain only a single Drusen, which is the left one of Figure 3.6a.

Figure 3.6: The final result of our data pre-processing pipeline, showing a) multiple Drusen and b) single Drusen image.

played by the generator and the discriminator. The discriminator tries to guess where its input came from, and the generator tries to deceive the discriminator. A commonly used analogy to understand this is to see the generator as a forger for paintings, and the discriminator is the analyst trying to find out if a painting is a real one or forged. In the beginning, both are not good at their tasks, but during training, they become better. The architectures of both models and their hyperparameters, such as learning rates, need to be balanced. Otherwise one becomes too good too fast and the other one is not capable of learning. In this process we can calculate the loss of the network as we know what was fed to the discriminator, allowing us to train a generative model for creating synthetic images similar to our dataset. To increase the variance in our dataset we augment by randomly flipping horizontally. We decided to not use any augmentations that change the pixel values or introduce features that are not present naturally.

The volume of Drusen is not distributed well, with over 90% of Drusen being in the smallest class of 10 total classes. To analyze the impact of the data distribution, we train with and without weighted sampling. For weighted sampling, we split our data into 5 classes, corresponding to the volume of the Drusen. The values for these classes were handpicked. When using the inverse of samples per class as the sampling weight, we observed *mode collapse*. Mode collapse describes when the generative model produces only a few different samples. An overview of mode collapse in GAN training with a more detailed description has been done by Kossale et al. [KAD22]. To combat this, we decided to use weights that do not fall off quite as rapidly, namely $\frac{1}{2^5}, \frac{1}{2^4}, \frac{1}{2^3}, \frac{1}{2^2}, \frac{1}{2}$. These weights lead to the GAN seeing big drusen more often, compared to random sampling, while still not allowing the generative model to learn to reproduce those few to fool the discriminator.

A DCGAN is a fully convolutional variant implementing the GAN approach and is described by Radford et al. [RMC15]. The proposed architecture of the generator G can be seen in Figure 3.8. The discriminator D is similar to it, but in reverse, and ends with

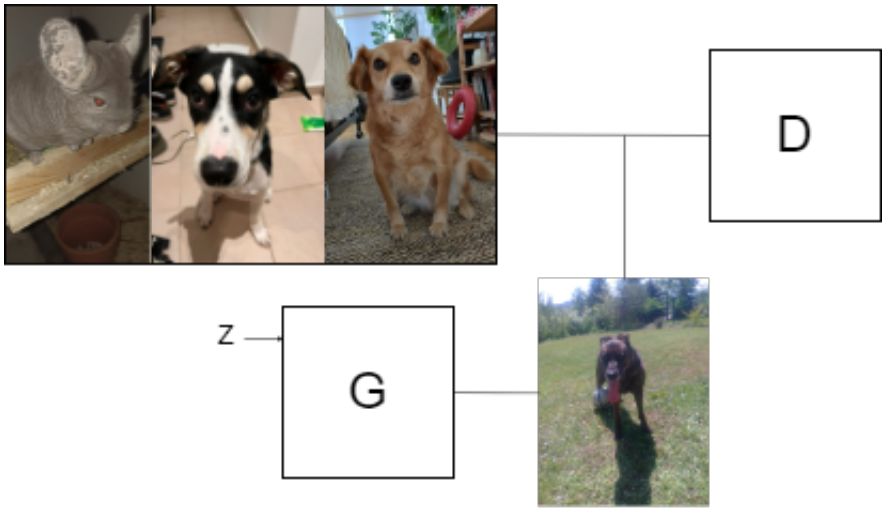


Figure 3.7: Generative Adversarial Net (GAN) approach for training a generative model, described by Goodfellow et al. [GPAM+14].

only one value, the class prediction. As our dataset seems to have few variables describing it, we decided on a smaller latent vector of size 20. The images are rectangular, with a size of 64 by 128, and not 64 by 64, as in the original architecture. To allow for this adaptation we added an additional layer with different parameters for height and width. Thus it doubles only one dimension in the generator and halves only one dimension in the discriminator. We train for 100 epochs, each containing the whole dataset in random order, with binary cross entropy as the loss function. This number of epochs has been chosen as loss and accuracy values show convergence and it allows to train a model in under 2 hours. Allowing us to explore different hyperparameters.

StyleGAN is one of the most used architectures for GANs, first described by Karras et al. [KLA19], with StyleGAN3 [KAL+21] being its newest update. The architecture of StyleGAN can be seen in Figure 3.9. It consists of two networks working together as a generative model to generate synthetic images. The mapping network f maps the latent vector $z \in \mathcal{Z}$ onto a style vector $w \in \mathcal{W}$, which is easier to use for the next network. The synthesis network g uses a constant, trained input instead of a latent vector z, affine transformations A of the style vector, and random noise B to create images that are, for humans, often indistinguishable from real ones. StyleGAN architectures are widely used for GAN inversion [XZY+22], and thus an interesting model for future work. It could allow altering the features of real extracted Drusen images, helping us to better our understanding of Drusen.
We decided to use the StyleGAN2 [KLA+20] model, following the guidelines by Karras et al. [KAL+21], which can be found on the corresponding GitHub page [sty]. The parameters they recommend for our image size are `gamma = 0.1024`, `cbase = 16384`, `map-depth = 2`, `glr = 0.0025`, and `dlr = 0.0025`. With these settings, we

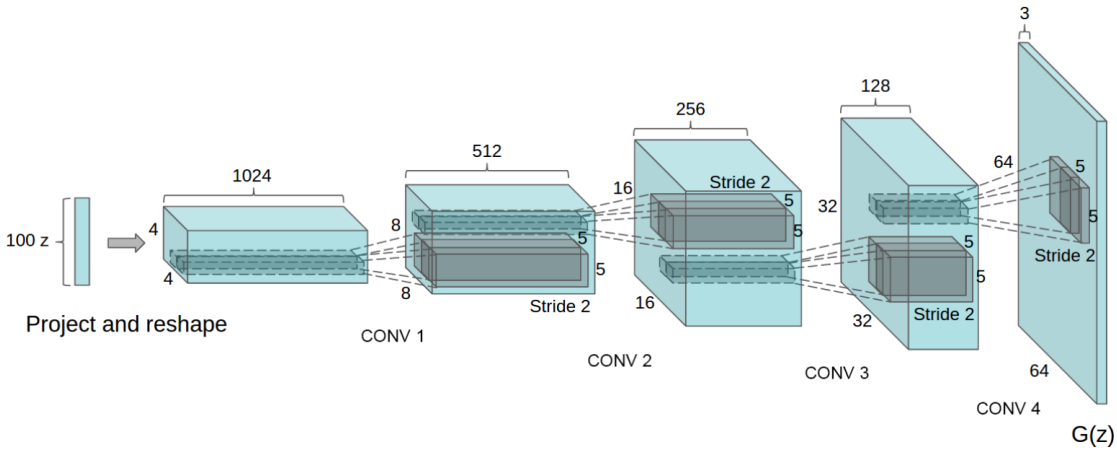


Figure 3.8: Architecture of the generator G of a DCGAN. Image taken from Radford et al. [RMC15].

trained for 1000 epochs with and without weighted sampling, and 2000 epochs without weighted sampling. The training for 2000 epochs was done before we decided on using weighted sampling. We decided to keep the results so that a comparison between with and without weighted sampling can be done and between training for 1000 and 2000 epochs. We trained on a single NVIDIA Titan X. GANs took about 1-2h for 100 epochs, and StyleGANs around 12h for 1000 epochs.

3.4 Paths

Our next goal is to discover features of Drusen. These features can be understood as paths, also called directions in the input space of generative models. When generative models learn to create synthetic data, they learn a mapping from the vectors they receive to an image similar to the ones it was trained on. In this, the values of the vector encode information on which image should be created. The dimensions of this vector are entangled and simply changing one value does not necessarily create a meaningful change of the output. Thus, there has been research done on how to find paths that show features that are interpretable for humans. We are interested in unsupervised methods, as we do not have features labeled in our data and are doing an explorative analysis of Drusen. For this unsupervised discovery of interpretable paths, we decided to use three papers that build on top of each other. Those are *GANLatentSpace* [VB20] by Voynov and Babenko, *WarpedGANSpace* [TTP21] by Tzelepis et al., and *PDETraversal* [SKSW23] by Song et al. which we respectively refer to as *Linear*, *Warp*, and *PDE*. They can be used similarly, and work with our trained generative models. As some time has passed since those papers were published, parts of the code had to be adapted to work with current Python packages. These models find paths in the latent space $\mathcal{Z}$ of our generative models. For a StyleGAN2, we were able to also search for paths in the style space $\mathcal{W}$.

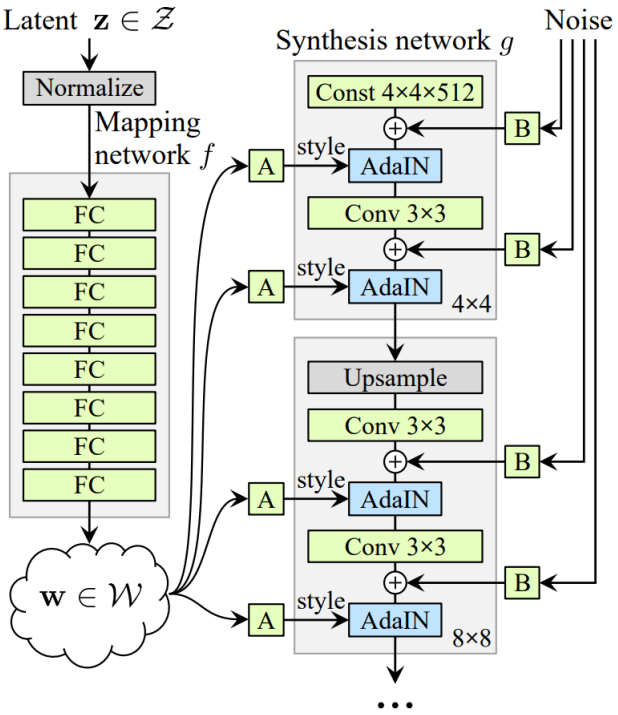


Figure 3.9: StyleGAN architecture showing the two networks and how images are generated. The mapping network learns a mapping from the latent space Z to the style space $\mathcal{W}$. Affine transformations A of the style vector $w \in \mathcal{W}$ are used as inputs throughout the layers of the synthesis network. This network takes a constant 4 by 4 sized image with 512 channels and upsamples it to the desired size. In each step, the affine transformation A and random noise B are added to generate an image of the desired style. The random noise causes random perturbations making images more realistic as areas of the same type (such as hair) do not look like repeating textures. Image taken from Karras et al. [KLA19].

These three methods have in common that they train a reconstructor $R(I_1, I_2)$ on pairs of images I_1, and I_2. Its goal is to reconstruct parameters of the shift between the latent codes of I_1, and I_2. The shift can be understood as moving the latent code along the path in the latent space. For this, they first sample $z \in \mathcal{Z}$, and if searching in $\mathcal{W}$, z is mapped into $\mathcal{W}$. As there is no further difference between searching in $\mathcal{Z}$ or $\mathcal{W}$, we will denote the latent code that will be shifted as z, even if it is in $\mathcal{W}$. After that, they sample parameters for the shift and create the shifted latent code $\hat{z}$. z, and $\hat{z}$ are passed through the generative model in order to get the corresponding images $I_1 = G(z)$, and $I_2 = G(\hat{z})$. By learning parameters of the shift, and the reconstructor R, the training is steered towards producing paths with distinct image transformations. Paths that are hard to distinguish create a higher loss for the reconstructor, as it fails to reconstruct the shift parameters. The Linear approach is visualized in Figure 3.10, with the two other ones being similar. In this chapter we will now discuss those 3 methods, and give short

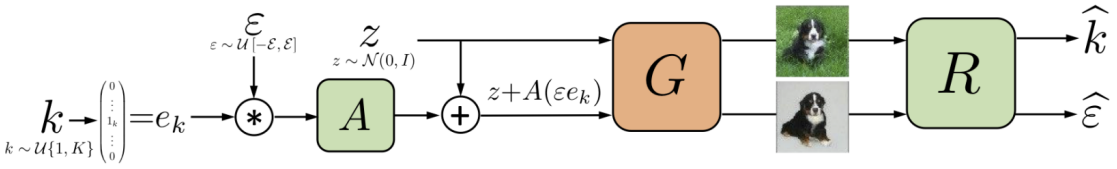


Figure 3.10: Approach for discovering linear directions in the latent space $\mathcal{Z}$ of a generative model G. Image taken from Voynov and Babenko [VB20].

explanations for them and what settings were used for training the models, depending on the type of generative model.

In Linear [VB20], directions are linear and encoded in the matrix A. The shifted latent code is then calculated with $\hat{z} = z + A(\epsilon \cdot e_k)$. $e_k = (0, \ldots, 0, 1_k, 0, \ldots, 0)$ is used to select the kth column of A that contains the kth shift direction. ϵ is how far we move the latent code in that direction. The two parameters k, ϵ are reconstructed by the reconstructor $R(I_1, I_2) = \hat{k}, \hat{\epsilon}$, with $\hat{k}$ being the prediction of k, and $\hat{\epsilon}$ the prediction of ϵ. As the ground truth is known, the loss can be calculated as $L = L_{cl}(k, \hat{k}) + \lambda \cdot L_r(\epsilon, \hat{\epsilon})$. with L_{cl} being cross-entropy loss, and for the regression loss L_r, the mean absolute error is used, with a weight coefficient $\lambda = 0.25$. We decided to follow the main approach of the paper, where the directions in A are orthonormal columns. The approach can be seen in Figure 3.10.

WarpedGANSpace [TTP21] builds on this approach by discovering n non-linear paths that follow the gradients of n RBF-based warping functions $f^1, \ldots, f^n$ of the latent space. A warping function f^k is a weighted sum of parametric Gaussian RBFs:

$$f^k(z) = \sum_{i=1}^{N} \alpha_i \exp(-\gamma_i \|z - s_i\|^2)$$

s_i are called support sets, α_i weight, and γ_i scale. γ_i corresponds to non-linearity, meaning small values create linear, large values non-linear paths. These warping functions can be understood as describing high-dimensional warpings of the latent space, creating a landscape. The gradient of this differentiable function is:

$$\nabla f^k(z) = -2 \sum_{i=1}^{N} \alpha_i \gamma_i \exp(-\gamma_i \|z - s_i\|^2)(z - s_i)$$

As we can see, the paths follow these gradients. The shifted latent code $\hat{z}$ of z into the kth direction by a distance of ϵ can be calculated in this way:

$$\hat{z} = \epsilon \frac{\nabla f^k(z)}{\|\nabla f^k(z)\|}$$

In the same way as the Linear method, the reconstructor R reconstructs k and ϵ. Each path has its own parameters s_i, α_i, γ_i describing the warping. To get those values, they

train a warping network $\mathcal{W}$, instead of the matrix A. The loss is calculated in the same way as in the Linear approach.

PDETraversal [SKSW23] views the multi-dimensional latent space $\mathcal{Z}$, or style space $\mathcal{W}$ as a dynamic potential landscape, in which the paths can be understood as flows down its gradient. This allows the paths to flow differently depending on the initial location of the sampled latent vector z. The landscape corresponding to the kth direction is a time-dependent scalar potential energy field $u^k(z_t, t)$. The time t is sampled from $(0, T-2)$, with T being a hyperparameter. It is learned with a Multilayer Perceptron. The time points z_t can be calculated with $z = z_0$, $z_t = z_{t-1} + \nabla_z u^k(z_{t-1}, t-1)$. The reconstructor R receives two images $I_t = G(z_t)$ and $I_{t+1} = G(z_{t+1})$. It only predicts the direction $\hat{k}$, not the timepoint t. This is because the problem of estimating t is challenging as the potential PDEs can be diverse, and they say it is neither necessary nor practically feasible. They introduce three new loss terms L_f, $L_{\mathcal{J}}$, and L_u in addition to the cross-entropy loss L_k. L_k is the same loss as for the Linear method. The three new losses help in guiding the model to find realistic-looking trajectories with improved semantic variations, and to keep $u(z_0, 0)$ at 0. The full loss for GANs is $L = L_u + L_f + L_{\mathcal{J}} + L_k$.

For choosing the number of epochs used in each approach, we decided to have them run for a similar time. This resulted in 50000, 20000, and 100000 for GANLatentSpace, WarpedGANSpace, and PDETraversal when trained on the DCGAN, and respectively 20000, 20000, and 50000 on the StyleGAN2. All these models are trained in about 2 hours. With a total of 48 path models, training them fast was important to us. We did not observe a qualitative difference when experimenting with training time and training some models for longer. However, it could be investigated whether longer training results in better paths.

3.5 Visualization

Paths that we discover consist of 37 images, with the middle one being the original image. It is possible to show all these images in one single image, but we decided not to, as for our analysis we also want to see the same path for multiple different Drusen. This is because paths might not consistently correspond to the same feature in different images of Drusen. As such we decided to use GIFs as our visualization technique, as shown in Figure 3.11. With our approach, we decided to show 5 different paths for up to 15 different Drusen. Ophthalmologists gave us positive feedback stating that these visualizations are interesting. These visualizations also helped us tremendously in analyzing all the found paths to summarize our findings in the next chapter, Results.

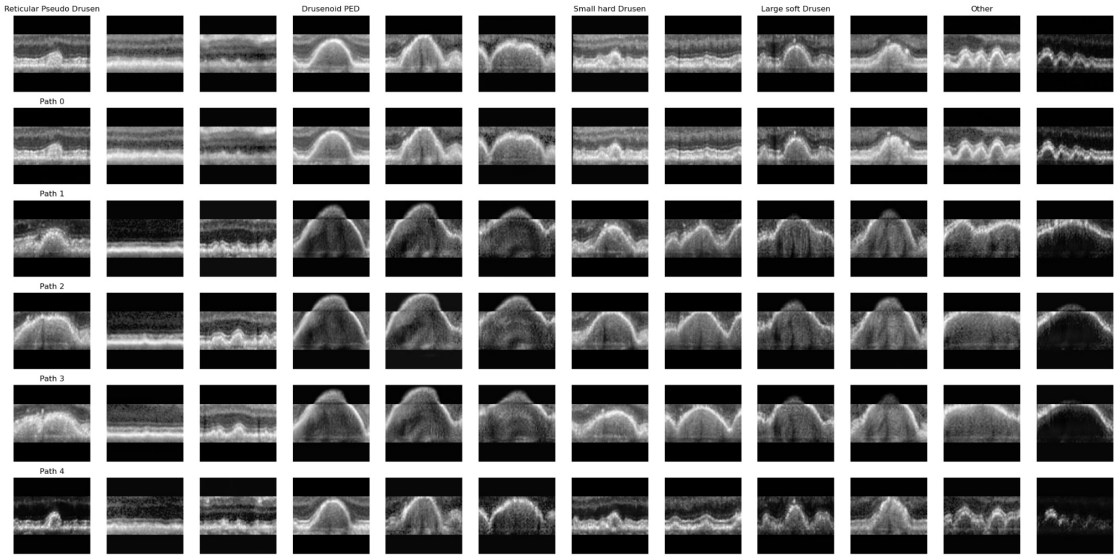


Figure 3.11: An example image of the created GIFs. The first row shows the original image and never changes. All the other rows show a given path and change over time. Each column corresponds to the image in its first row. The annotations for the columns show which type of Drusen we think those are. Here the first 3 are Reticular Pseudo Drusen, then 3 Drusenoid PED, 2 Small Hard Drusen, 2 Large Soft Drusen, and 2 Others. For more information on the types of Drusen, see our Results chapter.