UNIVERSITY OF THESSALY
SCHOOL OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND BIOMEDICAL INFORMATICS

Spatial image registration and applications through use of Deep Learning

GIAZITZIS SYMEON

THESIS
Supervisors
Delibasis Konstantinos
Professor

Lamia, 2023

UNIVERSITY OF THESSALY
SCHOOL OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND BIOMEDICAL INFORMATICS

Spatial image registration and applications through use of Deep Learning

GIAZITZIS SYMEON

THESIS
Supervisors
Delibasis Konstantinos
Professor

Lamia, 2023

Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. *Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί* ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. *Δέχομαι ότι η αυτολεξεί* ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. *Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια*

4. *Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.*

Ημερομηνία: ……/…../20……

Ο – Η Δηλ.

(Υπογραφή)

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.

Χωρική ταύτιση εικόνων και εφαρμογές, με χρήση βαθιάς μάθησης

ΓΙΑΖΙΤΖΗΣ ΣΥΜΕΩΝ

Τριμελής Επιτροπή:

Δελήμπασης Κωνσταντίνος, Καθηγητής (επιβλέπων)

Πλαγιανάκος Βασίλειος, Καθηγητής

Τασουλής Σωτήριος, Επίκουρος Καθηγητής

Abstract

Image registration is a challenge that has been around for many years and it is defined as the geometric transformation of images to achieve spatial matching. Through the concept of neural networks, the powerful yet complex tool of deep learning was introduced into computer vision. Considering that most clinical examination methods in the medical field contain some form of imagery result, an image registration problem is presented. The combination of the above, image registration and deep learning, proposes a hot topic in computer vision in the field of medical imaging. In this thesis, a deep neural network (DNN) is trained on image registration through iterative passes of medical images of retinas. The registration is done through an affine matrix, retrieved from the transformation values predicted from the DNN. Considering that there is not a lot of research related to this subject, this thesis could present a useful base guide for future studies to come.

Contents

1. Introduction

In computer vision, transformation matrices and transformation point grids are a fundamental way of warping an image. One of the most basic ones is the Affine transformation, which can be represented by a 3×3 matrix, which has $[0\ 0\ 1]$ as its last row elements, leaving 6 parameters. That last row makes the matrix applicable for use on a homogeneous coordinate system, without changing its position on the Z axis. The affine matrix contains the basic transformations: Translation, Rotation, Shearing and Scaling. When affinely transforming an image, lines and their parallelism are maintained. The Affine transformation has 6 degrees of freedom (DoF); hence it requires a minimum of 3 points to calculate its matrix.

On the other hand, through the Projective matrix, we can transform an image as being acquired by a different camera pose in 3D (also known as Homography). This transformation can be represented also by a 3×3 matrix, only this time, the 3rd row contains information about the Z axis as well. It has to be noted that the element H_{33} usually gets normalized to 1. That is because the Homography matrix can be calculated only up to a scale factor, which allows us to simply fix the scale of it by setting one of the parameters to 1. In most cases and in almost all implemented algorithms, this parameter is element H_{33}. This results in 8 DoF on the Homography, which leads to a minimum of 4 points required to calculate the Homography of an image. The Homography includes transformations as the Affine, but it does not preserve parallelism between lines. Both Affine and Projective are closed under composition. A preview of the matrices is presented on Table 1 and Table 2.

Affine matrix		
a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
0	0	1

Table 1. The Affine transformation matrix. It consists of 6 variables on the first two rows and two 0 and one 1 on the last row, indicating that there are no changes on the Z axis.

Projective matrix/Homography		
H_{11}	H_{12}	H_{13}
H_{21}	H_{22}	H_{23}
H_{31}	H_{32}	H_{33}

Table 2. The Projective transformation matrix, or Homography. It consists of 9 elements with 8 DoF. By normalizing one of the variables the degrees of freedom are reduced to 8. Usually, the one to normalize is h_{33}.

1.1. Image registration

A chapter of computer vision that has been extensively explored and plays crucial role in many computer vision applications, is image registration. Image registration is the process of aligning two images so that the common contents in them are spatially matching. This can be done in a variety of ways. In classic computer vision, the most widely used methodology to-date, utilizes a transformation matrix, like the ones mentioned before, to warp a moving (*M*) image so that it matches spatially a Fixed (*F*) one. The transformation matrix is based on given, matching, reference points between the two images, called homologous pair of points. The basic methodology is performed in 4 basic steps as shown in Figure 1.

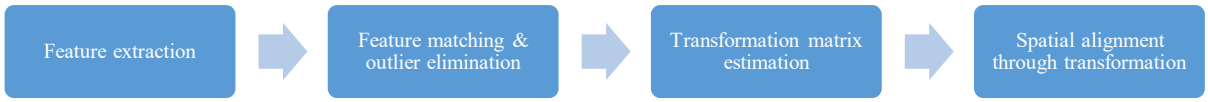


Figure 1. Basic steps of the classic approach for image spatial registration, divided into four generic parts.

This method is applied to image stitching or panorama creation. Each step has a variety of approaches through different algorithms.

i) When it comes to *local feature* extraction *around salient points*, there are many different proposed algorithms, each presenting differences between execution time and results.
   (1) A fundamental algorithm for image registration is the Scale-Invariant Feature Transform (SIFT) [1] algorithm. It creates a scale and rotation independent environment inside the image, where salient (key) points are identified and assigned a local feature descriptor. This is done through image filtering and local pixel-value calculations. Paradigm of such points can be seen in Figure 4 after the points have been matched and filtered for outliers (as will be discussed later). The points were extracted from the images presented in Figure 3.
   (2) Although accurate, SIFT is computationally demanding. This led to the creation of the Speeded-Up Robust Features (SURF) [2] algorithm, which is simply an upgraded version of SIFT. It is computationally lighter and its performance is about the same as SIFT.
   (3) Oriented FAST and Rotated BRIEF (ORB) [3] is currently one of the best feature extracting algorithm for real-time applications. It provides results comparable to SIFT's as seen in Figure 6 (SIFT's result applied on the image shown on Figure 5) in about half the time (features were sampled from the same Figure 3 pair). As its name implies, it utilizes improved versions of the FAST feature detector and the BRIEF feature descriptor algorithms to extract features with descriptors at very low cost.

ii) For feature matching, given a distance metric, each descriptor of image *M* is compared with all the descriptors of *F*. The lowest and second lowest distances for each key point on *M* will be saved. In case these two best match distances are too close with each other (difference or ratio between them is bellow or above a given threshold), the featured points are dumped as they are considered ambiguous. Although this gives spatially correct matches, in most cases bad matches are also presented. These are called outliers and there are outlier detection algorithms that eliminate them, leaving only the inlier features.

Random Sample Consensus (RANSAC) [4] and its variants [5–9], is a commonly used algorithm for both outlier detection and homography transformation. The advantage of this algorithm is that it can calculate robustly the parameters of a given model. Its cons are that it is effective only when at least half of the extracted features are inliers and that the only time limit is the exhaustive method, which is not an option in almost every real-life application.

Having extracted good matches of key points, between two images M and F, the alignment between them will be featured as the transformation matrix that can best match these points, within the two images. The basic idea of how to find the best fit is to solve the eigenvalue problem of $A^T Ah = \lambda h$, trying to find the eigenvector h, with smallest eigenvalue λ of square matrix AA^T so that it minimizes the point reprojection error. In this problem of least squares minimization, A is a $2n \times 9$ matrix, containing the n known feature point coordinates, with $n \geq 4$, whereas the eigenvector h contains each of the H_{ij} elements as shown in Table 2.

In the case of calculating the Affine transformation, things are easier since the minimum points needed are less and the problem we are set to solve is linear. Specifically, we need a bare minimum of $n \geq 3$ feature point pairs and to apply simple linear algebra methodology to find the numbers that would solve a multi-equational system. Mapping a plane as an affine equation will have the form of:

$$\boldsymbol{T(x', y') = (ax + by + c,\ dx + ey + f)}$$

Using that form on three points $A = (A_1, A_2), B = (B_1, B_2), C = (C_1, C_2)$ will give us:

$$\begin{array}{ll} \boldsymbol{A'_1 = aA_1 + bA_2 + c} & \boldsymbol{A'_2 = dA_1 + eA_2 + f} \\ \boldsymbol{B'_1 = aB_1 + bB_2 + c} & \boldsymbol{B'_2 = dB_1 + eB_2 + f} \\ \boldsymbol{C'_1 = aC_1 + bC_2 + c} & \boldsymbol{C'_2 = dC_1 + eC_2 + f} \end{array}$$

The equivalent of the above equation system in linear algebra will be:

$$\begin{bmatrix} \boldsymbol{A_1} & \boldsymbol{A_2} & \boldsymbol{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \boldsymbol{A_1} & \boldsymbol{A_2} & 1 \\ \boldsymbol{B_1} & \boldsymbol{B_2} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \boldsymbol{B_1} & \boldsymbol{B_2} & 1 \\ \boldsymbol{C_1} & \boldsymbol{C_2} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \boldsymbol{C_1} & \boldsymbol{C_2} & 1 \\ & & & \vdots & & \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} \boldsymbol{A'_1} \\ \boldsymbol{A'_2} \\ \boldsymbol{B'_1} \\ \boldsymbol{B'_2} \\ \boldsymbol{C'_1} \\ \boldsymbol{C'_2} \end{bmatrix}$$

At this point it is recalled that using more than three feature points will give us more robust results in the final affine transformation.

The RANSAC algorithm randomly samples the minimum number of feature pairs (4 in the case of homography, 3 in the case of affine) and using the previously mentioned methods respectively, the model is calculated. Using that same model, RANSAC checks which of the feature pairs fit (confirm) that model within a given error window. This process is repeated by a given x number of times. The model that had fitted the most features (inliers) is selected and a least-square version of the model (either homography or affine or any given model) is recalculated using all the selected inliers.

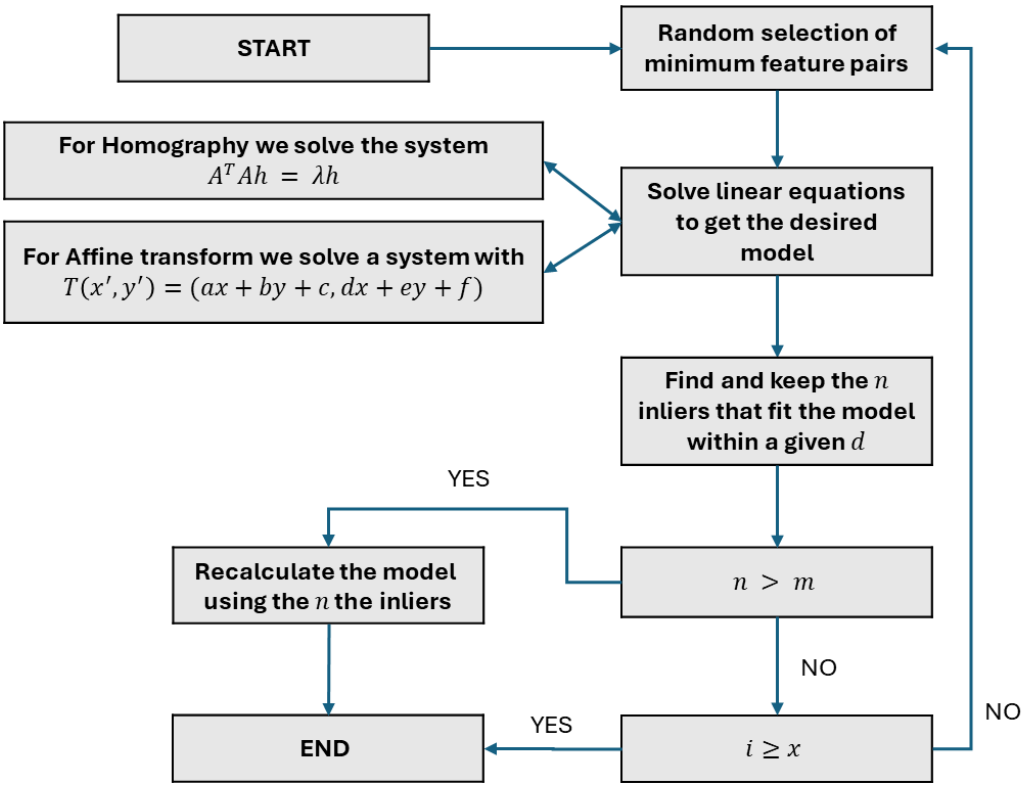


Figure 2. Basic RANSAC algorithm

In Figure 2, *m* stands for the threshold of acceptance for the number of inliers in the entire set of point pairs to be accepted as "good enough" to end the algorithm, *n* is the number of inliers found and stored, *i* is the current iteration and *x* is the max number of iterations that will be performed.

This is just the basic variant (and methodology) of the RANSAC algorithm in a very simplified and brief graph. There is a wide range of different approaches on this algorithm and we will see some of these variants in more detail in the Literature Review section further bellow.

iii) Lastly, the transformation matrix is applied on each one of the pixels of image M, potentially matching it with F, spatially. Two example sets of the complete methodology results are presented from Figure 4 till Figure 7. It is noticeable that

although the features extracted with SIFT are satisfactory, they are not as many as the ones produced with ORB. In both cases there seem to be no outliers. Considering that ORB found many more features compared to SIFT, may explain why it produced a near perfect match, compared to the satisfactory results of SIFT.

Figure 3. A random image showing a boat was warped with a random Homogeneous transformation. The original image (fixed) is on the left and the randomly transformed image (moving) is on the right.

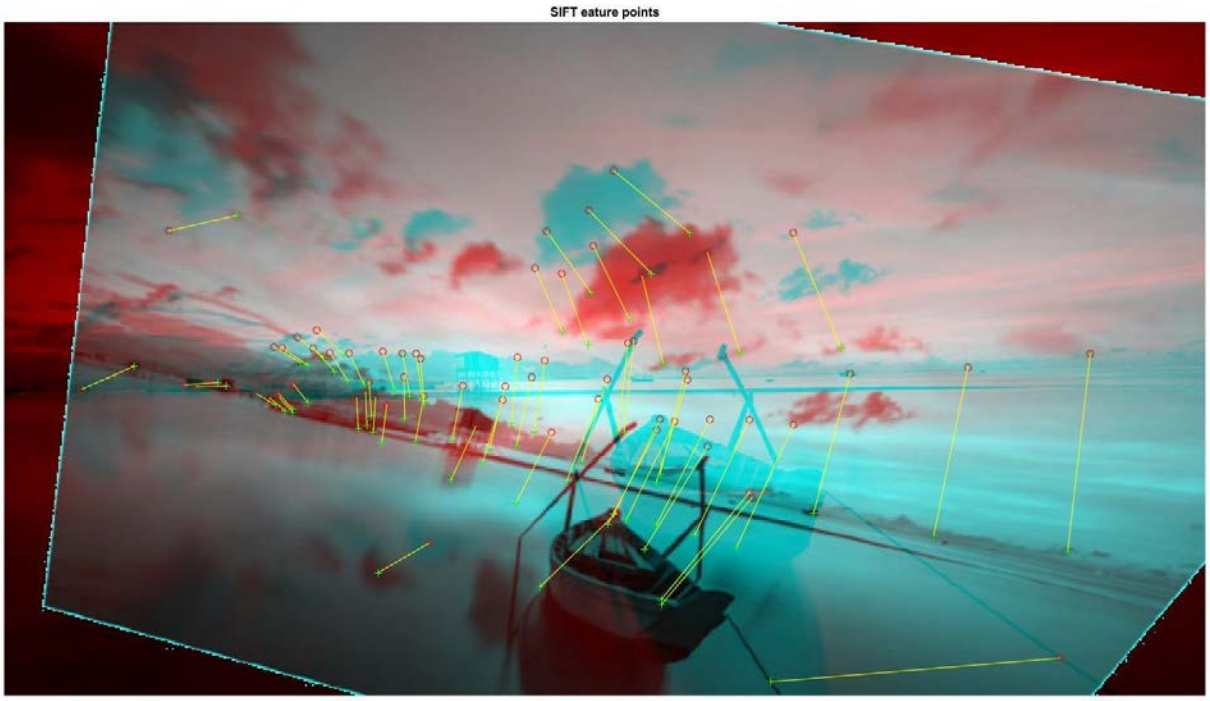


Figure 4.The image pair presented in Figure 3 was analyzed and warped as instructed in chapter 0 using the SIFT feature extraction method and warped the moving image to match the fixed one.

Figure 5. After finding the feature points through SIFT, the projection transformation matrix is estimated based on those points. The moving image is warped and fused with the fixed one to make results easier to the eye.

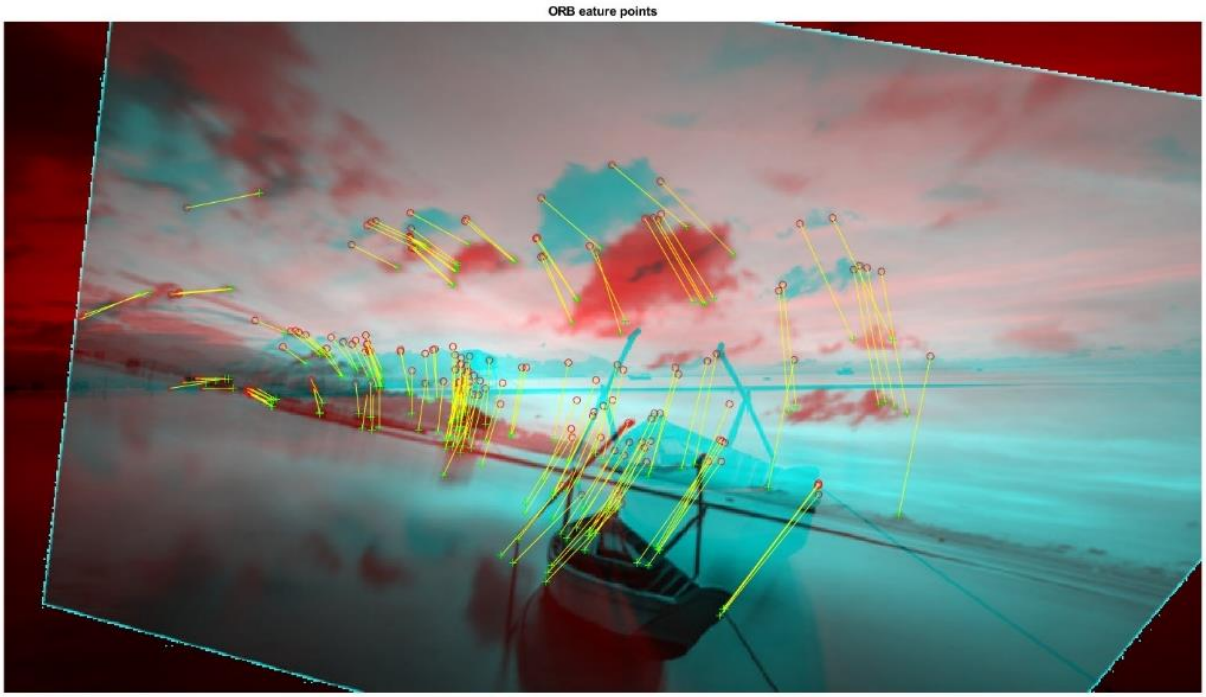


Figure 6. Similarly, as before in Figure 4, features are extracted, only this time ORB algorithm is utilized instead of SIFT. It is noticeable that ORB produced many more features compared to SIFT.

Figure 7. Unlike the warping conducted using the SIFT features (see Figure 5), the match produced from the ORB extracted features is notably improved.

Applications of the above-mentioned methodologies and algorithms can be found across a wide range of computing devices. They can be used for real time image registration applications such as military automatic target recognition, satellite image analysis, object detection in robotics etc. It can also be used for image stitching on phones. Similarly, it is used for medical imaging registration through many different image types (CT-scans, MRIs, ophthalmological images etc.).

1.2. Deep learning in computer vision

Since the discovery of (Artificial) Neural Networks (NNs or ANNs), their applications have been exponentially increasing within the past decade. When it comes to computer vision and the analysis of digital images, Convolutional NNs (CNNs) [10] are among the state-of-the-art techniques for fast, lightweight and robust analysis of multiple images and videos, in real time. The complexity of image analysis tasks lead to the increment of the number of layers in NN, leading to the concept of Deep NNs (DNNs).

DNNs are multi-layer NNs, often with complex structures, consisting of both convolutional layers and fully connected layers (as well as auxiliary input points and other methods that provide convergence to the network). These networks have the ability to decipher and "memorize" generic pattern-like features within images, giving them robustness in image variations and outliers. It is only natural that a tool with such potential is widely used on Image Registration. Medical imaging has also increased in quality and quantity in the past years, which lead to numerous studies exploiting medical image datasets for image registration, using DNN methods.

1.3. The objectives of this thesis

Image registration is a very useful tool both in everyday life applications and as a more serious scientific tool. In this thesis, we investigate the use of deep learning Neural Networks in image registration model problem. We experimented with spatial image matching and alignment of retinal images. Using a trained DNN, we aim to provide robust results on very low computational cost. The potential of expanding this method on other medical image types as well, is also highlighted in this study.

This thesis consists of 6 main sections. Each section can be found on the table of contents and each one presents a unique part of this study's inspiration, methodology and ideas. The main chapters are:

1. *Introduction*: where the reader is introduced to the idea of image registration and the applications behind it.

2. *Literature review*: where similar-related work to this one is presented, highlighting this study's inspiration and providing alternative approaches with similar tools.

3. *Materials and methods*: presenting the software, methodology, algorithms and data used both to experiment and extract our results.

4. *Experiments and results*: showing the different applications and approaches on the previously presented methodology and the numerical accuracy of those approaches.

5. *Discussion, future work and Conclusion*: in this section the results are further discussed, proposing improvements and possibilities about the methodology presented on this thesis.

2. Literature review

Examining the Deep Learning methodologies that exist today, we can find several of applications on Image Registration. Below are some related experimental works on image registration through the use of DNNs, that share basic concept and structure methodologies with our method as well as old and new algorithms from classic computer vision.

This methodology for **Homography Estimation** [11] heavily inspired the work of the current project, using Deep Image Affine instead of Homography Estimation. The large image dataset MS-COCO [12] is used to generate data for the estimation of homographies. The data generation methodology they use, is to calculate a projective transformation for each image, based on four randomly selected points. These points are sampled in a predefined range around the four corners of a randomly selected square patch from the original image. The calculated homography matrix was inverted and applied on the original image and a patch is taken from the transformed on the same position as the first one. These two patches are stacked and fed to the Convolutional Network they designed. The models they chose are two VGG-like networks, which differ only on the last, output layer. The output would either be a direct 8-coordinate displacement, for the four points of the two patches, using L2 loss (Regression Network), or by quantization of each of the 8 numbers into 21 bins, using cross-entropy loss (Classification Network). The optimizer they used was mini-batch Stochastic Gradient Decent (SGD). Their results seem promising as they provide a slight improvement in the case of the Regression Network on the Mean Average Corner Error compared to the traditional computer vision techniques such as ORB[3]+RANSAC[4] with 11.7 and their network achieving 9.2. The Classification Network did worse than the ORB+RANSAC with an error value of 24.1.

Another related approach [13] to this thesis, is with an end-to-end unsupervised **deformable image registration**, using a fully convolutional neural network. In that work they create a tool named DIRNet, which consists of a fully convolutional neural network, a spatial transformer and a resampler. It takes two stacked images as input, which are sampled patches from the dataset's images. One of the two images in the pair is defined as the fixed and the other one as the moving. The network returns a grid of deformation parameters to the spatial transformer which uses a cubic B-spline approach to generate a displacement vector field (DVF). Subsequently, the resampler takes the DVF and warps the moving image patch into the fixed one. The loss is provided by direct comparison of the warped image with the fixed image using Normalized Cross Correlation (NCC) function, though it is mentioned that any similarity method used in conventional image registration would perform well. The optimizer used was mini-batch Stochastic Gradient Decent (SGD). DIRNet was used on two datasets, the MNIST dataset, which contains handwritten number characters and the Sunnybrook Cardiac Data (SCD) dataset that contains multiple MRI scans, consisting of multiple cardiac slices, each of which present the specified cardiac part on 20 different points in time. On the MNIST dataset, DIRNet was used separately for each of the 10 digit characters, thus performing 10 different trainings on separate DIRNets. The results showed an average of about -0.9 NCC error, with the best results shown on numbers 1 and 0 with about -0.95 and -0.92 NCC error respectively. For the SCD dataset, seven different approaches were taken, each utilizing the base DIRNet, with minor tweaks with in it on each case. Results compare the seven DIRNets with a conventional image registration tool SimpleElastix [14], on three different error metrics, Dice, 95th percentiles of the surface distance (95th SD) and mean absolute surface distance (MAD). On average the DIRNets yielded results on about the same level of error as the conventional

tool SimpleElastix, being slightly outperformed by it. For comparison, the best results were with DIRNet version C1, with MAD error: 1.83± 0.89 and the SimpleElastix error being: 1.91±0.94. In every other version, DIRNet was slightly outperformed.

A major component of this thesis was the structure of the network, based on the basic concept of a **"Localization Network"**, which is part of a Spatial Transformer Network. As mentioned in [15], STNs contain three basic parts. The *localization network*, which consists of convolutional, as well as fully connected layers and plays major role in finding the proper spatial transformation, given two images with one being "moved" with respect to the other. The second part of the STN is the **"Parameterized Sampling Grid"**, which is a function called also called "grid generator". It takes as input a normal grid and applies on it the previously calculated transformation as seen on Figure 8.

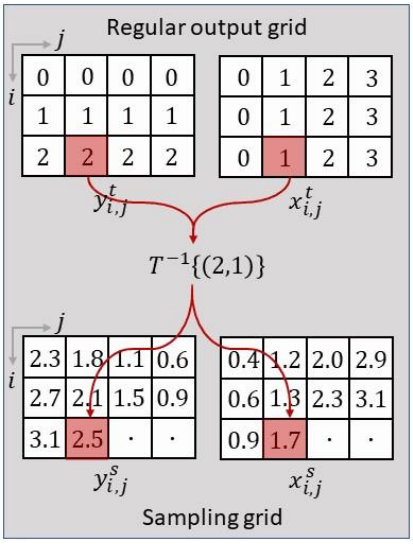


Figure 8. Grid generator. Image taken from [16]

After the grid is transformed, it is then passed on to the next and final part of the STN, which is the **"Differentiable Image Sampling"**. In that phase, the generated grid is applied to the initial moved image, transforming it accordingly, based on the prediction made by the localization network (in the first step). This structure can be used as many times as needed with in any kind of NN and can provide better results regarding image classification, by reducing the "noise" inside the data, produced by image distortions such as spatial transformations as well as elastic distortions. It is also worth noting that by applying these transformations on the data, the background noise is also reduced since the STN is paying attention on the areas of interest.

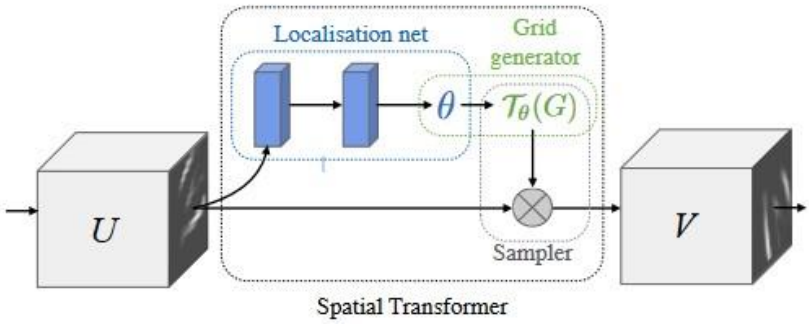


Figure 9. Basic structure of an STN. Image taken from [15]

A recent publication called **STN-Homography**: Direct Estimation of Homography Parameters for Image Pairs [17] was made few years after the previously mentioned [11] and tried to improve its performance with some more complex methodologies. Their method is also very similar to our approach, but its performance was far superior to ours. They used the MS-COCO [12] dataset like in [11], and they sampled 3 grayscaled 128×128 patches from each image, to which a random homography was applied. The homography was made with random dislocations on the four corners of the patches, with a max dislocation distance of 32 pixels (which is not included in the 128×128 patch size). The initial non-transformed version is considered the *"moving"* and the randomly transformed version is the *"fixed"* version. The two image patches are fed to the network as input. The homography generated using the random dislocations and a *"transformed"* version of the moving image, which is generated by applying the homography are used as ground truth data for the error functions of the NN. The NN model they used was similar to that of [11], containing 8 initial convolutional layers which gradually increase in number of features and decrease on the size of them. The homography matrix was normalized, with its 9^{th} element (H_{33}) being always equal to 1, providing 8 degrees of freedom (DoF). After the convolutions there is a 1×1024 sized fully connected layer, which then passes to a 1×8 fully connected layer, used as the output layer for the 8 normalized elements of the homography. As mentioned in the name of the research, the network was developed as an STN, hence this makes the previously mentioned architecture the *regression network* part of the STN network. The 8 normalized values of the homography are then passed to the g*rid generator* which recreates a version of the predicted transformation on a regular grid, with normalized coordinate values. This grid is passed to the sampler, which is applied to the initial moving patch, after being transformed back into the nonnormalized coordinates. The final output is a transformed version of the moving patch, made through the predicted homography of the regression network. They use two loss functions to update the network, one is the L2 loss between the predicted homography elements outputted from the regression network and the L1 loss for the pixelwise photometric loss between the final, transformed patch and the previously mentioned ground truth transformed patch that they created in the beginning. The L2 loss is the sum of all the squared differences between the ground truth value and the predicted value and the L1 loss is the average absolute differences between predicted and the target values. In Figure 10 we can see this model used for a sequential three-stage architecture.

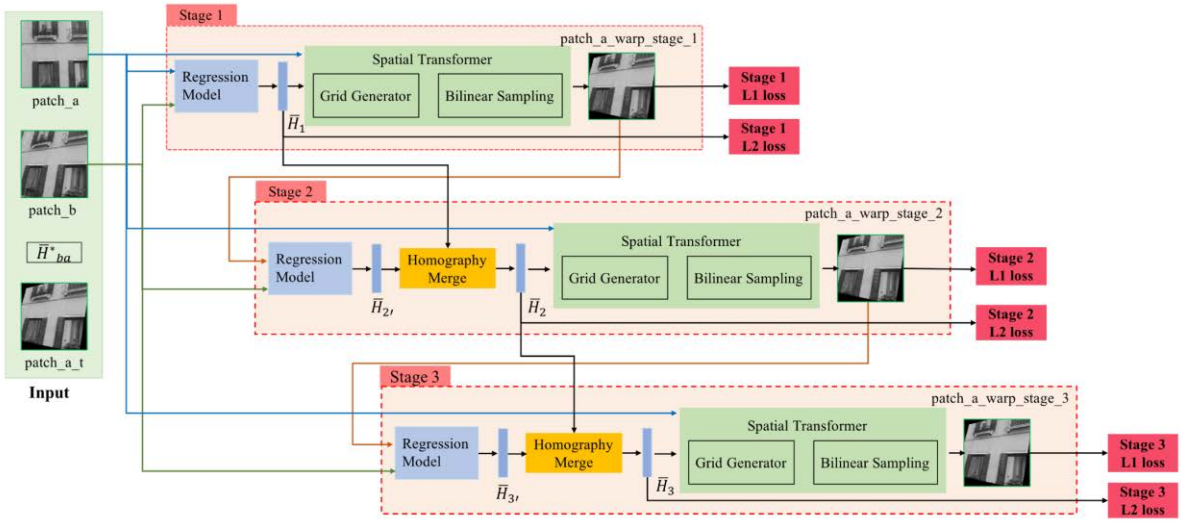


Figure 10. The sequential architecture used with the created model. Image was taken from [17].

Although the model they created would perform satisfactorily, they extended it by a hierarchical architecture and a sequential architecture, using the same model-structure on subsequent stages. The hierarchical architecture is similar on its basic ideology as the sequential, but their main differences are the fact that the second one is fully (end-to-end) differentiable, whereas the first one is not and has to be trained separately for each stage of it (e.g. 3 model trainings for a 3-stage hierarchical architecture). On the hierarchical case, models of the same structure are used in different stages, each stage trained on a separate session, using the previously trained stages as its input "providers". For example, in a 3-stage hierarchical architecture the 1st stage model is trained on its own, the 2nd stage model is trained together with the 1st one, taking those outputs as its inputs (the 1st stage *transformed moving* patch and the *fixed* patch). The 3rd stage model will be trained together with the already trained 1st and 2nd stage models behind it, again used as its input providers. Each stage returns a homography matrix which is multiplied with the ones predicted in the previous stages to get the final, combined homography. This method works well as every stage improves the previous stage's results. This method has a drawback though, it is not a single, end-to-end trainable model, rather it is *n* models trained in *n* subsequent sessions. To make this method more effective, they created the *sequence architecture* (shown in Figure 10) which is fully differentiable, since each stage's output is immediately passed on the next stage's model as input. They also created a custom, differentiable, *"Homography merge"* layer, which multiplies each stage's predicted homography with the previous ones. This methodology made up for a single, "big training", and even provided better results than the hierarchical one. The mean corner error of their approaches showed results that are equal or better than similar NN approaches such as [11] and far superior to classic computer vision methodology such as ORB[3]+RANSAC[4].

An interesting approach on image registration is that of [18], where the authors extract points of interest only from one of the images, through classic computer vision, using intensity variations. The corresponding points on the other image are found through a fairly new type of algorithm called **AIS** (Artificial Immune Systems) [19]. AIS performs multiple local feature-searching simultaneously. These local features are sampled in such a manner that combine local and global coverage when it comes to points of interest. The brief explanation of AIS is as follows:

- Beginning from the edges of the image, multiple points are selected, these points have a feature vector and a location.
- Then these points are "expanding" by selecting other points within a determined region around them.
- From these newly determined points, using their feature vectors, bad points are dumped based on a given distance error threshold.
- In the end, the algorithm stops when either all the feature points are found or when the given number of iterations is met.

One of the latest, proposed algorithms for Homography estimation between images is called ***H*-RANSAC** [8]. This algorithm utilizes the generic methodology of outlier elimination and Homography estimation, as simple RANSAC does. What makes this algorithm unique, it is that it accepts as input two sets of points from two images respectively, without the need for local feature vectors, or possible point pair formations, prior to the algorithm. It is specifically designed and intended for planar fields, best example of such being a football field. The data they used was images of football players inside the same field, taken from different perspectives. The points they extracted from the football fields were taken through a Yolo arcg NN [20], pretrained to locate football players. This NN is widely used as a multi-purpose computer vision tool, providing accurate and easy results. They propose a method that uses their own custom loss function as well as feature points between the two images that are close to a plane, in this case the football field. From there they match the images by targeting these specific points to be close to the plane, so they are targeting the player's feet. Their results seem to outperform, by a lot, many state-of-the-art algorithms for Homography estimation, such as USAC [5], VSAC [6], MAGSAC++ [9], combined with other state of the art feature extracting algorithms such as SIFT and ORB.

As mentioned in [5], **USAC** is a Universal RANSAC approach, taking into consideration many specifications and calculations added to the basic algorithm, that improve the final outcome and extend its limitations. More specifically, it consists of 4 basic steps plus an initial, not necessary, "prefiltering" step on the input data. This prefiltering step applies different algorithms on the input data such as SCRAMSAC [21] , reducing the number of matches, hence increasing the speed of the RANSAC algorithm by a lot. The first basic step of USAC is the sampling and sampling check. This step takes into consideration some form of bias when sampling the minimal subsets and then checks whether the sample is suitable for the purpose of calculating the model (before calculating the model parameters). The second step consists of the model generation and check. In this step the desired model is generated, but before it is confirmed on every other data point, a basic check is applied to determine whether the model is applicable on the base subject of interest (for example in a problem of scale, something that's bigger scale than the desired will be dumped), thus preventing the costly, unnecessary checks of models with multiple points. Next up, on the third step, model verification and degeneracy check are suggested by different means and methods that are designed to. The model verification suggests different types of statistical tests to be performed on a small amount of data, using the model and the results of the test determine whether the model will be kept or dumped. For the degeneracy, algorithms for data degeneracy, such as DEGENSAC [22] prevention are suggested, to help the algorithm find the model's unique solution. On the fourth and last step, it proposes existing methods to help with model refinement, reducing the noise included in the final predicted model and the also reducing the time needed to find it. Overall USAC is a RANSAC variation that increases its accuracy and reduces the time needed to

predict the desired model, by applying filtering algorithms on each step of RANSAC, getting rid of outliers and dumping unnecessary calculations.

Another interesting variant of the RANSAC algorithm is **MAGSAC++** [9] which is an improved version of the previous MAGSAC algorithm [7]. In MAGSAC++ they implement a new way to overcome the noise inside the given set of data. They propose an Iteratively Reweighted Least Squares (IRLS) instead of the multiple Least Squares fittings used in the original MAGSAC. Finally, they propose a different way of sampling points where it starts the sampling locally setting as a high likely hood the local points all being inliers and as it the algorithm tries to confirm whether these points are good to fit the model, the points are gradually sampled more and more in a random way like classic RANSAC resulting in a more global sampling. These improvements make MAGSAC++ outperform previous versions and similar algorithms in terms of accuracy, fail rate and speed.

A more recent version of RANSAC is **VSAC** [6]. VSAC is faster and more time efficient method than the previously mentioned versions using some technical improvements. It can also provide exceptional results on image pairs that do not match. The way they implement that is by applying a concept of *"independent random inliers"* which, as they mention, when dependent random inliers are not counted, the support score provided by RANSAC for the random models will show a distribution very close to that of Poisson. Taking that into account, they can return a second score for the random model, which shows how likely that model was predicted by chance (so basically, the confidence of the predicted model). This prevents the algorithm from reaching into false conclusions, hence improving greatly its accuracy. In USAC, DEGENSAC [22] algorithm was used to help with degenerate data. In this paper, VSAC introduced an improved version of this algorithm, DEGENSAC^{+}, which addresses some issues of the original algorithm that may occur in certain cases they mention in their study (such as false positives for fundamental matrices, predicted as degenerate). A calibration is also proposed to address more homography cases of the basic algorithm where it fails to produce a correct prediction and consumes more time until it is able to predict its fault. To further improve the speed of the algorithm, they propose a combination of the often time costly, local optimization and a threshold test that is looking, not for the best local model, but for a model that is good enough, since later on it can be further optimized, without the need of spending more time on local optimization. The process is made even faster by applying the Sequential Probability Ration Test (SPRT) [23] which checks whether a predicted model is likely to be better than the already *best-one-yet*, if it's not, the process is interrupted, saving time. Lastly, the algorithm also implements the Lindstrom [24] algorithm, to further improve the quality of its results. As the authors mention, this is one of the most effective and cost-efficient algorithms so far, providing test results with same accuracy as the current state-of-the-art RANSAC algorithms, with less errors and with a lower cost.

3. Materials and methods

The method we propose consists of the preprocessing of the data and the main training session of our network. Starting from our network, it is a supervised Deep Neural Network (DNN), its structure consists of $2 \times 256 \times 256$ input size for two stacked, grayscaled image patches and has an output of 3 pairs of numbers, each corresponding to the three basic transformations of translation, scale and rotation. Combining these three pairs in a 2×3 matrix we get the *local* affine matrix of each patch pair. During the data preprocessing, image pairs of Fixed (F) and Moving (M) images are patch sampled and their (global) affine transformation matrices are calculated, through given ground truth points between the images. The image patches are stacked and passed to the input of the DNN. With the top left corner (the one that's closest to the (0,0) point), we calculate the local affine transformation matrix between the M patch and its corresponding warped version on the F. These local affine matrices are the DNN's loss feedback (its targets).

3.1. Analysis of the utilized software

Some of the processing and preparation of the dataset was performed through the use of MATLAB R2023b. The function that played a very crucial role in this thesis was "fitgeotform2d". By utilizing it, we generated the desired transformation matrices, based on the two sets of ground truth points given by the dataset. The rest of the MATLAB functions that were used were mostly for graph and image recreations, both during the thesis process and after the final results were concluded. These were to help better visualize and conceptually understand the process of the work.

The code for both the network's dataset preparation and the network's training, was developed on Python 3.10.12, in PyCharm's 2023.2 code development environment. PyCharm is a useful tool, provided by JetBrains, that provides visual and operational assistance for easy-life code development. The main libraries used were PyTorch v2.0.1, NumPy v1.24.3, scikit-learn v1.2.0 and OpenCV2 v4.6.0. The GPU was also utilized for the training process through CUDA toolkit v12.2.128 library. PyTorch handles the training process and provides the network's building blocks. PyTorch also has the option of utilizing the GPU's multiple cores, for parallel processing of the data, through the CUDA toolkit, during the network's training. NumPy was used to generate NumPy arrays to pass the data that will be handled during the processes. It provides a lot of flexibility and a wide variety of tools to handle almost every kind of array-like object operation. OpenCV and scikit-learn were used for minor data handling such as: image editing (image reading, resizing, grayscale coloring) and food dataset splitting, respectively. The packages were handled in a virtual environment created by the Anaconda software v23.5.2

3.2. Structure of the neural network model

Multiple architecture models were made in this thesis, two of which are described:

- A model based on [11], following the same ideology on its structure, being a deep neural network.
- A custom-made model we created based on the initial structuring layers of STNs, containing multiple hidden convolutional layers, as well as fully connected layers. This was a more complex approach and it will be the main NN of this thesis.

Both models were built in python programming language, using the library named torch, referring to PyTorch.

3.2.1. Homography network based architecture

The "homography network" model was built based on [11], which as mentioned before heavily inspired us for the creation of both this first DNN and the upcoming, more complex networks. The input channels of this DNN are 2, meaning two images, stacked on top of each other will be "fed" to the network's input. The chosen size for the images is 128×128 pixels. The network consists of 8 2D convolutional layers and 2 fully connected ones. Every two layers, max pooling method is being used. On every layer, the chosen activation function is ReLU, together with applying 2D batch normalization right after it. Between the 8th convolutional layer and the 1st fully connected layer, the data changes form, by applying the reshape function on it. This results into transforming it from tensors of shape $16 \times 16 \times 128$ to a vector of 1×1024. The fully connected layers use linear calculations on the data. Between the 1st and the 2nd fully connected layers, data size is maintained. Taking it from there, since the data is on the last layer of the network, it must be reshaped again into the form that is desired in order to calculate the loss. The data is linearly condensed from a vector of shape 1×1024, into a 1×8 vector. The 1×8 vector is the neural network's output, which would later be used to calculate its loss comparing it to a ground truth target provided by the dataset. These 8 points would be the X and Y translation of the four corners between the M image's patches and the F's.

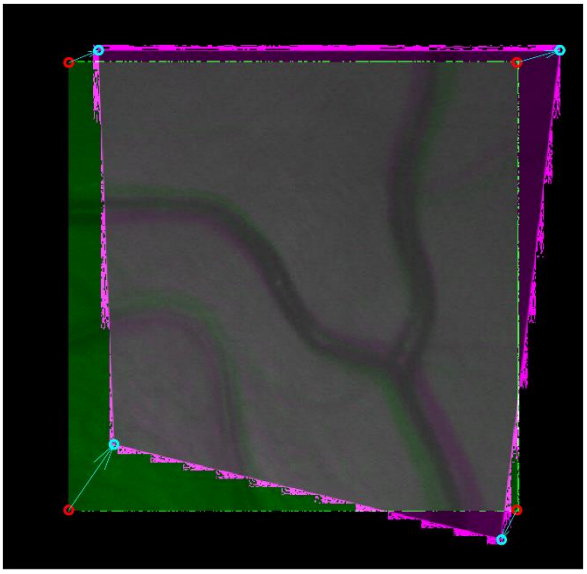


Figure 11. The 4 corner displacements of a patch from image P29_1. Random 3D transformation was applied; hence it can be reversed back only with Homography. Min displacements required for transformation matrix: 4.

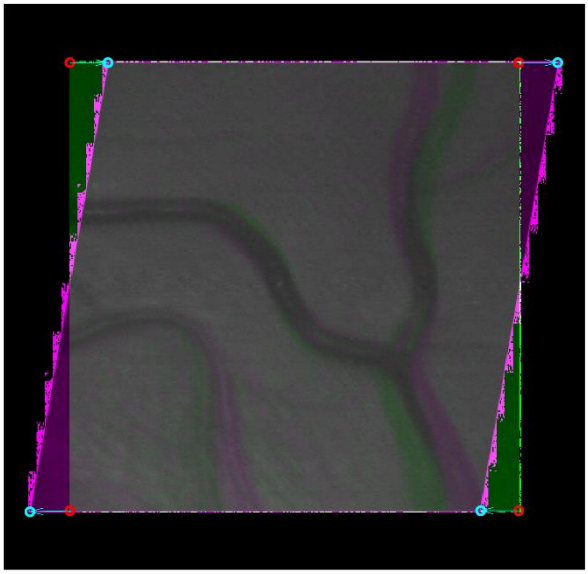


Figure 12. The 4 corner displacements of a patch from image P29_1. Patch was randomly sheared in 2D, meaning it can be reversed both with affine and Homography. Min. displacements required for transformation matrix: 3.

The loss was defined as the MSE of the predicted x and y translations and the ground truth translations of the four corners, that were precalculated and saved in the respective dataset that was prepared for this DNN. This model is a simple, deep, convolutional neural network. Since it has no special structure and because its results were not good as well, we chose to not analyze it further and just briefly mention it.

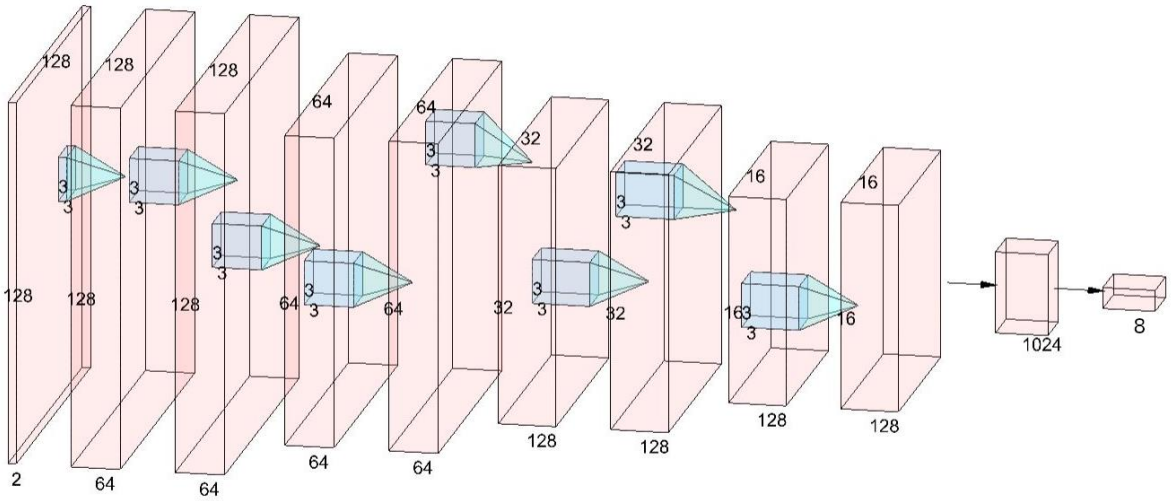


Figure 13. Deep Image Homography-based Neural Network Structure.

3.2.2. Parallel-output network architecture

We initially started exploring the possibilities of STNs, building different architectures as the NN model and trying different values on the hyper-parameters. A basic architecture we tested as an STN was a 4 convolutional layer block passing its output to a block containing 2 fully connected layers in a row and a 1×6 output used as the affine matrix. This output would then be passed to a grid generator function which would create an image grid with the size of the initial patches, based on the predicted transformation that was passed to it. This grid would then be passed to a sampler, which would also take the initial M image and apply the transformed grid on it. The output of the sampler was also the output of the STN and it was a transformed version of the initial M patch, using the predicted affine transformation passed from the final fully connected layer to the grid generator. The loss function used for this network would be the MSE between the pixels of the F and the transformed M. This network would perform poorly overall as the loss function would not provide to the STN transformation-specific information, hence the STN's weights would not be updated accurately during the backward pass.

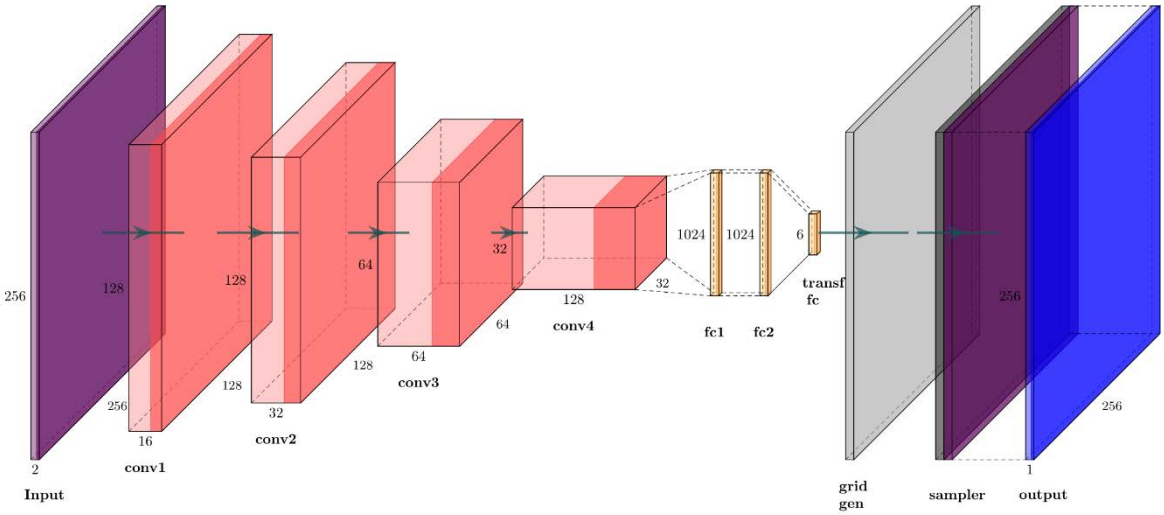


Figure 14. Structure of one of the STN-based approaches we used as a NN model during experimentation. The output is a full-size transformed version of M.

After trying the basic STN, we decided to discard the grid generator and the sampler all-together so that we could focus more on the transformation itself. We decided to create a NN with 3 parallel outputs. The structure of the parallel-output model is built using the idea of producing an affine matrix. It consists of a convolutional block which decrypts and learns the patterns and features inside the images and 3 parallel blocks, consisting of equal numbers of fully connected layers (per block), plus the output layer.

The convolutional block performs gradual size reduction of the feature's size as well as increase the number of features. Specifically, the feature size ("F size") and their number (# of F) are defined on some of the basic models we tried as such:

Table 3. The input is always 256 with 2 feature images. The number of features and the size of them are gradually decreasing within 3 convolutional layers on the small 3-layer NN model we made.

	Input layer	1st CL	2nd CL	3rd CL
F size	256	128	64	32
# of F	2	64	128	256

Table 4. The gradual increase and decrease on the number of features and the feature size respectively on the 7-layer pooling-unpooling NN model we made.

	Input layer	1st CL	2nd CL	3rd CL	4th CL	5th CL	6th CL	7th CL
F size	256	128	128	64	64	32	16	16
# of F	2	16	16	32	32	64	128	128

Looking at the tables above, we can see that the output on Table 3 will be of size $256 \times 32 \times 32 = 262{,}144$ and on Table 4, it will be $128 \times 16 \times 16 = 32{,}768$ which of course means that's the number of features the first linear layers will be taking from the output of the convolution block.

The extracted features are passed into each one of the three linear blocks containing the fully connected layers. From there, the features are processed specifically for each one of the basic transformations in an affine matrix: translation, scaling and rotation. For each one of the basic transformations, 2 numbers are extracted from each block, one for the x and one for the y coordinate. These 6 numbers combined give us the affine transformation, which when applied to the moved image, it should match it spatially with the fixed one.

To help generalize the learned features and prevent the network from overfitting, between the convolutional layers are introduced pooling-unpooling layers, as well as dropout layers between the fully connected layers. The activation function used was ReLU, as well as a 2D batch normalization after every convolution.

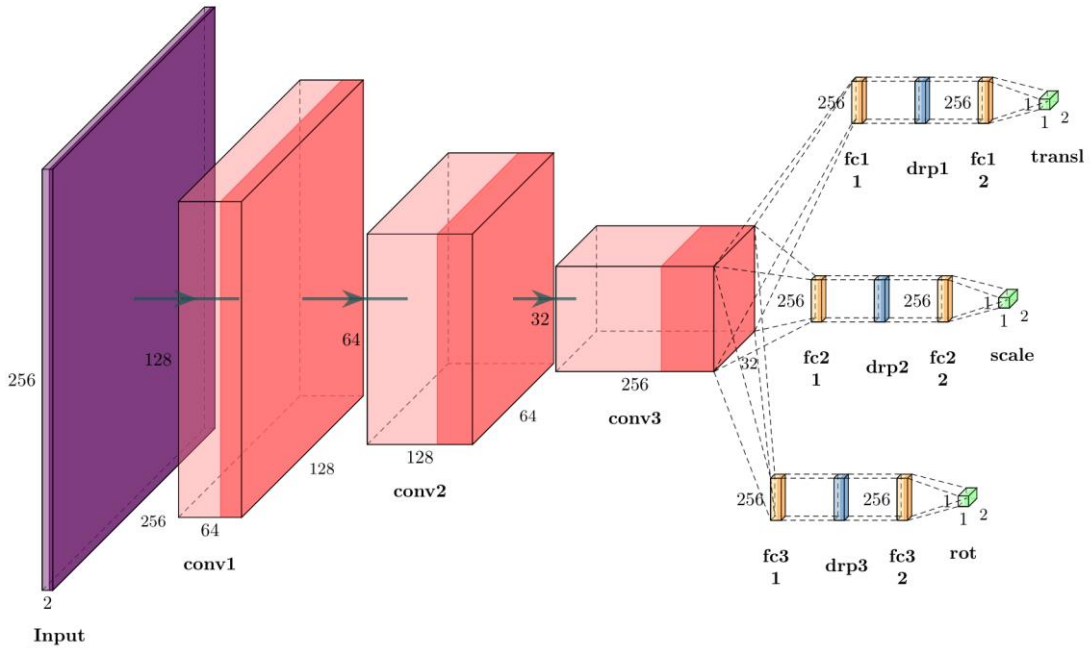


Figure 15. Structure of the Parallel-output NN with 3 convolutional layers and 2 fully connected layers. There are dropout disruptive layers in the fully connected blocks.

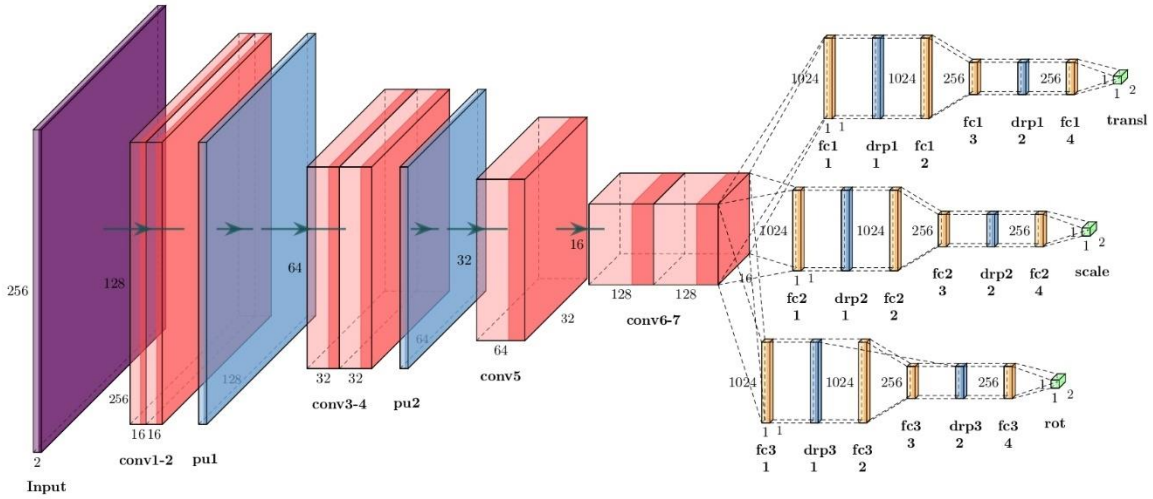


Figure 16. Structure of the parallel-output CNN with 7 convolutional layers and 4 fully connected layers. There are Disruptive layers such as pooling-unpooling and dropout layers.

3.3. Neural network model training process

For the overall training of the network, the code was again constructed in a python developing environment (IDE). The main libraries used for that purpose were PyTorch, NumPy, scikit-learn, copy and the python code file containing the neural network's model, which will be used in the training process. A batch size and a number of total epochs are given in the beginning as general variables to train the network on the previously prepared and saved network dataset. If available the CUDA toolkit library will be utilized, resulting in training time being drastically reduced. To use that option, a variable "Device" is created, pointing to the GPU, if it is available, or the CPU otherwise. The training code consists of the three functions that perform the data unpacking, training, results back-up and the initializations of hyper-parameters, iterators and lastly data splitting.

3.3.1. Initializations

First, the libraries that will be used are imported. The global variables containing hyper-parameters and paths are initialized. Hyper-parameters such as the batch size, the number of training epochs are essential, whereas others such as learning rate and momentum may or may not be needed, depending on the optimizer that will be used. In this case we are using Adam as the optimizer. Adam turned out to be easier to handle the training process through experimenting, as the SGD optimizer would get extremely high or extremely low values during training, depending on the chosen values of the hyperparameters, which made it difficult to use. The network model was already predefined in another file, which we already imported and we load into the memory and then directly pass it to the GPU (if available).

3.3.2. Loading and unpacking of the input dataset

Initially the input's path is passed to the "datasetloader" function where it will be used to retrieve all of the file paths. The list variable containing these path names is "filename". The list is then sorted alphabetically and passed through a loop to the custom-made class "Loader".

The class takes the filenames one by one, returning the contents of each file to two variables, one containing the image patches and the other one containing the ground truth targets. These are appended to new lists until all the food files are loaded. From there the lists containing the food data, are returned to the out-of-scope variables "Images" and "Targets", in the form of floating points.

3.3.3. Dataloaders and iterators

A function named data loader handles the loading of the dataset files. It retrieves each file's path and then proceeds to load it into the memory. Since there are two kinds of files, one with the patch pairs and one with the target affine matrix, both are loaded separately into different lists, which are then returned to the data loader function. From there the data is shuffled manually and then divided into $70-30\%$ splits for training and validation-testing. The 30% used for validation-testing is again split, between images, into $70-30\%$ giving us a rough $70-20-10\%$ for the train-validation-testing phases respectively. The image splitting is done indirectly through the use of the function *"train_test_split"* taken from the library *Scikit-Learn*. The shuffling is done manually using another function from the *Scikit-Learn* library called *"shuffle"*. We performed the data splitting this way because we wanted to divide the dataset between the image samples and not the patch samples, so that the training and validation phases would be 100% unbiased. After the splitting and shuffling, the four lists containing training images and targets, as well as validation images and targets, are passed to PyTorch's data loader function which will return each individual data loader variable that will later be used on the network's training and validation. In these data loaders, variables such as batch sizes are passed to determine how you want the dataset to be handled later.

3.3.4. Training and validation phase

The optimizer and loss function(s) used are defined and passed the required hyper-parameters. The most commonly used optimizers are SGD and Adam, with SGD being manually handled on its learning rate as well as the momentum, whereas the Adam algorithm is adaptively changing its parameters to conclude the fastest possible convergence. In this case we chose to use Adam as our optimizer, as SGD appeared to be more sensitive with respect to its parameters, resulting in big changes in the outputs, which were not optimal for the process. For the loss function, we chose to use the classic MSE (squared L2 norm distance) between the predicted transformation numbers from the network and the precalculated ground truth from the dataset we prepared beforehand. MSE, standing for: mean squared error, is the mean of the squared distances between *n* pairs of numbers: $\frac{1}{n}\sum_{i=1}^{n}(X-X')^2$ .Since the loss was calculated with in the relative ranges of each transformation type, the fully connected blocks of the network were also being updated accordingly to the same scale (for example the range of the rotation transformation is $0-1$ whereas the translation transformation starts from 0 and could get up to few hundreds of pixels in this case). It is also for that reason that we used L2 instead of a normalized version of each loss, as will be mentioned later.

To define the network's architecture, we pass the function "NetworkModel" to a variable named "model". That function was developed on another python file named "Network". After defining the model, a summary of its architecture is printed.

In order to speed up the process of training and validation, we pass to the GPU every variable and data we will be using during those phases. That includes the loaders and the model right after being defined.

The "trainer" function handles the training process and takes as parameters: the model, the loss function, the optimizer, as well as the four data loaders for the training-validation phases with the stacked patches and the target affine matrices.

3.4. Dataset

The dataset that is used is called "Fundus Image Registration" or "FIRE" [25] for short. It consists of 134 pairs of retinal images, where both images on each pair showcase the same sample of retina, spatially transformed. This dataset was chosen because it provided high resolution retinal images as well as feature points. Those points significantly reduced the preparation of the dataset for the training phase. On every pair, one of the two images is considered the "fixed" one, whereas the other one is the (spatially) "moved", with respect to the first. All images are of size 2912×2912 pixels and they are RGB colored. The feature points consist of ten coordinate pairs, for each pair of retinal images. Each pair contains the (x, y) coordinates, of specific points in the plane of the images as seen on Figure 17. Each ten points in one image correspond to ten points in the other image (common), which are considered moved with respect to the originally fixed image. These points constitute the ground truth between the two images and can be considered as the reference points for generating the inverse transformation matrix of the moved image to the stationary (or fixed) one, based on standard computer vision algorithms.

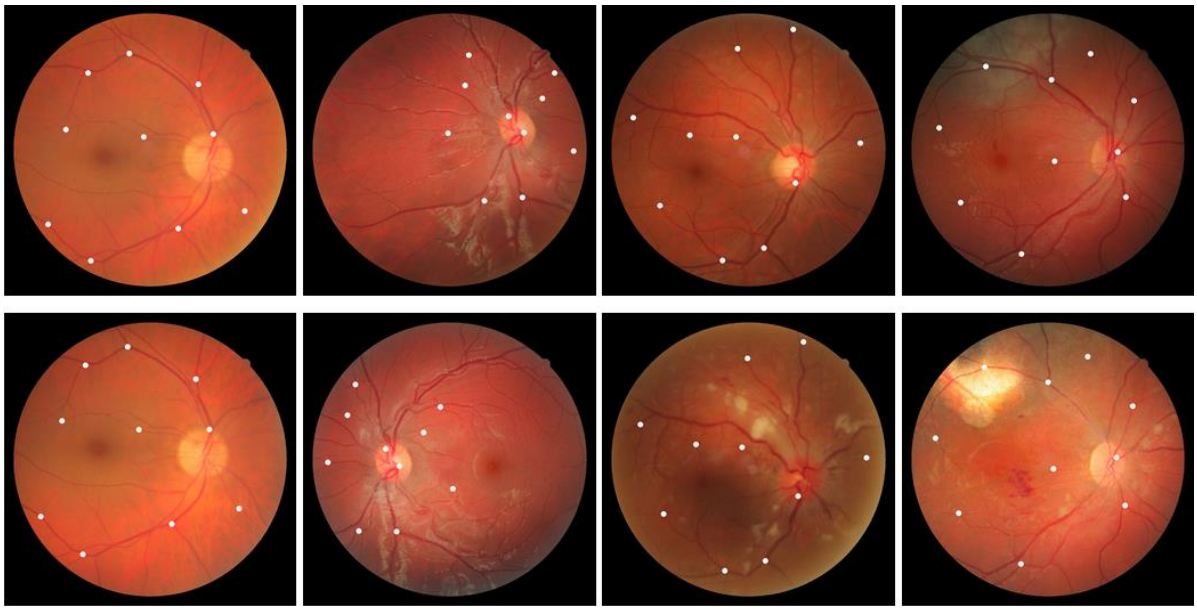


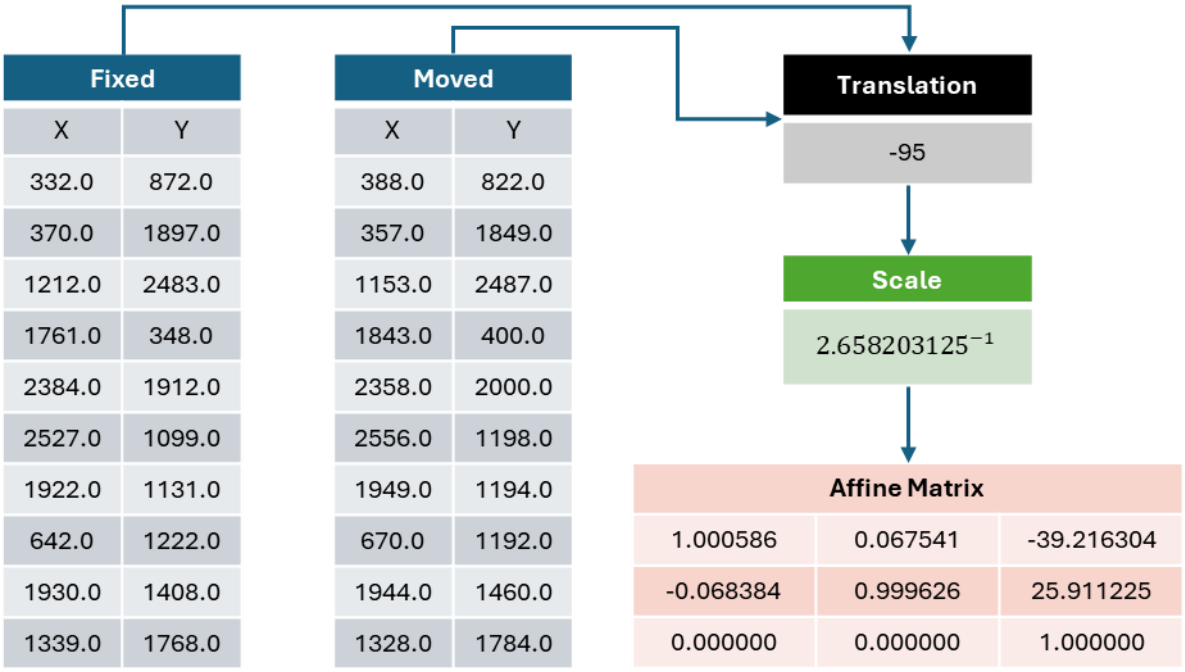
Figure 17. Four FIRE dataset samples. Each column shows a pair. The white spots represent the ten ground truth points. The first column's pair is from the S category, the second column contains a pair from the P category and the third and fourth columns contain pairs from the A category.

3.5. Ground truth exploitation

Initially, we have to calculate the transformation matrices of the moved (M) images to the fixed (F) ones. This is done using the MATLAB programming environment, exploiting the given reference points between each image pair. The methodology followed involves first examining the ".txt" files, which contain the coordinate values of the reference points. This is done with three different loops, one for each type of file name (A, P, S). Each loop opens the files one by one, reading and transferring the tokens contained in them, to variables. The separation of the points between M and F is done as the files are read.

At this point it is recalled that we are working with image data of size 2912×2912, which is a problem due to the high computational demand of the algorithms for processing and storing them. The simplest way to deal with this is to reduce the size of the images, by cropping out as much as possible from the black frame around each one of them and reducing their resolution. We chose to crop 95 pixels from each side of the image and then scale down the image from the new resolution of 2722×2722 to 1024×1024. These two image pre-process transformations are translation and scaling respectively, which should also be applied to all of the image feature point data, which was given to us based on the original 2912×2912 resolution. Specifically, we subtract the number of pixels we removed from the starting image (95), with respect to the starting point $(0,0)$ which is the top left corner of the image, so that way it can be considered a "translation" towards that direction. Next, we divide the point numbers with the number that divides 2722 and returns 1024, which is exactly 2.658203125.

The next step, once the reference data has been processed and separated, is to find the global transformation matrix. Applying this matrix on M will give a warped version of it, where the reference points between the two images should match. An example of this process can be seen below on Figure 18. This is implemented using the function *"fitgeotform2d"* which takes as input the reference points of F, the reference points of M and the type of transformation to return as output. The way this function works is similar to the previously mentioned methodology in chapter 0 part ii). By choosing "affine" as the output variable, the function will return a 3×3 matrix which is the transformation matrix that will do the aforementioned transformation on M. Choosing "projective" as the output variable will return a 3x3 matrix which is the projective transformation matrix or Homography of M to F. The differences between the two matrices have been discussed in Chapter 1. Finally, the tables are stored in text format in a file. This process is done for each pair of images and the transformation tables are stored in the same order that the images are stored on the disk (nominally: A→P→S).



Fixed	
X	Y
332.0	872.0
370.0	1897.0
1212.0	2483.0
1761.0	348.0
2384.0	1912.0
2527.0	1099.0
1922.0	1131.0
642.0	1222.0
1930.0	1408.0
1339.0	1768.0

Moved	
X	Y
388.0	822.0
357.0	1849.0
1153.0	2487.0
1843.0	400.0
2358.0	2000.0
2556.0	1198.0
1949.0	1194.0
670.0	1192.0
1944.0	1460.0
1328.0	1784.0

Affine Matrix		
1.000586	0.067541	-39.216304
-0.068384	0.999626	25.911225
0.000000	0.000000	1.000000

Figure 18. Affine Transformation Matrix Calculation for Image Pair "A01". The first two tables contain the resized Ground Truth points for the Fixed and Moved Image respectively. These are used to calculate their Affine Transformation Matrices through the 'fitgeotform2d' MATLAB function.

3.6. Construction of usable data for the network model

The preparation of the dataset that will be used to train the neural network model was done in a python language environment. The data must go through some processing before being given to the network as its input. The images are being processed in pairs, in name order, same as before with the transformation matrices. Just as mentioned before, the images are cropped and scaled down to 1024×1024, greatly reducing the memory required. Next step is turning the images from RGB coloured to grayscale coloured. After the image processing, the code accesses the txt file that was created before, containing the transformation matrices in the same order as the images are being processed (nominally). The retrieved transformation matrix is in the form of a 1×9 array, which will then be edited and transformed into a 3×3 array. The 3rd row of the array is being deleted since the homogeneous coordinates of the 3rd dimension will not be needed, resulting in the final transformation matrix form of 2×3.

Since the size of the dataset is as small as 134 pairs of images, multiple patches of size 256×256 pixels, are randomly sampled from the pre-processed image pairs. The random patches are being selected by generating two random values, which will then be used as the coordinates of the centre point of the moved image patches. It has to be noted that not all image pairs are usable. That happens in cases where the moved image is highly translated in comparison to the fixed one, or there are too many black pixels (from the black frame in the corners). In the first case the randomly selected patch on the moved image, may have very little or no common ground at all with the patch coming from the fixed image. This of course means that the network would never be able to find a suitable transformation for that pair. To fix that we define a threshold and we accept only image pairs that their translations (either on x or y axis) do not exceed it. In the second case, we define another threshold which we use to check the black pixels in the patch, if they exceed it simply resample the patch with another random location

in the image. In this case we used as thresholds 220 translation pixels and 10 black pixels respectively. Figure 19 provides numeric details about this process.

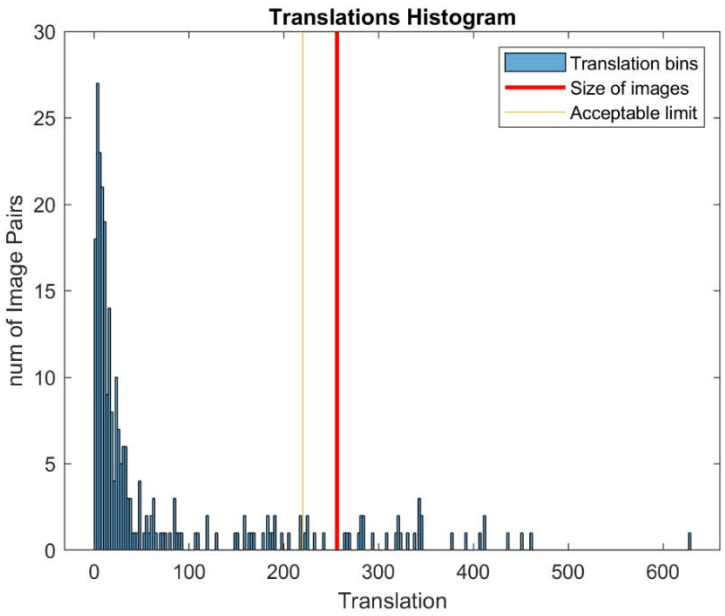


Figure 19. In the image we see a histogram of the x, y translations of all 134 pairs of images from the FIRE dataset. The red vertical line indicates the 256 mark, which is the size of the patches. The yellow vertical line indicates the threshold of acceptance for the translations. Any image pair with translation value passing the yellow line will not be used to train the network. In total, 102 pairs out of the 134 were used and 32 were excluded from the dataset.

The patch's four edge coordinates are generated by adding and subtracting 128 pixels respectively to the centre coordinates. This way if the coordinates of its centre are: (c_x, c_y), the edges (lines) of the patch will be:

$$top = c_x - 128 \qquad bottom = c_x + 128$$
$$left = c_y - 128 \qquad right = c_y + 128$$

Given the edge coordinates, the four corner coordinates of the patch will be:

$$top_L = (top, left) \qquad top_R = (top, right)$$
$$bot_L = (bottom, left) \qquad bot_R = (bottom, right)$$

with the edge names indicating which corner it is. Since the size of the image pairs is 1024×1024 and the size of their patches is 256×256, the minimum and maximum values of the randomly generated patch centre coordinates will be: $0 + 128 = 128$ and $1023 - 128 = 767$ respectively, making sure that way that the patches will not break out of the boundaries of the image. All of the above indicate, as mentioned before, that the beginning of the image axes (point $(0, 0)$) is on the top left corner of the images (that's where the rows and columns of the images start). These four sides will be used to crop the patches from both the fixed and moved images on each pair. The neural network will be taking as input these two 256×256 grayscale image patches, stacked as one sample. On the output the network will return 6 numbers, corresponding to each basic transformation that's contained in the affine transformation matrix (translation, scale, rotation).

It is important to note here that a pixel closer to $(0, 0)$ will get transformed differently compared to a pixel further away. This is because a pixel closer to the beginning of the axes will move less through the rotation of it with in the image, compared to another pixel further away. This issue presents itself when we sample the patches, hence the patch's position spatially is lost and its top left corner is no longer top_L, rather it is now, locally, the start of axes for the patch.

To resolve this issue we consider the image's transformation as "*global*", and the patch's transformation as "*local*". We turn the affine matrix from global to local by applying it to the (global) top_L and then passing (to the affine matrix [1]) the following differences:

$$a_{13} = (top_L - a_{13})$$

$$a_{23} = (top_L - a_{23})$$

After having the local affine we can use that as the network's target. As we will see further bellow, we can easily reverse back to the global affine matrix if we know the local one and the coordinates of the top left position from where the patch was sampled.

The corresponding 4 corner points on the fixed image, will be calculated by applying the retrieved (local) transformation matrix on each one of the moved image's patch edges. Given that the corners of the moved image's patch are the four points: $[top_L, top_R, bot_L, bot_R]$ and T the previously mentioned transformation matrix, the corresponding four points on the fixed image's patch will be:

$$T \times [top_L, top_R, bot_L, bot_R] = [top'_L, top'_R, bot'_L, bot'_R]$$

Following bellow is the visualization of the sampling process on the image pair "$A01$" and "$P02$", showing the sampling in Figure 20 and Figure 21 as well as their matching results on Figure 22 and Figure 23.

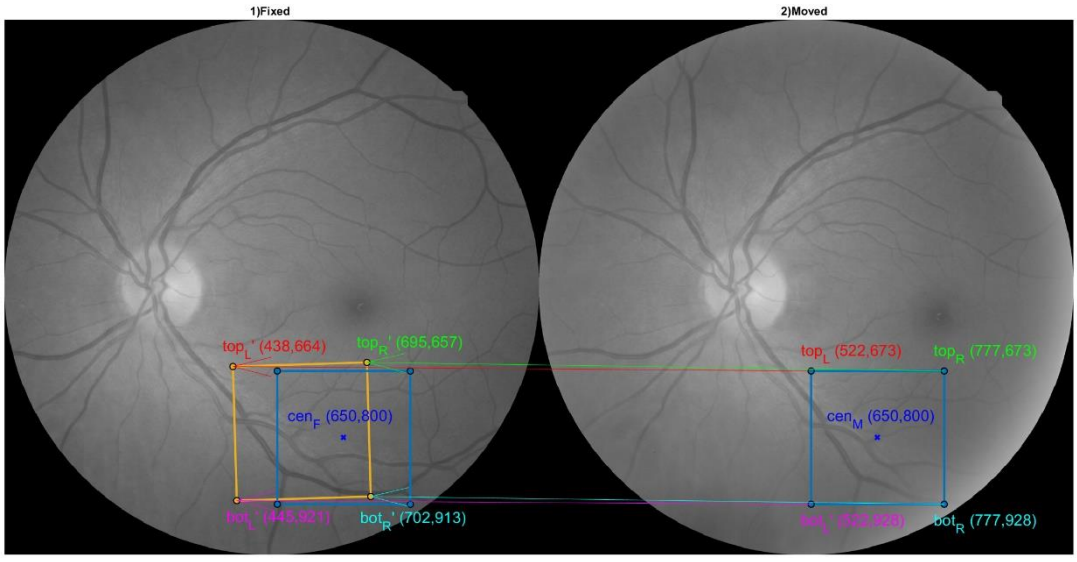


Figure 20. Image Pair "P02". The two blue squares are sampled and stacked as each image's patch. The center is the same for both images. By multiplying the corner points (top_L, top_R, bot_L, bot_R) with the respective Transformation Matrix (T), we get the four points (top_L', top_R', bot_L', bot_R') which create the yellow quadrilateral.

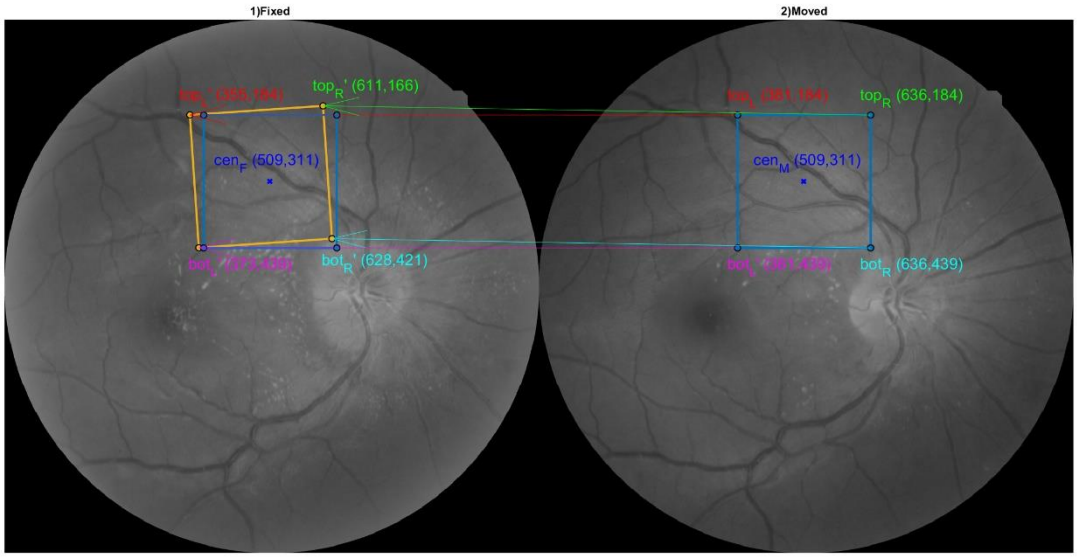


Figure 21. Image Pair "A01". The two blue squares are sampled and stacked as each image's patch. The center is the same for both images. By multiplying the corner points (top_L, top_R, bot_L, bot_R) with the respective Transformation Matrix (T), we get the four points (top_L', top_R', bot_L', bot_R') which create the yellow quadrilateral.

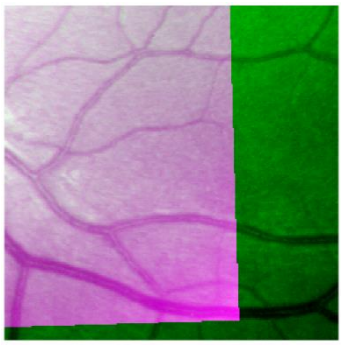


Figure 23. Image Pair's "P02" transformed patch (purple) stacked on top of the fixed patch (green).

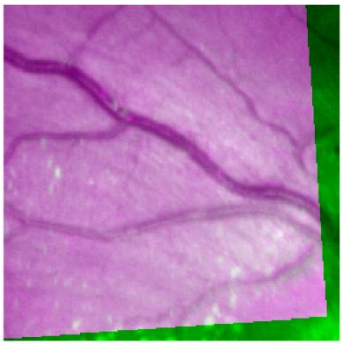


Figure 22. Image Pair's "A01" transformed patch (purple) stacked on top of the fixed patch (green).

The stacked image patches are saved as discrete data files in the disk, with the fixed image patch first and the moved one second. The targets are saved as 1×6 arrays containing the local affine matrices. There is one target file for each patch sample file. The files saved on the disk are ".npy" type, meaning they are created and can be accessed by the NumPy python library. It is a convenient file format to save the data, since NumPy contains all the essential tools to edit array-like objects. The names of the files are based on the category, the number and the pseudo ID number of each sampled pair of patches, on top of which is added a "(T)" to mark the target files for each sample. For example, for the pair of images: "`A11_1.jpg`" and "`A11_2.jpg`", the produced file for the 10th patch pair will be named: "`A11_009.npy`" (starting from 0, the 10th number will be 9) as well as its own affine matrix file will be "`A11_009_(T).npy`".

3.7. Trained Network Testing

Since the network is taking patches of 256×256 and not the full image (hence, the extraction of local features), we used a clever way to extract the global transformation matrix, by finding multiple local transformations. With the network model trained, we randomly sample an x number of patches and then proceed to find their local transformations. The following methodologies were performed, again, in a Python environment, constructing and providing multiple sets of data necessary for the transformations, as will be mentioned bellow.

3.7.1. Testing methodology

It was previously mentioned that to transform the whole image, we need to convert the local (patch) transformation back to global image transformation. The way to do that is by applying some linear algebra on the local affine matrix and the translation of the top left corner of the patch, with respect to the top left corner of the image. The equations that appear will return the global matrix. Let L_A be the local affine matrix that we predicted and let G_A be the global affine matrix. Let also be P the difference between the top_L of the patch and the top left corner of the image (point $(0,0)$) and R be any point inside the chosen patch ($P + 255 \geq R \geq P$).

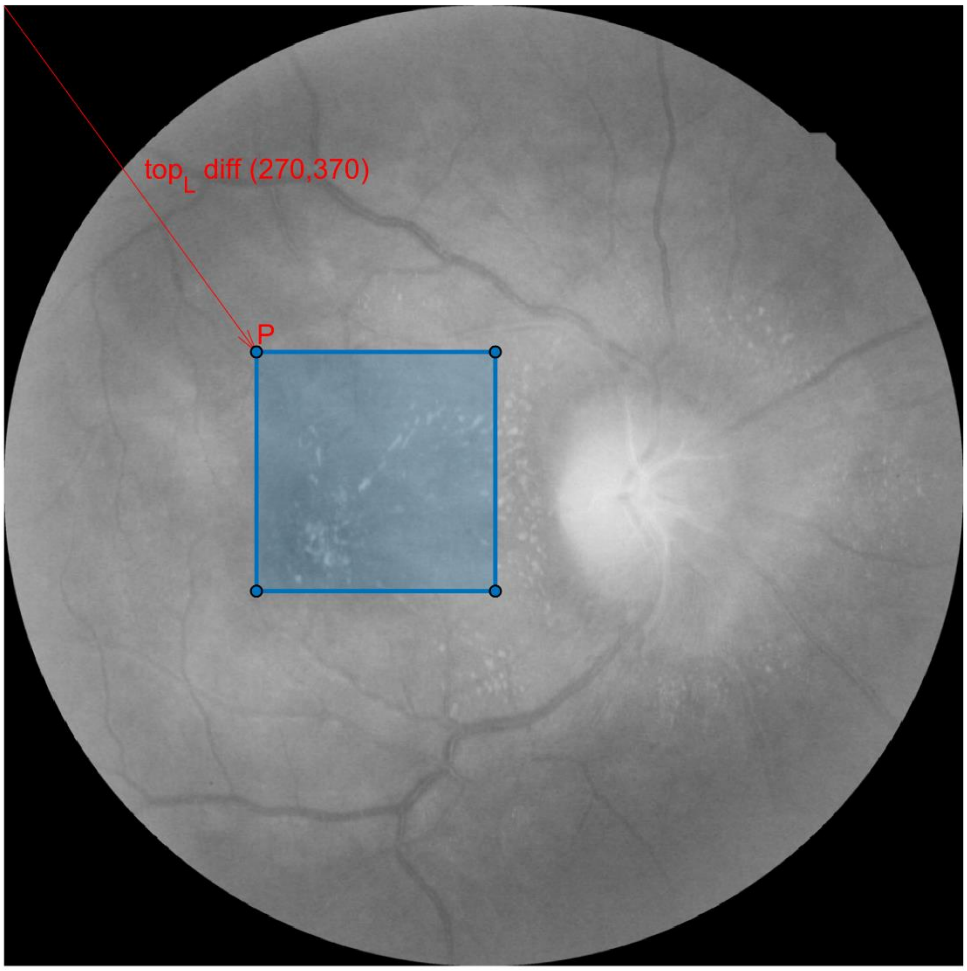


Figure 24. The top left corner of the image (global point 0,0) and its translation compared to the top left point P of the patch (local point 0,0).

$$top_L\, diff = P = (p_x - 0, p_y - 0)$$

Since the only difference between global and local transformations is the translation, that implies the following:

$$l_{11} = g_{11} \quad l_{12} = g_{12} \quad l_{21} = g_{21} \quad l_{22} = g_{22}$$

Putting everything together we can conclude into the equation:

$$G_A \times R = L_A \times (R - P) + P$$

$$\begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} r_x \\ r_y \\ 1 \end{bmatrix} = \begin{bmatrix} l_{11} & l_{12} & l_{13} \\ l_{21} & l_{22} & l_{23} \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} r_x - p_x \\ r_y - p_y \\ 1 \end{bmatrix} + \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

Using linear algebra to the above matrices, we can calculate the unknown elements g_{13} and g_{23} from the equations as shown below:

$$g_{13} = -l_{11} \times p_x - l_{12} \times p_y + l_{13} + p_x$$
$$g_{23} = -l_{21} \times p_x - l_{22} \times p_y + l_{23} + p_x$$

To retrieve the final global matrix, we calculate the mean of each element of the number of G_A patches we calculated, as mentioned above. Putting it to the test, that seemed to work very well, when used on the good results of the training samples. It is obvious that since the network did not perform well on unknown data, the results on the construction of the global transformation were also bad. This method is useful in general when it comes to image processing with a neural network, because it is very hard to have the images in full size, since they require a lot of processing power and memory, as well as time to finish each pass through the network.

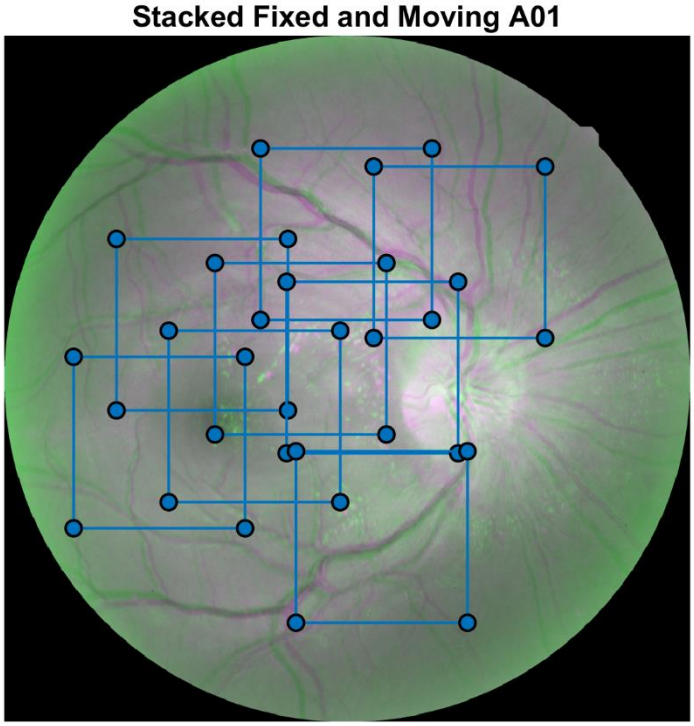


Figure 25. Image pair A01 (images are stacked on each other) and its 8 randomly sampled 256 × 256 patches.

Following are the results of the sampling case of Figure 25. The results are the prediction of the NN after converting the transformation matrix from global to local and calculating the mean of each element

from the matrix. The pair A01 was used during the training phase so the predicted values are pretty good.

The loss function we used was the MSE between the 6 local, output elements $(l_{11}, l_{12}, l_{13}, l_{21}, l_{22}, l_{23})$ and the 6 local, ground truth elements of the affine transformation matrices. More specifically, as mentioned in a previous chapter, we would take each basic type of transformation contained in the affine matrix (translation: l_{13}, l_{23}, rotation: l_{12}, l_{21} and scaling: l_{11}, l_{22}) and we would get the MSE for each of them. The final update on the network, as mentioned before, was uniquely done for each one of the three losses, starting from each fully connected block and moving backwards all the way to the start of the block containing he convolutional part of the network.

Table 5. NN's matrix prediction for A01. The predicted values of the affine matrix can be seen in rows 1 and 2. These values are the mean of the randomly sampled

Predicted affine matrix		
1.0	0.051	-28.745
-0.051	0.998	18.996
0.0	0.0	1.0

Table 6. The ground truth values of the specified pair A01. This transformation matrix, when applied on the M image, it will return a spatially accurate version of it, with respect to the first one.

Target affine matrix		
1.0	0.067	-39.21
-0.068	0.999	25.91
0.0	0.0	1.0

Table 7. Here we see each pair of numbers, representing a different transformation type contained in the affine matrix (translation, scale and rotation), is compared with the ground truth matrix, using the MSE distance. This error was similarly calculated and fed back to the network during training phase.

Individual losses		
Translation	Rotation	Scale
78.729	0.0	0.569

Table 8. The three individual losses are summed to show the overall error.

Summed losses
79.298

3.7.2. Implementation of network testing

Putting all the above into computer code, we required two basic functions, the trained NN weights (previously saved in a "`.pt`" type of file) and the data on which we will perform the testing.
The custom-made functions are similar to the previously used ones for training and data loading, with some adjustments. More specifically, a data-loading function named *"imageloader"* was used, which would take as input the test image's path and the path for the text file containing the transformation matrices. This function loads the test images and randomly samples 8 patches from it, in a similar manner to the dataset generator code. The function, the image pair, the sampled patches, the target affine matrix, the top left corner and the name of the image pair's file are appended in 5 lists, which are returned in the end of the function. These lists are then used to create their equivalent dataloaders and then passed to the second function, which is used for testing. The testing function, called *"tester"*, is taking the 5 mentioned dataloader, together with the criterion loss function (in this case MSE was used) and the loaded, pretrained NN model. In the *tester* function, the data is passed from the dataloaders to the NN and we retrieve the three output predictions corresponding to each of the three basic transformations, just like in the training phase. From there, the outputs of the patches, sampled from the same image pair, are transformed from local back to global (as shown before) and the final affine matrix is calculated through their mean value. Finally, that matrix can be applied to the M image to transform it to match spatially with the F.

4. Experiments and results

Overall, most of the NN models we tested seemed to perform in a similar manner. The training would converge relatively fast and would provide very low training loss. However, during the validation phase, the network would not perform equally well and it would converge poorly. The introduction of "disruptive" layers such as dropout layers, as well as pooling and unpooling layers reduced the network's overfitting and improved the loss of the validation, but would still not produce results that are good enough.

4.1. Results of our NN model

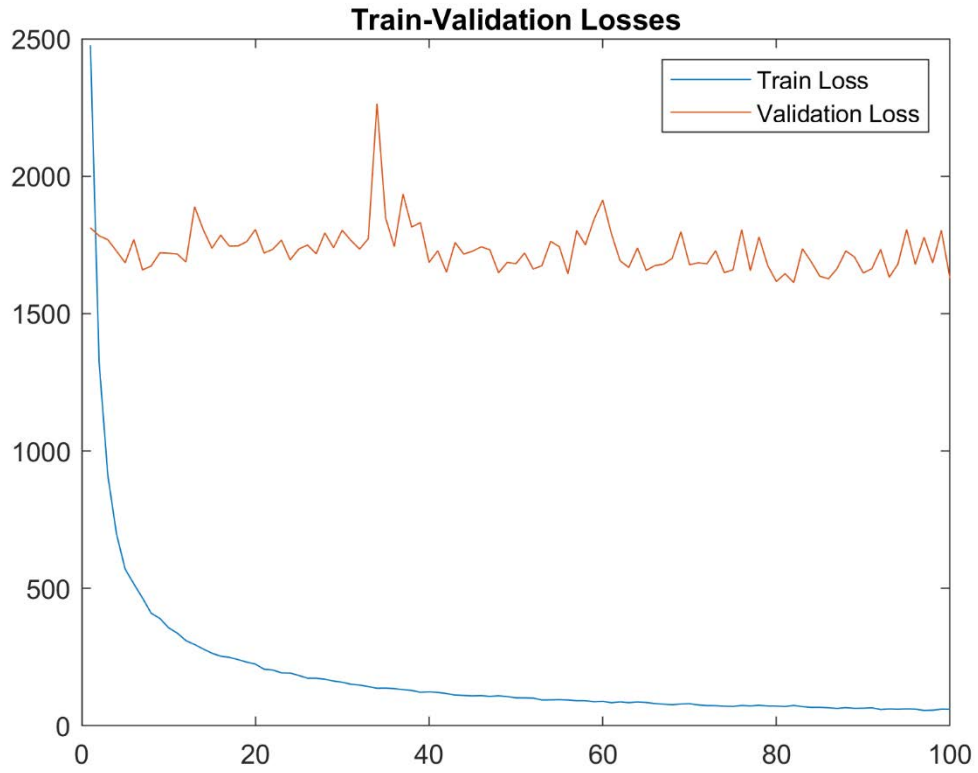


Figure 26. Training and validation phase errors graph of the 3 convolutional layer network thats shown above. It is noticeable that the NN although it learns the dataset well, it performs poorly when samples that it has never "seen" before are presented, resulting in a non-convergence.

Reducing the number of convolutional layers and increasing the number of features extracted on each convolution, appeared to assist the NN to generalize a little more. When using slow, gradually increasing convolutions and number of features, the NN appeared to perform much worse in the validation phase.

When using the green channel from the RGB color channels of the images to create the dataset, the NN appeared to underperform significantly compared to the mean of all three color channels combined to one grayscale channel.

It is important to mention that the reason we used the MSE loss, and not a normalizing loss that would, for example, normalize all of the loss ranges in a fixed 0-1 range, is because in practice, the fully-connected block that's responsible for the translation, would not update the its weights correctly on each type of transformation, hence returning wrong values in the end.

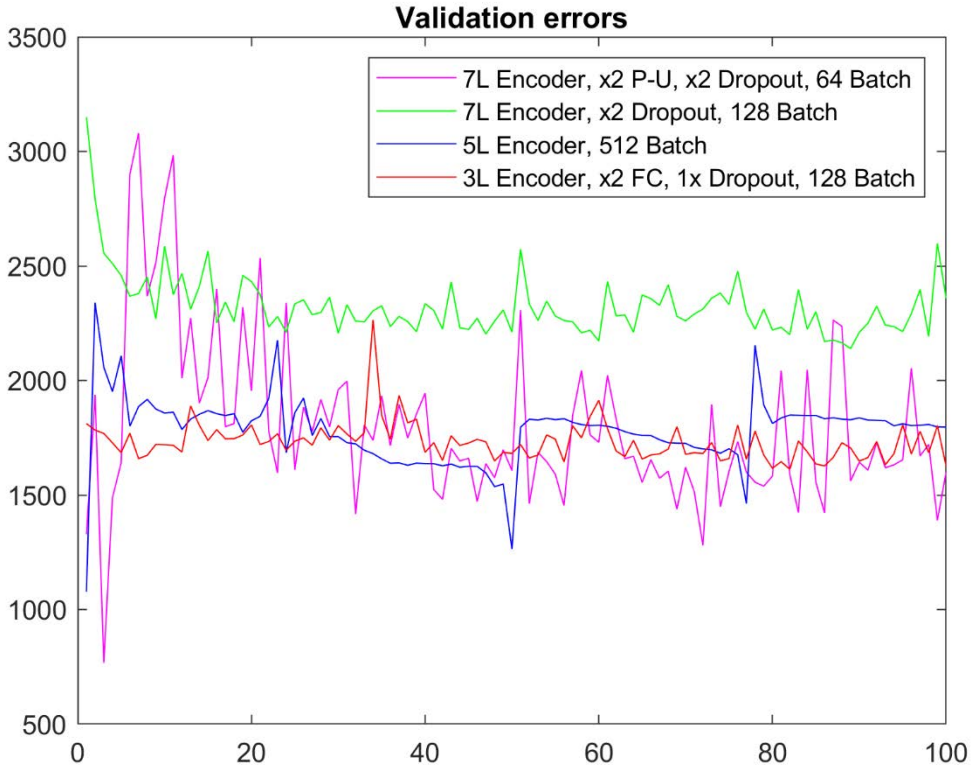


Figure 27. Comparison of the validation errors taken from four different types of model structures, following the same base architecture of parallel fully connected layers. The base differences are the number of features and convolutional/fc layers as well as dropout layers and pooling-unpooling layers.

On Figure 27 we see the validation performance of 4 different architecture attempts on the NN. On the legend of the graph, we can see the basic structural attributes of each architecture. The first attribute in every case is the number of convolutional layers contained in the model which are resembled as a number and *L* next to it. The 1st (magenta) and 2nd (green) models have 7 convolutional layers (7L) and 4 fully connected layers with the first two being 1024 in size and the other two 256. Their main difference is the pooling and unpooling layers existing on one but not the other. The 3rd model (blue) has 5 convolutional layers and 4 fully connected ones (same as before) but has no pooling-unpooling layers and no dropout layers. The 4th model (red) has 3 convolutional layers and 2 fully connected layers with 1 dropout layer in between. The sizes of the fully connected layers are 256 on both. The way the convolutional layers are gradually decreasing and increasing the size and number of features is displayed on Table 4 for the first 3 models. The last model's convolution layer details can be seen on Table 3.

The difference between the two models with the 7 convolutional layers is the performance on generalization, which is much better on the 1st model with the included pooling-unpooling and dropout layers. The performance of the 2nd, 3rd and 4th models is very similar, proving that differences in layer numbers and sizes are not affecting the performance a lot.

Considering the above, it is suggested that the problem lies in the generalization of the learned patterns and features. The network appears to struggle learning useful features and instead overfits on the introduced training samples, resulting in poor performance when new, unknown samples are presented to it.

4.2. Results from classic computer vision methodology

When using the traditional methods of computer vision, such as the initially mentioned image registration in 0, it appears to underperform when used on the raw data of the retina images. Even though the images contain a lot of distinct features, the top performing feature extracting algorithms such as SIFT, SURF, ORB, appear to struggle to produce good features. Sometimes the algorithms produce an acceptable number of features initially, but after filtering them through an "acceptable" threshold algorithm, very few remain. Having that in mind, the next step of matching, appears to eliminate almost all valid feature point pairs. In many cases the results would produce too few or even no feature pairs and even those pairs would many times be invalid, resulting in not being able to produce a transformation matrix, since the minimum number of matching pairs we need is 3 for the affine matrix. These occurrences appeared on multiple image samples, regardless of the detail or definition of the image in pixels, as it can be seen in Figure 28.

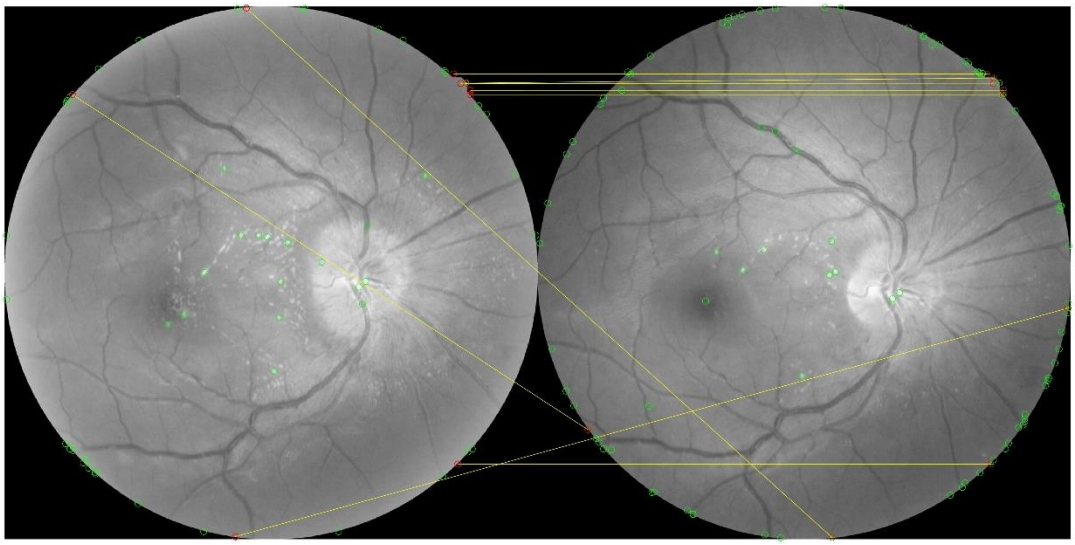


Figure 28. Image pair A01_1 and A01_2 are shown here after SIFT was applied both for feature point detection and feature extraction. Most, if not all, the feature points are noticeably in regions that lack real spatial information. Hence the produced transformation matrix is inaccurate.

4.2.1. SIFT after vessel exposure through Hessian matrix

We chose to use the *eigenvectors* extracted from the *Hessian* matrix of the images as a means of enhancing the vessels in the retinal images as explained in [26]. When performing this method, a map containing the edges of the features inside the images (vessels) is extracted using the custom made functions by [27], then we simply apply some filtering to cut off unwanted parts, such as the circular edges around the retina. Lastly, we apply the SIFT feature detector method on the extracted map, extracting both feature points and feature vectors around those points using the library [28]. The rest of the procedure is the same as previously mentioned. Points are matched, outliers are filtered out and then used to calculate the transformation matrix. In this case the results were much better. An acceptable number of features were extracted in every case, what's more is that the initial map extracted from the eigenvectors can be adjusted on different images, when there are too many features (hence more outliers) or when there are too few features (hence providing little information on the overall transformation).

Bellow we can see an example of this methodology applied on the pair "$A01\_1$" and "$A01\_2$" of the images, with the second one being the moving image (Figures Figure 29, Figure 30 and Figure 31).

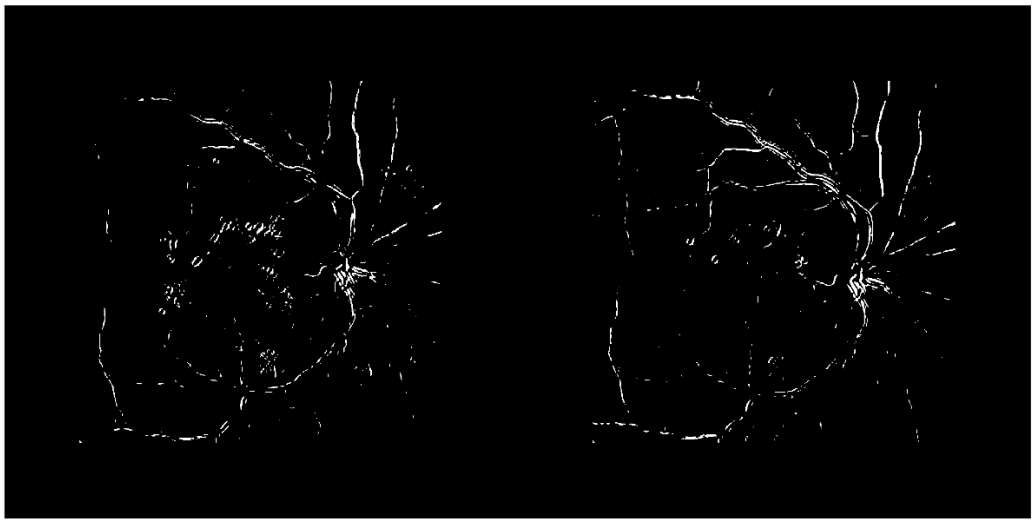


Figure 29. Derived maps of the images through the eigenvectors of the Hessian matrix of the images.

Figure 30. The two images stacked, on top of which, the matched points of the fixed image (red circles) are shown with their equivalents of the moving image (green crosses).

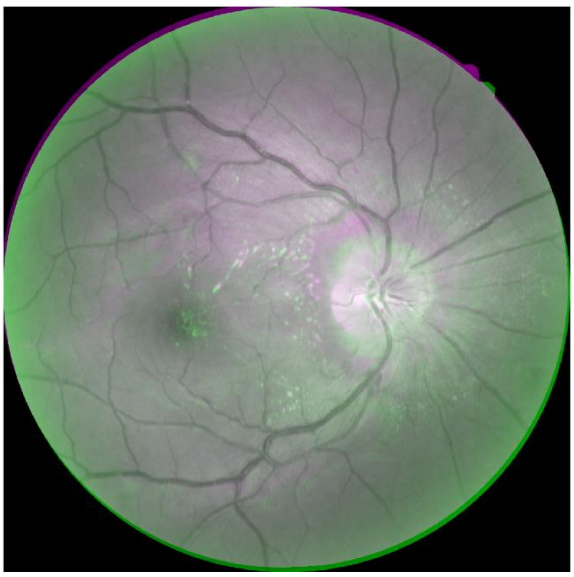


Figure 31. The two images stacked, after the predicted affine matrix was applied on the moving image using the matched points found, which are shown in Figure 30.

4.2.2. Harris corner and edge detection algorithm

Another very popular method of finding points of interest inside the image is through the Harris corner detection algorithm [29]. By itself it can provide sufficient points, which we can use to extract features through SIFT, as we did before. The results although acceptable, they were not as good as before, as it can be seen in Figures 32 and 33.

Figure 32. Harris corner algorithm detected and matched feature points on the images. It is noticeable that there are plenty of outliers and miss-matches, as well as duplicate matches on the same points.

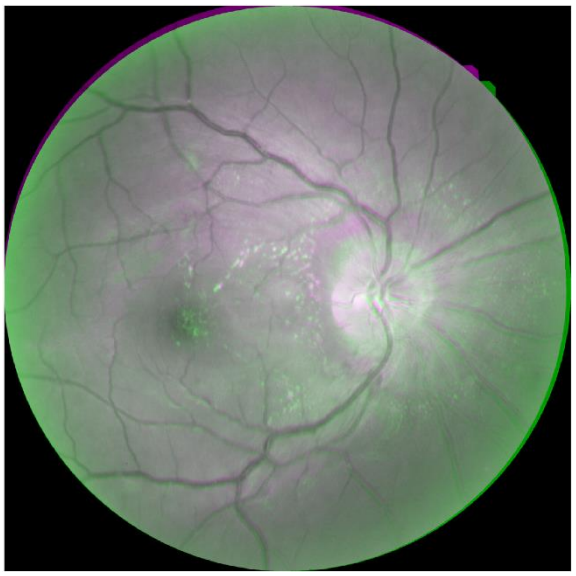


Figure 33. Matched and stacked images after predicted affine transformation matrix was applied on the moving image using the points shown in Figure 32. The match is good but not as good as before.

4.2.3. SIFT applied on Harris extracted points from Hessian maps

The combination of the two previous methods can produce better results since there are more point matches to use and there appear to be fewer outliers. The image maps extracted from the Hessian matrix for each of the two images are used to extract the most essential information. This way the number of feature points is reduced, speeding up the process and increasing the accuracy since only points of interest are kept by the Harris algorithm. An example of such can be seen below in Figure Figure 34 - Figure 36.

Figure 35. Matched Features between the two images, using a combination of Hessian maps and Harris corner/edge detection algorithm.

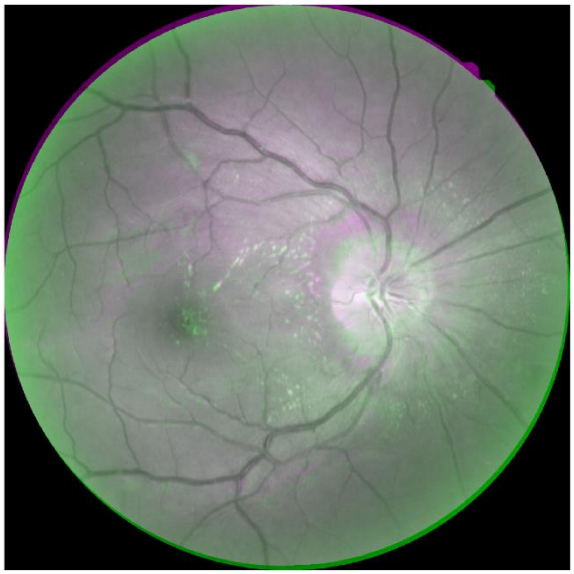


Figure 34. The pair of images after the transformation using the points found in Figure 35. The results are similar when using just the Hessian maps.

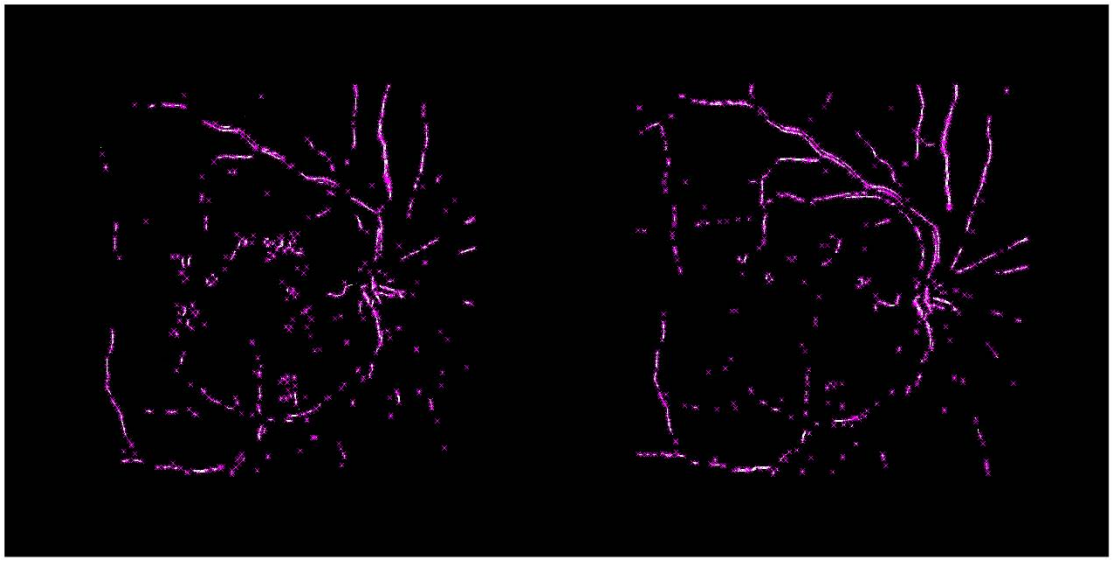


Figure 36. We can see the mappings produced through the Hessian matrix shown in white color and the Harris points found with magenta '×' signs.

Although the results appear to be (in this case) similar to the first method (Hessian maps), the results in general will be more robust, since there are more points all across the image, which provide better global generalization of the predicted affine transformation.

4.3. Ground truth comparison

Since the validation results of the deep neural network were poor, they were outperformed by the classic computer vision methods. Below, we are comparing both method's results using the given reference points contained in the dataset. We chose to split the dataset into 70-20-10%, with 10% being the testing. The way we chose to show the difference between each method's effectiveness is the reprojection error of the ground truth points of the mentioned image pairs. Specifically, we apply each method's result matrices (the transformation matrix from classic computer vision and our NN) on the ground truth points of the moving image, from each pair and then we find the root of the mean squared errors of each pair's ground truth points, using the transformed moving points and the fixed points.

Table 9. The average reprojection errors in mm of each method, for each pair of test image pairs.

	A04	A05	A06	A11	P03	S04	S17	S22	S23	S33
H+H	1.05	3.89	2.59	1.23	4.61	0.85	0.64	0.44	1.01	0.35
NN	18.13	39.62	17.12	10.4	37.25	3.36	4.68	4.54	4.86	5.05

On Table 9 the first row shows the reprojection errors from the affine matrices, predicted with the Hessian + Harris methodology. On the second row the reprojection error of our NN's predicted affine matrices is presented v. The NN model used for the testing was the 3 CL with the 2 FC layers.

During training, the overall error of the validation part of the dataset was showing that the NN was performing poorly by a significant margin. Seeing the test results on Table 9, we can see in certain cases that the NN-based method, although still outperformed by classic computer vision, it provides a result that could potentially be good.

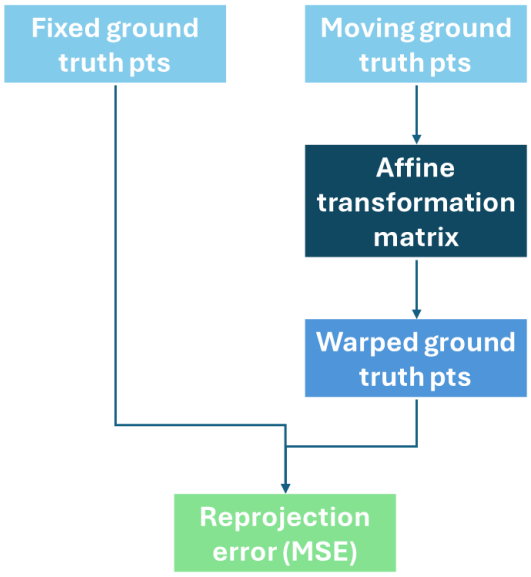


Figure 37. Reprojection error acquisition steps.

The main difference in these samples that makes the NN perform differently is the magnitude of the translation. For example, on the P03 image pair, the translation is on scale of hundreds of pixels, whereas on A and S image pairs, the translation is below 100 pixels and sometimes even below 10 pixels.

This behavior is expected since the NN "saw" a lot more samples with low translation magnitude during training compared to the samples with higher translation numbers. This issue mostly likely occurred because in the FIRE dataset only 49 images out of the 134 had high translations. Out of the 49 image pairs 32 were not used because their respective translation was too large for the selected

patch size to have enough overlay. Hence the NN did not “see” enough samples to be able to “learn” how to predict transformations of big magnitude.

The NN approach we created provided good training results but did not generalize well enough to be able to perform in samples it has not seen before (such as the validation and test image pairs with big translations). Although overall it may not be able to perform in every case of image transformation, it may still be able to provide decent results for images with transformations of small magnitude.

5. Discussion and future work

5.1. Improvements and performance

The homography based DNN [11] was briefly mentioned in the beginning without providing any more details about performance and results. The reason we chose to keep the same structure but change the output to affine transformation matrix instead of homography, was to simplify the problem into the two dimensions, hence making it easier for the network to learn as well. Although in many cases the application of a simple DNN works fairly well, in this case the network did not perform well. As seen in the original paper as well, the reported performance compared to classic methodology was not exceptionally good either.

When it comes to our second attempt on a NN structure, the parallel-based model has more interesting architecture, which could be useful in the future for similar research. Considering that this network performs exceptionally well in training, it appeared to provide decent results on certain test cases. We believe that there could be a possibility for a very well performing method to match (spatially) pairs of images. The benefits of such a NN in comparison to classic computer vision would be the speed and low computational cost it would require calculating the results, as well as obtaining an end-to-end approach.

An introduction of a better-defined loss function, could be a means of improving the network's performance. Defining a more elaborate way to handle the three differently scaled losses, could increase the sensitivity of the NN on learning generic patterns on each type of transformation. This could also potentially assist on the generalization of learning how to predict the translation parameters of the affine matrix in such a manner that it would not be affected by the translation's magnitude.

Finally, the dataset is crucial for the effectiveness of the training and the convergence of the NN. Although publicly available datasets, especially on biomedical data, have come a long way on being free and easy to access, there are still very few of them out there, with most of them being not well defined. FIRE was an example of a very well-constructed and ready to use, public dataset. With a bigger number of data samples, a bigger variety of image types and transformations could be introduced to the NN, resulting in better generalization and easier convergence, resulting in a better performance overall, without overfitting.

When we applied the classic computer vision algorithm for image registration through simple feature extraction with the SIFT algorithm, we noticed that the results were poor. That shows that in many cases, classic algorithm application in image registration does not work and more complicated methods need to be applied. The retina imagery is just one example of such cases where computer vision still lacks routine algorithms that generalize and perform well on the specific category of images.

5.2. Conclusion

The need for classic computer vision has been around for decades. Throughout the years, more and more algorithms were developed, trying to improve already existing ones or creating new ideas, with each algorithm being faster and more effective. Although the introduction of deep learning has been a useful idea and many times more effective than classic algorithmic

computer vision, there are still cases that it lacks research and knowledge, hence it is not as effective or accurate as classic methodology.

Considering the above, it is also worth mentioning that in most cases, classic computer vision requires a variety of variables needed to be set differently, as different images tend to have points of interest on different levels. Considering this, together with the fact that these algorithms tend to be costly both in terms of time and memory, deep learning can sometimes provide solutions that may not be as accurate, but they can be much faster and less costly than their classic computer vision equivalents, while achieving an end-to-end approach, which is a very desirable characteristic.

This thesis has shown that although there appear to be many possibilities, it is also unclear and requires a lot of experimentation when it comes to applying NNs and deep learning in image registration. Although not irrelevant, the results of deep learning are still lacking compared to conventional methodologies, such as the ones our network is compared to. With more people experimenting and researching this sector of computer vision, it is only a matter of time before feature extraction and spatial image registration becomes effectively implemented in deep learning as well.

6. References

[1] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[2] H. Bay, T. Tuytelaars, and L. Van Gool, “SURF: Speeded Up Robust Features,” in *Computer Vision – ECCV 2006*, 2006, pp. 404–417.

[3] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571.

[4] M. A. Fischler and R. C. Bolles, “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography,” *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981.

[5] R. Raguram, O. Chum, M. Pollefeys, J. Matas, and J.-M. Frahm, “USAC: A Universal Framework for Random Sample Consensus,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 2022–2038, 2013.

[6] M. Ivashechkin, D. Barath, and J. Matas, “VSAC: Efficient and Accurate Estimator for H and F,” *CoRR*, vol. abs/2106.10240, 2021.

[7] D. Barath and J. Matas, “MAGSAC: marginalizing sample consensus,” *CoRR*, vol. abs/1803.07469, 2018.

[8] G. Nousias, K. Delibasis, and I. Maglogiannis, “H-RANSAC, an algorithmic variant for Homography image transform from featureless point sets: application to video-based football analytics.” 2023.

[9] D. Barath, J. Noskova, M. Ivashechkin, and J. Matas, “MAGSAC++, a fast, reliable and accurate robust estimator,” *CoRR*, vol. abs/1912.05909, 2019.

[10] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, “Deep Learning for Computer Vision: A Brief Review,” *Computational Intelligence and Neuroscience*, vol. 2018, p. 7068349, Feb. 2018.

[11] D. DeTone, T. Malisiewicz, and A. Rabinovich, “Deep Image Homography Estimation.” 2016.

[12] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft COCO: Common Objects in Context.” 2015.

[13] B. D. de Vos, F. F. Berendsen, M. A. Viergever, M. Staring, and I. Išgum, “End-to-End Unsupervised Deformable Image Registration with a Convolutional Neural Network,” in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, 2017, pp. 204–212.

[14] K. Marstal, F. Berendsen, M. Staring, and S. Klein, “SimpleElastix: A User-Friendly, Multi-Lingual Library for Medical Image Registration,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2016.

[15] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, “Spatial Transformer Networks,” *CoRR*, vol. abs/1506.02025, 2015.

[16] T. Kurbiel, “Spatial Transformer Networks.” 2021.

[17] Q. Zhou and X. Li, “STN-Homography: Direct Estimation of Homography Parameters for Image Pairs,” *Applied Sciences*, vol. 9, no. 23, 2019.

[18] K. K. Delibasis, P. A. Asvestas, and G. K. Matsopoulos, “Automatic point correspondence using an artificial immune system optimization technique for medical image registration,” *Computerized Medical Imaging and Graphics*, vol. 35, no. 1, pp. 31–41, 2011.

[19] J. Timmis, “Artificial Immune Systems,” in *Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2010, pp. 40–44.

[20] G. Jocher, A. Stoken, J. Borovec, NanoCode012, ChristopherSTAN, L. Changyu, Laughing, tkianai, A. Hogan, lorenzomammana, yxNONG, AlexWang1900, L. Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, F. Ingham, Frederik, Guilhen, Hatovix, J. Poznanski, J. Fang, 于力军 L. Y., changyu98, M. Wang, N. Gupta, O. Akhtar, PetrDvoracek, and P. Rai, "ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements." Zenodo, Oct-2020.
[21] T. Sattler, B. Leibe, and L. Kobbelt, "SCRAMSAC: Improving RANSAC's efficiency with a spatial consistency filter," in *2009 IEEE 12th International Conference on Computer Vision*, 2009, pp. 2090–2097.
[22] O. Chum, T. Werner, and J. Matas, "Two-view geometry estimation unaffected by a dominant plane," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2005, vol. 1, pp. 772–779 vol. 1.
[23] J. Matas and O. Chum, "Randomized RANSAC with sequential probability ratio test," *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 2, pp. 1727–1732 Vol. 2, 2005.
[24] P. Lindstrom, "Triangulation made easy," *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1554–1561, 2010.
[25] C. Hernandez-Matas, X. Zabulis, A. Triantafyllou, P. Anyfanti, S. Douma, and A. A. Argyros, "FIRE: Fundus Image Registration dataset," *Modeling and Artificial Intelligence in Ophthalmology*, vol. 1, no. 4, pp. 16–28, 2017.
[26] A. F. Frangi, W. J. Niessen, K. L. Vincken, and M. A. Viergever, "Muliscale Vessel Enhancement Filtering," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 1998.
[27] D.-J. Kroon, "Hessian based Frangi Vesselness filter."
[28] A. Vedaldi and B. Fulkerson, "Vlfeat: an open and portable library of computer vision algorithms," in *Proceedings of the 18th ACM International Conference on Multimedia*, 2010, pp. 1469–1472.
[29] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *Proceedings of the 4th Alvey Vision Conference*, 1988, pp. 147–151.