Project 1 - MyAutoPano Report

Using 1 late day

Ameya Patil
Department of Computer Science
University of Maryland
College Park, Maryland 20740

Sigurthor Bjorgvinsson
Department of Computer Science
University of Maryland
College Park, Maryland 20740

I. Introduction

The aim of this project was to stitch multiple images having considerable content common between them. The homography between the images was computed to get a measure of how much a certain image needed to be transformed to make it merge with the other image. This homography estimation was done using 2 approaches - the classical approach and the deep learning approach.

II. Phase 1: Classical Approach

This approach is based on finding corners in the image, followed by calculating features of the image at all these corner points. These two steps are performed for all the images to be stitched. After this, images are checked for adjacency based on the corner features; better or more matches imply that the images are adjacent. Corner pairs are then chosen to compute the best homography matrix and apply it on the corresponding image to bring it in the same plane as the other image. The last step is to blend the two images. Each of the above mentioned steps are described in detail below.

A. Corner Detection

Corners are characterized by changes in intensity in more than one direction. The Harris corner detection algorithm uses the Sobel filter to calculate the image gradients in both the X and Y directions for each pixel. Based on these gradients, it then calculates a score per pixel which tells us how likely the pixel is to be a corner. We used the opencv cv2.CornerHarris function for this part. The output of this function is an image with a per pixel corner score. We then performed a thresholding operation on the image to filter out the pixels with a low corner score and retain only the strong corners. The results are shown in Figure 1, Figure 2, Figure 3 and Figure 4.

B. Adaptive Non-Maximal Suppression

Since corners cannot be perfectly detected, many pixels around a true corner may also get marked as a corner after the application of the Harris corner detection algorithm. To get the true corners, we evaluated the strongest corner in a sliding window of a user configurable size and retained it, while all other corners in that window were rejected. This step is called the local maxima filtering and it makes sure that detected corners are not clustered around one another. However, another possible issue that can arise is the unequal distribution of corners throughout the image, for example, most of the good corners are in the top left quarter of the image while the bottom right quarter has only a few corners. To avoid such a situation, we performed Adaptive Non-Maximal Suppression (ANMS) of the corners. ANMS computes a compression radius metric for each corner pixel. Compression radius of a pixel refers to the radius of a circle centred at the pixel in which all the corner pixels are weaker than the centre pixel. It then chooses a user configurable number of corner pixels having the highest suppression radii. This ensures that corners are not densely populated in a certain region of the image. This is similar to the local maxima filtering but takes into account the entire image rather than a small window which also makes it an expensive operation in terms of execution time. Performing local maxima filtering before ANMS reduces the workload for ANMS. The results are shown in Figure 5, Figure 6, Figure 7 and Figure 8.

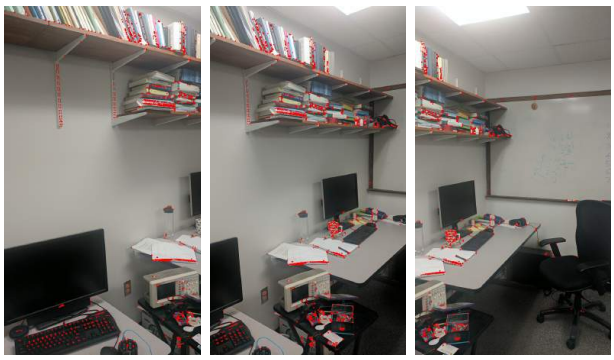
C. Feature Matching

The next step is to extract features around all the corner pixels in an image. This feature is used a characterization of that corner pixel which enables us to match corners in different images. A 40x40 patch was extracted around each corner pixel from the image and a gaussian blur was applied to it. Corner pixels around which such a patch could not be extracted because they were very close to the edges, were skipped. The patch was then downsampled to 8x8 and reshaped to a 64 element vector. So each corner in an image was characterized by a 64 element feature descriptor. The images to be checked for adjacency were then iterated over all its corners and a matching corner in the second image was searched for using the L2 loss between their feature descriptors. To visualize these matches, we created a custom implementation of the opencv cv2.drawMatches function. This was done in order to avoid the syntactic requirements of the arguments passed to the cv2.drawMatches function. The results are shown in Figure

(a) Corner Detection for Train Set 1 images

(b) Corner Detection for Train Set 2 images

(c) Corner Detection for Test Set 3 images



(d) Corner Detection for Test Set 4 images

Fig. 1. Corner Detection for Train and Test images, marked in red, each corner is 1x1 pixel

9, Figure 10, Figure 11 and Figure 12. Figure 13 shows an example of bad match found in one of the test sets.

One issue that we were seeing here was the matching of multiple corners in one image to the same corner in the other image. To work around this issue, we changed the feature matching code such that we have a strict one-to-one mapping between the corners of the two images.

D. RANSAC

After matching features between two images, their matches are not necessarily correct. With RANSAC, we select the

(a) Corner Detection for Custom Set 1 images

(b) Corner Detection for Custom Set 2 images

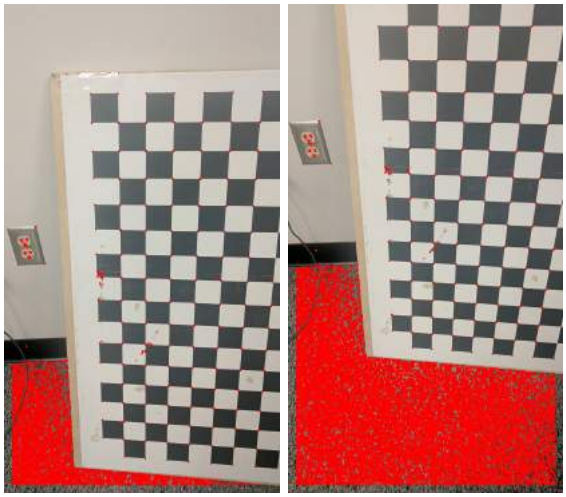
Fig. 2. Corner Detection for custom images, marked in red, each corner is 1x1 pixel

largest number of inliners that have similar transformation in terms of distance from their matched feature. In a loop, we select 4 random matched points (i.e. 'src' and 'dest') from the list of matches. We calculate the homography matrix (H) for the 4 pairs and then calculate the distance from the projected points (by applying the H matrix on the 'src' point) to the 'dest' point. The distance is calculated from the sum of the squared differences of the projected 'src' and the 'dest' points. If the sum of the squared differences is below a threshold, it is added to the list of inliners. Once 90% of the matches are in the list of inliners or a number of iterations are exhausted, the search for inliners ends. The next step of the RANSAC algorithm is supposed to compute the Least Squared $\hat{H}$. We were unable to implement this step correcly and instead took a shortcut. Instead of computing the Least Squared $\hat{H}$ we used the H that included the most inliners. The results are shown in Figure 14, Figure 15 and Figure 16.

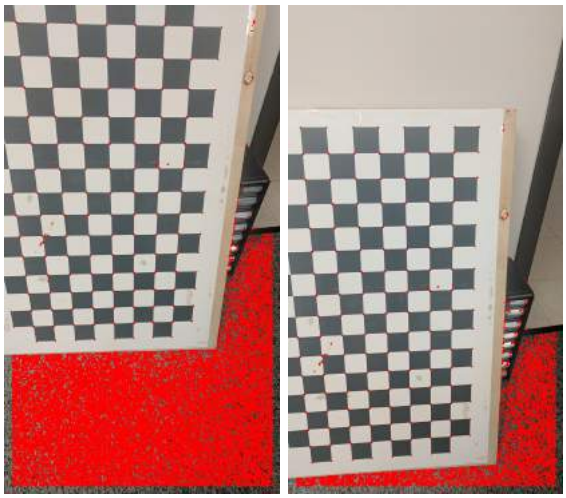
E. Picking Best Match

For our algorithm to be robust, we wanted to be able to take in a folder of images and stitch together any images that fit together. The wrapper performs feature matching and RANSAC for all pairs of images and stores a score for each pair. The score is calculated by the $\frac{NumInliners^2}{NumMatchedFeatures}$ for each pair and each image is given an ID (1...N). Pairs are created between 1 and 2...N and then between 2 and 3...N and so on, creating a unique pairs without repetition. Before two images are stitched, each image in the range of 1..N-1 searches for the image that has the highest matching score. From those lists, the highest pair score is selected for stitching.

When two images are stitched, their IDs are removed from the stitching pool and a new image with ID N+1 is created. The Corner Detection and ANMS is applied on the newly stitched



(a) Corner Detection for Test Set 1 imagesCorner Detection for Test Set 2 images



(b) Corner Detection for Test Set 1 images

Fig. 3. Corner Detection for Test Set 1 images, marked in red, each corner is 1x1 pixel

image, which is then paired with every other image in the pool where Feature matching and RANSAC is performed. If images have less than 8 matched features or less than 6 inliners, the images are not stitched together. If there are no images that pass this requirement, the images that are left in the pool are written to the 'panos' folder.

F. Blending Images

Aligning and blending the images was a difficult task on its own. We attempted padding the image that was going to be transformed so the its edges would be included in the transformed image. The padding would be decided by the values of the transformed corner points and the padding applied on the left and top of the image we were stitching to. This approach seemed like the logical approach but led to offsets we had a hard time figuring out. The blending started as overlap where we set the transformed image as a background and applied the other image on top. This created a shortcut for us where we did not have to worry about the seams of the transformed image where pixels bled into the background because of the tilt of the image.

After multiple iterations we finally got the locations of the images right. We applied a transformation to the homography matrix which moved the warped image into the frame of the other image. Padding to the non-warped image was required if the warped location was projected outside the non-warped image boundaries.

To handle overlapping regions, we created a blended image using cv2.addWeights() function. When combining the warped and non-warped image, if both images had non-black pixels at a given location, we selected pixels from the blended image; if only the warped image or non-warped image had pixels, we selected the pixels from them, respectively.

Here is where the seams started to really show. Because we were selecting pixels from 3 different arrays, the gradient of the pixel intensities resulted in the seam being more visible (or close to black pixels because of pixel bleed after warping). We had a few attempts at addressing this issue without much success for example changing the alpha/beta value of the addWeights by giving the non-warped image more weight and setting a condition for the warped image pixels to have more that a combined value of x to be included in the image to counteract the pixel bleed on the edges after warping.

Another issue was that the edges of an stitched image were black because of the change of perception. When this black padding intersected the actual pixels, the gradient was large and could be interpreted as a part of a corner. Our solution to this was to create a alpha channel array that was initialized after blending two images. This array had the same size as the stitched images and was set to -1 if a pixel was too dark (i.e. because the sum of the pixel value of edges were 0). During corner detection on that image, corners that would be within 20pixels of a negative value would be filtered out. This alpha channel array could also have been populated with the x,y coordinates of where the stitched lines intersect which would have partially mitigated the seam issue. A better stitching algorithm would have been a priority before implementing something like that.

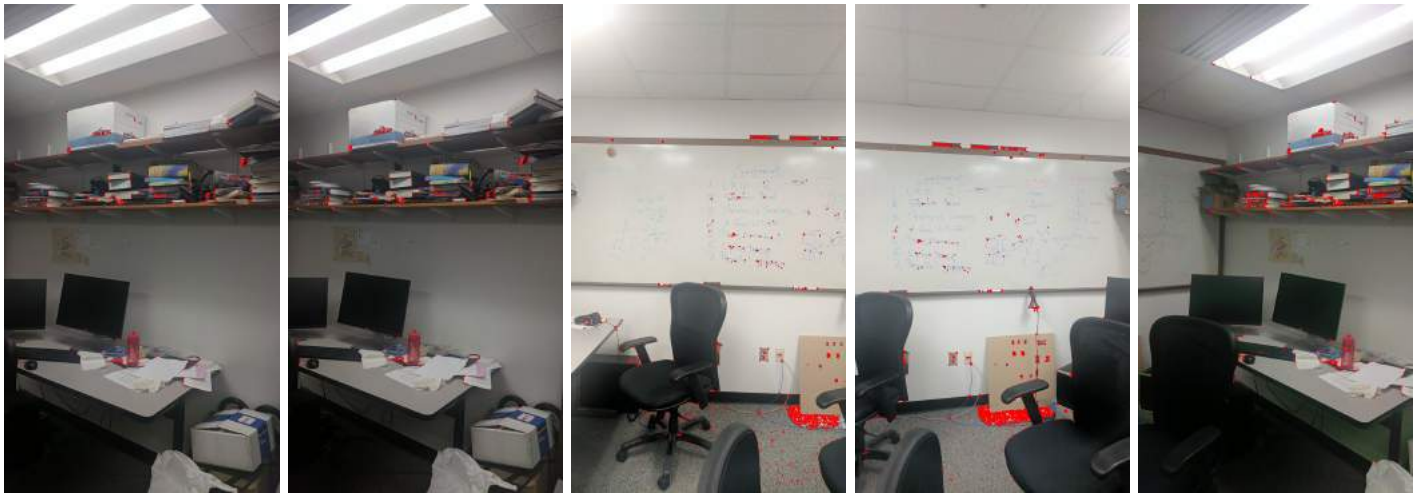
G. Results

Our approach works well with few images but as more images are stitched together, the more stitches and distortions are added to the final image. We obtained the best results with the configuration values shown in table I. The largest set we applied the algorithm on (Train set 3) took about 12 minutes to run. Resulting panoramas are shown in Figure 17 and Figure 18

H. Possible Improvements

1) Better Blending: The blending was definitely where we could have improved most. The edges of where the images intersect are taken as corners in the next iteration of the algorithm creating a lot of noise. This noise is then propagated and increases at each stitch making future stitching

(a) Corner Detection for Test Set 2 imagesCorner Detection for Test Set 2 images



(b) Corner Detection for Test Set 2 images

Fig. 4. Corner Detection for Test Set 2 images, marked in red, each corner is 1x1 pixel

Configuration Variable	Value
NumFeatures	400
WindowSize	7
RansascDistanceThreshold	15.0
RansacIterations	5000
FeatureMatchingRatioThreshold	0.65

TABLE I
CONFIGURATION VALUES

harder and harder. Applying a better blending algorithm would have allowed for greatly improving the performance of our algorithm.

2) Weight to Location of Features: When images become very large, there is a problem with features being located in places that no image that will be stitched will contain. for example, if the image is 3000px width, the features on the edges should have more weight than the features in the center. This might have created more features that would have been more likely to be matched to other images. A possible approach to that would be to apply ANMS on sections of the image with different feature values.

I. Failed Panoramas

Some panoramas failed to be created when there were more than 4 or 5 images. When our algorithm had combined so many images with such a primitive blending algorithm, the images became distorted. This resulted in images not being able to finish their full panorama but ended up with two partial panoramas. This distortion also creates homography matrices

(a) ANMS for Train Set 1 images

(b) ANMS for Train Set 2 images

(c) ANMS for Test Set 3 images

(d) ANMS for Test Set 4 images

Fig. 5. ANMS for Train and Test images, marked in red, each feature is a 5x5 pixel block

that are not correct like in Test set 2. When trying to combine the last two partial panoramas the homography matrix between them created a warped image of 33500x25740 which ended up crashing because of lack of memory. These failed or partial panoramas can be seen in figure 19

III. Phase 2: Deep Learning Approach

In the deep learning approach, we try to create a network which learns the homography between a pair of images. There are two kinds of approach within the deep learning approach which have been described in detail in the subsequent sections.

(a) ANMS for Custom Set 1 images

(b) ANMS for Custom Set 2 images

Fig. 6. ANMS for custom images, marked in red, each feature is a 5x5 pixel block

A. Data Generation

We decided to use the example in the paper with image size of 128x128 using ρ value range of $-32 < \rho < 32$ as suggested from the paper. First we started by generating patches and labels and saving them in a folder. Later on we decided to move this functionality to be done on the fly during training and testing.

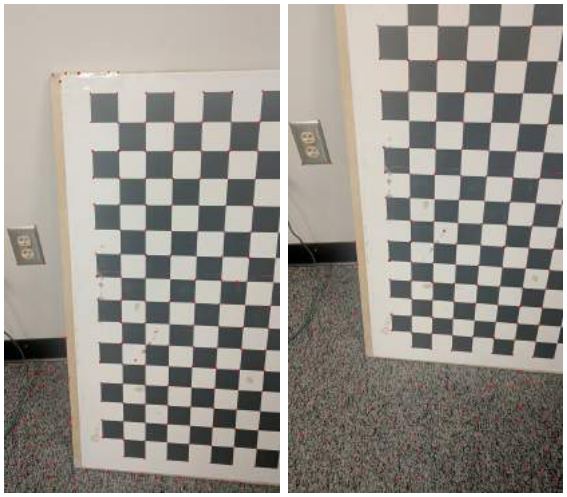
B. Supervised Approach

The supervised approach requires lots of data to be able to be effective and to be able to learn the function of perspective matching. Using the data generation for unlimited training data, we attempted at implementing a network that would perform well on finding out a pixel shift of patches and from that figure out the homography matrix between the two patches.

1) Network: The original network [2] had 8 convolution layers with a fully connected layer and 8 layer output. We built on that and implemented the network in Tensorflow. Our network followed the architecture described but added shortcuts. There are 3 shortcuts in the network, all leading from the input layer to the input of every other convolution layer. This was added just to play with the network a bit and try to make it our own. We used ReLU activation function along with batch normalization right after that. The network has 34194568 parameters and is show in in Figure 20.

2) Results: We trained a model for 500 iterations using batch size 64 and learning rate of 0.0001. the loss in the table was the average of $||\widetilde{H_{4Pt}} - H_{4Pt}||_2$ for all predictions. See table II.

One forward pass of the supervised network took around 0.002 sec after the network was loaded and graph was initialized on a GTX 1080 and about 0.01 sec on a GTX 1060M



(a) ANMS for Test Set 1 images

(b) ANMS for Test Set 1 images

Fig. 7. ANMS for Test Set 1 images, marked in red, each corner is a 5x5 pixel block

Image Set	Avg. Loss	Avg. Corner Error	Avg. Forward Pass
Train	30.320	17.735	0.00215s
Validation	30.728	18.023	0.00271s
Test	29.416	17.249	0.00270s

TABLE II
EXECUTION RESULTS OF SETS ON OUR TRAINED MODEL

The results of the supervised model were very disappointing because of the results reported in the paper. The paper states that they had a average pixel error of little more than 9. When running the validation set, our average corner error was reported around 8.5. When we tried to plug our model into the wrapper tested the prediction in Figure 21, the results were not what we expected as we will describe in the last sub-section.

3) Effect of Changing Batch Size During Testing: We were unable to figure out why this was happening but when we changed the batch size of the network during testing, the loss of the network more than doubled. We tried talking with some students but no one else seemed to be having the same issue. We talked with UMIACS faculty Prof. Abhinav Srivastava who told us that this was most likely because because the batch size was being baked into the network during training. the batch size was hard coded into the image placeholder but changing that to None and retraining the network did not help. Table III shows execution results when batch size is 64 which is obviously much better.

Image Set	Avg. Loss	Avg. Corner Error
Train	14.845	8.549
Validation	14.639	8.446
Test	14.668	8.459

TABLE III
EXECUTION RESULTS USING BATCH SIZE 64

The homography estimation for our supervised network is shown in Figure 21. This estimation was performed with one image at a time, meaning a batch size of 1. As discussed above, we were facing reduced accuracy with batch size of 1 which is evident in the figure.

C. Unsupervised Approach

Unlike the supervised approach, the unsupervised approach computes the homography between a pair of images without using explicit labels for the 4-point homography description. The network used for this approach consists of 4 parts as described below:

1) Regression Homography Model: This is exactly the same network as was used in the supervised homography detection approach. This network takes as input, 2 patches extracted from the same image and one of them is warped with certain homography. This model learns and predicts the homography with which the two patches are related, in the 4 point formulation. This 4 point estimate is then added to the 4 corners of the original patch to get an estimate of the 4 corners of the warped patch.

2) TensorDLT Network: This network takes as input the 4 corners of both the original and the warped patches. It then formulates a system of linear equations and solves it using singular value decomposition in a differentiable way so that errors can be backpropagated through the network. This gives us the (3x3) homography matrix. The network in a way, learns the functionality of the opencv cv2.getPerspectiveTransform() function in a differentiable way.

3) Spatial Transformer Network: This network takes as input, the homography matrix predicted by the TensorDLT network and the original image from which the original patch was extracted. It then learns the application of the homography on the original **image** to get the warped **patch**. The network in a way, learns the functionality of the opencv cv2.warpPerspective() function in a differentiable way.

4) Calculating Photometric loss: The warped patch extracted from the Spatial Transformer Network is then compared with the ground truth warped patch using the L1 photometric loss. This loss is backpropagated through the network.

(a) ANMS for Test Set 2 images

(b) ANMS for Test Set 2 images

Fig. 8. ANMS for Test Set 2 images, marked in red, each corner is a 5x5 pixel block

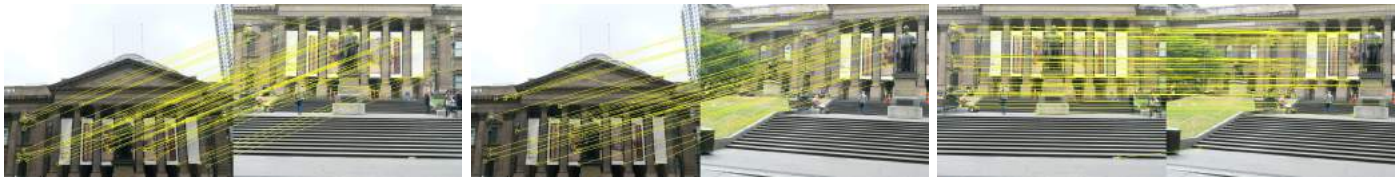
5) Result: We were unable to train the unsupervised network and there for do not have any results for it. The network did not become better over time which we thought indicated that it was not working. We gave it our best shot.

D. Wrapper

The wrapper in Phase 2 takes in a source folder and tries to match images together. This wrapper is rather unsophisticated because we had a hard time conceptualizing what part of the classical approach was supposed to be used and what the model should take care of and what we needed to implement. Our wrapper expects that all images go together, starting from the one with the name first in alphabetical order. The images are resized to 128x128 and passed through the network to get a prediction. This prediction is changed into a homography matrix using cv2.getPerspectiveTransform and applied to a image using the same transformation logic as in the phase 1 wrapper. This combined image is then put back into the pool. We tried applying this wrapper on the test sets but we were only able to generate one output image from test set 1 22. All other images exploded in size resulting in output images as 36000x13800 and 41700x10955, making us run out of memory.

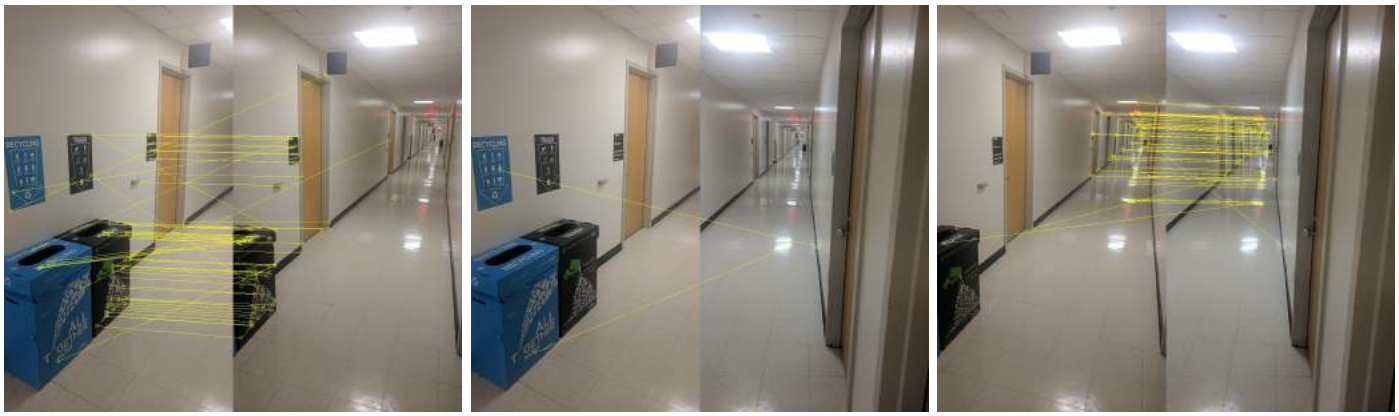
IV. Conclusion

By working on this project, we have learned about the homography relationship between images, how is a homography specified in the 4 point form or the 3x3 matrix form and how can the homography equation be solved using singular value decomposition. Further, we got an idea of how a system of

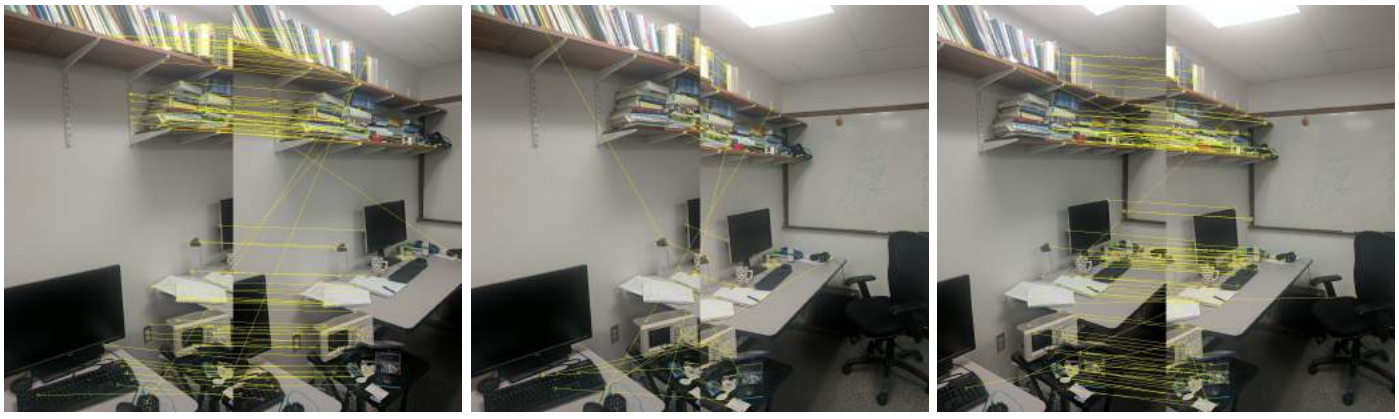


(a) Feature matches for Train Set 1 images

(b) Feature matches for Train Set 2 images



(c) Feature matches for Test Set 3 images



(d) Feature matches for Test Set 4 images, other images did not have enough matches

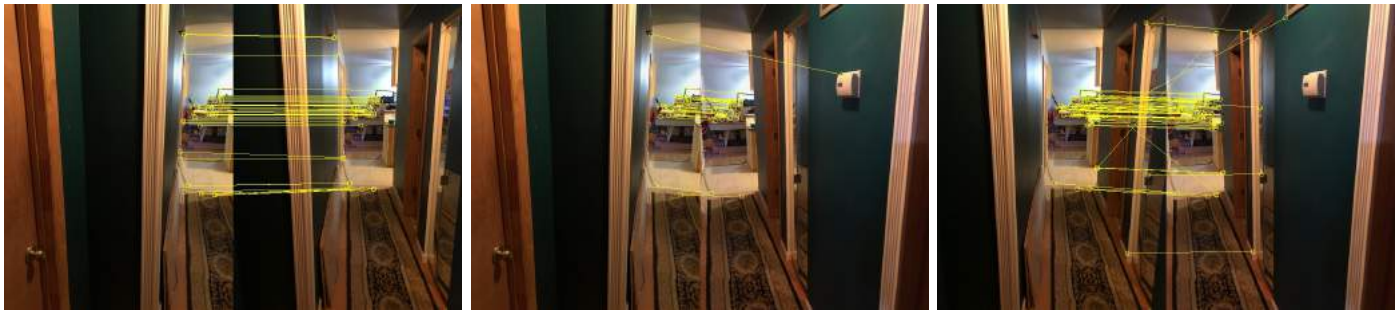
Fig. 9. Feature matches for Train and Test images

linear equations can be solved using a neural network, in a differentiable way. Finally, we also gained an understanding of the applications of homography in stitching panoramas which can be useful in a wide range of fields ranging from UAV navigation to general aesthetic photography

REFERENCES

[1] Unsupervised Deep Homography: A Fast and Robust Homography Estimation Model; Nguyen, Ty and Chen, Steven W and Shivakumar, Shreyas S and Taylor, Camillo J and Kumar, Vijay. Available: https://github.com/tynguyen/unsupervisedDeepHomographyRAL2018

[2] Deep Image Homography Estimation; Daniel DeTone, Tomasz Malisiewicz, Andrew Rabinovich. https://arxiv.org/pdf/1606.03798.pdf

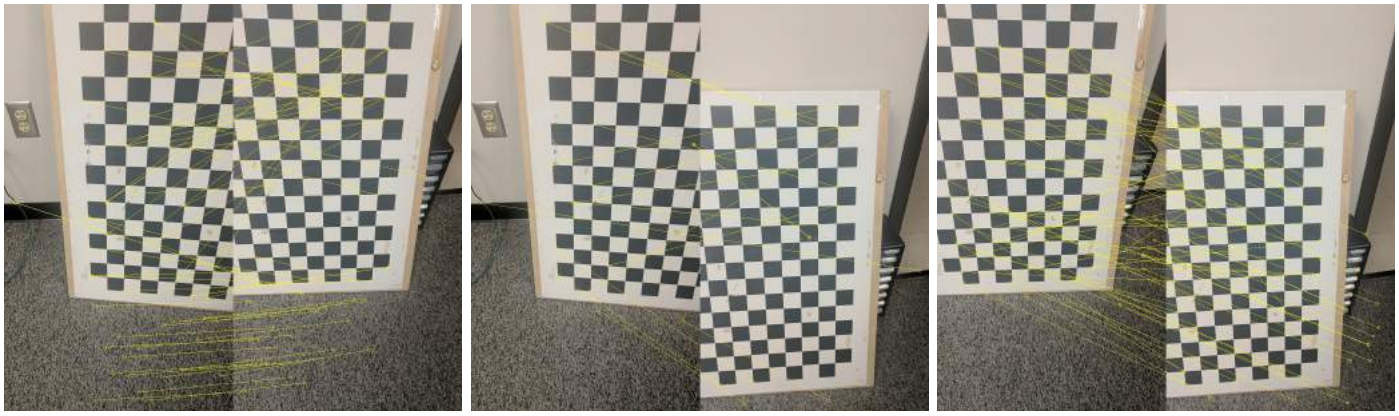


(a) Feature matches for Custom Set 1 images

(b) Feature matches for Custom Set 2 images

Fig. 10. Feature matches for custom images

(a) Feature matches for Test Set 1 images



(b) Feature matches for Test Set 1 images

Fig. 11. Feature matches for Test Set 1 images

Fig. 12. Feature matches for Test Set 2 images

(a) Bad feature matches for Test Set 2 images

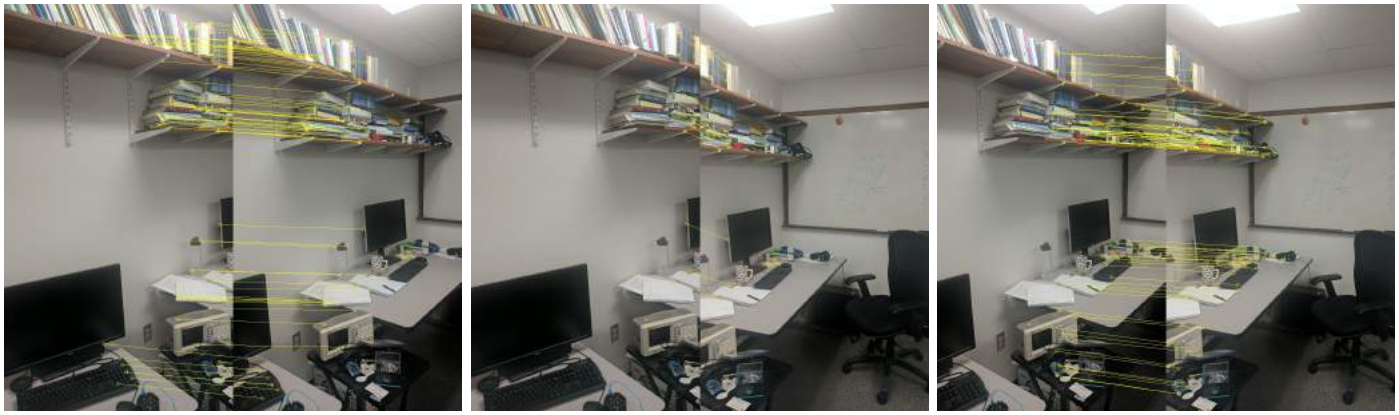
(b) Bad feature matches for Test Set 4 images

Fig. 13. Bad feature matches for Train and Test images. These pairs did not come out of the RANSAC because of the poor feature matches

(a) RANSAC for Train Set 1 images

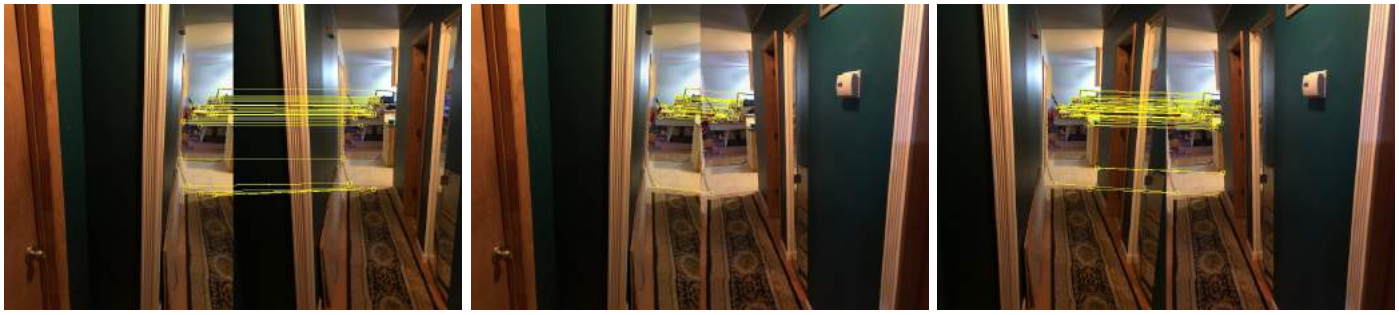
(b) RANSAC for Train Set 2 images, the third pair did not make it through RANSAC

(c) RANSAC for Test Set 3 images, the third pair did not make it through RANSAC



(d) RANSAC for Test Set 4 images, the bad matches for Test Set 4 (Figure 13) did not make it through RANSAC

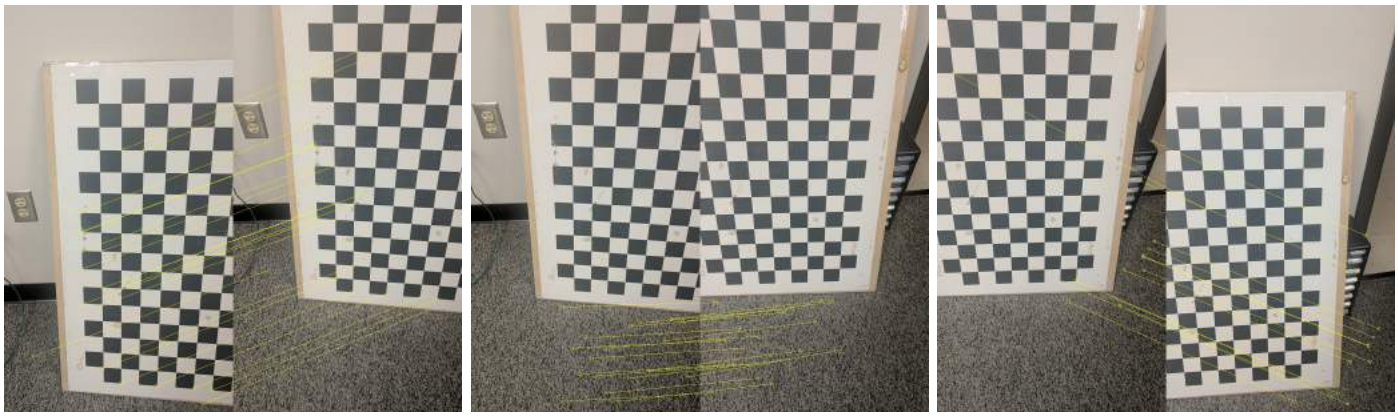
Fig. 14. RANSAC for Train and Test images



(a) RANSAC for Custom Set 1 images

(b) RANSAC for Custom Set 2 images

Fig. 15. RANSAC for custom images

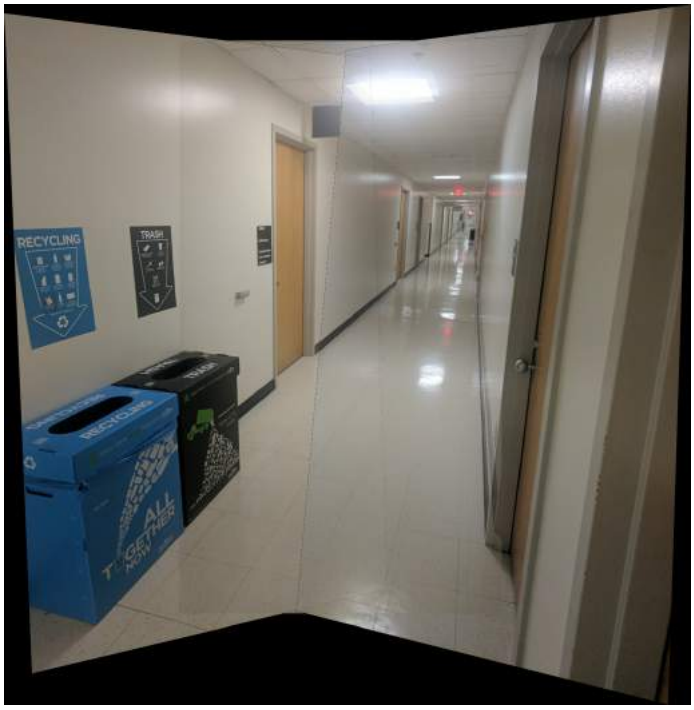


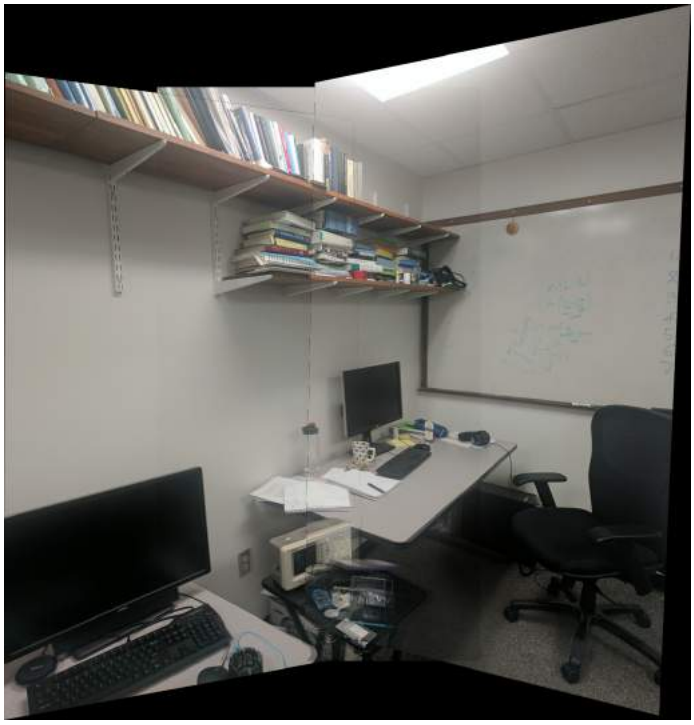
(a) RANSAC for Test Set 1 images

(b) RANSAC for Test Set 2 images

Fig. 16. RANSAC for Test Set images

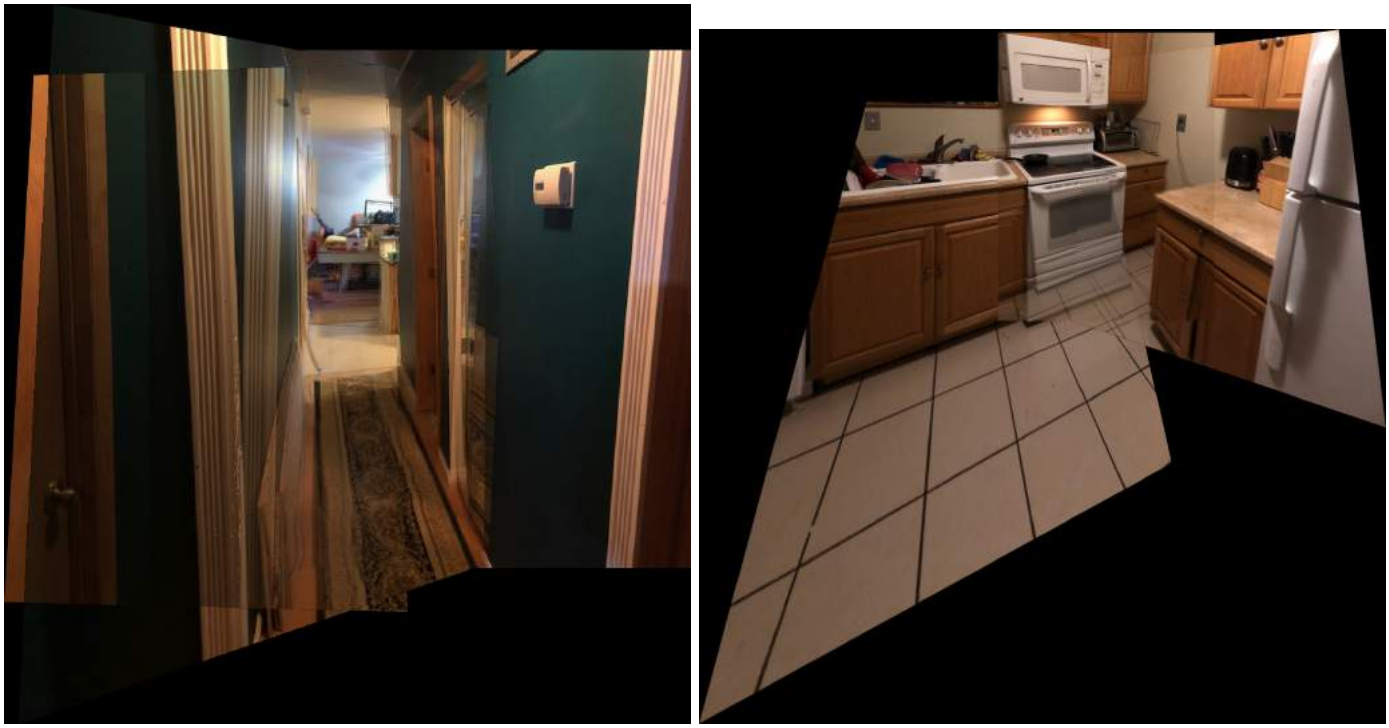
(a) Panoramas for Train Set 1 and Set 2 images





(b) Panoramas for Test Set 3 and Set 4 images

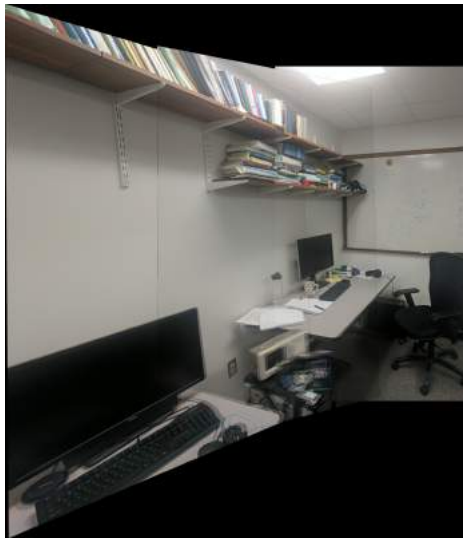
Fig. 17. Panoramas for Train and Test images

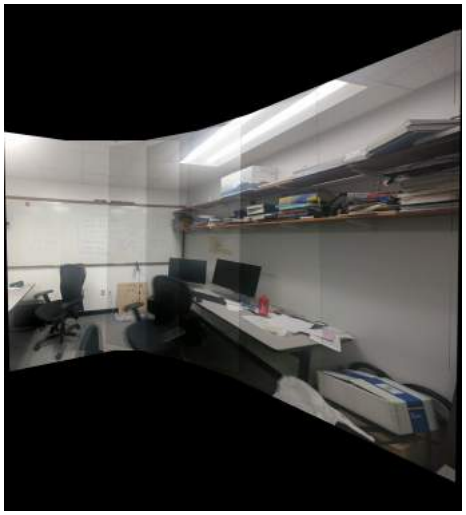


(a) Panoramas for Custom Set 1 and Set 2 images

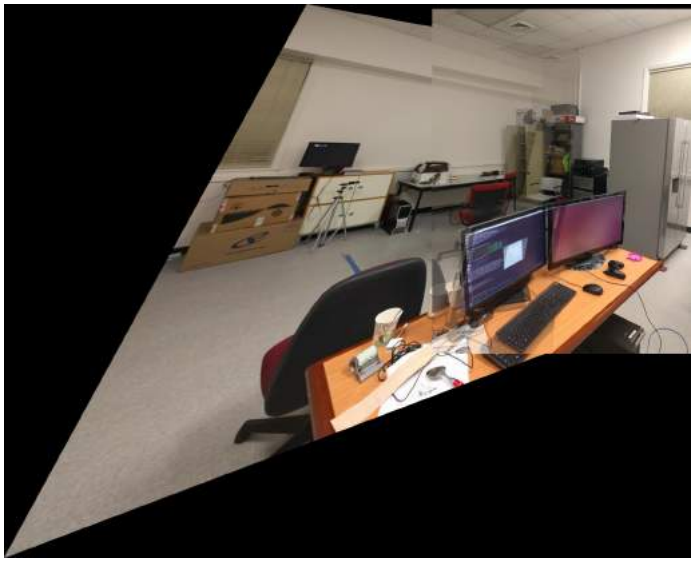
Fig. 18. Panoramas for Custom images

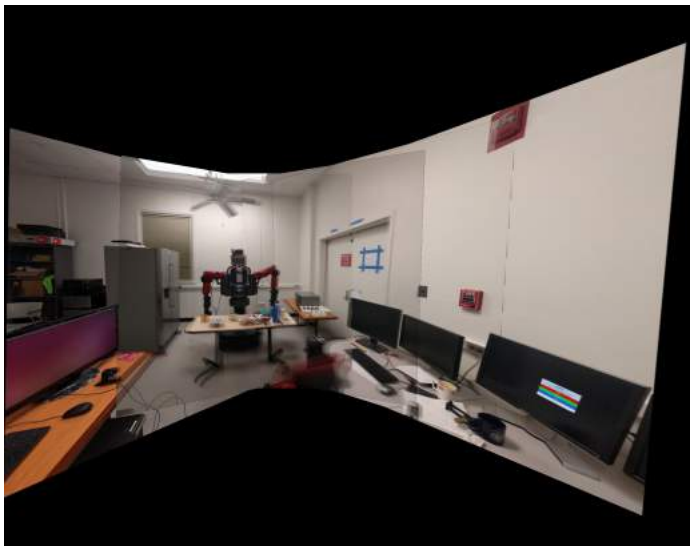
(a) Failed panorama for Test Set 1





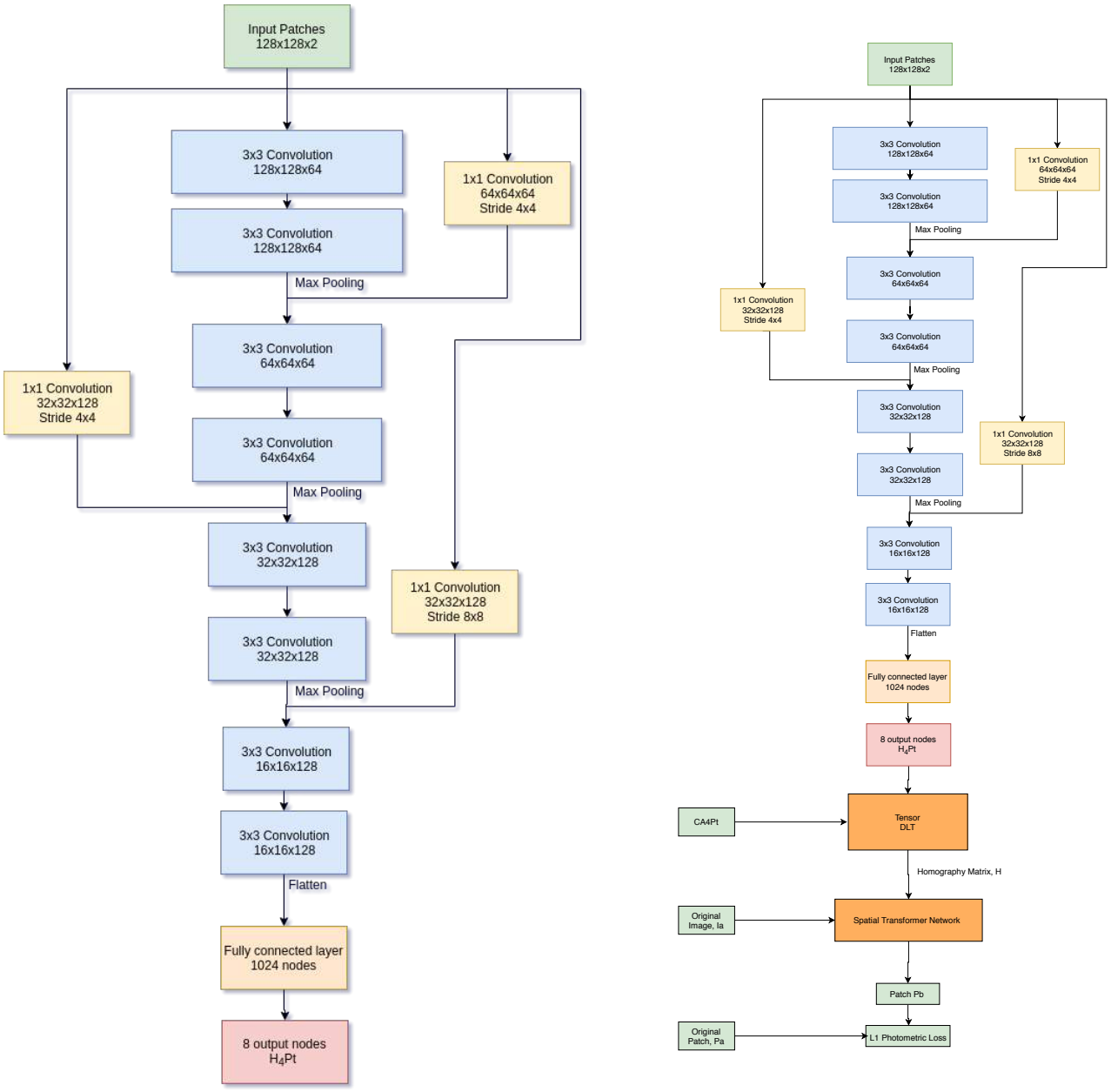
(b) Partial Panoramas for Test Set 2





(c) Partial Panoramas for Train Set 3

Fig. 19. Failed and Partial Panoramas

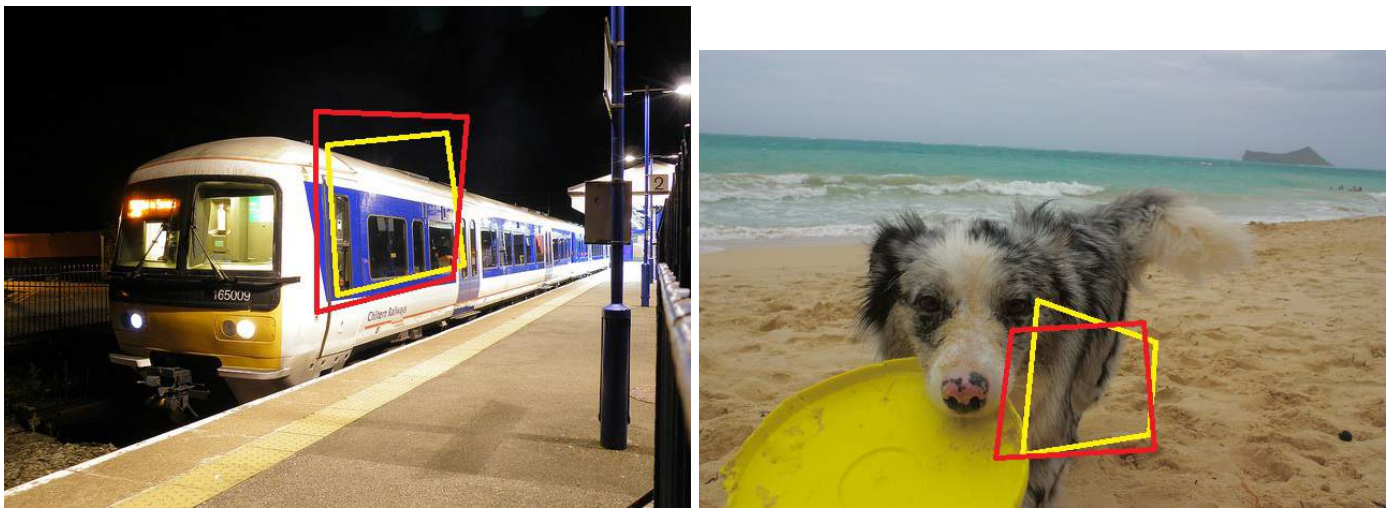


(a) Supervised Network

(b) Unsupervised Network

Fig. 20. Network Architectures

(a) Homography estimation for images from train set



(b) Homography estimation for images from test and validation set

Fig. 21. Actual homography (yellow) and homography estimated (red) by the supervised network

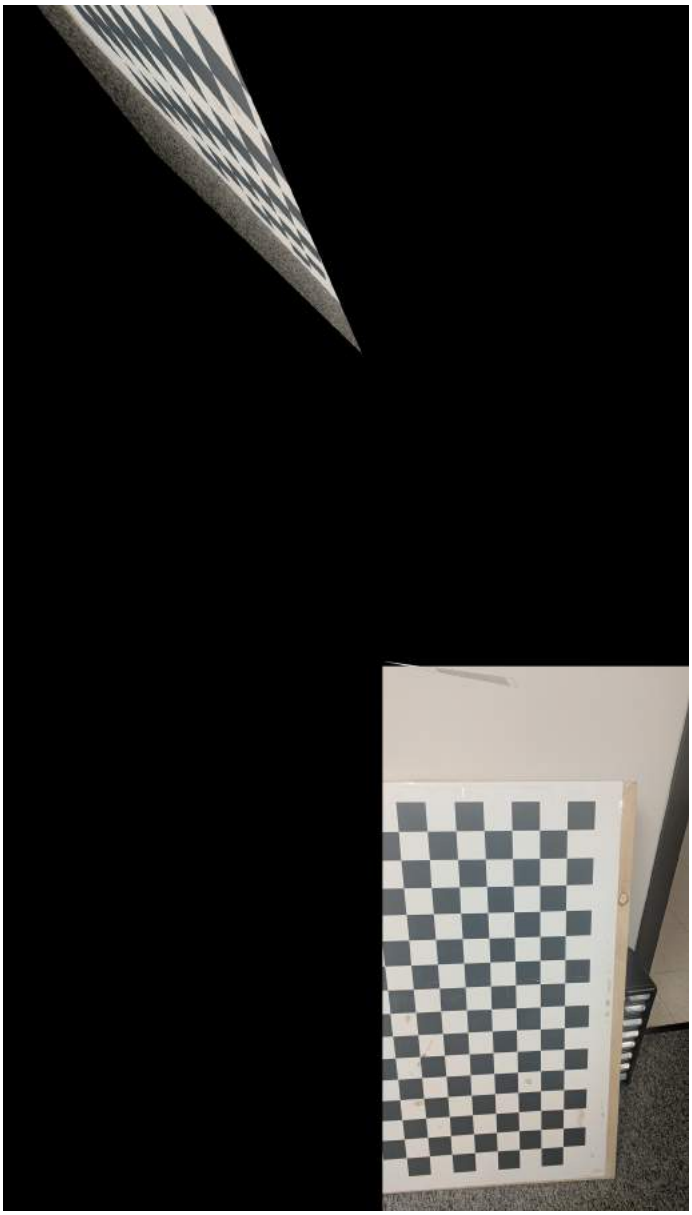


Fig. 22. Panorama of Test Set 1 by Wrapper using supervised model