AutoPano, Project-1
Used two late days

Shreyas Kanjalkar
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, USA
skanjalkar@wpi.edu

Khizar Mohammed Amjed Mohamed
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, USA
kamjedmohamed@wpi.edu

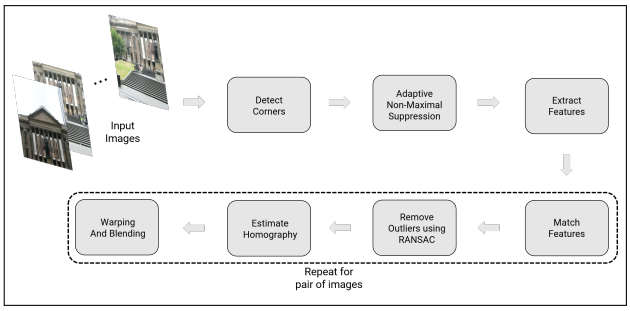

Fig. 1: Flowchart

Fig. 2: ANMS Example

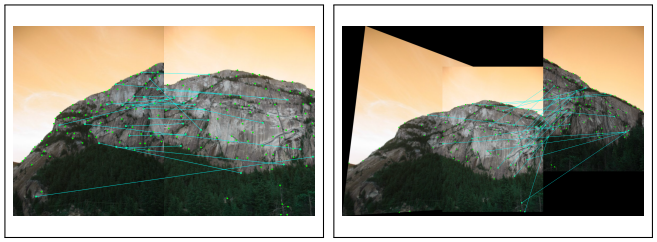

Fig. 3: Feature Matching Example

I. Phase1

A. Introduction

The goal of this project is to stitch given images like a panorama. Each image is made up of pixels, and each pixel has features. The gist of the project is to find these pixels which are repeated in the other images, to match the local features.

B. Data Collection

Each sequence of images should contain at least 3 images, if not more, and each image should have 30% to 50% overlap.

C. Traditional Approach

The traditional approach is shown in Figure 1, is to detect corners. Corners are considered as strong features to match an image. This is because corners have the most change in gradient. They need not necessarily depict the "corner" that we usually interpret. Then we apply Adaptive Non-Maximal Suppression to those corners, which essentially spreads out the corners evenly in the image. We then match features using sum of square distance(ssd) and match only good features using RANSAC. We then estimate the homography and warp and blend the image to finally display the output.

D. Corner Detection

We get the corners by using Corner Harris method in **cv2**. This gives us corner which are above 1% of the strongest corner. This gives us certain number of corners.

E. ANMS

In this algorithm, we attempt to find the N_{best} corners. We find the Cornermetric, which is the Corner Score Image(local maxima). We then iterate through the corners in image, and compare it with every other point in the image. If another point has Corner value greater than the current point, then we take the ssd between those points. We repeat this for all the points, and sort them by the N_{best} ssd in descending order.

F. Feature Descriptor

Now that we have the N_{best} corners, we need to describe each corner with a feature vector. This is called encoding of the information in each corner by a vector. The process is as defined in the assignment.

G. Feature Matching

Now we take the corner points in image1 and image2 that we to stitch. We do a brute force matching between the features of each point using ssd. We take the ratio of best match with the second best match, and if it is below a certain threshold,

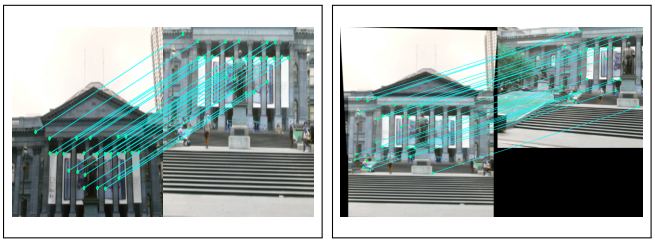

Fig. 4: RANSAC

Fig. 5: RANSAC

which in our case was 0.9 then we add the pair as a feature match. I used the BFMatcher() function in cv2, however I had also implemented it manually. I found that BFMatcher gave better results than manual, which was surprising. Then I used cv2,.drawMatches to visaulize it. Figure 3 shows example of feature matching

H. RANSAC

In feature matching, it can be noted that there are some good matches, but there are some bad matches as well, multiple points matching to one point in the next image. We use RANSAC algorithm to fix this. RANSAC also known as Random Sample Concensus is used to compute homography and remove the bad matches. The algorithm is described in the assignment. When implementing it there were quite a lot of issues. One such is that homography matrix is a 3x3 matrix which you get from using cv2.getPerspectiveTransform(). In order to apply the homography matrix to the entire image to find the number of inliers, we need to convert the image1 to a w*h*dummy. This ensures that the matrix multiplication property is satisfied. We then divide the resulting with the dummy column to normalize it. We then compute the ssd across transformed image and the old image to find the number of inliers. The number of inliers are the number of ssd's less than user set threshold. I found 1e3 to work really well for the given sets and customset that I made.We repeat this until we run out of N iterations or we find some % of inliers. For set 1, around 70% worked, and for set 2 and 3 I had to reduce it to 35% for it to work.

I. Blending

For blending we used a third party code. However, it seems to work well only for up to 3 images. I thought of using pairs of images to create stitches and repeat this until there is only one image left. However, the corners in the stitched image

Fig. 6: Final Stitch Example 1

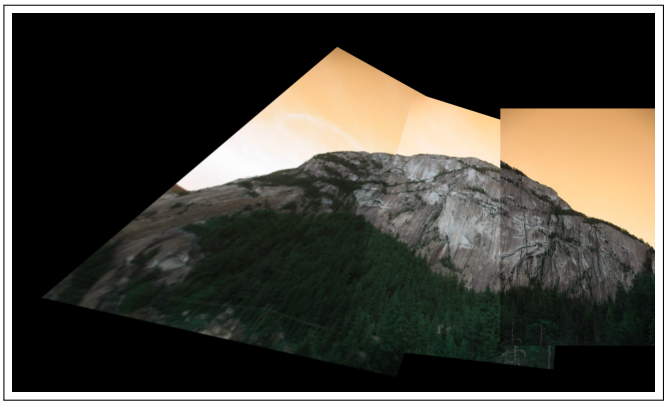

Fig. 7: Final Stitch Example 2

were not as strong and the overlap was not found. I found that I had to tweak the threshold for each iteration which seems rather weird. Perhaps some bug in the code. But it works well for sequences which have 3 images.

J. Custom Set

We tried our algorithm on custom images that we captured while working, and the outputs are in order shown as feature matches, ANMS, RANSAC and Final Stitch. We also tried to stitch the images given in the but we could only stitch the first set. The rest of the images required a lot of fine tuning, and a lot noise was being generated or data was being lost when the images were being stitched, due to introduction of empty black patches.

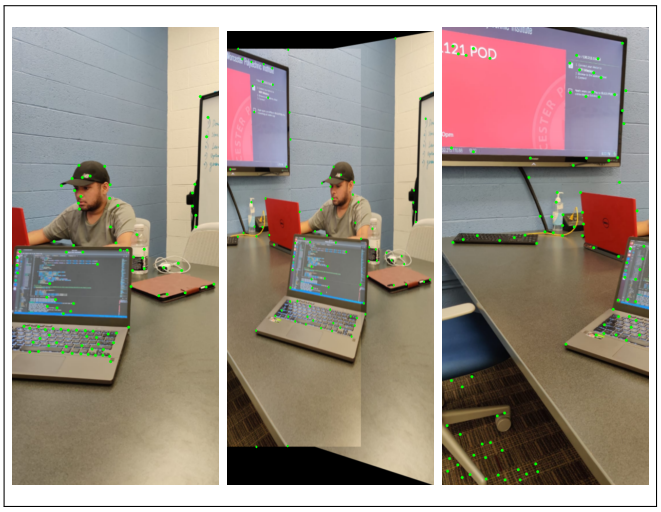

Fig. 8: Custom ANMS

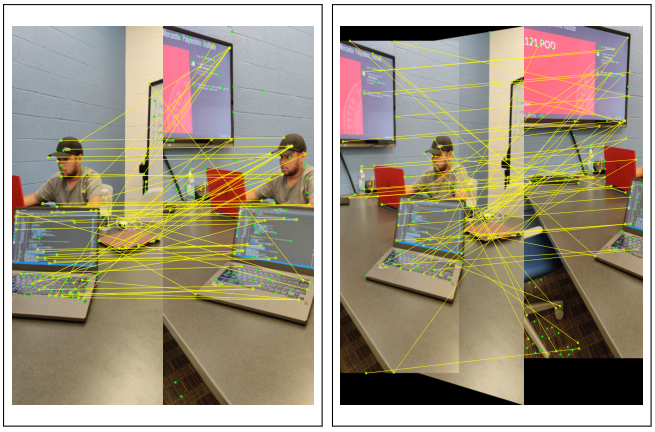

Fig. 9: Feature Matching

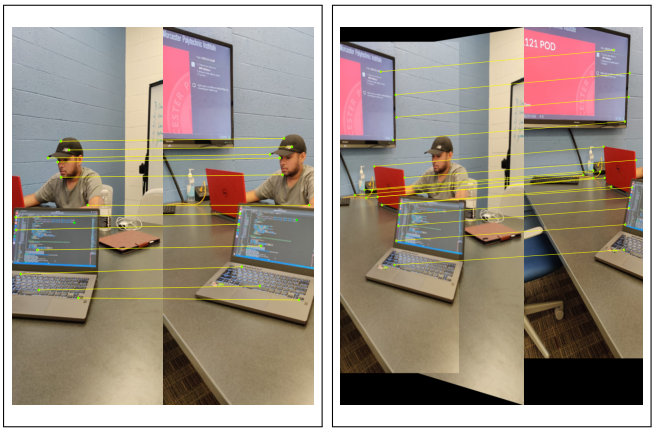

Fig. 10: RANSAC Custom Set

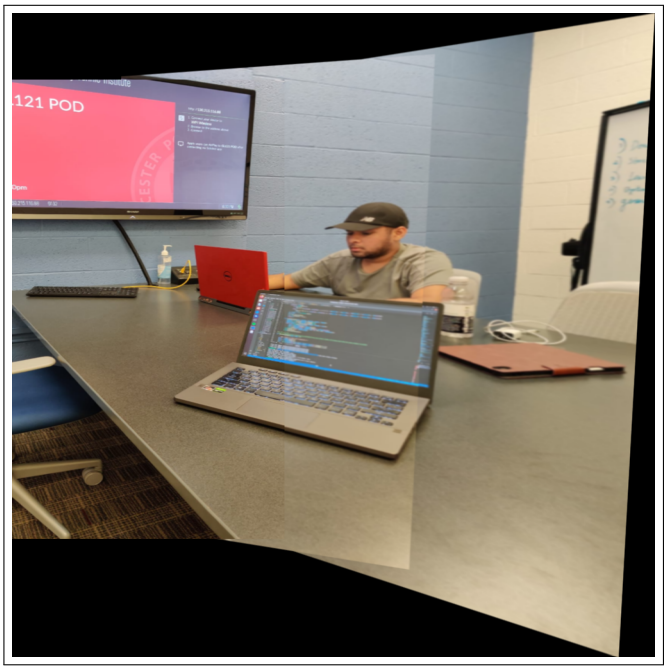

Fig. 11: Final Output

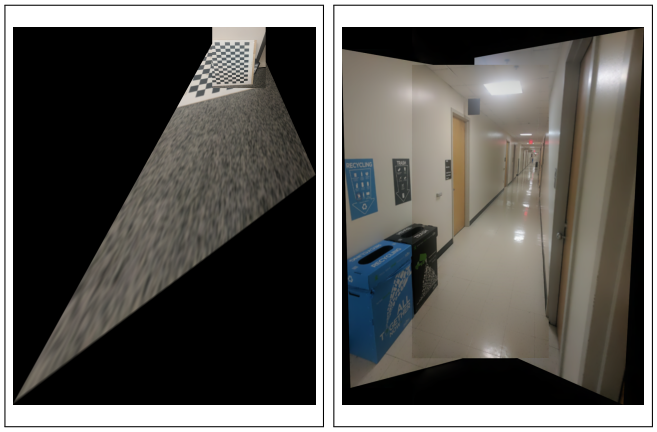

Fig. 12: Final Stitch for Test Set 1 and 3

II. Phase2

A. Supervised Learning

1) Data generation: For the patch generation, we are generating two patches of size 128x128. The first patch is generated by randomly selecting a point in the image and making that the top right corner our patch. The second patch is created by perturbing the four corners of the first patch. Comparing the corners of the first patch and the perturbed corners of the second patch, we find the Homography matrix. We then warp our original image using the inverse of the Homography matrix. The second patch is then generated form the newly created warped image, in the same locations as the first patch. It is important to note that the first patch is selected in such a way that we try avoiding going out of the picture area when we warp it. This is achieved by finding the right

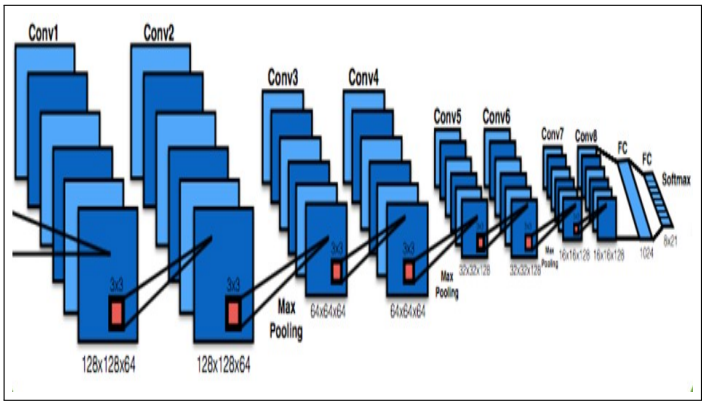

Fig. 13: Homography Network architecture

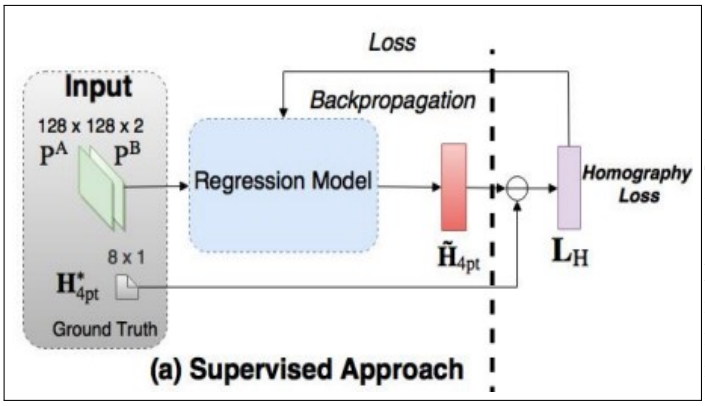

Fig. 14: Homography Network architecture

patch size and the right perturbation maximum. Using a high perturbation maximum resulted in losing pixels for the second patch. Therefore we selected a perturbation maximum of 16. We also select the first patch randomly from an area nearer to the center of the image. The smaller the patch size the more freedom we get in random selection of the location of the patch. However a smaller patch size will not help the network learn well, since there will be lesser warp seen. Therefore we settled with a patch size of 128x128. Through this we were mostly able to avoid losing information from the image when we warp.

2) Network architecture: Once the two patches are generated, we stack them together and pass it through our network. The network we have used is similar to (1). We also use a 4 point parameterized homography called H4pt rather than the using the actual transformation matrix H which is of 3x3 dimensions. The H4pt matrix is of the form 1x8 and is just the difference between the corner locations of patch 1 and patch 2. . The network architecture used is similar to VGGNet and is depicted in *Figure 13*. The output of this network is a predicted H4pt. We use a L2 norm loss and we use an adam optimizer.

3) Implementation: In the given code, the GenerateBatch function does not guarantee it will span the entire data set since we are randomly picking an image. It is possible that it might pick the same image, although very unlikely but in order to

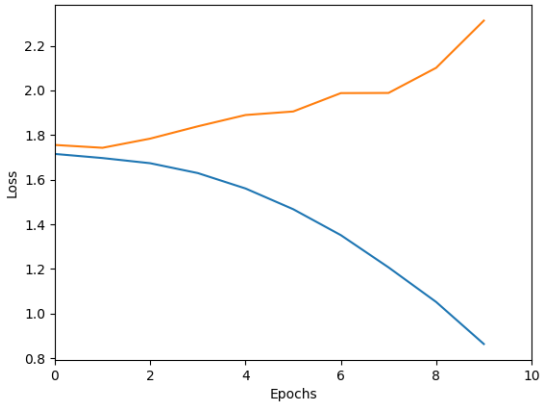

Fig. 15: SGD, Overfitting

change that, We used a generator and shuffled the array every time we are generating the batch to create a random sequence. This ensures that entire data set is spanned and maintains the last index it was working with the array.

We were initially trying to run it on a local machine which has a GTX 1650, but We were running out of memory for mini batch size of 1 too. In order to fix this, We had to make few changes to the given code. We modified the network to not return a dictionary and ran the code on Google Colab. The ipynb file is also in the code. Please run that file in order to run the code. This saved a lot of memory as dictionary allocates extra space because it can contain any type of data set and since we are only returning the loss from it it, we calculate the loss from the delta that we get from the model inside the iteration it self.

We also made some changes to the validation step. We combined the train and validation data set and instead of doing training and testing separately for each data set, we randomly shuffle the data set every time to guarantee randomizing data set. We then are training 80% of the data and using 20% of the data to validate the data set. Initially, using the SGD optimizer, without any weight decay I noticed that the model was over fitting a lot. This means that there is a lot of noise in the model. I used AdamW optimizer and added dropout layer with a probability of 0.1 to the network to try to smoothen the noise. The results are shown in the plot.

This showed consistency with training and validation loss. Since they were almost nearly equal after epochs and not deviating much. So we implemented this model on google colab with a batch size of 32 for 6000 images. We then tried to visualize using this model for the test set. However, the testing showed really weird results as shown in the figure.

B. Unsupervised Learning

1) Data generation: In unsupervised learning we use the entire Supervised learning implementation. The data generation and the architecture is the same. However after the neural

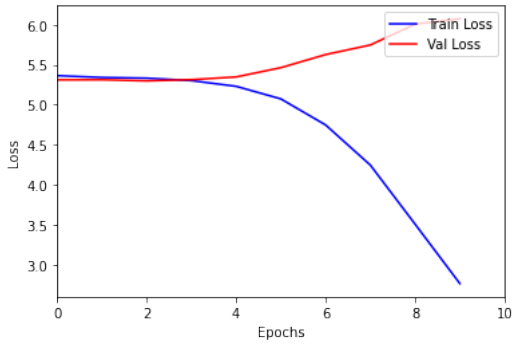

Fig. 16: AdamW, no dropout layers

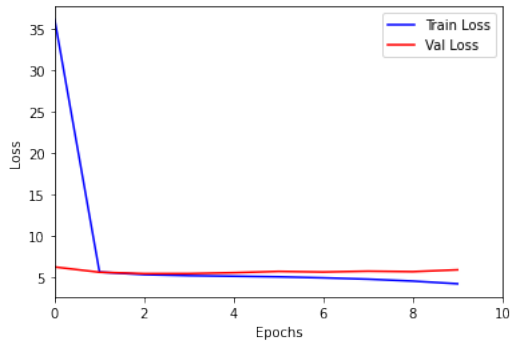

Fig. 17: AdamW with dropout layers

network implementation we have a few additional functions which help in computing the loss.

2) Architecture: We have used the architecture in (2) in our project. Figure 18 explains our architecture. As mentioned in the above paragraph, the architecture used is very similar to the supervised learning part. In addition we have two more components called the DLT(Direct Linear Transform) and STN(SPatial tranformer network).

3) Implementation:

4) DLT: DLT takes in the H4pt we get from the supervised part and the location of corners from of patch 1 from our data generation. Using this we calculate the predicted corners of patch 2. This is then used to calculate the 3x3 Homography matrix. The authors in (2) discuss a method to calculate the

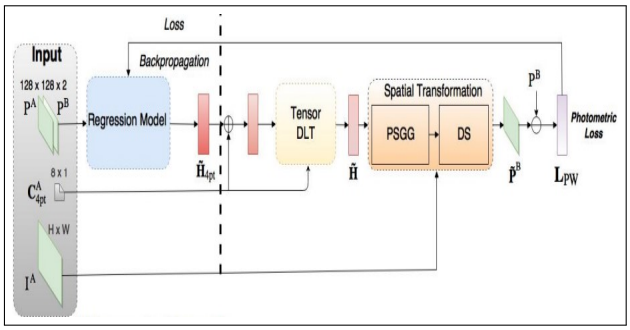

Fig. 18: Unsupervised network

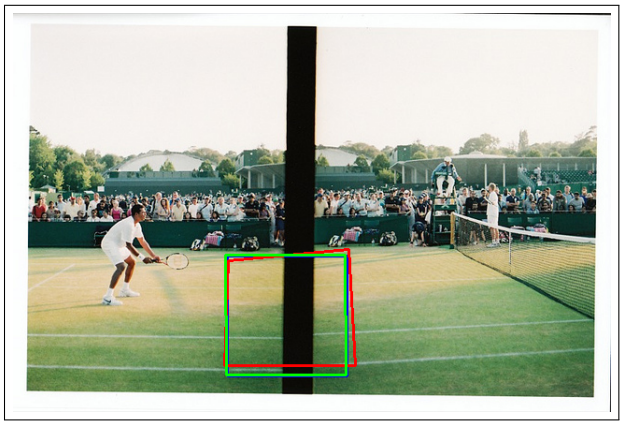

Fig. 19: Visualization

DLT. Consider the equation $x' = H * x$. Here x=[u,v,1] where u and v are the x and y coordinates of a point. As seen in (2) we can solve for H in $\hat{A}_i * \hat{H} = \hat{b_i}$. Here are the first 8 elements of H written in a 8x1 shape. *i* represents the corner number. In our case i=1,2,3,4. Through this we can solve for the

$$\hat{A}_i = \begin{bmatrix} 0 & 0 & 0 & -u_i & -v_i & -1 & v'_i u_i & v'_i v_i \\ u_i & v_i & 1 & 0 & 0 & 0 & -u'_i u_i & -u'_i v_i \end{bmatrix}$$

$$\hat{b_i} = \begin{bmatrix} -v'_i \\ v'_i \end{bmatrix}$$

We noted that the (2) solved for H using auxiliary matrices. This method requires initializing 10 sets of auxiliary matrices for this purpose. We have instead used for loops to iterate through all the instances in the minibatch and iterate through all the corners.Although easy to implement, this method is computationally expensive.

5) STN: The spatial uses the output of the DLT, the 3x3 H matrix and the patch 1 to predict the patch 2. We have used the kornia library to warp patch 1. Once warped, the photometric loss is computed by comparing the predicted patch and the actual patch

C. RESULTS

The model that we trained, showed good results for loss in both training and validation and testing. However, when we tried to visualize it, we were not able to see the same effect. We have a strong suspicion that it is trying to learn the wrong patch and hence it has bad h_4pt. The figure shows the same. (2)

REFERENCES

[1] D. DeTone, T. Malisiewicz, and A. Rabinovich, "Deep image homography estimation," *arXiv preprint arXiv:1606.03798*, 2016.

[2] T. Nguyen, S. W. Chen, S. S. Shivakumar, C. J. Taylor, and V. Kumar, "Unsupervised deep homography: A fast and robust homography estimation model," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2346–2353, 2018.