Image Matching and 3D Reconstruction

DATS 6303

Deep Learning-Final Project

Individual Report- Tanmay Ambegaokar

I. Introduction

1.1 Overview

We have chosen the problem of image matching for the final project, where we will be reconstructing 3D images from a set of regular 2D images. This challenge intrigues us due to its practical applications in various domains such as augmented reality, 3D reconstruction, and navigation systems. Image matching is a cornerstone for understanding and interpreting the world visually through machines, making it an essential task in computer vision.

To go from a 2D image to a 3D view requires a lot of steps. First, we must extract features of the image and make sure that there are similar features in different images. For this, we make image pairs and compare similar features between them. Next, we must extract keypoints and descriptors of the image. Keypoints in an image are specific locations in an image which are invariant to image scaling, rotation, and sometimes lighting. Once keypoints are identified, descriptors are used to encode the patches around the keypoint to a high dimensional vector. The last step is to match these keypoints and descriptors across various images. Once this is done, we can reconstruct these 3d points using Python's COLMAP library.

As we were researching for image matching techniques, we came across a Kaggle Competition hosted by Google. The dataset can be found here. The dataset has scenes from various different landmarks. All image files of 12.3 GB. Each scene has close to 500-600 images.

1.2 Shared Work

Parv: LightGLUE for matching keypoints & Bash scripts

Sajan: ALIKED for extracting keypoints & COLMAP 3D Reconstruction

Tanmay: DINOv2 for getting matching image pairs and Streamlit App

II. Individual Work

2.1 Overview of Individual Work

The first task for me was to find a way to get a pair of matching images. This is a crucial phase because accurate reconstruction of a landmark or scene relies on analyzing images that are visually similar. For example, if we are reconstructing the Lincoln Memorial, we want to get keypoints and descriptors of only those images which have Lincoln Memorial. We do not want a matching pair for a picture of Lincoln Memorial and a picture of Mars because that will interfere with the precision of the reconstruction process.

I read up on various Kaggle Notebooks as well as research papers on how to go ahead with image matching. Some of the methods mentioned were SIFT and DELF. I found Meta's Vision Transformer DINO and read up on their paper to understand how their transformer is used to obtain normalized image embedding. DINO uses a student-teacher network where the student model tries to replicate the output of the teacher.

The first step in this architecture is that resizes the image to a fixed size. Then it makes patches of the image. The student and the teacher are fed the same image with different augmentations. The student model is trained using SGD to mimic the outcome of the teacher based on different augmentations. Thus, the model learns to extract important keypoints from the image in its embedding. For our case, we extract these embeddings from the last layer of the model.

The second task I did was to visualize the extracted keypoints we got of each image. And lastly, I made majority of the streamlit app. I found pretty good templates on this site. I got design ideas from here and started implemented for our app.

2.2 Explanation

1. ***Getting Image Pairs by obtaining normalized embeddings***

   We first set the path of the dataset. From our dataset, we select a scene which we want to reconstruct. We create a list of these images. We also download the model weights of dinov2 and set the path for this.

```
# Download and extract DinoV2 model checkpoints
echo "Downloading and extracting DinoV2 model checkpoints..."
kaggle models instances versions download metaresearch/dinov2/pyTorch/base/1
mkdir -p dinov2/pytorch/base/1
tar -xzf dinov2.tar.gz -C dinov2/pytorch/base/1
```

Fig 1: Downloading Dinov2 weights and setting the path.

Once this is done, we pass the image list and the path to the dinov2 weights to a function to get normalized embeddings and image pairs.
In this function, the first thing I did was to get a list of all exhaustive image pairs. For example, if the image list from the dataset we selected has 100 images, we get a list of all exhaustive image pairs.

```
def get_pairs_exhaustive(lst: list[Any]) -> list[tuple[int, int]]:
    """Obtains all possible index pairs of a list"""
    return list(itertools.combinations(range(len(lst)), r: 2))
```

Fig 2: Code to get exhaustive images.

Now that we have all exhaustive image pairs. We get the normalized embeddings by loading the model from the dino path. Each image pair is looped over while converting the image to a tensor after doing preprocessing the image. The processed image is then forwarded to the output. From the last layer of the model, we get the embeddings as mentioned before. We also perform L2 normalization.

```
for i, path in tqdm(enumerate(paths), desc="Global descriptors"):
    image = load_torch_image(path)

    with torch.inference_mode():
        inputs = processor(images=image, return_tensors="pt", do_rescale=False).to(device)
        outputs = model(**inputs)  # last_hidden_state and pooled


        embedding = F.normalize(outputs.last_hidden_state[:, 1:].max(dim=1)[0], dim=-1, p=2)

    embeddings.append(embedding.detach().cpu())
return torch.cat(embeddings, dim=0)
```

Fig 3: Code to get embeddings.

The last step is to compute the distance between each pairs' embedded tensors. We calculate this by using the Euclidian distance between the pairs.

```
distances = torch.cdist(embeddings, embeddings, p=p)
```

Fig 4: Code to get the Euclidian distance.

Here, I tried changing the p value to 1 (Manhattan Distance/ L1 Norm) and to inf (Chebyshev Distance). We also set a threshold value which we can change as per our convenience. If this distance is above the threshold, we

do not keep that image pair. This is how we have achieved image matching pairs using a vision transformer.

2. ***Streamlit App***

Got some template ideas from the internet and set up the base app running. Added sidebar, functionalities to load a custom or to upload the images the user wants. Did extensive error handling so that if the user gets an error if no images are uploaded or if the user skips a step. For example, the user should get an error message if after uploading the images, the user goes directly to match keypoints before extracting the keypoints. Made us of streamlits' session state function to keep track of all the selections the user makes to control the flow of the app.

```
st.session_state['images_list'] = images_list
st.session_state['scene'] = scene
st.session_state['dataset'] = dataset
st.session_state['path'] = path
```

Fig 5: Code to set session states.

```
if 'images_list' not in st.session_state or not st.session_state['images_list']:
    st.error("Image list is not available.")
    return
```

Fig 6: Code to get error messages.

As and when I encountered any traceback errors or path issues, I kept on adding more try catch exceptions.

```
try:
    selected_pair_index = st.selectbox("Select Image Pair to Visualize:", range(len(index_pairs)),
                                       format_func=lambda x: index_pairs[x])
    idx1, idx2 = index_pairs[selected_pair_index]
    visualize_matches(images_list, idx1, idx2, feature_dir)
    st.session_state['sparse'] = 'sparse'
except IndexError as e:
    st.error(f"Selected index out of range: {e}")
except Exception as e:
    st.error(f"An error occurred: {e}")
```

Fig 7: Code with try catch exceptions to handle errors.

2.3 Results

1. *Index pairs using ViT.*

I got all the exhaustive index pairs first. To get the best possible number of matches, I played around with all these parameters to see how if the number of matches changed or not.

```
similarity_threshold: float = 0.6,
tolerance: int = 1000,
min_matches: int = 20,
exhaustive_if_less: int = 20,
p: float = 2.0,
```

Fig 9: Parameters to get different matches.

If we increase the similarity threshold, the chances of wrong images getting matched together increases whereas if we set it too low, the chances of no images getting matched increases. Thus, I had to find a balance between the two.

Fig 10: Matching Images Visualization

2. *Streamlit*

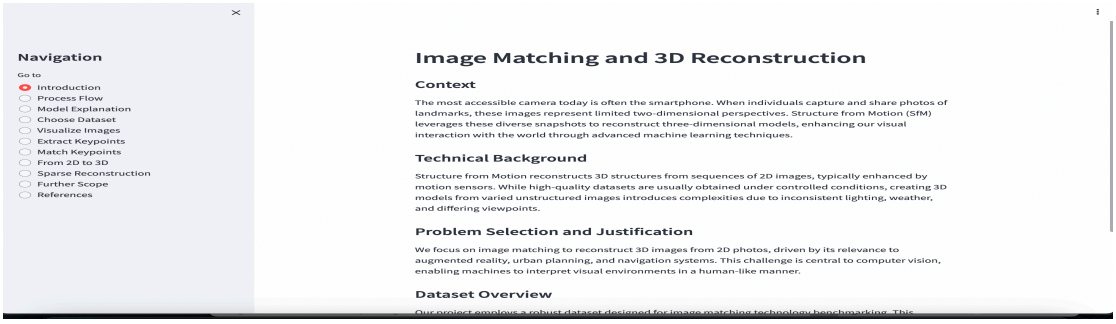
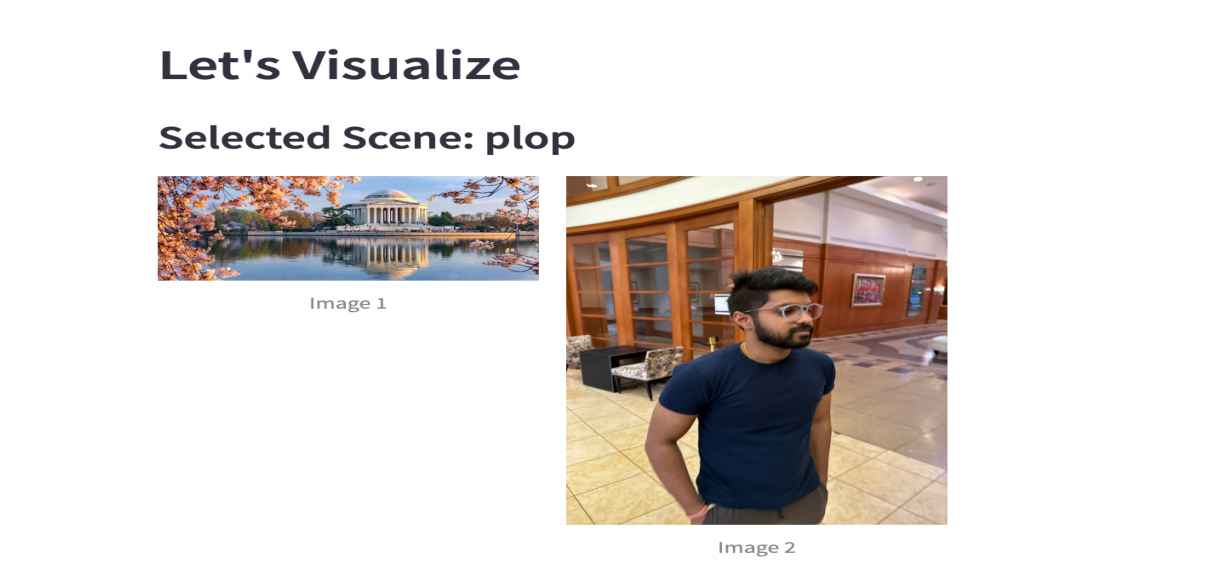
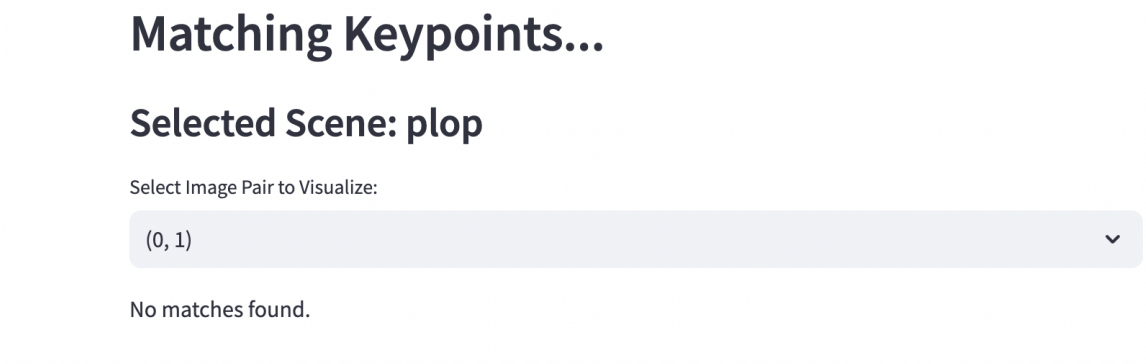


Fig 11: Streamlit landing Page.

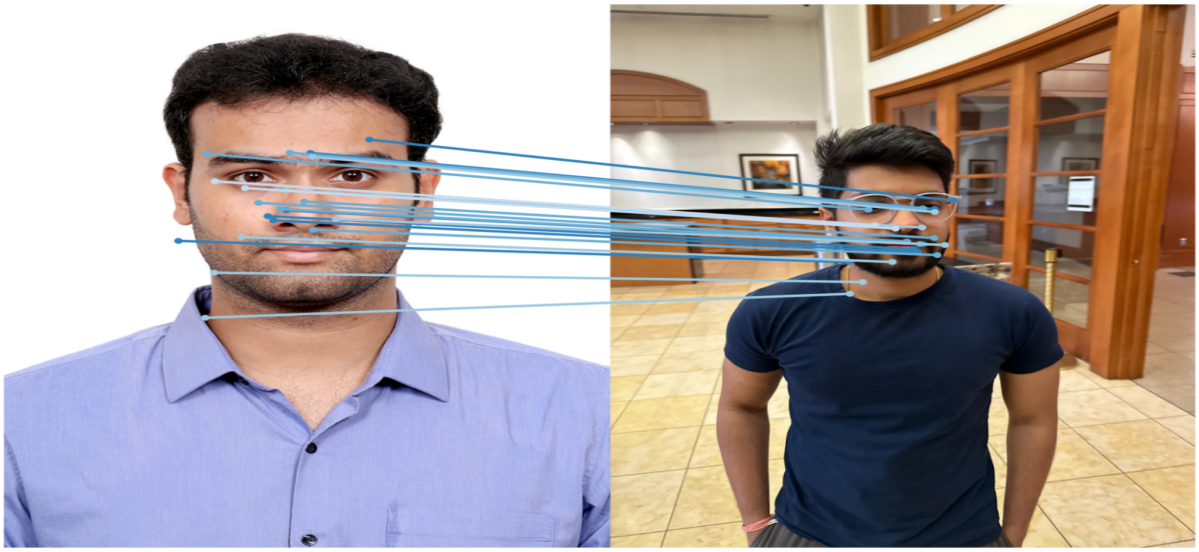
The user can also upload images as per the users' convenience. To show how image matching with different images works, I have uploaded two pictures of different scenes.



The transformer is fed with patches with different augmentations of both the images and then the embedded output's distance is calculated between the two images. For this scene, all the colors and edges are totally different, so no feature should be matched. Which is exactly the case.



I also tried uploading 2 pictures of different people. Our model tried matching the image based on the spatial edges of the noses and ears etc.



It excludes all the keypoints of the background but tries to match the features of the face based on the nose edges, color of the skin, jaws.

III. Summary and Conclusion

My key findings and learnings from this project were:

- Thoroughly understood how a transformer works. Not just the architecture but the difference between multi-head attention and self-attention. Understood how different augmentations are fed as input to the model and how it extracts features. In our DINOv2's case, it employs PCA for feature extraction and each patch is color coded with a color. Similar looking patches across different images are then compared and matched if the region around the patch has similar patch embeddings.
- Understood how streamlit works. Put myself in the shoes of the user and kept on user testing to remove any traceback issues. Encountered as many issues as possible while trying different inputs or going to the wrong navigation page. Put in a lot of try catch clauses for error handling.
- Since our project was mostly inferencing, for future work I would like to write my own custom transformer to get image matching embedding.

IV. Code Percentage: 69.13%

V. References:

- P. Lindenberger, P. -E. Sarlin and M. Pollefeys, "LightGlue: Local Feature Matching at Light Speed," 2023 IEEE/CVF International Conference on Computer Vision (ICCV), Paris, France, 2023, pp. 17581-17592, doi: 10.1109/ICCV51070.2023.01616. keywords: {Adaptation models;Visualization;Technological innovation;Codes;Three-dimensional displays;Computational modeling;Memory management} (https://ieeexplore.ieee.org/document/10377620)

- Oquab, Maxime & Darcet, Timothée & Moutakanni, Théo & Vo, Huy & Szafraniec, Marc & Khalidov, Vasil & Fernandez, Pierre & Haziza, Daniel & Massa, Francisco & El-Nouby, Alaaeldin & Assran, Mahmoud & Ballas, Nicolas & Galuba, Wojciech & Howes, Russell & Huang, Po-Yao & Li, Shang-Wen & Misra, Ishan & Rabbat, Michael & Sharma, Vasu & Bojanowski, Piotr. (2023). DINOv2: Learning Robust Visual Features without Supervision. (https://www.researchgate.net/publication/370058767_DINOv2_Learning_Robust_Visual_Features_without_Supervision)
- J. Dai et al., "Deformable Convolutional Networks," 2017 IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 2017, pp. 764-773, doi: 10.1109/ICCV.2017.89. keywords: {Convolution;Kernel;Object detection;Standards;Feature extraction;Two dimensional displays}, (https://ieeexplore.ieee.org/document/8237351)
- Zhao, Xiaoming & Wu, Xingming & Chen, Weihai & Chen, Peter C. Y. & Xu, Qingsong & Li, Z.G.. (2023). ALIKED: A Lighter Keypoint and Descriptor Extraction Network via Deformable Transformation. (https://arxiv.org/abs/2304.03608)
- https://www.kaggle.com/code/asarvazyan/imc-understanding-the-baseline#Sparse-Reconstruction