Stereo Image Matching

Ayush Garg

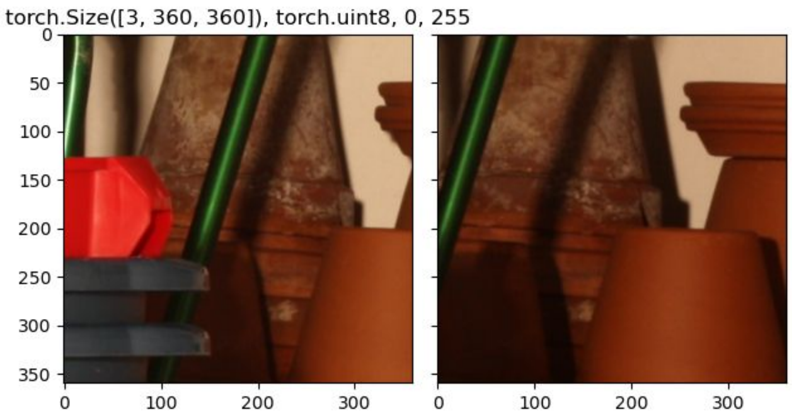
Overview

- Introduction
- Methods: Architecture
- Methods: Loss function
- Experiments
- Improvements
- Discussion

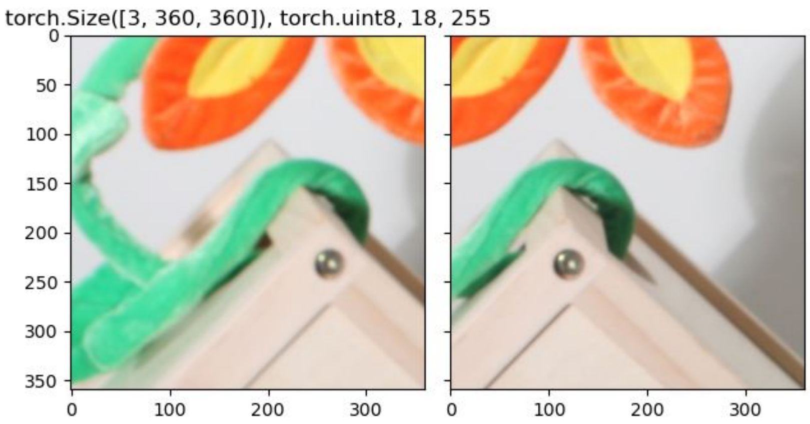
Introduction

Problem:

The matching between stereo-image pairs in a dataset, has been lost for some reason. How can we match the image pairs without using metadata like filenames / disparity maps?



Test sample A

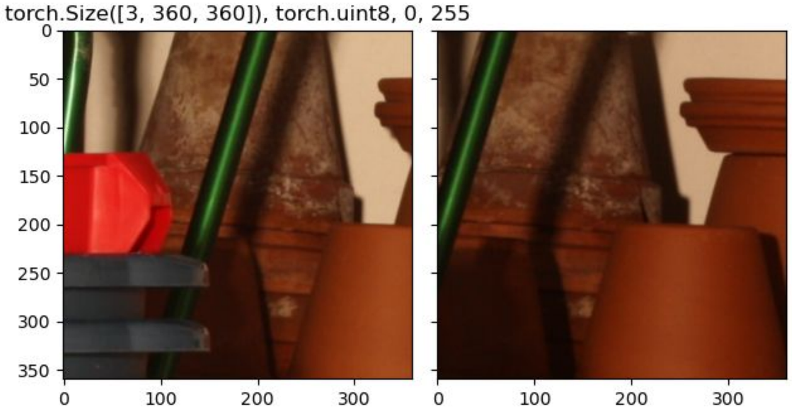


Test sample B

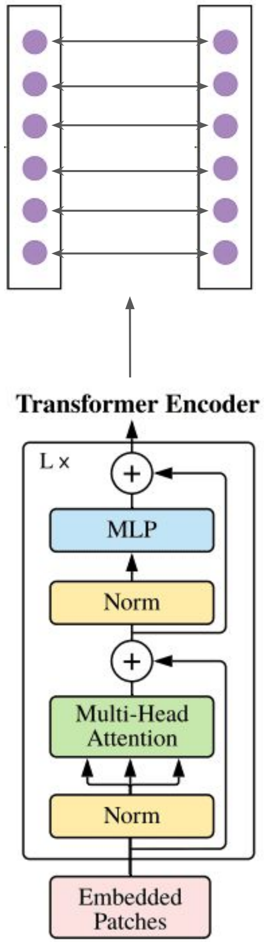
Introduction

Possible solution:

- Use an image transformer to generate embeddings for all images.
- Use a similarity metric to match the embeddings into pairs



Matching using a similarity metric



Embeddings

Image transformer

Transformer image from: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, Dosovitskiy et al.

Introduction

Transformer

- Initially introduced as a sequence transduction model that does away with recurrent or convolutional neural networks using **“self-attention”**.
- The idea is to learn **contextual embeddings** of the sentence by allowing each token/word to “attend” to any other token in the sentence.
- Such a contextual embedding is useful in learning **long-range information** within a sentence without recurrence and is helpful in many applications that require outputs based on the context of the input.
- The model achieved SOTA performance in many NLP tasks ranging from language-translation to text summarization and QnA.

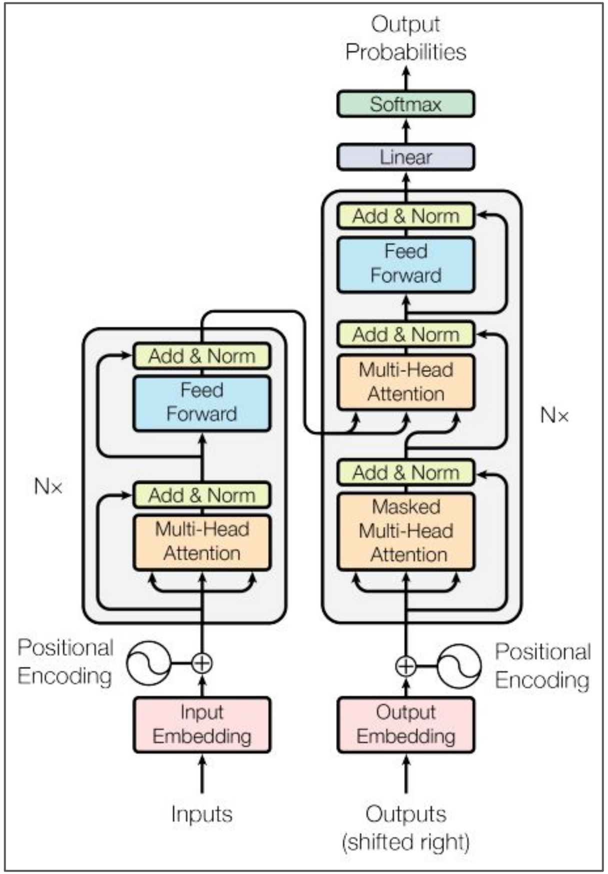
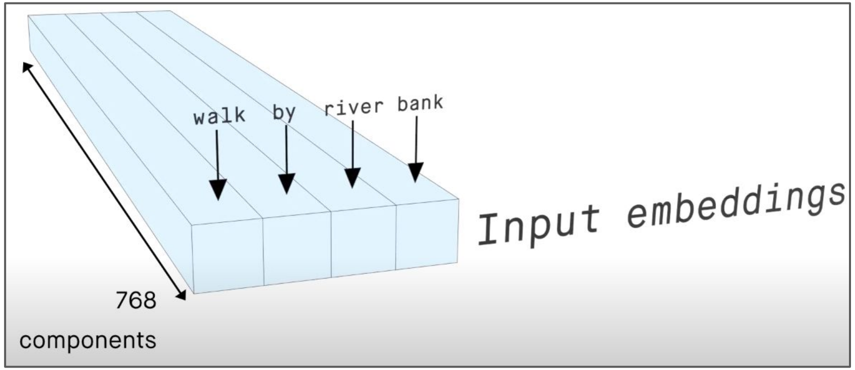


Image from: Attention Is All You Need, Vaswani et al. 2017

Self-attention mechanism

In order to create **contextual** embeddings, the dot product of the embedding of each token is calculated.

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{D_k}}\right)\mathbf{V} = \mathbf{A}\mathbf{V}$$



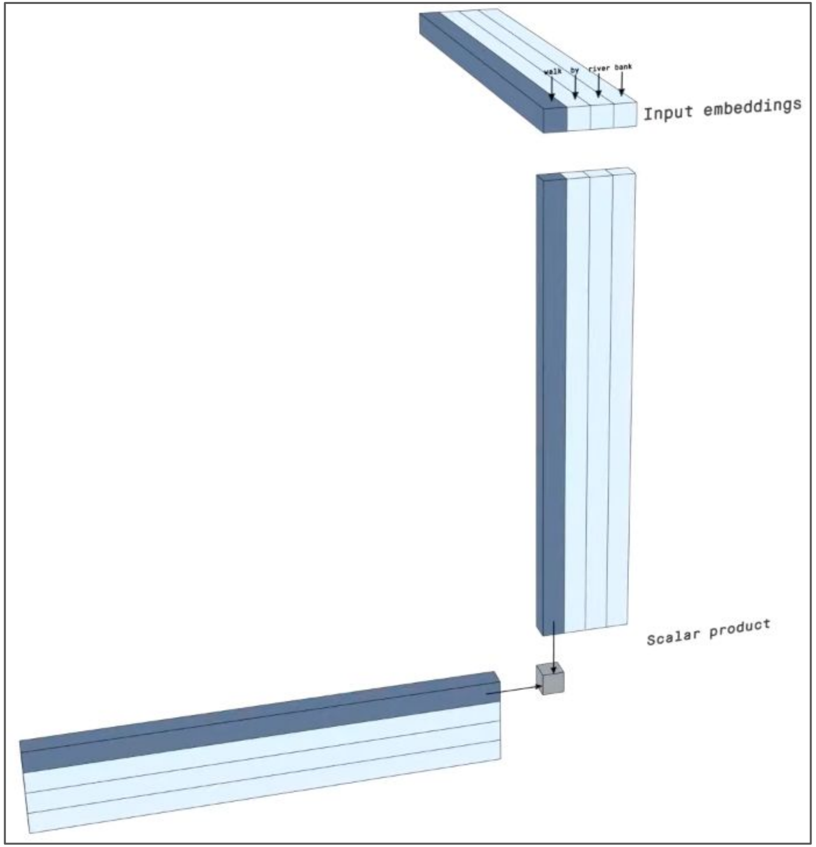


Image from: How to get meaning from text with language model BERT, Peltarion

Self-attention mechanism

- This scalar product is a measure of the **similarity or correlation** between the embeddings.
- For each embedding, a softmax is taken to **squash or amplify** the contributions of the scalar products, and also to **normalize** the scores to 1.

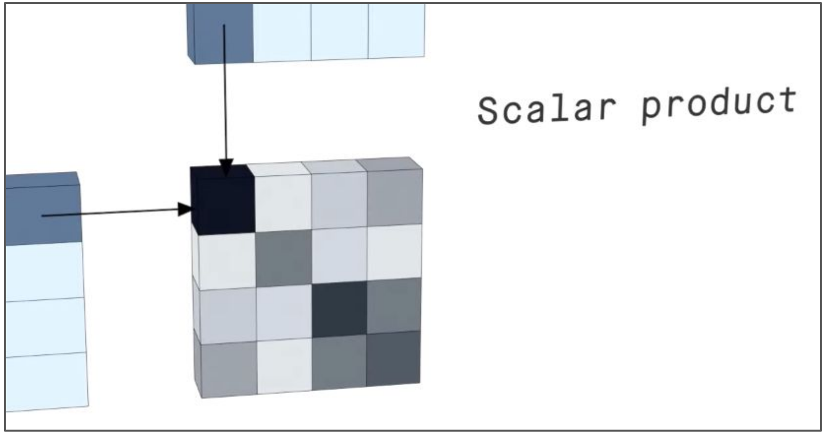
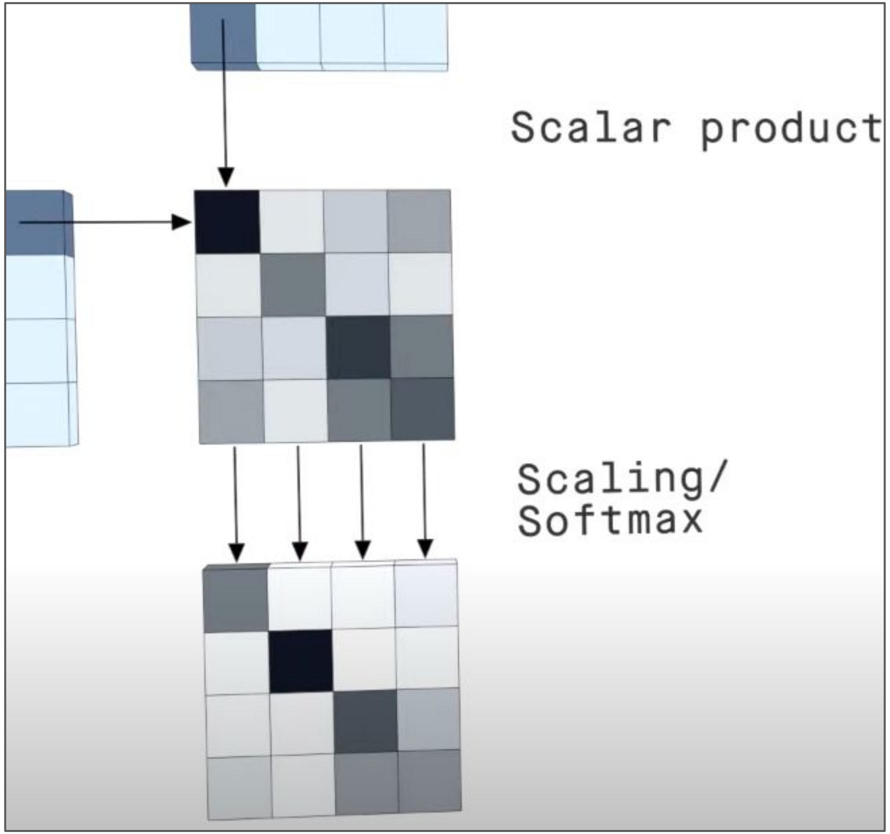


Image from: How to get meaning from text with language model BERT, Peltarion



Self-attention mechanism

Finally, the embedding for each token is recalculated using a **linear combination** of the softmax scores, and hence the new embeddings are **contextualized** with contributions from other embeddings.

Position-wise feedforward networks further process these embeddings to create higher level features by mixing different dimensions within a vector.

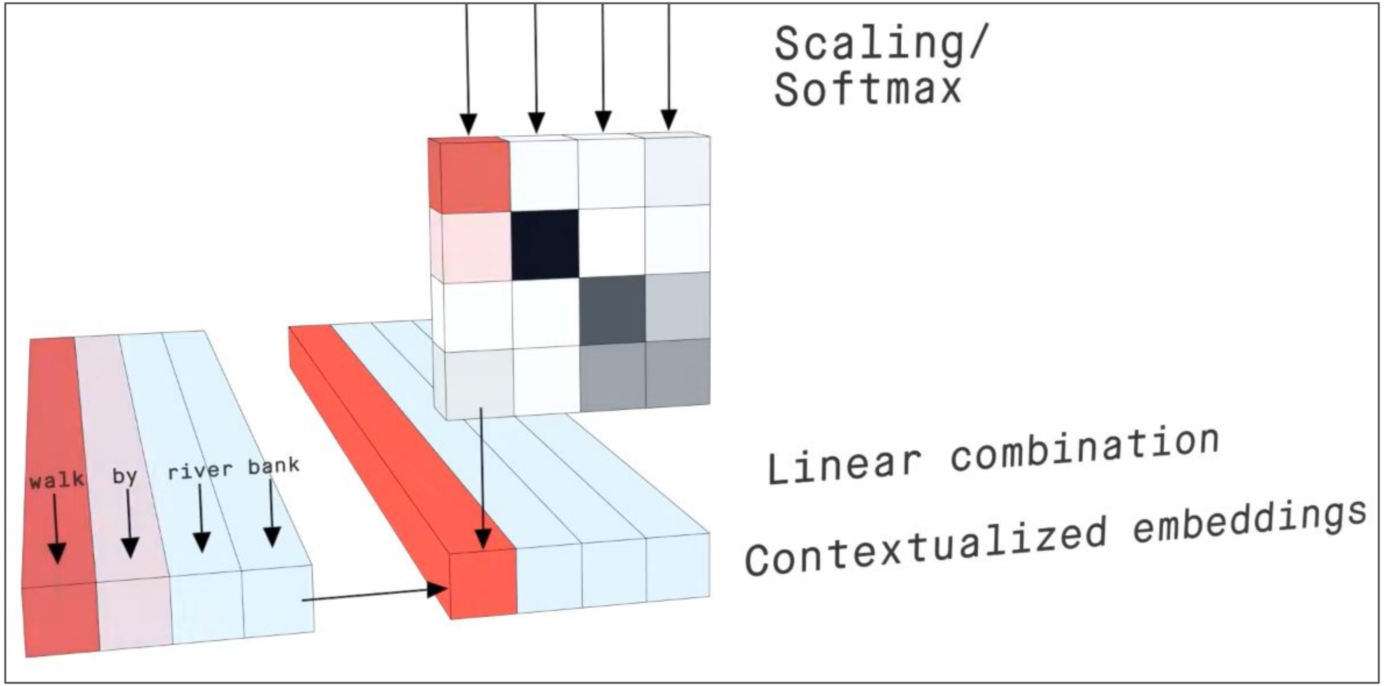


Image from: How to get meaning from text with language model BERT, Peltarion

Methods: Architectures

Baseline: Vision Transformer (ViT)

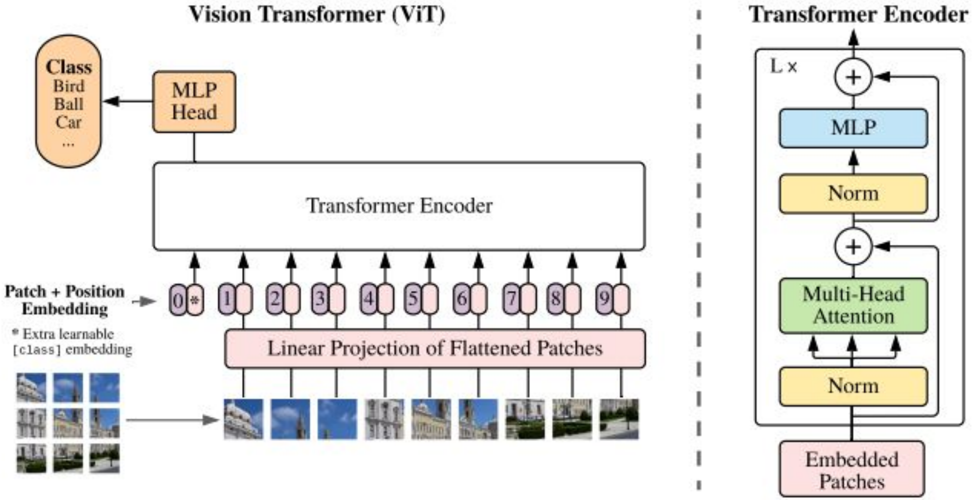


Image from: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, Dosovitskiy et al. 2021

- A vision transformer (ViT) aims to use the transformer architecture for 2-D images.
- Designed to inherently operate on 1D sequential data, the image needs to be broken down into patches and presented as a sequence **in raster order,** in order to use the transformer architecture for image processing.
- The 2D image patches are then converted to vector embeddings using a projection layer, following which the transformer can be used as usual.
- A **special [class] token** is also prepended, whose corresponding output at the final layer, is taken as the representation of the whole image.

Methods: Architectures

Adapters for Transformers

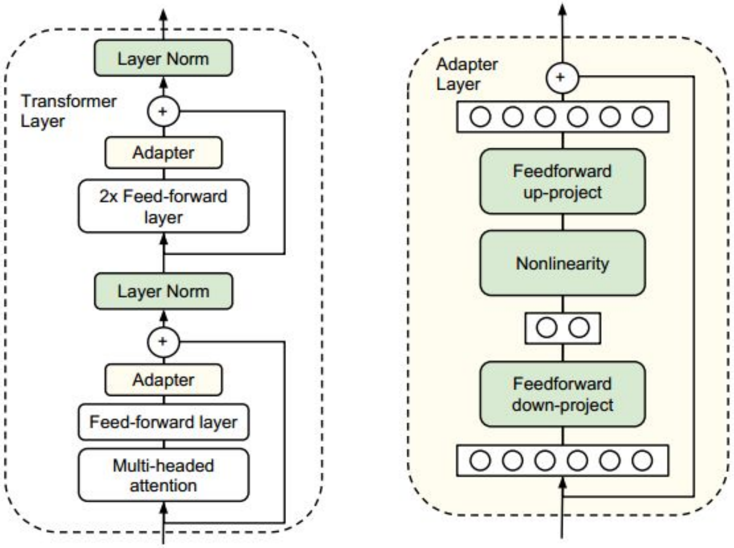


Image from: Parameter-Efficient Transfer Learning for NLP: Houlsby, Giurgiu et al. 2019

- Transformers require a huge amount of data to train, due to their very generic architecture and hence, the lack of inductive biases for textual or image data.
- In order to fine-tune a pre-trained transformer for a different task, a huge number of parameters need to be retrained on the new data.
- In addition, overwriting the same weights would inevitably lead to catastrophic forgetting.
- Adapters resolve this situation by introducing two new adapter layers (shown on the right), inside a single layer of a pre-trained transformer (shown on the left).
- Only the adapter layer is trainable, and the rest of the layers are frozen.
- Having a simple bottleneck architecture, the adapter layer is very parameter efficient.
- Its weights are initialized near zero, in order to implement a near-identity function at initialization.
- By using different adapter layers for each new task, the transformer is made extensible and hence not susceptible to catastrophic forgetting while maintaining low number of parameters per new task.

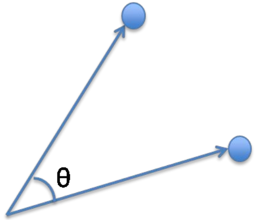
Adapters: Implementation

```python
class Adapter(nn.Module):
    def __init__(self, adapter_config: dict):
        super(Adapter, self).__init__()
        self.down_project = nn.Linear(adapter_config["hidden_size"], adapter_config["bottleneck_dim"])
        self.activation = adapter_config["activation"]()
        self.up_project = nn.Linear(adapter_config["bottleneck_dim"], adapter_config["hidden_size"])
        self._init_params()

    def _init_params(self):
        for param in self.down_project.parameters():
            # torch.nn.init.constant_(param, 0.)
            torch.nn.init.normal_(param, 0., 1e-5)
        for param in self.up_project.parameters():
            # torch.nn.init.constant_(param, 0.)
            torch.nn.init.normal_(param, 0., 1e-5)

    def forward(self, hidden_states):
        # hidden_states = (B, T, D)
        outputs = self.down_project(hidden_states)
        outputs = self.activation(outputs)
        outputs = self.up_project(outputs)

        adapter_outputs = outputs + hidden_states
        return adapter_outputs
```

Methods: Loss function

Similarity metric: Cosine Embedding Similarity, measures the similarity using the angle between the embeddings of two images, say.

$$sim(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



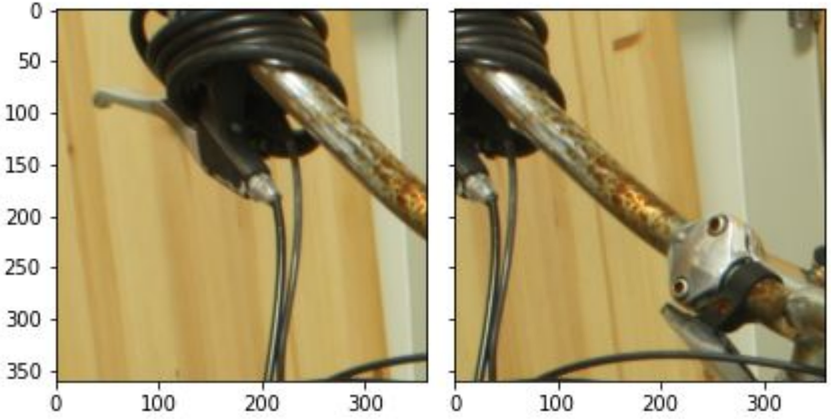
Loss function: Cosine Embedding Loss (torch.nn.CosineEmbeddingLoss)

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$

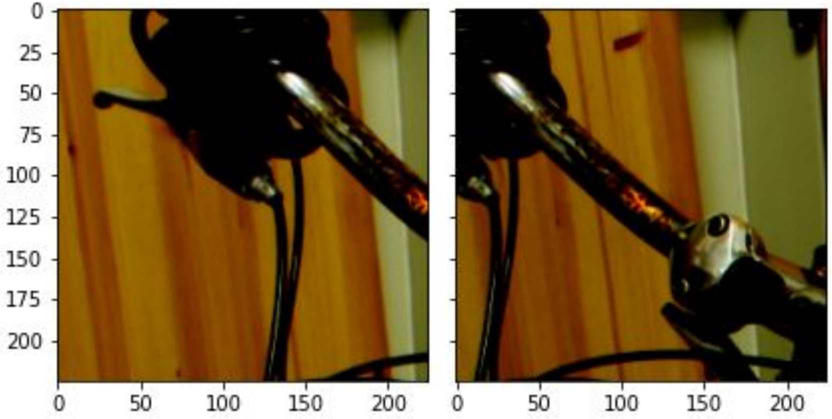
Experiments

Data preprocessing

1. **Resize**: (3, 360, 360) → (3, 224, 224) (PIL.resize() with bilinear interpolation)
2. **Rescale**: Values between 0 and 255 → Values between 0 and 1
3. **Normalize**: Subtract 0.5 and divide by 0.5 → Values between -1 and 1



Before



After

Experiments

Baseline results

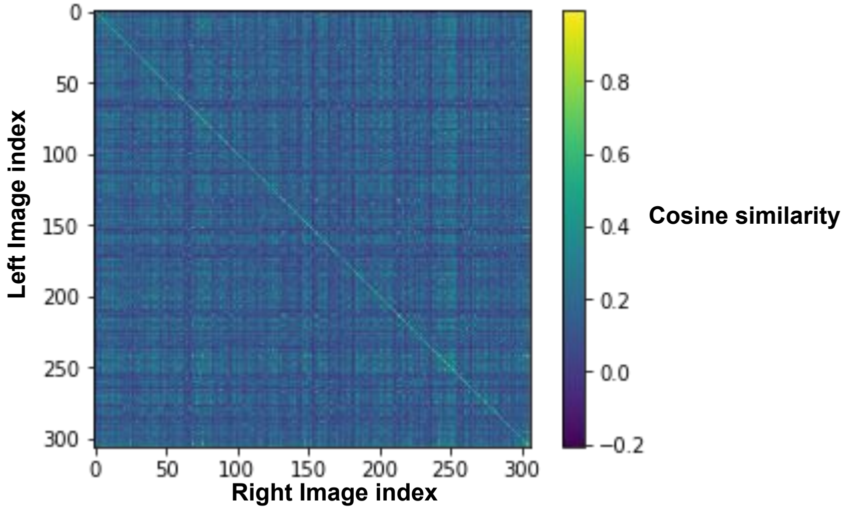
Model used: google/vit-base-patch16-224 (From huggingface)
Patch-size = 16x16, Image resolution = 224x224
Number of patches per image = 1 + 14x14 = 197

This model is chosen because its a vision transformer (ViT) trained on the ImageNet dataset which contains similar classes of images as the evaluation dataset seems to contain.

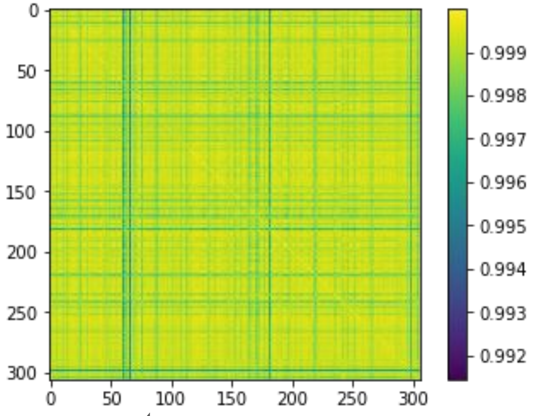
Chance accuracy: 1/306 = 0.32%

Matching accuracy: 53.92% ~ 165/306

Matching confusion matrix:



Experiments with Adapters



The table below shows the test performance and parameter overload for different

- adapter sizes
- batch-sizes
- learning-rates
- number of epochs
- learning-rate schedules

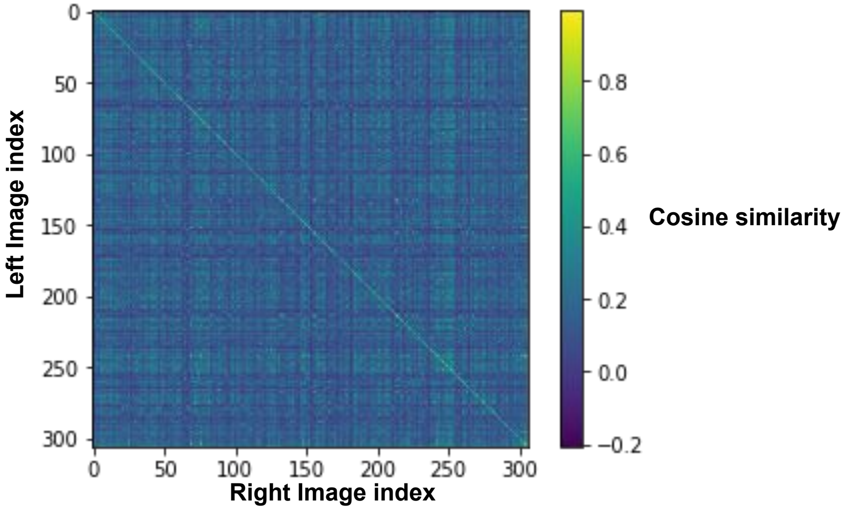
The optimizer and adapter activation functions are kept fixed.

Index	Adapter size	Adapter actvn	Batch-size	Optimizer	Learning Rate	Epochs	LR Schedule	Test perf		Param overload (%)	Trainable params
1	8	GELU	-	-	-	-	No	53.27%		0.362934054	313536
2	8	GELU	64	Adam	3.00E-04	3	No	54.25%		0.362934054	313536
3	16	GELU	32	Adam	3.00E-04	3	No	58.82%		0.7045321196	608640
4	16	GELU	32	Adam	3.00E-04	20	No	57.19%		0.7045321196	608640
5	32	GELU	16	Adam	3.00E-04	3	No	56.86%		1.387728251	1198848
6	32	GELU	32	Adam	3.00E-04	3	No	58.50%		1.387728251	1198848
7	16	GELU	64	Adam	3.00E-04	3	Yes. OneCycle	56.50%		0.7045321196	608640
8	32	GELU	32	Adam	3.00E-03	3	Yes. OneCycle	56.86%		1.387728251	1198848
9	64	GELU	64	Adam	3.00E-04	3	No	**61.11%**		2.754120513	2379264
10	128	GELU	64	Adam	3.00E-04	3	No	60.78%		5.486905037	4740096
11	64	GELU	128	Adam	3.00E-04	3	No	**61.11%**		2.754120513	2379264
12	64	GELU	64	Adam	3.00E-05	3	No	**61.11%**		2.754120513	2379264

- From this small hyperparameter search, we can see that in all cases introduction of the adapter layers brings improvement in the matching accuracy of the stereo-image pairs.
- The best improvement is **~7.2%** from the baseline for an adapter size of 64, **~ 21** more images in the evaluation dataset

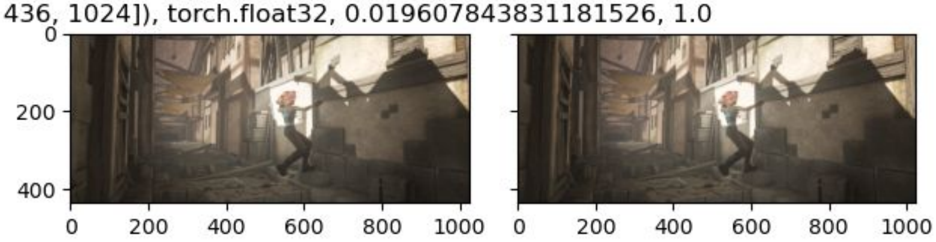
Possible avenues for improvement

1. Use **both axes** of the similarity matrix to calculate the best match.
    - This results in a baseline accuracy of **62.42%** vs **53.92%**
2. Use a **contrastive loss** to prevent degenerate solutions.




```python
match_matrix = torch.zeros(306, 306)
accuracy = []
for i in range(306):
    for j in range(306):
        match_matrix[i][j] = cos(left_embeddings[i:i+1], right_embeddings[j:j+1])
for i in range(306):
    scores = match_matrix[i, :] + match_matrix[:, i]
    match = torch.argmax(scores)
    if match == i:
        accuracy.append(1.)  # Correct match
    else:
        accuracy.append(0.)  # Incorrect match
```

Training data: Sintel Stereo Dataset



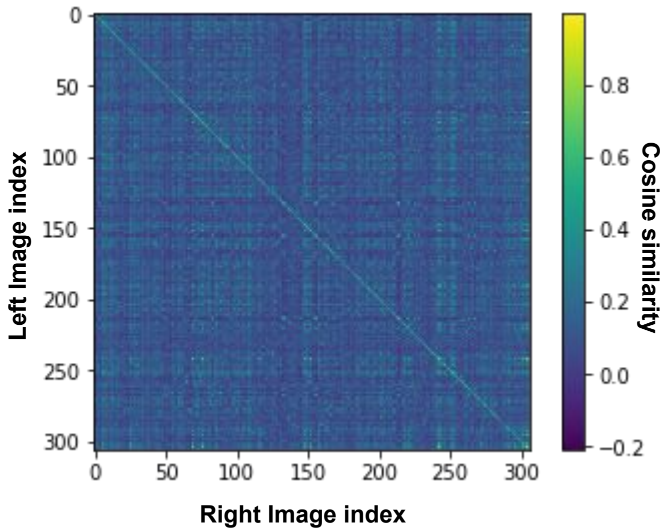
Sample A

Sample B

Remark: The training data is peculiar because many pairs of images look very similar and are only a few frames apart. A **naive similarity inducing loss** will thus easily lead to a trivial solution that maps **every image to the same embedding**. We thus, need a **contrastive loss** to counter this degenerate solution.

Experiments with Contrastive Loss

```python
mask = 2*torch.eye(batch_size)-1.

def contrastive_loss(outputs_l, outputs_r, mask, batch_size):
    loss = torch.tensor(0.).cuda()
    for b in range(batch_size):
        x = loss_fn(torch.tile(outputs_l[b:b + 1], [batch_size, 1]), outputs_r, mask[b])
        y = loss_fn(torch.tile(outputs_r[b:b + 1], [batch_size, 1]), outputs_l, mask[b])
        loss += (x[b] + (torch.sum(x[:b]) + torch.sum(x[b + 1:])) / (batch_size - 1)) / 4.
        loss += (y[b] + (torch.sum(y[:b]) + torch.sum(y[b + 1:])) / (batch_size - 1)) / 4.
    return loss/batch_size
```



Index	Adapter size	Adapter actvn	Batch-size	Optimizer	Learning Rate	Epochs	LR Schedule	Test perf		Param overload (%)	Trainable params	Remarks
16	64	GELU	-	-	-	-	-	62.42%		2.754120513	2379264	
17	64	GELU	64	Adam	3.00E-04	20	No	68.30%		2.754120513	2379264	
18	128	GELU	64	Adam	3.00E-04	20	No	69.28%		5.486905037	4740096	
19	64	GELU	64	Adam	3.00E-04	20	No	**72.55%**		2.754120513	2379264	Loss = 2/3, 1/3
20	64	GELU	64	Adam	3.00E-04	20	No	70.92%		2.754120513	2379264	Loss = 3/4, 1/4
21	128	GELU	64	Adam	3.00E-04	20	No	**73.20%**		5.486905037	4740096	Loss = 2/3, 1/3

Discussion

- A more thorough hyperparameter optimization still needs to be performed.
- What kind of mistakes is the model still making, and can we alleviate the issue?
- Are there better loss functions that can be devised to use the stereo-image generation process more efficiently?
- Thoughts/questions?

Thanks for your attention!