COTR: Correspondence Transformer for Matching Across Images

Supplementary Material

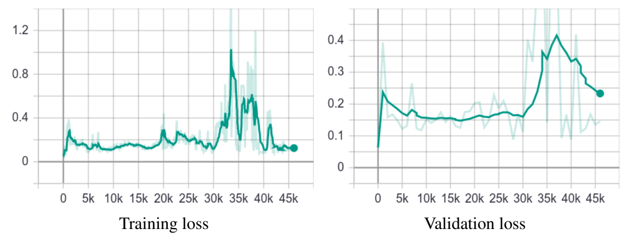

Figure B. Unstable training and validation loss for **COTR** with *log-linear positional encoding*. We terminate the training earlier as the loss diverges.

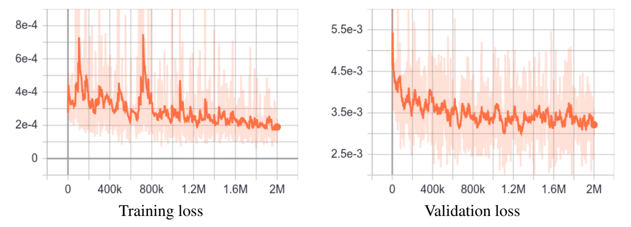

Figure A. Training and validation loss for **COTR** with *linear positional encoding*. Both losses slowly converge to a stable status.

A. Compute

The functional (and recursive) nature of our approach, coupled with the use of a transformer, means that our method has significant compute requirements. Our currently non-optimized prototype implementation queries *one* point at a time, and achieves 35 correspondences per second on a NVIDIA RTX 3090 GPU. This limitation could be addressed by careful engineering in terms of tiling and batching. Our preliminary experiments show no significant drop in performance when we query different points inside a given crop – we could thus potentially process any queries at the coarsest level in a single operation, and drastically reduce the number of operations in the zoom-ins (depending on how many queries overlap in a given crop). We expect this will speed up inference drastically. In addition to batching the queries at inference time, we plan to explore its use on non-random points (such as keypoints) and advanced interpolation techniques.

B. Log-linear vs Linear

Here, we empirically demonstrate that linear positional encoding is important. We train two **COTR** models with different positional encoding strategies; see Section 3.2. One model uses log-linear increase in the frequency of the sine/cosine function, and the other uses linear increase instead. Fig. A shows that **COTR** successfully converges using the linear increase strategy. However, as shown in Fig. B, **COTR** fails to converge with the commonly used log-linear strategy [73, 10]. We suspect that this is because the task of finding correspondences does not involve very high frequency components, but further investigation is necessary and is left as future work.

C. Architectural details for COTR

Backbone. We use the lower layers of ResNet50 [23] as our CNN backbone. We extract the feature map with 1024 channels after layer3, *i.e.*, after the fourth downsampling step. We then project the feature maps with 1024 channels with 1×1 convolution to 256 channels to reduce the amount of computation that happens within the transformers.

Transformers. We use 6 layers in both the transformer encoder and the decoder. Each encoder layer contains an 8-head self-attention module, and each decoder layer contains an 8-head encoder-decoder attention module. Note that we disallow the self-attention in the decoder, in order to maintain the independence between queries – queries should not affect each other.

MLP. Once the transformer decoder process the results, we obtain a 256 dimensional vector that represents where the correspondence should be. We use a 3-layer MLP to regress the corresponding point coordinates from the 256-dimensional latent vector. Each layer contains 256 neurons, followed by ReLU activations.

D. Architectural details for the MLP variant

Backbone. We use the same backbone in **COTR**. The difference here is that, once the feature map with 256 channels is obtained, we apply max pooling to extract the global latent vector for the image, as suggested in [21]. We also tried a variant where we do not apply global pooling and use a fully-connected layer to bring it down to a manageable size of 1024 neurons but it quickly provided degenerate results, where all correspondence estimates were at the centre.

MLP. With the latent vectors from each image, we use a 3 layer MLP to regress the correspondence coordinates. Specifically, the input to the coordinate regressor is a 768-dimensional vector, which is the concatenation of two global latent vectors for the input images and the positional encoded query point. Similarly to the MLP used in **COTR**, each linear layer contains 256 neurons, and followed by ReLU activations.

E. Comparing with RAFT [65]

RAFT [65] performs better in KITTI-type of scenarios, not necessarily so for other cases. To show this, we provide

Method	ETH3D							
	AEPE↓	rate=3	rate=5	rate=7	rate=9	rate=11	rate=13	rate=15
RAFT [65] ECCV'20		1.92	2.12	2.33	2.58	3.90	8.63	13.74
COTR		**1.66**	**1.82**	**1.97**	**2.13**	**2.27**	**2.41**	**2.61**
COTR +Interp.		1.71	1.92	2.16	2.47	2.85	3.23	3.76

Method	KITTI 2012		KITTI 2015		HPatches			
	AEPE↓	Fl↓	AEPE↓	Fl↓	AEPE↓	PCK-1px↑	PCK-3px↑	PCK-5px↑
RAFT [65] ECCV'20	2.15	9.30	5.00	17.4	44.3	31.22	62.48	70.85
COTR	**1.28**	**7.36**	**2.62**	**9.92**	**7.75**	**40.91**	**82.37**	**91.10**
COTR +Interp.	2.26	10.50	6.12	16.90	7.98	33.08	77.09	86.33

Method	Image Matching Challenge		
	Num. Inl.↑	mAA(5°)↑	mAA(10°)↑
RAFT [65] ECCV'20+DEGENSAC (N= 2048)	1066.1	0.163	0.259
COTR +DEGENSAC (N= 2048)	**1686.2**	**0.515**	**0.678**

Table A. RAFT on ETH3D, KITTI, HPatches, and IMC2020.

results for RAFT [65] on all other datasets in Table A. On KITTI, sparse COTR still performs best, and with the interpolation strategy it is roughly on par with RAFT [65]. On other datasets, COTR outperforms RAFT [65] by a large margin[1].

[1] Note that RAFT [65] requires two input images of the same size. We resize them to 1024×1024 for HPatches and the Image Matching Challenge.