End-to-End Rotation Averaging with Multi-Source Propagation Supplementary Material

Luwei Yang[1] Heng Li[1] Jamal Ahmed Rahim[1] Zhaopeng Cui[2*] Ping Tan[1*]
[1] Simon Fraser University [2] State Key Lab of CAD & CG, Zhejiang University
{luweiy, lihengl, jrahim, pingtan}@sfu.ca, zhpcui@zju.edu.cn

To make our submission self-contained, the supplementary material provides additional details about: 1) The Network and Modules; 2) Training; 3) Test & Iterative Edge Weight Refinement, and lastly 4) additional experiments on execution time and outlier impact. The code demonstration and dataset (*YFCC100*) are available at https://github.com/sfu-gruvi-3dv/msp_rot_avg.

1. Network Details

In this section, we provide additional details of the network modules, as we mentioned in the main paper.

Feature Extraction Module: The Feature Extraction Module extracts useful information for the graph nodes and edges from images with matched correspondences. Recall that it employs cluster-based aggregation for obtaining fixed-size node features and edge features regardless of arbitrary input dimensions and the number of feature matches.

For constructing the node feature at a node i, we first extract feature maps $\mathbf{F}_i^4$ (512 channels) from VGG16 [14]'s layer group 4. The VLAD pooling operation applies k-means to cluster all the pixels from the feature maps $\mathbf{F}_i^4$ into $P = 64$ groups, resulting in a pooled VLAD feature $\mathbf{g}_i$. We then finally reduce the channel size of $\mathbf{g}_i$ with a 1×1 convolution, producing the node feature, $\hat{\mathbf{g}}_i$, with a size of 32×64. For constructing the edge feature for an edge i, j, we first reduce the channels of the feature maps $\mathbf{F}^4$ to 64 channels with another 1×1 convolution, resulting in $\hat{\mathbf{F}}^4$. Given that the edge connects node i and j with n pairs of matches, we gather the features from each pair of correspondence:

$$\{\mathbf{f}_{ij}\}_n = \{[\tilde{\mathbf{x}}_i^a, \tilde{\mathbf{x}}_j^b, \mathbf{f}_i^a, \mathbf{f}_j^b]\}_n$$

where $\mathbf{f}_i^a = \hat{\mathbf{F}}_i^4(\mathbf{x}_i^a)$ and $\mathbf{f}_j^b = \hat{\mathbf{F}}_j^4(\mathbf{x}_j^b)$ are the features of points a, b in the feature map $\hat{\mathbf{F}}_i^4, \hat{\mathbf{F}}_j^4$. The $\tilde{\mathbf{x}}_i^a, \tilde{\mathbf{x}}_j^b$ are 2D positions of keypoints a, b on frames i, j, and normalized to [-1, 1]; Next, a two-layer MLPs $\phi(\cdot)$ with $\{128, 64\}$ hidden units is used to generate a vector encoding the features, and

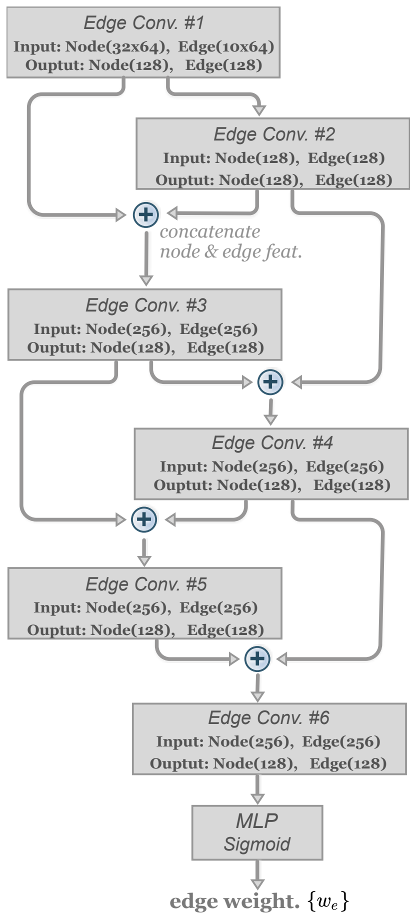
Edge Conv. #1
Input: Node(32x64), Edge(10x64)
Ouptut: Node(128), Edge(128)
Edge Conv. #2
Input: Node(128), Edge(128)
Ouptut: Node(128), Edge(128)
concatenate node & edge feat.
Edge Conv. #3
Input: Node(256), Edge(256)
Ouptut: Node(128), Edge(128)
Edge Conv. #4
Input: Node(256), Edge(256)
Ouptut: Node(128), Edge(128)
Edge Conv. #5
Input: Node(256), Edge(256)
Ouptut: Node(128), Edge(128)
Edge Conv. #6
Input: Node(256), Edge(256)
Ouptut: Node(128), Edge(128)
MLP
Sigmoid
edge weight. $\{w_e\}$

Figure 1. Structure of Appearance-Geometry Fusion Network. The size of input and output feature is listed in each `Edge Convolution` layer.

yield a $n \times 64$ feature: $\{\hat{\mathbf{f}}_{ij}\}_n = \{\phi(\mathbf{f}_{ij})\}_n$; Lastly, to remove any dependency on the number of matches, we apply k-means clustering to the positions of keypoint $\{\tilde{\mathbf{x}}_i\}_n$, resulting in $K = 10$ different groups. Then, for all features that belongs to a cluster group m, $\{\hat{\mathbf{f}}_{ij}\}_n^m$, we use `Maxpool` to aggregate the feature set to a single vector feature and get

*Corresponding authors.

our edge features $\mathbf{e}_{ij} = \{\texttt{maxpool}(\{\hat{\mathbf{f}}_{ij}\}_n^m)|m < K\}$. The aggregate function `Maxpool` is chosen as it is symmetric for all items regardless of ordering dependency [11]. This results in edge features with a fixed dimension of 10×64.

AGF Network Structure: The Appearance-Geometry Network takes the node and edge feature, along with the measured relative geometry as input, and predicts a weight for each edge, which indicates whether the edge is reliable or not; The Figure 1 shows the structure of AGF Network. The AGF consists of 6 Edge Convolution layers[5], each with 128 hidden units. Note that from layer #3 to #6, the input node and edge features are concatenated output node and edge features from the previous two `Edge Convolution` layers, respectively. For example, the output node features from `Edge Conv.#1` and `Edge Conv.#2` are concatenated as the input node features for `Edge Conv.#3`. Similarly, the input edge features for `Edge Conv.#3` is the concatenation of the edge features `Edge Conv.#1` and `Edge Conv.#2`. Lastly, a MLP layer followed by a `sigmoid` activation is appended to the output edge features from `Edge Conv.#6`, and generates the normalized weight on each edge $\{w_{ij}|(i,j) \in \mathbf{E}, 0 < w_{ij} < 1\}$.

Self-loops in MSP: In MSP, we add a virtual self-loop for each node in the view-graph with a preset weight 1.0. This will fix the rotation at the node when it reaches a high confidence score and will not propagate much information from neighbors in the subsequent iterations.

FineNet Structure: With the initialized orientation as the node input and relative rotation as the edge input, the FineNet [10] is used to produce the refined orientation for all nodes in a view-graph. FineNet consists of 4 layers of Edge Convolution. The input node features are the initial camera orientations $\mathbf{q}_i$, and edge features are the residual of the estimated and measured relative rotations: $f_{ij} = \mathbf{q}_j^{-1}\tilde{\mathbf{q}}_{ij}\mathbf{q}_i$. With the input node and edge features, the FineNet produces the residual orientation $\Delta\mathbf{q}_i$ over the initial orientation $\mathbf{q}_i$ for each node $i \in V$. Later, its final refined orientation is obtained by $\mathbf{q}_i^f = \Delta\mathbf{q}_i + \mathbf{q}_i$, and further normalized to unit with $\frac{\mathbf{q}^f}{||\mathbf{q}^f||}$.

2. Training Details

In this section, we introduce more details of training data generation and training process.

Constructing View-Graph for YFCC100: We use the *YFCC100*'s reconstruction as the ground-truth camera orientation for training. To build the missing view-graph for every *YFCC100* scene, the COLMAP [13] is used to extract 2D key-point features in every image at beginning, and the vocabulary tree is built to match each image against its visual 20 nearest neighbours. Then, any two nodes are connected by an edge if the five-points algorithm [8] yields a valid relative orientation. Lastly, for all *YFCC100* and *1DSfM* scenes, the view-graph is transformed from directed to un-directed by adding reverse edges with the inverse relative orientation $\{\mathbf{q}_{ij}^{-1}, (i,j) \in E\}$.

Sampling Training Sub-graphs: Recall that we sample multiple sub-graphs during training due to the limited GPU memory. Given the full view-graph $G = \{V, E\}$, we aim to find a sub-graph $G_s = \{V_s, E_s\}$ with maximum N_s nodes. Firstly, a node $v_r \in V$ is randomly chosen as the root node, then we start Breadth-first Search (BFS) from node v_r, and add all visited nodes into sampled node set $\{V_s\}$ until $|V_s| = N_s$. Later, the sampled edge set E_s is formed by adding all edges (u, v) that connects any two nodes $u \in V_s$ and $v \in V_s$ in sampled node set $\{V_s\}$. We randomly sample 10-200 sub-graphs from each view-graph of *YFCC100* scene, depending on its total number of nodes. Typically, this generates 3,000 – 4,000 sub-graphs for training. As for the *1DSfM* dataset, we randomly sample 250-300 sub-graphs from each scene, and form a training set with a total of 3000-3500 sub-graphs.

Implementation and Parameter Initialization: Our model is implemented with the PyTorch [9] framework. We use the `pytorch-geometric` [4]'s `Edge Convolution` implementation for constructing the Appearance-Fusion Network while the Multi-Source Propagator is implemented using the DGL library [15].

For training on the *YFCC100* dataset, the parameters of VGG16 [14] and VLAD layer in Feature Extractor are copied from the trained NetVLAD[1] model, and the FineNet is initialized with the model trained by NeuRoRA's synthetic data [10]. `Edge Convolution` layers in AGF and other `MLPs` are randomly filled by PyTorch's default initialization generator. Later, the model trained on *YFCC100* is used to initialize the parameters when fine-tuning the *1DSfM* dataset.

Training: Recall that due to the memory limitation, our model is trained in three stages, with each stage containing 30, 80 or 180 nodes in sampled sub-graphs. We first use sub-graphs with 30 nodes to train our full model till it converges. Then, we fix the parameters of Feature Extractor for memory efficiency and train subsequent modules with 80 nodes. When training on sub-graphs with 180 nodes, in order to get better generalization, a pair of FineNet that shared the parameters are chained sequentially such that the output of the first FineNet is fed into the second FineNet. Thus, the total loss for the sub-graph with 180 nodes is:

$$L^{180} = L_{\mathbf{e}} + L_{\mathbf{init}} + L_{\mathbf{opt}}^1 + \omega_{opt} L_{\mathbf{opt}}^2, \qquad (1)$$

where the $L_{\mathbf{opt}}^1$ and $L_{\mathbf{opt}}^2$ are the same loss function (Equation 13 in main paper) applied to the output of the first and the second FineNet, while weight $\omega_{opt} = 2.0$ balances the two term.

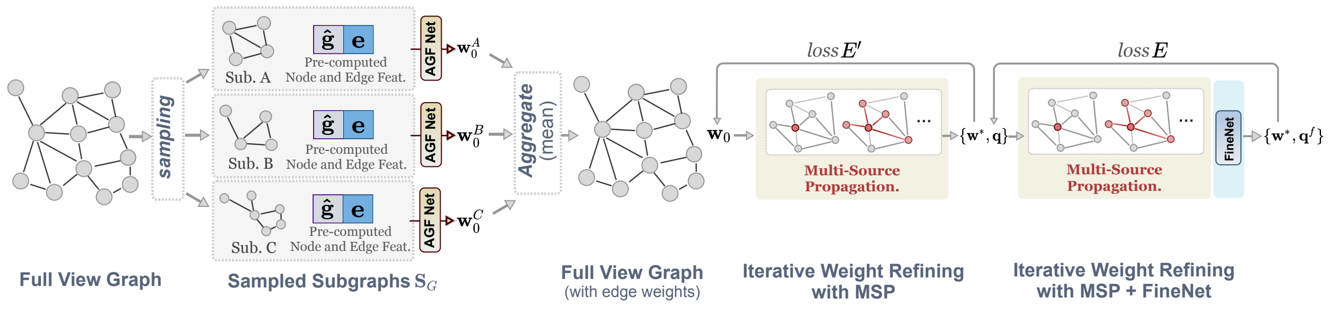
sampling
Sub. A
Sub. B
Sub. C
Pre-computed Node and Edge Feat.
AGF Net
Aggregate (mean)
loss E'
loss E
Multi-Source Propagation.
FineNet
Full View Graph
Sampled Subgraphs S_G
Full View Graph (with edge weights)
Iterative Weight Refining with MSP
Iterative Weight Refining with MSP + FineNet

Figure 2. Test pipeline, please refer to the main text for specific details.

Adam optimizer [7] is used to train our pipeline with different learning rates in various modules, as shown in Table 1. It takes 36 hours (for 15 epochs) to train the first stage with 30 nodes in each sub-graph. For the following two stages, the training can be accelerated by caching the node and edge features to disk during the first epoch. We then train the model with another 39 epochs, with the two stages costing 24 hours and 36 hours, respectively.

Module	lr
Node and Edge Feature Extraction	1.0×10^{-3}
AGF Network	1.0×10^{-4}
FineNet	1.0×10^{-4}

Table 1. The learning rates used in training various modules.

3. Test Details

In this section, we provide additional testing details. The Fig. 2 shows the overview of the test procedure, including 1) sampling multiple sub-graphs from a full view-graph that covers all edges; 2) generating edge weights for each sub-graph using AGF and later aggregate into full view-graph; 3) iterative edge weight refinement with MSP and FineNet in testing time. The following sections will detail the test pipeline.

3.1. Sub-graph Sampling During Testing

Recall that limited GPU memory prevents us from loading neither all images and correspondences nor all node and edge features at once. Therefore, we also sample sub-graphs from a view-graph to circumvent the limitations. However, compared to the training, the sampled sub-graphs set must cover all edges in the graph G during testing. Therefore, we aim to sample a sub-graph set $\mathbf{S}_G = \{S_1, \dots, S_k\}$ from the original view-graph $G = \{V, E\}$, where each sub-graph has $N_s = 80$ nodes and all edges in the original graph G are covered by any sub-graph in $\mathbf{S}_G$.

The sub-graph set is constructed iteratively, we build a residual graph $G^R = \{V^R, R^R\}$ by duplicating G at beginning; Next, we sample a sub-graph $S_k = \{V_k, E_k\}$ from G^R using BFS, and then delete the S_k from residual graph G^R. Specifically, we delete edges $e \in E^R$ that were previously sampled in S_k, and a node $v \in V^R$ is removed if no adjacent edge exists. Later, the remaining residual graph G^R is used for sampling new sub-graphs in the following iterations. The sampling stops when $|E^R| = 0$, where all edges in E are covered in the sub-graph set $\mathbf{S}_G$. Note that the number of nodes $|V_k|$ of the sampled sub-graph S_k might be less than N_s after several iterations. Therefore, another function *ExpandSubGraph* is called to apply BFS on the original G to extend V_k until $|V_k| = N_s$. The steps are summarized in Algorithm 1.

Algorithm 1: Sub-graph Sampling During Testing

```
Input  : $G = \{V, E\}$
Output: Sub-graph sets $\mathbf{S}_G = \{S_1, \dots, S_k\}, S_i = \{V_i, E_i\}$
Initialization:
1  $\mathbf{S}_G \leftarrow \emptyset; E^R \leftarrow E; V^R \leftarrow V$      ▷ copy original view-graph $G$
2  $k \leftarrow 1$
   while $(|E^R| \neq 0)$ do
3      $v_r \leftarrow random(V^R)$      ▷ select a random node to BFS
4      $S_k \leftarrow BFS(V^R, E^R, v_r)$      ▷ find sub-graph nodes
       if $|V_k| < N_s$ then
5          $S_k \leftarrow ExpandSubGraph(V, E, S_k)$
       end
6      $\mathbf{S}_G \leftarrow \mathbf{S}_G \cup \{S_k\}$
7      $E^R \leftarrow E^R - E_k$      ▷ update residual graph $G^R$
8      $V^R \leftarrow \{u, v | e = (u, v), e \in E^R\}$
9      $k \leftarrow k + 1$
   end
```

3.2. Building Complete Graph from Sub-graphs

Given a sampled sub-graphs set $\mathbf{S}_G$ as input, we first construct the complete view-graph with weights on all edges. Specifically, this is done by first feeding the sub-graphs into the Feature Extractor and AGF modules, and caching the output edge weights. Note that the input node and edge features can be pre-computed and cached onto the disk at the beginning. Computation of relative poses and construction of the view-graph can similarly be done before the feed-forward for efficiency. Next, a complete weighted view-graph is aggregated where the edge's weight is computed by the average of its weights in all sub-graphs containing it, resulting in $\mathbf{w}^0$. Later, the full view-graph with initial edge-weights is then fed together into the MSP followed by Finenet, and finally refined together using Iterative Edge Weight Refinement.

3.3. Edge Weight Refinement

Recall that the edge weight $\mathbf{w}^0$ might be imprecise. With the fully-differentiable MSP and FineNet modules, we can refine the edge weights by minimizing the discrepancy error towards relative orientation measurement. In practice, to speed up the overall refining process, we first obtain an approximate initial solution without involving the FineNet and minimize the following energy term w.r.t initial candidates ($M = 15$) produced by MSP

$$E' = \frac{1}{|M||E|} \sum_{m \in M} \sum_{(i,j) \in E} w^0_{ij} \rho(d_q(\tilde{\mathbf{q}}_{ij}, \mathbf{q}^m_{ij})). \quad (2)$$

Here, the $d_q(\mathbf{q}_a, \mathbf{q_b}) = min(||\mathbf{q}_a - \mathbf{q}_b||, ||\mathbf{q}_a + \mathbf{q}_b||)$ defines the distance of two quaternions and ρ denotes smooth L_1 costs:

$$\rho(\mathbf{x}) = \begin{cases} |\mathbf{x}| - 0.5\alpha & \text{if } |\mathbf{x}| > \alpha \\ \frac{0.5\mathbf{x}^2}{\alpha} & \text{if } |\mathbf{x}| \leq \alpha \end{cases}, \quad (3)$$

where $\alpha = 5 \times 10^{-2}$. The rotation $\mathbf{q}^m_{ij}$ is computed using absolute orientations generated by MSP. We use Adam optimizer and set the learning rate $lr = 1.0$ to minimize the E' with 20 iterations, resulting in the approximate refined weight $\mathbf{w}'$. Note that the weight $\mathbf{w}$ is normalized at any iteration by a `sigmoid` function to clamp the range from 0 to 1 before feeding into the MSP.

Later, the weight $\mathbf{w}'$ is further refined by fixing the parameters of FineNet module, and use the same Adam optimizer (with $lr = 1.0$) to minimize the following energy until convergence:

$$E = \frac{1}{|M||E|} \sum_{m \in M} \sum_{(i,j) \in E} w'_{ij} \rho(d_q(\tilde{\mathbf{q}}_{ij}, \mathbf{q}^m_{ij})) + \frac{1}{|E|} \sum_{(i,j) \in E} w'_{ij} \rho(d_q(\tilde{\mathbf{q}}_{ij}, \mathbf{q}^f_{ij})). \quad (4)$$

Similar to [6, 10], given the outputs from MSP, we empirically select the initial candidate that has the highest sum of adjacent weights of the reference node, and then feed it into FineNet to get refined absolute orientations $\{\mathbf{q}^f_i, i \in V\}$. Later, the relative rotation $\mathbf{q}^f_{ij}$ is computed using refined absolute orientations.

4. Additional Experiments

Execution Time: Table 2 reports the execution times of our method using the 1DSfM dataset. It includes the time costs of both, AGF and Iterative Refinement steps. Note that the features for nodes and edges can be extracted and pre-cached when computing the relative poses between any two images at initialization. The running time of NeuRoRA[10] and Chatterjee[2] (IRLS) are also shown for comparison[1]. NeuRoRA is regression-based and is the fastest since it produces results with a single forward pass, while, as discussed in [12], pose regression is generally less accurate. Our method is slower because of the iterative edge reweighting, which produces more accurate results. Compared to IRLS[2], our method runs faster in larger scenes (e.g. `Piccadilly`, 2152 nodes) while slower in smaller scenes, as large scenes could benefit from massive GPU parallelization. Our overall running time is acceptable for the current SfM pipeline, e.g. Global SfM [3] takes 578s for `Alamo` and 1480s for `Piccadilly`.

	AGF	Iter. Refine	Ours Total	IRLS[2]	NeuRoRA[10]
Alamo	16s	65s	81s	20.5s	2.2s
Ellis Island	6s	38s	44s	2.5s	0.4s
GendrmMarket	4s	49s	53s	11.1s	0.5s
Madrid	6s	36s	42s	3.2s	0.2s
Montreal N. D.	13s	40s	53s	9.1s	1.0s
Piazza D. P.	8s	31s	39s	3.3s	0.4s
Roman F.	18s	97s	115s	20.2s	1.3s
T. London	5s	42s	47s	1.9s	0.3s
U. Square	7s	48s	55s	6.8s	0.6s
V. Cathedra	33s	114	147s	48.1s	2.1s
Yorkminster	6s	77s	83s	4.0s	0.4s
NYC Lib.	6s	43s	49s	4.8s	0.2s
Piccadilly	98s	262s	360s	449s	6s

Table 2. Time costs (sec.) of each step (AGF, Iterative Refinement) and the comparison with Chatterjee[2](IRLS) and NeuRoRA[10] on *1DSfM*.

Outlier Impact: We follow NeuRoRA[10] and test our pipeline w.r.t different levels of outliers in the `louvre` scene (624 nodes, *YFCC100*). In order to construct different outlier levels, similar to [10], for all edges in view-graph, we first corrupt their ground-truth relative rotations by a Gaussian noise with std σ=30 °. Then, we randomly select edges by the percentage of γ as outliers and replace their orientations with random orientations. From Table 3, when γ=40% of edges are outliers, the performance drops considerably.

	γ=5%	γ=10%	γ=40 %	γ=50 %
mean err.	2.1°	2.2°	3.3°	4.97°
median err.	3.13°	3.44°	4.83°	6.64°

Table 3. The rotation error in deg. w.r.t ratio of outliers (γ).

[1]The execution time is reported in [10].

References

[1] Relja Arandjelović, Petr Gronat, Akihiko Torii, Tomas Pajdla, and Josef Sivic. Netvlad: Cnn architecture for weakly supervised place recognition. *IEEE Conf. Comput. Vis. Pattern Recog.*, 2016. 2

[2] Avishek Chatterjee and Venu Madhav Govindu. Efficient and robust large-scale rotation averaging. *Int. Conf. Comput. Vis.*, 2013. 4

[3] Zhaopeng Cui and Ping Tan. Global structure-from-motion by similarity averaging. *Int. Conf. Comput. Vis.*, 2015. 4

[4] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 2

[5] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. 2017. 2

[6] Richard Hartley, Khurrum Aftab, and Jochen Trumpf. L1 rotation averaging using the weiszfeld algorithm. *IEEE Conf. Comput. Vis. Pattern Recog.*, 2011. 4

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 3

[8] David Nistér. An efficient solution to the five-point relative pose problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(6):756–770, 2004. 2

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 2

[10] Pulak Purkait, Tat-Jun Chin, and Ian Reid. Neurora: Neural robust rotation averaging. *Eur. Conf. Comput. Vis.*, 2020. 2, 4

[11] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *IEEE Conf. Comput. Vis. Pattern Recog.*, 2017. 2

[12] Torsten Sattler, Qunjie Zhou, Marc Pollefeys, and Laura Leal-Taixe. Understanding the limitations of cnn-based absolute camera pose regression. *IEEE Conf. Comput. Vis. Pattern Recog.*, 2019. 4

[13] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise View Selection for Unstructured Multi-View Stereo. *Eur. Conf. Comput. Vis.*, 2016. 2

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1, 2

[15] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. 2