13

Boosting SLAM with Deep Learning

Zachary Teed, Jia Deng, Boris Chidlovskii, Jérome Revaud, Felix Wimbauer and Daniel Cremers

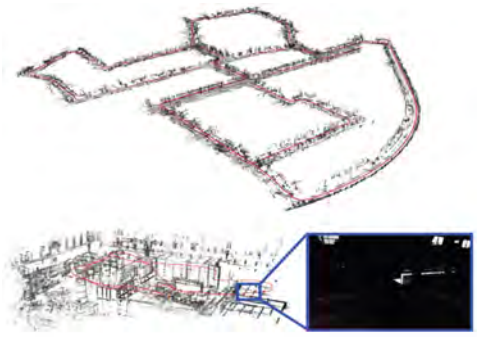
13.1 Introduction

Following the introduction of large, curated datasets [616], researchers have deployed deep neural networks for a multitude of challenges ranging from image segmentation [697] and optical flow estimation [286] to protein prediction [396, 535]. These systems are able to leverage large, labeled datasets and use highly overparameterized neural networks with powerful internal representations to learn a mapping from the set of inputs to the space of labels. A natural question is how we can extend the success of deep learning to SLAM.

As we saw in the previous chapters, modern SLAM systems contain many interconnected components including feature detection, feature matching, outlier rejection and relocalization. One promising line of work has investigated replacing individual components with learned systems. For example, networks for feature detection [275], descriptors [222], matching [1057, 1141, 976, 1142, 678] and outlier rejection [976] can be trained on large datasets and can outperform their classical counterparts. Once these components are trained, they can be plugged back into the full SLAM pipeline to produce more accurate and robust systems. Deep neural networks can also be trained to predict useful 3D properties directly from input images such as single image depth [97, 1223, 310] or relative pose between pairs of frames [1224]. Depth or relative pose can then be naturally integrated into existing SLAM systems, both as additional factors in the factor graph and as additional terms in the loss functions [1224].

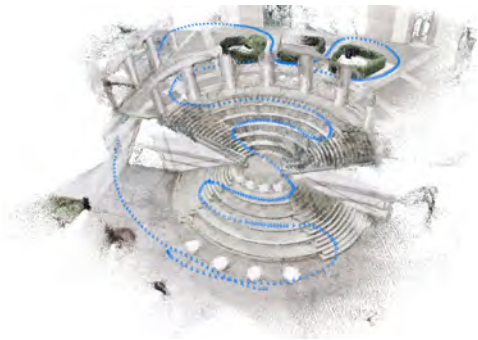
One issue with the modular approach is that errors from individual components can feed into each other and compound. Increasingly, deep networks have been shown capable of combining more and more of the pipeline into end-to-end trainable modules, including the integration of optimization problems such as bundle adjustment as neural network layers [1086, 1072, 677]. The DUSt3R [1162] line of works takes this idea even further by replacing the full stack with a network that predicts point maps directly from uncalibrated images.

In this chapter, we will discuss several promising ways to bring the power of neural networks into the SLAM pipeline. An overview of some of the works presented in

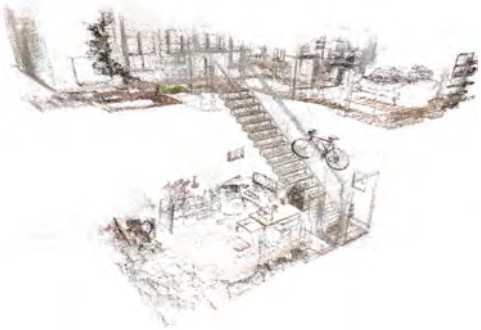


(a) D3VO [1224] (©2020 IEEE)

(b) MonoRec [1184] ©2021 IEEE)



(c) DROID-SLAM [1086]



(d) DPVO [1088]

(e) DuSt3R [1162] (©2024 IEEE)

(f) MUSt3R [141] (©2025 IEEE)

Figure 13.1 Over the last years, a multitude of efforts have been made to bring the predictive power of deep networks into visual SLAM. In this chapter, we will review a number of these approaches.

this chapter is shown in Figure 13.1. In many ways these are techniques that try to combine the best of both worlds. Some feed depth, pose and uncertainty predictions from neural networks into an optimization-based SLAM pipeline. Some combine

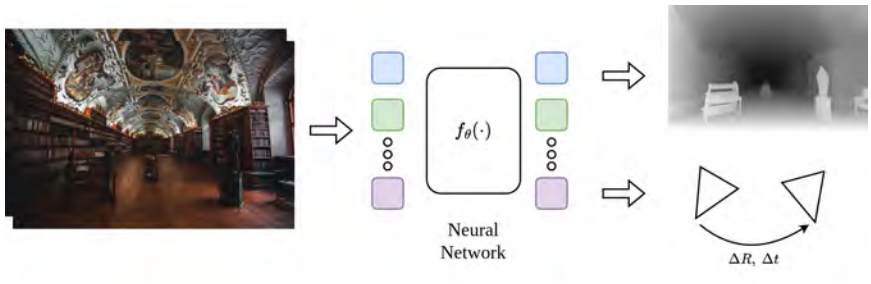


Figure 13.2 A neural network can be trained to predict 3D properties, such as depth or relative pose, from an input image or collection of images. These 3D properties can used to augment SLAM systems.

end-to-end learned components with factor graphs. And some explore novel ways of estimating inter-frame motion and 3D structure with suitably trained networks. The resulting methods can significantly boost visual SLAM methods, providing higher precision and robustness and often generalize well beyond their training domain.

13.2 Deep Learning for Depth and Camera Pose

Deep neural networks can be trained to learn a mapping from an input space $\mathcal{Z}$ (e.g images) to the label space $\mathcal{X}$ (e.g depth). Modern neural networks consist of a large number of differentiable layers which can be composed into end-to-end trainable systems. The parameters of the network are optimized using variations of gradient descent over a loss function which measures how well the prediction of the network matches the groundtruth label.

With modern depth sensors and laser scanners, it is now possible to collect large amounts of 3D data. For example, depth sensors or lidar can be used to collect images paired with depth maps. In addition, rendering engines can be used to produce large scale, synthetic 3D datasets. As shown in Fig 13.2, neural networks can be trained on this data to predict 3D properties directly from input images, such as depth or relative camera pose. In this section, we show how the predictions from these networks can be used to augment SLAM systems.

13.2.1 Deep Learning for Depth Prediction

Given a large collection of image and depth pairs, neural networks can be trained predict depth (or some parameterization of depth) directly from a single input image. Eigen et al. [310] first showed that convolutional neural networks could be used to predict depth from a single image. Over the years, single image depth networks have continued to advance on account of both improved network architectures and

without mask module with mask module

Figure 13.3 MonoRec: The mask module in the network architecture allows to filter out moving objects based on brightness inconsistency across time [1184] (©2021 IEEE).

the expanding collection of 3D datasets. Modern depth networks often use a pre-trained vision transformer (ViT) backbone [288] with a convolutional decoder to upsample the coarse resolution ViT features and produce a final depth estimate [1160, 488]. To improve generalizability, these networks are often trained on combinations of over 20 datasets including both synthetic and real data.

Depth predictions have many immediate applications in SLAM, particularly in the monocular case. Monocular SLAM systems typically only produce reconstructions up to a scale factor and are subject to scale drift. Yang et al. [1223] and Hu et al. [488] showed that metric depth predictions could be used to combat scale drift of a monocular SLAM system. Depth predictions can also be used to robustify systems and avoid failure cases in challenging situations such as pure camera rotation, changing camera parameters and dynamic objects [667].

Furthermore, visual SLAM systems often only track a sparse subset of feature points, leading to sparse reconstructions of the scene. For example, areas with little texture are barely present in the resulting point cloud. While sparse point clouds are already useful, many downstream applications such as robot navigation or autonomous driving require a *dense* reconstruction.

MonoRec: Monocular Dense Reconstruction [1184]: One way to densify the output of a sparse SLAM system is to predict dense depth maps for every frame of the video. A dense reconstruction, such as the one shown in Figure 15.5, can be obtained by lifting all the pixels into 3D and accumulating them in a global reference frame using the poses produced by a SLAM system. However, simply using the depth maps produced by a depth network is suboptimal for a couple reasons. First, single image depth networks tend to produce geometrically inconsistent outputs between adjacent frames. Second, many scenes contain dynamic objects which would introduce trailing artifacts in a static reconstruction. *MonoRec* [1184] improves geometric consistency by augmenting the depth network with a cost volume and filters dynamic objects by predicting a moving object mask.

To improve geometric consistency, for each frame I_t, MonoRec [1184] builds a cost volume $C_t(d)$ aggregated over consecutive frames $\{I_{t'_1}, \ldots, I_{t'_n}\}$. The cost volume is

built by computing the photometric consistency of each pixel at a number of discrete depth steps d aggregated into a 3D volume. MonoRec uses a depth network which takes in both the input image I_t and its cost volume $C_t(d)$ to predict depth. The cost volume provides the network with additional context for producing geometrically consistency depths.

Cost volumes assume a static world and provide wrong signals if an object in the scene is moving. Therefore, MonoRec relies on an additional masking network which predicts a moving object segmentation mask M_t from the cost volume. The moving object segmentation mask is used to filter out dynamic points when lifting the predicting depths into 3D. Figure 15.5 shows that the masking module can filter out moving objects in the dense reconstruction.

MonoRec is trained using the sparse 3D point predictions of the SLAM system. Since the *Mask* module can benefit from having accurate depth and the *Depth* module can benefit from accurate dynamic masks, joint training of the two components can further boost performance, yielding accurate and consistent depth maps.

Behind the Scenes (BTS) [1185]: Dense pointclouds, as produced by MonoRec [1184], can accurately represent the 3D world observed in the video. However, regions that were not occluded are consequently not part of the reconstructed 3D scene. *Behind the Scenes (BTS)* tackles this shortcoming by predicting a volumetric reconstruction instead of a depth maps. Here, everything in the camera frustum is reconstructed, even the areas that are occluded in the current frame. In BTS, scenes are represented as volumetric density fields $\phi(\mathbf{x}) : \mathbb{R}^3 \rightarrow \mathbb{R}_+$, which map any point $\mathbf{x}$ in the camera frustum to a volumetric density σ. Through the established volume rendering technique, it is possible to synthesize any novel view from this scene representation. Overall, a volumetric reconstruction contains useful additional information for many downstream applications, especially in the autonomous driving and robotics domain.

13.2.2 Deep Learning for Camera Pose Prediction

Similar to how a neural network can be trained to predict depth, a neural network can also be trained to predict camera pose between pairs of frames or video sequences. Letting I_1 and I_2 be a pair of input images, we can design a neural network f_θ which takes the two images as input and predicts the relative camera pose from I_1 to I_2 denoted $f_\theta(I_1, I_2) = \boldsymbol{T}_1^2 \in SE3$. Given groundtruth poses $\boldsymbol{T}_1^{2^*} \in SE3$, we can define a loss

$$\mathcal{L}_{pose} = \left\| \left(\boldsymbol{T}_1^2\right)^{-1} \cdot \boldsymbol{T}_1^{2^*} \right\|, \qquad \text{where } \boldsymbol{T}_1^2 = f_\theta(I_1, I_2) \tag{13.1}$$

which measures the relative transformation between the predicted pose and the groundtruth pose. It is also possible to define other losses such as minimizing the

rotation angle or computing distance between predicted and ground truth quaternion representations of relative poses.

Early works for camera pose estimation used networks which concatenated the two frames as input and regressed camera pose parameters such as axis-angle [1118] or Euler angles [1294]. Given the relative pose between pairs of frames, one can recover the full trajectory of the camera by chaining the poses together by composition. TartanVO [1165] improved relative posed estimation by augmenting the input images with predicted optical flow and camera intrinsics, leading to better generalization on downstream datasets.

One issue with two view pose prediction is that relative pose errors can compound for longer video sequences. DeepVO [1161] proposed using a recurrent neural network which takes in a sequence of frames as input and predicts a sequence of camera poses. DeepVO extracts features from the input video sequence to generate a sequence of feature maps. These feature maps are then processed by an LSTM to predict a sequence of camera pose estimates.

While these works showed promising results, directly predicting camera pose is often less accurate than more classical SfM or SLAM techniques where geometry is more deeply embedded in the system [1086]. Furthermore, these techniques have a tendency to overfit to their training distribution and require training on large combinations of datasets to encourage better generalization.

13.2.3 Unsupervised Learning of Depth and Camera Pose

Supervised learning of depth and pose require groundtruth 3D data which might not always be available or can be difficult to obtain. It turns out there are ways to train *pose* and *depth* networks without explicit 3D supervision. One common technique uses view synthesis as a supervision signal [395, 1294]. Given two images, I_1 and I_2 with relative poses $\boldsymbol{T}_1^2$, we can generate a new image from the viewpoint of I_1 using pixels sampled from I_2.

To see how this is done, let $\pi_c : \mathbb{R}^3 \mapsto \mathbb{R}^2$ be the camera projection which takes a 3D point to its 2D pixel coordinates and $\pi_c^{-1} : \mathbb{R}^2 \times \mathbb{R}^+ \mapsto \mathbb{R}^3$ be the inverse projection function which takes a 2D pixel with depth d to its 3D point. Given these functions and relative pose $\boldsymbol{T}_1^2$, we can map a $(\boldsymbol{x}^1, d^1)$ pair in I_1 to its location in I_2

$$\boldsymbol{x}^2 = \pi_c \left(\boldsymbol{T}_1^2 \pi_c^{-1}(\boldsymbol{x}^1, d^1) \right) = \omega_{1\to 2}(\boldsymbol{x}^1, d^1) \tag{13.2}$$

barring occlusions and using $\omega_{1\to 2}$ used for shorthand. Given a depth map $\boldsymbol{d}$, (13.2) can be used to warp I_2 into the viewpoint of I_1 by defining the warped image

$$\tilde{I}_2(x) = I_1(w_{1\to 2}(\boldsymbol{x}, \boldsymbol{d}(x))) \tag{13.3}$$

where $I_1(\cdot)$ denotes bilinear interpolation. Since bilinear interpolation is locally differentiable with respect to pixel coordinates, it is possible to compute the deriva-

Figure 13.4 DVSO: By feeding monocular depth predictions from a deep neural network into a classical SLAM method like DSO [316], the monocular method "Deep Virtual Stereo Odometry" (DVSO) [1223] achieved highly accurate camera trajectories that were on par with state-of-the-art stereo-based methods. The deep network was trained in a self-supervised manner using stereo videos and lead to significant boost in precision and the capacity to recover scaled reconstruction from a single moving camera.

tive of the synthesized image $\tilde{I}_2$ with respect to depth and camera pose. Godard et. al [395] showed that a view synthesis loss could be used to train a depth network on stereo images. Following works showed that this idea can be taken even further and jointly trained a *depth* and a *pose* network using view synthesis supervision [1294, 1133].

Deep Virtual Stereo Odometry (DVSO) [1223]: On limitation of monocular SLAM systems is that they tend to be less robust and more prone to drift than stereo systems. DVSO showed that a classical visual odometry system, namely DSO[316], could be augmented using depth predictions from a neural network trained using view-synthesis supervision between stereo pairs of frames [395]. DVSO works by taking the depth predictions from the neural network and adding an additional factor that encourages the consistency between the predicted depth maps and the depth computed by DSO as shown in Figure 13.4.

Using the depth predicted by the neural network improved the reconstruction quality along with the accuracy and robustness of the poses predicted by the VO system. Additionally, since the depth predictions were in metric units, the method was able recover scale-accurate reconstructions even from a single camera. This work was coined "virtual stereo odometry" since in the monocular application the deep network essentially hallucinated the effect of a second camera.

D3VO: Deep Depth, Deep Pose and Deep Uncertainty for Monocular Visual Odometry [1224]: Beyond predictions of monocular depth, one can go further and also feed predictions of relative camera motion into the classical SLAM pipeline in order to ensure consistency of the recovered trajectory with the predictions of camera motion for consecutive frames. Direct visual SLAM methods like DSO [316] rely on color consistency across frames to infer localization and mapping. In the real world, this color consistency is not always guaranteed–in particular for non-Lambertian structures like shiny, metallic or glass structures corresponding points will likely not have the same color. While it is conceivable to explicitly model

the reflection properties of the world, this is computationally demanding and likely infeasible in a real-time setting.

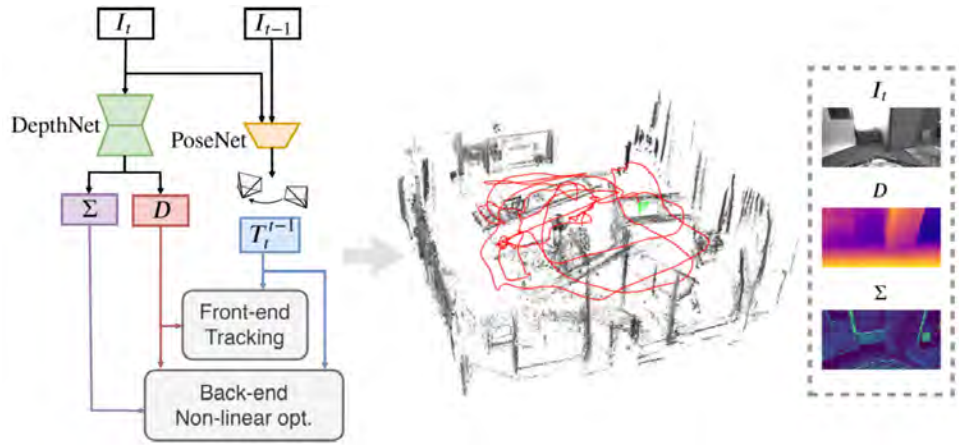


Figure 13.5 D3VO [1224] combines self-supervised neural networks for pose, depth, and uncertainty prediction with a direct SLAM system [316]. The pose and depth predictions from the networks are integrated into the SLAM system as additional factors in the factor graph, encouraging the state variables to the SLAM system to align with the predictions of the neural networks. By combining deep learning with factor graphs, D3VO benefits from the robustness of neural networks with the accuracy of direct SLAM. Image from [1224] (©2020 IEEE).

It turns out that one can train a neural network to predict locations where color is likely preserved across frames. Such predictions can also be fed into a direct SLAM pipeline, essentially down-weighting the residuals where color consistency is not expected. In this manner, D3VO[1224] feeds deep network predictions of monocular depth, relative inter-frame camera motion and uncertainty into a SLAM pipeline. As shown in Figure 13.5, D3VO use a *depth* and a *pose* network trained using view synthesis losses in a self-supervised manner. The predictions enter on the frontend in form of respective factor graph terms, but also in the backend in the form of respective loss terms. By combining depth and pose predictions from a neural network with direct visual odometry [316], D3VO is able to achieve highly accurate camera trajectories and is robust to failure cases common for monocular VO.

13.3 Deep Learning for Feature Matching and Optical Flow

The previous sections showed how neural networks can be used to predict 3D information such as depth and camera pose directly from input images. This feed-forward approach contrasts with the typical SLAM pipeline, where inference typically takes

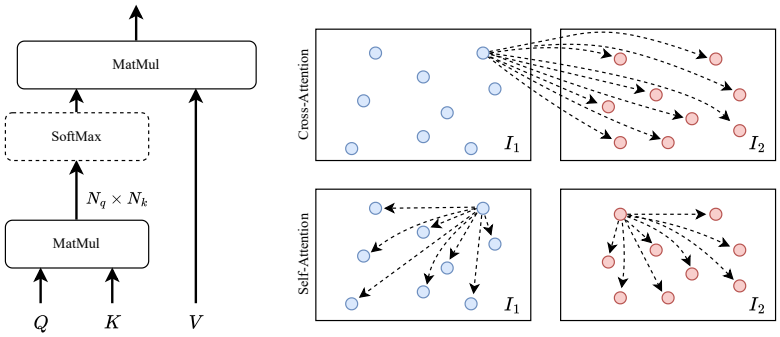


Figure 13.6 Illustration of Multi-Head self and cross attention commonly used in feature matching networks such as SuperGlue[976], LightGlue[678] and LoFTR[1057]. SuperGlue and LightGlue perform self and cross attention over feature vectors associated with image keypoints. LoFTR does not rely on keypoint detection and performs self and cross attention over dense feature grids.

the form of an optimization problem: solving for $\boldsymbol{x}^{\text{MAP}}$ given the noisy measurements $\boldsymbol{z}$ such as feature matches.

Predicting camera pose or depth is not an easy problem. To do so, the network essentially needs to learn multi-view geometry from scratch, such as the relationships between depth, camera pose and camera intrinsics. An alternative to this approach is to predict the measurements $\boldsymbol{z}$, such as feature matches, instead of the state variables directly.

13.3.1 Learning Feature Detectors and Descriptors

Indirect SLAM approaches rely on feature detectors and descriptors to extract a sparse set of visual measurements from the input images. First, salient regions of the images are identified, commonly refered to as *keypoints*. Then the local neighborhood of each keypoint is used to compute a vector descriptor which can be used to match keypoints between images.

Neural networks can be trained to replace the hand designed feature detectors and descriptors commonly used in SLAM and SfM pipelines. On such method, SuperPoint [275], proposed using a convolutional neural network to both (1) detect salient image regions and (2) assign feature descriptors to the detected keypoints. Similiarly, Universal Correspondence Network [222], showed that feature descriptors produced by a neural network can improve downstream feature matching accracy. Such networks are typically trained using a *contrastive* loss, which encourages the feature descriptors for matching points to be similar and encourages the distance between the feature descriptors for non-matching keypoints to be large.

13.3.2 Feature Matching

Given a set of keypoints produced by a pair of images, neural networks can also be used to improve the matching of keypoints between images. SuperGlue[976] introduced the idea of using a neural network to guide feature matching. SuperGlue operated over keypoints extracted from a pair of input images. SuperGlue iteratively updated feature vectors associated with each keypoint with self and cross attention layers as shown in Figure 13.6. The network outputs an assignment matrix, providing the likelihood of each keypoint in the first image with each keypoint in the second image (with an additional column for unmatched keypoints). SuperGlue showed that the predicted keypoint matches could improve accuracy on several downstream tasks such pose estimation. LightGlue[678] improved the accuracy and speed of SuperGlue with a collection of architecture and training improvements.

One potential limitation of SuperGlue and LightGlue is that they operate over a sparse set of image keypoints. While this has the advantage of reducing computation, it cannot use the full image which can limit the number of high quality matches on difficult examples such as images with limited texture. LoFTR [1057] proposed a network for dense pixel matching. Similar to SuperGlue and LightGlue, LoFTR leveraged self attention and cross attention (Figure 13.6). In order to reduce compute, LoFTR performed dense matching over coarse resolution feature grids, then refined the predictions at higher resolution.

The feature matches produced these systems can be directly integrated into indirection SLAM systems in the form of visual measurements and their corresponding factors. Deep feature matching can improve the robustness of SLAM systems in cases where traditional feature descriptors are not reliable.

13.3.3 Optical Flow and RAFT

Optical flow is closely related to the problem of feature matching. It is the task of estimating per-pixel motion between a pair of frames. This per-pixel motion–or flow field–can be used as measurements in a factor graph-based visual SLAM system.

Originally, optical flow was formulated as an optimization problem with a cost function that balanced two terms: a *data* term and a *regularization* term. The data term encourages the alignment of visually similar image regions while the regularization term encourages physically plausible motion fields. In more recent years, deep learning has emerged as a promising alternative [286, 1056, 510, 914]. In this section, we describe one such approach, RAFT [1085] (Figure 13.7), in more detail as it serves as the basis for DROID-SLAM [1086]. More recent optical flow networks have improved upon RAFT with the addition of multi-head attention and transformer blocks [501] such as those illustrated in Figure 13.6.

Figure 13.7 illustrates the overall design of RAFT. RAFT combines elements of both deep learning and traditional first order optimization. Features are first ex-

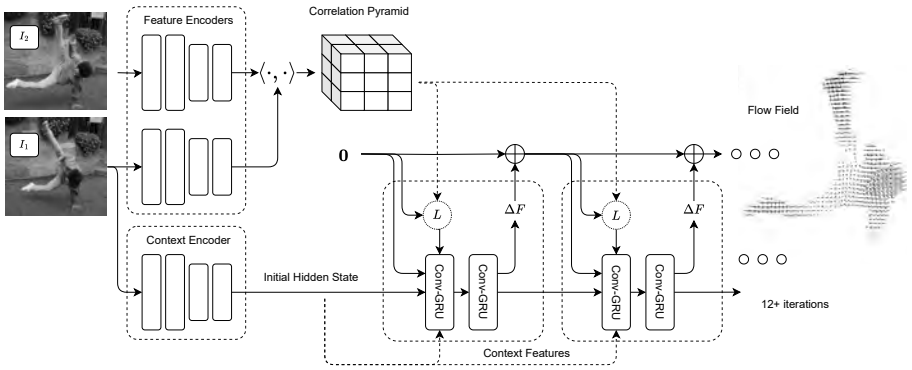


Figure 13.7 RAFT: Recurrent All Pairs Field Transforms for Optical Flow architecture. Features are extracted from the input images using a pair of residual networks. RAFT then builds a correlation volume by taking an all-pairs inner product between the features from I_1 and the features from I_2. Starting from a flow field initialized at 0, RAFT iteratively using the current estimate of optical flow to perform lookups from the correlation pyramid and applies a recurrent convolutional GRU to update the flow estimate.

tracted from the input images then used to build a 4D correlation volume. Starting with a flow field initialized at zero, the flow field is used to perform lookups from the correlation pyramid. The correlation features, alongside the current estimate of the flow field, are plugged into a Convolutional-GRU which produces an update to the flow field in addition to an updated *hidden state*. RAFT is trained on a combination of synthetic datasets with groundtruth optical flow [750, 286] but shows good generalization to real data [764].

Feature Extractor: The feature extractor is used to encode the input images into dense feature maps at lower resolution. RAFT uses a pair of feature extractors: (1) a appearance encoder g_θ and (2) a context encoder c_θ which map the input image with dimensions $(h_{in},\ w_{in},\ 3)$ to a lower resolution feature map with shape $(h,\ w,\ c)$ where c is the number of feature channels. By default, RAFT extracts features as 1/8 resolution so $(h,\ w) = (h_{in}/8,\ w_{in}/8)$ but some other networks use higher resolution.

Correlation Pyramid: The appearance features are used to build a multi-resolution correlation pyramid. The correlation volume is 4-dimensional tensor which has shape $h \times w \times h \times w$ where h and w are the dimensions of the appearence feature map as defined above. The correlation volume $\mathbf{C}$ is constructed by taking the inner product between all pixels in $g_\theta(I_1)$ with all pixels in $g_\theta(I_2)$

$$\mathbf{C}_{ijkl}(I_1, I_2) = \langle g_\theta(I_1)_{ij},\ g_\theta(I_2)_{kl} \rangle = \sum_m g_\theta(I_1)_{ijm} \cdot g_\theta(I_2)_{klm}. \tag{13.4}$$

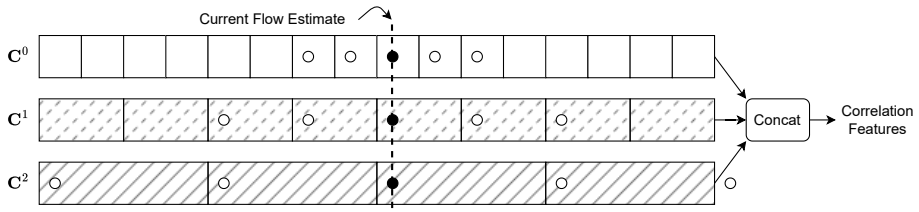


Figure 13.8 RAFT correlation lookup operator. This figure illustrates a 1D slice of the lookup operation. For the current flow estimate, RAFT constructs a local neighborhood and retrieves features from each level of the correlation pyramid which are then flattened and concatenated.

This operation can be implemented as a single matrix multiplication (by flattening the spatial dimensions and taking the transpose of the first argument) which is highly optimized on the GPU. RAFT doesn't just use a single correlation volume but constructs a multi-resolution correlation pyramid. The pyramid is constructed by iteratively performing average pooling on the last two dimensions of the correlation volume to produce a 4 level pyramid $\{\mathbf{C}^0, \mathbf{C}^1, \mathbf{C}^2, \mathbf{C}^3\}$ where the l^{th} level volume $\mathbf{C}^l$ has shape $h \times w \times h/2^l \times w/2^l$.

Correlation Lookup RAFT defines a *lookup* operator which uses uses the current estimate of the flow $\mathbf{f}$ to index from the correlation pyramid (Figure 13.8). For a given pixel (u, v) in $g_\theta(I_1)$, the flow field $\mathbf{f}$ is used to compute its estimated correspondence in I_2 by following the flow vector $(u', v') = \mathbf{f}(u, v) + (u, v)$.

RAFT then constructs a $(2r + 1) \times (2r + 1)$ neighborhood grid around (u', v')

$$N(u', v') = \{(u' + \Delta u, v' + \Delta v) \mid \Delta u, \Delta v \in \{-r, ..., r\}\} \tag{13.5}$$

where r is a hyper-parameter which defines the lookup radius. The neighborhood grid is used to index from $\mathbf{C}^0$ via bilinear interpolation. Features are sampled from the lower resolutions $\mathbf{C}^l$ by using neighboorhood grids centered around the rescaled correspondence coordinates $N(u'/2^l, v'/2^l)$. The final feature vector is formed by concatenating the features from each level as shown in Figure 13.8 and contains both fine resolution features about small displacements and coarse resolution information about large displacements.

Update Operator: The RAFT weight-tied update operator estimates of flow fields $(\boldsymbol{f}^1, ..., \boldsymbol{f}^N)$ starting from the zero field. With each iteration of the update operator, it produces an update direction which is applied to the current estimate to produce the next estimate $\boldsymbol{f}^{k+1} = \boldsymbol{\delta f}^k + \boldsymbol{f}^k$.

The update operator uses gated activation unit based on the GRU [225] where the linear layers are replaced with convolutions. The convolutions allow information to be spatially propagated over the image and weight tying and bounded activations

encourage convergence to a fixed point. Each iteration takes the following four inputs (each in the form of a 2D feature map): (1) the *hidden state* from the previous iteration, (2) the *context features*, (3) the current *flow* estimate and (4) *correlation features* from the lookup operator. These features are taken as input to the ConvGRU which produces an update hidden state and the flow update $\boldsymbol{\delta f}^k$.

13.3.4 Optical Flow as Visual Measurements

In a static scene the optical flow between a pair of frames is a function of depth and camera pose. To see this, let I_i and I_j be two images with known camera poses $\boldsymbol{T}_i^w$ and $\boldsymbol{T}_j^w$ (as camera to world transformations). Given a pixel (u, v) with depth d, we can compute the induced optical flow by first unprojecting the point (u, v) to depth d, then transforming the 3D point from frame i to j, followed by projecting onto camera j

$$\hat{\boldsymbol{f}}(u,v) = \boldsymbol{\pi}\left(\boldsymbol{T}_i^j \cdot \boldsymbol{\pi}^{-1}(u,v,d)\right) - (u,v) \qquad \boldsymbol{T}_i^j = (\boldsymbol{T}_j^w)^{-1} \cdot \boldsymbol{T}_i^w. \tag{13.6}$$

where $\boldsymbol{\pi} : \mathbb{R}^3 \mapsto \mathbb{R}^2$ which takes a 3d point in pixel coordinates and projects it to pixel coordinates, and the inverse projection function $\boldsymbol{\pi}^{-1} : \mathbb{R} \times \mathbb{R}^2 \mapsto \mathbb{R}^3$ which backprojects pixel coordinates (u, v) with depth d to its 3D coordinate.

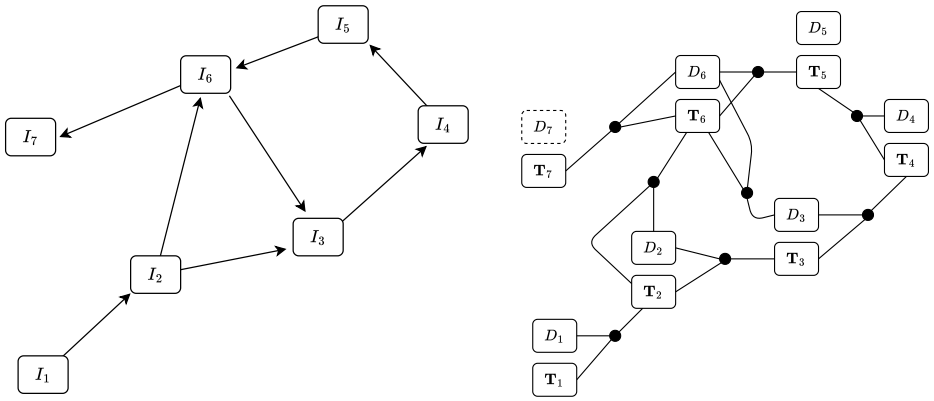
Here we assume pinhole cameras but complicated camera models with radial or tangential distortion can also be used if they are undistorted as a preprocessing step which is possible assuming the distortion model is known. For notational convenience, we overload $\boldsymbol{\pi}$ and $\boldsymbol{\pi}^{-1}$ with vectorized versions of these operators that act on dense $h \times w$ grids of pixels $(\boldsymbol{u}, \boldsymbol{v})$, such that the flow field can be represented in the form

$$\boldsymbol{h}_{ij}(\boldsymbol{T}_i^w, \boldsymbol{T}_j^w, \boldsymbol{d}_i) = \boldsymbol{\pi}\left(\boldsymbol{T}_i^j \cdot \boldsymbol{\pi}^{-1}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{d}_i)\right) - \begin{bmatrix}\boldsymbol{u} \\ \boldsymbol{v}\end{bmatrix} \tag{13.7}$$

where $\boldsymbol{d}$ is taken to be a $h \times w$ dense depth map.

13.3.5 Estimating Pose and Depth from Optical Flow

We can use factor graphs to go in the reverse direction: using estimates of optical flow to solve for depth and camera pose. Consider a set of images $\{I_1, ..., I_n\}$ and let $\mathcal{G}$ be a graph with edges connecting images with flow estimates (Figure 13.9a), such that $(i, j) \in \mathcal{G}$ means that we have an estimate of optical flow between images I_i and I_j. Let's denote the flow estimate between frames I_i and I_j as the measurement $\boldsymbol{z}_{ij}$ (because each edge in the frame graph gives rise to a single factor, we index the factors with the tuple (i, j) for notional convenience) and let the state variables $\boldsymbol{x}$ be the poses $(\boldsymbol{T}_1^w, ..., \boldsymbol{T}_n^w)$ and dense depth maps $(\boldsymbol{d}_1, ..., \boldsymbol{d}_n)$. We can construct a cost function over the states describing the flow that we should observe given the



(a) Flow Frame Graph (b) Flow Factor Graph

Figure 13.9 Flow frame graph and flow factor graph.

state variables

$$J(\boldsymbol{x}) \triangleq \sum_{(i,j)\in\mathcal{G}} \left\| \boldsymbol{z}_{ij} - \boldsymbol{h}_{ij}(\boldsymbol{T}_i^w, \boldsymbol{T}_j^w, \boldsymbol{d}_i) \right\|_{\boldsymbol{\Sigma}_{ij}}^2 \tag{13.8}$$

where the *measurement prediction function* $\boldsymbol{h}_{ij}$ was defined in (13.7).

This cost function gives rise to the factor graph in Figure 13.9b where each factor involves 3 state variables: the source pose, the destination pose and the the source depth. A couple notes on parameterization: in practice, it is better to parameterize the depth variables using inverse depth because this leads to better local approximations of the cost function. Since poses are manifold state variables, we also want to use the local parameterization $\exp\left(\boldsymbol{\xi}_i^{\wedge}\right)\boldsymbol{T}_i^w$.

13.4 Differentiable Bundle Adjustment and DROID-SLAM

We can use RAFT (or another off-the-shelf method) to compute optical flow, then minimize a least-squares optimization problem to solve for pose and depth. Alone, this doesn't work particularly well. Optical flow networks are trained on the task of optical flow, not 3D reconstruction or pose estimation. The loss function doesn't necessarily do a good job modeling what we care about in SLAM, particularly the accuracy of the camera poses. The second issue is that the output of our flow network is subject to outliers and all pixels are weighted equally in the cost function.

One solution is to use intermediate representations, such as optical flow, as inputs to a differentiable optimization layer. Using the techniques outlined in Chapter 4, we can perform backpropagation through the optimization layer to supervised this intermediate representation and, by extension, train the weights of our network. Furthermore, we can use the network to output corresponding confidence weights

to further shape the optimization landscape. To make this idea more concrete, recall in Chapter 1 that a factor graph gives rise to a cost function in the form

$$J(\boldsymbol{x}) \triangleq \sum_i \|\boldsymbol{e}_i(\boldsymbol{x}_i)\|^2_{\boldsymbol{\Sigma}_i} = \|\boldsymbol{e}(\boldsymbol{x})\|^2_{\boldsymbol{\Sigma}} \qquad \boldsymbol{e}_i(\boldsymbol{x}) = \boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i) \tag{13.9}$$

where $\boldsymbol{e}_i$ is the error vector representing the difference between the model of the world and the observed measurements. The error vector $\boldsymbol{e}_i$ typically represents reprojection error (indirect methods) or photometric error (direct methods).

Instead of modeling the error directly, we can instead use a neural network to produce the error and corresponding confidence. We can define a neural network as a function $\boldsymbol{e}_\theta$ which maps the current state $\boldsymbol{x}$ (e.g poses) and measurements $\boldsymbol{z}$ (e.g images) to an error vector $\boldsymbol{e} \in \mathbb{R}^m$ and parametrize covariance in a similar way. We can rewrite the cost function plugging in the learned functions $\boldsymbol{e}_\theta$ and $\boldsymbol{\Sigma}_\theta$

$$J(\boldsymbol{x}) \triangleq \|\boldsymbol{e}_\theta(\boldsymbol{z}, \boldsymbol{x})\|^2_{\boldsymbol{\Sigma}_\theta(\boldsymbol{z},\boldsymbol{x})} \tag{13.10}$$

allowing the neural networks to shape the cost function.

BANet [1072] was among the first works to integrate bundle adjustment into an end-to-end trainable architecture by defining a photometric error over feature maps extracted by a neural network. Lindenberger et. al [677] used a similar feature-metric error to improve the outputs of SfM systems. In this section, we outline DROID-SLAM [1086] and DPVO [1088] which build SLAM and VO systems using optical flow as an intermediate representation for differentiable BA.

13.4.1 DROID-SLAM Architecture

The architecture of DROID-SLAM (Figure 13.10) reuses many of the components of RAFT. It takes in two inputs: the sequence of input images $(I_1, ..., I_n)$ and a frame graph $\mathcal{G}$ with edges between co-visible pairs of images. It outputs a sequence of pose and depth estimates.

Feature Extraction and Visual Similarity: The input to the network is a sequence of frames $(I_1, ..., I_n)$ and a frame graph $\mathcal{G}$. Just like RAFT, DROID-SLAM uses a two feature extractors: a context encoder $\boldsymbol{c}_\theta$ and an appearance encoder $\boldsymbol{g}_\theta$–both residual networks which output features at 1/8 the input resolution. The appearance features are used to build a set of correlation pyramids following RAFT. A correlation pyramid is constructed for each edge (i, j) in the frame graph using the correlation between appearance features of I_i and I_j. In total, if there are E edges in the frame graph, E correlation pyramids will be constructed.

Update Operator: The update operator acts on the frame graph $\mathcal{G}$ to produce global pose and depth updates. Each edge (i, j) in the frame graph has an associated

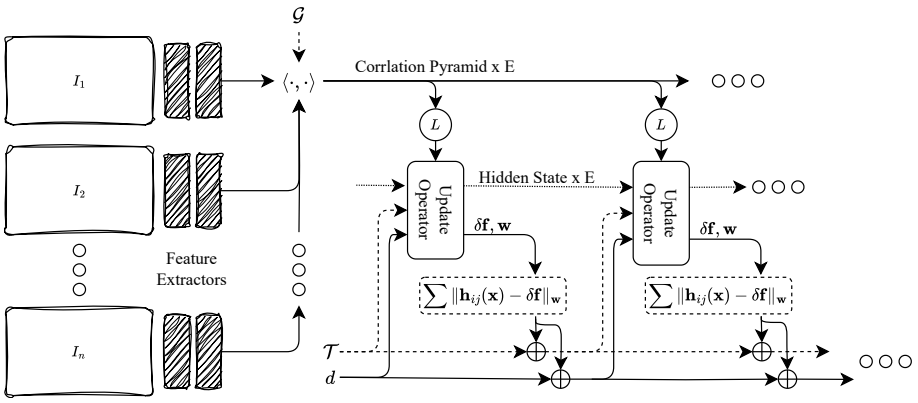


Figure 13.10 DROID-SLAM architecture: Features are extracted from the set of N input images and used to construct E correlation volumes (one for each edge in the frame graph). The update operator acts symmetrically on each edge in the frame graph. In each iteration, it uses the current estimate of the poses $\mathbf{T}_1, ..., \mathbf{T}_n$ and depths $\boldsymbol{d}_1, ..., \boldsymbol{d}_n$ to index from the correlation pyramid. The correlation features and hidden state are used to produce a flow revision $\delta \boldsymbol{f}$ and confidence weights $\boldsymbol{w}$ which parametrize the cost function. A couple Gauss-Newton updates are applied to generate pose and depth updates.

correlation pyramid $\boldsymbol{C}_{ij}$ and hidden state $\boldsymbol{q}_{ij}$. The hidden state $\boldsymbol{q}_{ij}$ is initialized from the context features of the source frame I_i.

Each iteration of the update operator uses the current estimate of poses and depth to compute the induced flow for each edge in the frame graph, using the measurement function defined in (13.7). Like RAFT, the update operator is a convolutional GRU, but here it is replicated for each edge in the frame graph. During each iteration, the update operator uses (1) the hidden state (2) correlation features (3) flow features and (4) the residual from the previous iteration to produce an updated hidden state. The updated hidden state is used to predict the measurement errors $\boldsymbol{\delta f}_{ij}$ (e.g. flow field updates) and an associated confidence map $\boldsymbol{w}_{ij}$.

Differentiable Bundle Adjustment: The error estimate and the confidence map are used to construct a NLS cost function of the form

$$J(\boldsymbol{x}) \triangleq \sum_{(i,j)\in\mathcal{G}} \left\| \boldsymbol{\delta f}_{ij} - \left[\boldsymbol{h}_{ij}(\boldsymbol{x}_{ij}) - \boldsymbol{h}_{ij}(\boldsymbol{x}_{ij}^0) \right] \right\|_{\boldsymbol{\Sigma}_{ij}}^2 \tag{13.11}$$

where the covariance is constructed by turning $\boldsymbol{w}_{ij}$ into a block diagonal matrix $\boldsymbol{\Sigma}_{ij} = \text{diag } \boldsymbol{w}_{ij}$ and $\boldsymbol{x}_{ij}^0$ are the initial pose and depth estimates at the start of the iteration. This cost function is saying that we want to update are states to drive the change in the measurement function $\boldsymbol{h}_{ij}(\boldsymbol{x}_{ij}) - \boldsymbol{h}_{ij}(\boldsymbol{x}_{ij}^0)$ to match the error $\boldsymbol{\delta f}_{ij}$ as predicted by the network. The weights $\boldsymbol{w}_{ij}$ give the network additional control of the optimization landscape by allowing it to place more weight on regions of the

image where it is confident. The depth and camera pose are updated by unrolling Gauss Newton iterations. The techniques outlined in Chapter 4 are used to perform backpropagation through each individual Gauss Newton step allowing the network to be trained end-to-end.

Training: DROID-SLAM is trained on the synthetic TartanAir dataset [1164] which includes groundtruth depth and camera pose. Training involves a weighted combination of a *pose* and a *flow* loss. The *pose* loss penalizes the difference between the predicted poses of the network and the ground truth poses. Likewise, the *flow* loss penalizes the difference between the optical flow induced by the predicted pose and depth (13.6) and the optical flow induced by the groundtruth pose and depth.

13.4.2 DROID-SLAM Inference

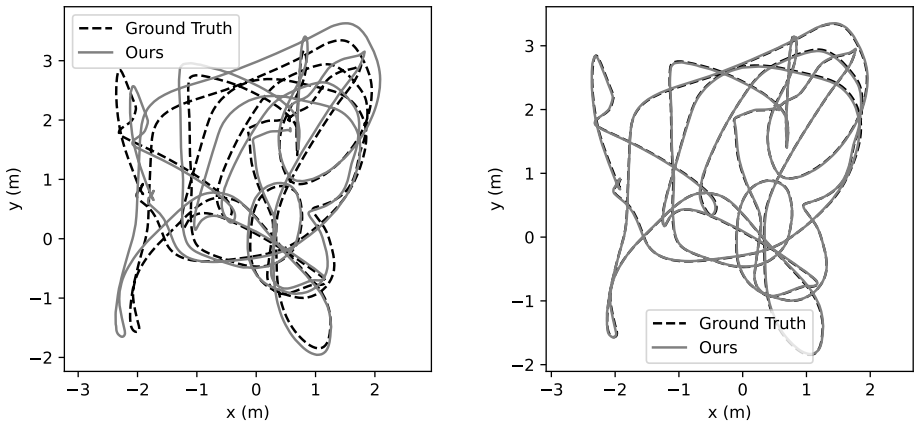
The network is trained in an *in-place* setting, where it receives a fixed number of frames as input. There are several modifications that need to be made to operate in the online setting where the frame graph is dynamic and new frames are being constantly added and removed.

Motion Filtering: For initialization, it is important to select frames where there is sufficient motion. DROID-SLAM uses a simple probe to filter frames based on relative motion from the last keyframe. If I_{t-1} is the last keyframe and I_t is the incoming frame, it constructs a frame graph with one edge: $I_t \rightarrow I_{t+1}$, then runs the update operator to compute the flow revisions $\delta \boldsymbol{f}$. If the magnitude of $\delta \boldsymbol{f}$ is below a threshold, the frame is discarded, otherwise it is kept.

Initialization: Once a sufficient number of frames have been accumulated, DROID-SLAM attempts to initialize. It constructs a frame graph where each frame is connected to its temporal neighbors, then runs several iterations of the update operator. Despite the simplicity of this initialization strategy, it works quite well on all benchmark datasets. Of course, this would fail in cases of pure rotation.

Frontend: Once a new frame is added, edges are added to the frame graph to include the new frame. Temporal edges are always added bidirectionally between the keyframe and the previous two frames. Induced optical flow is also used to compute the distance between all pairs of keyframes and new edges are added between frames with high co-visibility. DROID-SLAM takes steps to avoid adding too many edges such as applying non-maximal suppression to neighboring edges.

Global Optimization: DROID-SLAM also uses a backend process to optimize the full frame graph jointly. It first adds edges between temporally adjacent frames then uses the current estimate of poses and depths to compute optical flow between all pairs of frames. Long distance edges are added to the graph if the optical flow is



(a) Frontend Only (b) After Global Optimization

Figure 13.11 Estimated trajectory compared to ground truth on the `V1_02_medium` scene in the EuRoC benchmark. Global optimization reduces the rmse ATE from 16.5cm to 1.2cm.

below a certain threshold. The long distance edges enable min-range loop closure assuming the drift isn't too severe but DROID-SLAM doesn't include any relocalization module, so large loops with lots of drift cannot be closed. As Figure 13.11 shows, global optimization with mid-range loop closure can significantly improve the accuracy of the estimated trajectory.

13.4.3 Generalizing to Other Modalities

The factor graph formulation makes it easy to generalize DROID-SLAM to RGB-D and stereo video without needing to retrain the network. In addition to RGB-D and stereo, this approach has also been generalized to visual-inertial SLAM [871] and event cameras [767].

RGB-D Video: With depth video, it is sufficient to simply add another factor encouraging the predicted depth to be close to the sensor depth. Sensor depth estimates are typically sparse, so this is only applied on pixels which have a valid sensor reading, producing the updated cost function

$$J(\boldsymbol{x}) \triangleq \sum_{(i,j)\in\mathcal{G}} \|\boldsymbol{z}_{ij} - \boldsymbol{h}_{ij}(\boldsymbol{x})\|^2 + \sum_i \left\|\bar{\boldsymbol{d}}_i - \boldsymbol{d}_i\right\|^2_{\boldsymbol{\Sigma}_d} \tag{13.12}$$

where $\bar{\boldsymbol{d}}_i$ is the sensor depth observation and $\boldsymbol{\Sigma}_d$ is the sensor-covariance which is represented as a diagonal matrix with zeros on the missing sensor values.

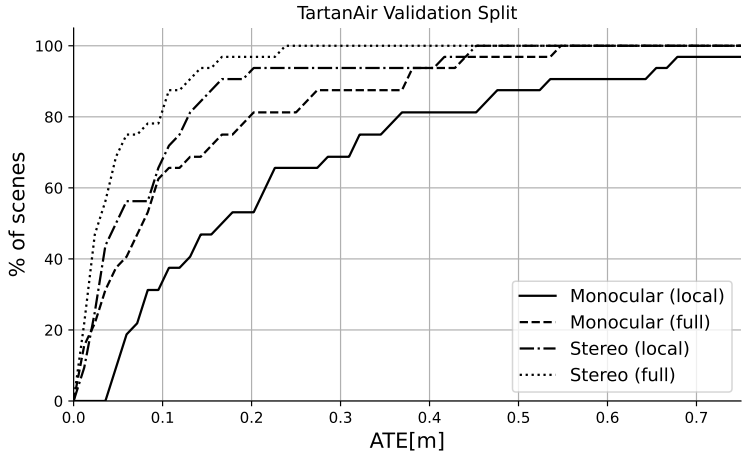


Figure 13.12 Comparison of monocular and stereo versions of DROID-SLAM with and without global optimization. DROID-SLAM is run all combinations on the 32 TartanAir validation sequences and plot the percentage of trajectories with absolute trajectory error (ATE) less than each threshold. Having stereo frames significantly helps–partially because it solves the problem of scale drift. Both the monocular and stereo system benefit from global optimization.

Stereo Video: Generalizing to stereo videos is not much harder. In the stereo case, each input consists of a time-synchronized left and right frame taken from a stereo rig. It is assumed that the relative pose between the left and the right frame is known. DROID-SLAM simply adds both the left and right frames to the frame graph and, for every left frame, and edge is added connecting it to its right pair. The relative pose between the left and right frame is fixed during optimization. In Figure 13.12, DROID-SLAM shows that stereo pairs of frames an improve performance in addition to the performance benefit of using global optimization.

13.4.4 Deep Patch Visual Odometry (DPVO)

Deep Patch Visual Odometry [1088] is closely related to DROID-SLAM but instead operates on sparse *patches* instead of dense frames. This turns out to have several advantages, not just for computational efficiency, but also accuracy.

Patch Graph: DPVO operates on patches instead of full frames and uses a *patch graph* data structure to store the association between patches and images. The patch graph is a bipartite graph representing the patch lifetimes; edges in the graph connect patches with frames.

Each patch is represented as a fronto-parallel plane in the image from which it

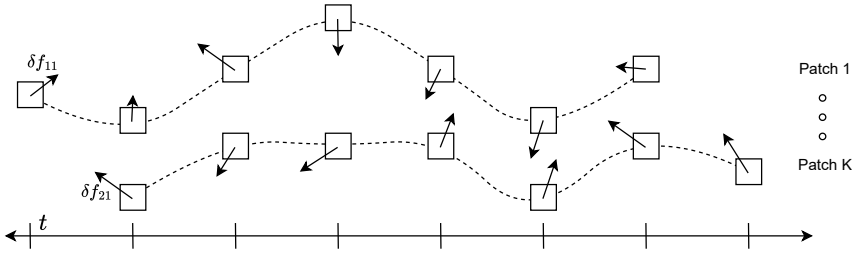


Figure 13.13 DPVO patch graph: each patch is tracked through time. The update operator outputs flow revisions $\boldsymbol{\delta f}$ for each edge in the graph, denoted by the error vector in the figure.

was extracted. Thus, each patch is parameterized by a single (inverse) depth. To reproject patches between frames, we update the measurement function

$$\boldsymbol{h}_{ij}(\boldsymbol{T}^w_{s(i)}, \boldsymbol{T}^w_j, d_i) = \boldsymbol{\pi}\left(\boldsymbol{T}^j_{s(i)} \cdot \boldsymbol{\pi}^{-1}(\boldsymbol{u}, \boldsymbol{v}, d_i)\right) - \begin{bmatrix}\boldsymbol{u}\\ \boldsymbol{v}\end{bmatrix} \tag{13.13}$$

where i is the patch index and $s(i)$ is the source index, or the index of the frame from which patch i was extracted. $\boldsymbol{u}$ and $\boldsymbol{v}$ are the pixel coordinates of the patch and d_i is a single scalar depth, shared over the full patch.

Patch Extraction: Each incoming image is processed by a feature encoder and a context encoder which outputs features at 1/4 resolution. Patch centroids are randomly sampled and used to construct a 3×3 patch with one pixel offsets (1px in the resolution of the extracted feature maps so actually this corresponded to 4px in the original image). DPVO uses a very simple method for constructing the patch graph. Each patch is connected to all the frames within a radius of r timesteps. Computation can be traded off for accuracy by selecting more patches and increasing their lifetime.

Update Operator: The update operator acts on edges in the patch graph. Like RAFT and DROID-SLAM, the update operator used by DPVO is a recurrent weight-tied unit but incorporates several new components. It performs temporal convolutions across the patch lifetime in addition to aggregation layers which pool features across frames and patches. The update operator produces a per-edge flow update $\boldsymbol{\delta f}$ and accompanying confidence weights $\boldsymbol{w}$. These are used to construct a cost function using (13.7) as the measurement function and the pose and depth updates are produced by unrolling two GN iteration.

Inference: DPVO starts by accumulating frames for initialization. Once a set num-

ber of frames is accumulated, DPVO initializes by running several iterations of the update operator. Subsequently, when each new frame is added, patches are extracted from the new frame and edges are added to the patch graph–both forward edges connecting previous patches with the new frame and backward edges connecting the new patches with previous frames. Next, the update operator is run to refine the depth and pose estimates. Like DROID-SLAM, DPVO also implements keyframing to ensure that the optimization window is sufficiently large.

Loop Closure and Global Optimization: DPVO is purely a visual odometry system and cannot perform loop closures outside the optimization window. Deep Patch Visual SLAM (DPVS) [679] expands on DPVO by adding global optimization and loop closure. DPVS can perform long-range loop closures using a relocalization module and short-range loop closures based on proximity detection. Unlike DROID-SLAM, DPVS can maintain real-time performance on a single GPU.

13.5 DuSt3R

As shown in the previous sections, modern SLAM systems can be described from a high level point of view as pipelines involving a series of steps: first, obtaining pixel correspondences between frames, being as keypoints, patches or dense optical flow; then, modeling the factor graph associated between the state variable $\boldsymbol{x}$ (*e.g.,* camera parameters and depth maps) and the observed correspondences; and finally solving for the state variable update that would minimize the overall cost function $J(\boldsymbol{x})$. Note that, in the larger context of SfM, camera parameters to be estimated would also include the intrinsic calibration of each camera – those are generally assumed to be given in the context of SLAM. In previous chapters, we have seen how learning-based approach can be inserted within each of these steps (*e.g.,* neural network for estimating the optical flow), but the SLAM pipeline remains heavily handcrafted and rather complex overall.

Recently, a novel type of learning-based approach has emerged. Instead of breaking down the simultaneous-localization-and-mapping problem as a series of atomic steps, DUSt3R [1162] proposes instead to let a neural network solve all steps at once: solely given images, *it directly outputs camera poses, dense depth maps, and even intrinsic parameters*, without requiring any explicit solvers nor iterative algorithms. Note that applications for this type of network go well beyond SLAM, as it additionally makes no assumption on the temporal relation between the input images, and therefore more generally applies to SfM and dense 3D reconstruction in general.

In reality, however, tasking a network with reconstructing a large scene from possibly thousands of frames still seems out of reach, and in practice, DUSt3R only solves a *local* version of the 3D reconstruction problem involving just two input images. As we will see later, in the end, a form of factor graph is still necessary

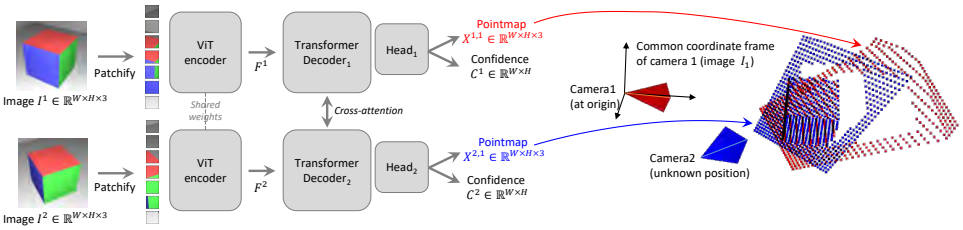


Figure 13.14 **Architecture of the network.** Two views of a scene (I^1, I^2) are first encoded in a Siamese manner with a shared ViT encoder. The resulting token representations F^1 and F^2 are then passed to two transformer decoders that constantly exchange information via cross-attention. Finally, two regression heads output the two corresponding pointmaps and associated confidence maps. Importantly, the two pointmaps are expressed in the same coordinate frame of the first image I^1. The network is trained using a simple regression loss (Eq. (13.17)). Image from [1162] (©2024 IEEE).

for performing SLAM (or large-scale 3D reconstruction) given a set of local 3D reconstructions. The difference with previous methods is that, due to the richness of its output, DUSt3R simplifies and expands the possibilities of SLAM systems, making them far more robust and enabling new features like handling varying camera intrinsics (*e.g.,* dynamic zoom-in / zoom-out).

Overview. In a nutshell, it is able to perform Dense Unconstrained Stereo 3D Reconstruction from *un-calibrated* and *un-posed* cameras. Its main component is a network that can regress a dense and accurate scene representation solely from a *pair* of images, without prior information regarding the scene nor the cameras (not even the intrinsic parameters). The resulting scene representation is based on *3D pointmaps* with rich properties that simultaneously encapsulate (a) the scene geometry, (b) the relation between pixels and scene points and (c) the relation between the two viewpoints. From this output alone, practically all scene parameters (*i.e.,* cameras and scene geometry) can be straightforwardly extracted. This is possible because the network jointly processes the input images and the resulting 3D pointmaps, thus learning strong geometric and shape priors which are reminiscent of those commonly leveraged in multi-view stereo, like shape from texture, shading or contours [1003].

Pointmap. We denote a dense 2D field of 3D points as a *pointmap* $X \in \mathbb{R}^{W\times H\times 3}$. In association with its corresponding RGB image I of resolution $W \times H$, X forms a one-to-one mapping between image pixels and 3D scene points, *i.e.,* $I_{i,j} \leftrightarrow X_{i,j}$, for all pixel coordinates $(i, j) \in \{1 \dots W\} \times \{1 \dots H\}$. We assume here that each camera ray hits a single 3D point, *i.e.,* ignoring the case of translucent surfaces.

Given camera intrinsics $K \in \mathbb{R}^{3\times 3}$, the pointmap X of the observed scene can be straightforwardly obtained from the ground-truth depthmap $D \in \mathbb{R}^{W\times H}$ as $X_{i,j} = K^{-1} D_{i,j} \, [i, j, 1]^\top$. Here, X is expressed in the camera coordinate frame. In

the following, we denote as $X^{n,m}$ the pointmap X^n from camera n expressed in camera m's coordinate frame:

$$X^{n,m} = P_m P_n^{-1} h\left(X^n\right) \tag{13.14}$$

where $P_m, P_n \in \mathbb{R}^{3\times4}$ are the world-to-camera poses for images m and n, and $h : (x, y, z) \to (x, y, z, 1)$ is the homogeneous mapping.

13.5.1 A Network for Generalized Stereo Reconstruction

Problem definition. We build a network that solves the 3D reconstruction task for the generalized stereo case through direct regression. The network f takes as input two RGB images $I^1, I^2 \in \mathbb{R}^{W\times H\times3}$ and outputs two corresponding pointmaps $X^{1,1}, X^{2,1} \in \mathbb{R}^{W\times H\times3}$ with associated confidence maps $C^{1,1}, C^{2,1} \in \mathbb{R}^{W\times H}$. Note that both pointmaps are expressed in the *same* coordinate frame of I^1, which radically differs from existing approaches but offers multiple key advantages (see 13.5.3).

Network architecture. As shown in Fig. 13.14, the architecture of the network is composed of two identical branches (one for each image) comprising each an image encoder, a decoder and a regression head. The two input images are first encoded in a Siamese manner by the same weight-sharing ViT encoder [288], yielding two token representations F^1 and F^2:

$$F^1 = \text{Encoder}(I^1), F^2 = \text{Encoder}(I^2).$$

The network then reasons over both of them jointly in the decoder, a generic transformer network equipped with cross attention. Each decoder block thus sequentially performs self-attention (each token of a view attends to tokens of the same view), then cross-attention (each token of a view attends to all other tokens of the other view), and finally feeds tokens to a MLP. Importantly, information is constantly shared between the two branches during the decoder pass. This is crucial in order to output properly aligned pointmaps. Namely, each decoder block attends to tokens from the other branch:

$$\begin{aligned} G_i^1 &= \text{DecoderBlock}_i^1\left(G_{i-1}^1, G_{i-1}^2\right), \\ G_i^2 &= \text{DecoderBlock}_i^2\left(G_{i-1}^2, G_{i-1}^1\right), \end{aligned}$$

for $i = 1, \ldots, B$ for a decoder with B blocks and initialized with encoder tokens $G_0^1 := F^1$ and $G_0^2 := F^2$. Here, $\text{DecoderBlock}_i^v(G^1, G^2)$ denotes the i-th block in branch $v \in \{1, 2\}$, G^1 and G^2 are the input tokens, with G^2 the tokens from the other branch. Finally, in each branch a separate regression head takes the set of decoder tokens and outputs a pointmap and an associated confidence map:

$$\begin{aligned} X^{1,1}, C^{1,1} &= \text{Head}^1\left(G_0^1, \ldots, G_B^1\right), \\ X^{2,1}, C^{2,1} &= \text{Head}^2\left(G_0^2, \ldots, G_B^2\right). \end{aligned}$$

It should be noted that this generic architecture never explicitly enforces any geometrical constraints. Hence, pointmaps do not necessarily correspond to any physically plausible camera model (but they closely fit in practice). Rather, we let the network learn all relevant priors present from the train set, which only contains geometrically consistent pointmaps. Using a generic architecture allows to leverage strong pretraining techniques, ultimately surpassing what existing task-specific architectures can achieve.

13.5.2 Regression Loss and Confidence-Aware Loss

3D Regression loss. The sole training objective is based on regression in the 3D space. Let us denote the ground-truth pointmaps as $\hat{X}^{1,1}$ and $\hat{X}^{2,1}$, obtained from Eq. (13.14) along with two corresponding sets of valid pixels $\mathcal{D}^1, \mathcal{D}^2 \subseteq \{1 \dots W\} \times \{1 \dots H\}$ for which the ground-truth is defined. The regression loss for a valid pixel $i \in \mathcal{D}^v$ in view $v \in \{1, 2\}$ is simply defined as the Euclidean distance:

$$\ell_{\text{regr}}(v, i) = \left\| \frac{1}{z} X_i^{v,1} - \frac{1}{\bar{z}} \hat{X}_i^{v,1} \right\|. \tag{13.15}$$

To handle the scale ambiguity between prediction and ground-truth, we normalize the predicted and ground-truth pointmaps by scaling factors $z = \text{norm}(X^{1,1}, X^{2,1})$ and $\bar{z} = \text{norm}(\hat{X}^{1,1}, \hat{X}^{2,1})$, respectively, which simply represent the average distance of all valid points to the origin:

$$\text{norm}(X^1, X^2) = \frac{1}{|\mathcal{D}^1| + |\mathcal{D}^2|} \sum_{v \in \{1,2\}} \sum_{i \in \mathcal{D}^v} \|X_i^v\|. \tag{13.16}$$

Confidence-aware loss. In reality, there are ill-defined 3D points, *e.g.,* in the sky or on translucent objects. More generally, some parts in the image are typically harder to predict than others. We thus jointly learn to predict a score for each pixel which represents the confidence that the network has about this particular pixel. The final training objective is the *confidence-weighted regression loss* from Eq. (13.15) over all valid pixels:

$$\mathcal{L}_{\text{conf}} = \sum_{v \in \{1,2\}} \sum_{i \in \mathcal{D}^v} C_i^{v,1} \ell_{\text{regr}}(v, i) - \alpha \log C_i^{v,1}, \tag{13.17}$$

where $C_i^{v,1}$ is the confidence score for pixel i, and α is a hyper-parameter controlling the regularization term [229]. To ensure a strictly positive confidence, we typically define $C_i^{v,1} = 1 + \exp c_i^{v,1} \gg 0$, with $c_i^{v,1} \in \mathbb{R}$. This has the effect of forcing the network to extrapolate in harder areas, *e.g.,* those ones covered by a single view. Training network f with this objective allows to estimate confidence scores without an explicit supervision. Examples of input image pairs with their corresponding outputs are shown in Fig. 13.15.

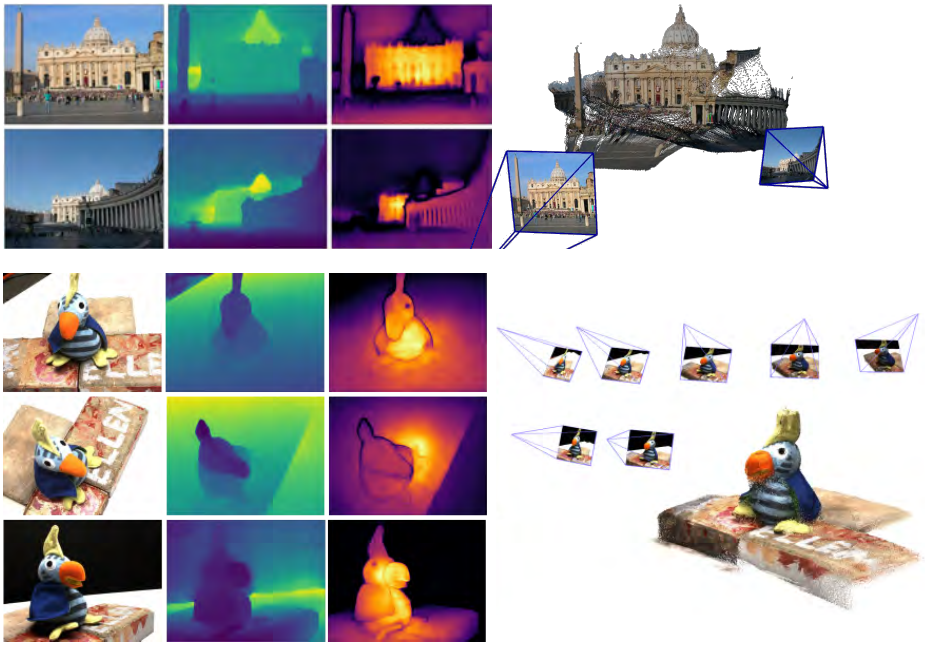


Figure 13.15 **Reconstruction examples** on two scenes never seen during training. From left to right: RGB, depth map, confidence map, reconstruction. The top scene shows the raw result output from $f(I^1, I^2)$. The both scene shows the outcome of global alignment (Sec. 13.5.4) Image from [1162] (©2024 IEEE).

13.5.3 Downstream Applications

Rich properties of the output pointmaps allows us to recover all scene parameters as described below.

Recovering intrinsics. By definition, the pointmap $X^{1,1}$ is expressed in I^1's coordinate frame. It is therefore possible to estimate the camera intrinsic parameters by solving a simple optimization problem. We assume that the principal point is approximately centered and pixels are squares, hence only the focal length f_1^* remains to be estimated:

$$f_1^* = \arg\min_{f_1} \sum_{i=0}^{W} \sum_{j=0}^{H} C_{i,j}^{1,1} \left\| (i', j') - f_1 \frac{(X_{i,j,0}^{1,1}, X_{i,j,1}^{1,1})}{X_{i,j,2}^{1,1}} \right\|,$$

with $i' = i - \frac{W}{2}$ and $j' = j - \frac{H}{2}$. Fast iterative solvers, *e.g.,* based on the Weiszfeld algorithm [885], can find the optimal f_1^* in a few iterations. For the focal f_2^* of the second camera, the simplest option is to perform the inference for the pair (I^2, I^1) and use above formula with $X^{2,2}$ instead of $X^{1,1}$.

Relative pose estimation. One way is to perform 2D matching and recover intrinsics as described above, then estimate the Epipolar matrix and recover the relative

pose [444]. Another, more direct way is to compare the pointmaps $X^{1,1} \leftrightarrow X^{1,2}$ (or, equivalently, $X^{2,2} \leftrightarrow X^{1,2}$) using Procrustes alignment [716] to get the scaled relative pose $P^* = \sigma^*[R^*|t^*]$:

$$P^* = \underset{\sigma,R,t}{\arg\min} \sum_i C_i^{1,1} C_i^{1,2} \left\| \sigma(RX_i^{1,1} + t) - X_i^{1,2} \right\|^2,$$

which can be achieved in closed-form. Procrustes alignment is, unfortunately, sensitive to noise and outliers. A more robust solution is to rely on RANSAC [336] with PnP [444, 647].

13.5.4 Global Alignment

The network f presented in the previous section can only handle a pair of images. DUSt3R [1162] also proposed a simple post-processing optimization for larger scenes. It enables the alignment of pointmaps predicted from multiple images into a joint 3D space. This is possible thanks to the rich content of the pointmaps, which encompasses by design two aligned point-clouds and their corresponding pixel-to-3D mapping.

Pairwise graph. Given a set of images $\{I^1, I^2, \ldots, I^N\}$ for a given scene, we first construct a connectivity graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where N images form vertices $\mathcal{V}$ and each edge $e = (n, m) \in \mathcal{E}$ indicates that images I^n and I^m share some visual content. To that aim, we either use existing off-the-shelf image retrieval methods, or we pass all pairs through network f (inference takes NEW ≈25ms on a H100 GPU) to measure their overlap from the average confidence in both pairs, and then filter out low-confidence pairs.

Global optimization. We use the connectivity graph $\mathcal{G}$ to recover *globally aligned* pointmaps $\{\chi^n \in \mathbb{R}^{W \times H \times 3}\}$ for all cameras $n = 1 \ldots N$. To that aim, we first predict, for each image pair $e = (n, m) \in \mathcal{E}$, the pairwise pointmaps $X^{n,n}, X^{m,n}$ and their associated confidence maps $C^{n,n}, C^{m,n}$. For the sake of clarity, let us define $X^{n,e} := X^{n,n}$ and $X^{m,e} := X^{m,n}$. Since the goal involves to express all pairwise predictions in a common coordinate frame, we introduce a pairwise pose $P_e \in \mathbb{R}^{3\times4}$ and scaling $\sigma_e > 0$ associated to each pair $e \in \mathcal{E}$. We then formulate the following optimization problem:

$$\chi^* = \underset{\chi,P,\sigma}{\arg\min} \sum_{e\in\mathcal{E}} \sum_{v\in e} \sum_{i=1}^{HW} C_i^{v,e} \left\| \chi_i^v - \sigma_e P_e X_i^{v,e} \right\|. \tag{13.18}$$

Here, with some abuse of notation, we write $v \in e$ for $v \in \{n, m\}$ if $e = (n, m)$. The idea is that, for a given pair e, the *same* rigid transformation P_e should align both pointmaps $X^{n,e}$ and $X^{m,e}$ with the world-coordinate pointmaps χ^n and χ^m, since $X^{n,e}$ and $X^{m,e}$ are by definition both expressed in the same coordinate frame. To avoid the trivial optimum where $\sigma_e = 0$, $\forall e \in \mathcal{E}$, we enforce that $\prod_e \sigma_e = 1$.

Recovering camera parameters. A straightforward extension to this framework enables to recover all cameras parameters. By simply replacing

$$\chi^n_{i,j} := P_n^{-1} h(K_n^{-1} D^n_{i,j}[i, j, 1]^\top)$$

(*i.e.,* enforcing a standard camera pinhole model as in Eq. (13.14)), we can thus estimate all camera poses $\{P_n\}$, associated intrinsics $\{K_n\}$ and depthmaps $\{D^n\}$ for $n = 1 \dots N$.

13.6 MASt3R

Two-view 3D reconstruction priors, pioneered by DUSt3R [1162] has created a paradigm shift in structure-from-motion (SfM) by capitalizing on curated 3D datasets. Beyond intrinsics and relative poses, DUSt3R can also obtain reliable pixel correspondences from pointmaps, by looking for reciprocal matches in some invariant feature space [1162, 303, 995, 1190]. While such a scheme works well even in presence of extreme viewpoint changes, the resulting correspondences are rather imprecise, yielding suboptimal accuracy. This is a rather natural result as (i) regression is inherently affected by noise, and (ii) because DUSt3R was never explicitly trained for matching.

13.6.1 Matching Head

For these reasons, MASt3R proposed to extend DUSt3R by adding a second head that outputs two dense feature maps D^1 and $D^2 \in \mathbb{R}^{H \times W \times d}$ of dimensionality d:

$$D^1 = \text{Head}^1_{\text{desc}}([H^1, H'^1]), \tag{13.19}$$

$$D^2 = \text{Head}^2_{\text{desc}}([H^2, H'^2]). \tag{13.20}$$

The head is implemented as a simple 2-layers MLP interleaved with a non-linear GELU activation function [455]. Each local feature is normalized to the unit norm.

13.6.2 Matching Objective

MASt3R's matching objective is to encourage each local descriptor from one image to match with at most a single descriptor from the other image that represents the same 3D point in the scene. To that aim, MASt3R leverages the infoNCE [1124] loss over the set of ground-truth correspondences $\hat{\mathcal{M}} = \{(i, j) | \hat{X}^{1,1}_i = \hat{X}^{2,1}_j\}$:

$$\mathcal{L}_{\text{match}} = - \sum_{(i,j) \in \hat{\mathcal{M}}} \log \frac{s_\tau(i, j)}{\sum_{k \in \mathcal{P}^1} s_\tau(k, j)} + \log \frac{s_\tau(i, j)}{\sum_{k \in \mathcal{P}^2} s_\tau(i, k)}, \tag{13.21}$$

$$\text{with } s_\tau(i, j) = \exp\left[-\tau D^{1\top}_i D^2_j\right]. \tag{13.22}$$

Here, $\mathcal{P}^1 = \{i|(i,j) \in \hat{\mathcal{M}}\}$ and $\mathcal{P}^2 = \{j|(i,j) \in \hat{\mathcal{M}}\}$ denote the subset of considered pixels in each image and τ is a temperature hyper-parameter. Note that this matching objective is essentially a cross-entropy *classification* loss: contrary to the regression loss in DUSt3R the network is only rewarded if it gets the correct pixel right, not a nearby pixel. This strongly encourages the network to achieve high-precision matching. Finally, both regression and matching losses are combined to get the final training objective:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{conf}} + \beta \mathcal{L}_{\text{match}} \tag{13.23}$$

13.7 Extending MASt3R to SfM and SLAM

DUSt3R and MASt3R are designed to estimate pointmaps between pairs of images. As we saw, global optimization can extend MASt3R and DUSt3R to produce multi-frame reconstructions. In this section, we explore works which integrate MASt3R into full SfM and SLAM systems which are capable of reconstruction full video sequences and large collections of frames.

13.7.1 MASt3R-SfM

As MASt3R [649] is able to perform local 3D reconstruction and matching in a single forward pass, MASt3R-SfM [297] proposed to extended it to a fully-integrated SfM pipeline that can handle completely unconstrained input image collections, *i.e.,* ranging from a single view to large-scale scenes, possibly without any camera motion. Since MASt3R is fundamentally limited to processing image pairs, it scales poorly to large image collections. To address this issue, MASt3R-SfM modified its frozen encoder to perform fast image retrieval with negligible computational overhead, resulting in a scalable SfM method with quasi-linear complexity in the number of images. Thanks to the robustness of MASt3R to outliers, the proposed method is able to completely get rid of RANSAC. The SfM optimization is carried out in two successive gradient descents based on frozen local reconstructions output by MASt3R: first, using a matching loss in 3D space; then with a 2D reprojection loss to refine the previous estimate.

13.7.2 MUSt3R

MASt3R-SfM works seamlessly with dozens of images, but when feeding even many images, the pairwise nature of DUSt3R and MASt3R becomes a drawback rather than a strength. Since the predicted pointmaps are expressed in a local coordinate system defined by the first image of each pair, all predictions live in different coordinate systems. This design hence requires a global post-processing step to align all predictions into one global coordinate frame, which quickly becomes intractable

Figure 13.16 Qualitative example of MUSt3R reconstructions of Aachen Day-Night [1279] nexus4 sequence 5 (offline, *top*) and TUM-RGBD [1050] room sequence (online, *bottom*). Images from [141] (©2025 IEEE).

for large collections when done naively. This raises the problem of the quadratic complexity of a pairwise approach and a robust and quick optimization in the real-time scenario. While these concerns are partially addressed in MASt3R-SfM [297], MUSt3R [141] takes a different stance and designs a new architecture that is scalable to large image collections of arbitrary scale, and that can infer the corresponding pointmaps in the same coordinate system at high frame-rates. To achieve this, MUSt3R extends the DUSt3R architecture through several crucial modifications – *i.e.,* making it symmetric and adding a working memory mechanism – with limited added complexity. The model, beyond handling *offline* reconstruction of unordered image collections in a Structure-from-Motion (SfM) scenario, can also tackle the task of dense Visual Odometry (VO) and SLAM, which aims to predict *online* the camera pose and 3D structure of a video stream recorded by a moving camera. MUSt3R can seamlessly leverage the memory mechanism to cover both scenarios such that no architecture change is required and the same network can operate either task in an agnostic manner (see Fig. 13.16).

13.7.3 MASt3R-SLAM

We compete this chapter by presenting an alternative approach to make MASt3R scalable. MASt3R-SLAM [795] is a real-time SLAM framework to leverage MASt3R's two-view 3D reconstruction priors as a unifying foundation for tracking, mapping, and relocalisation, as shown in Fig. 13.17. While MASt3R-SfM as applied these priors to SfM in an offline setting with unordered image collections [297], SLAM receives data incrementally and must maintain real-time operation that requires solutions on low-latency matching, careful map maintenance, and efficient methods for large-scale optimisation. On one side, MASt3R-SLAM adopts filtering and optimisation techniques in SLAM systems [1084, 1085], and performs local filtering of pointmaps in the frontend to enable large-scale global optimisation in the backend. On the other hand, like in MASt3R, it makes no assumption on each image's camera model beyond having a unique camera center that all rays pass

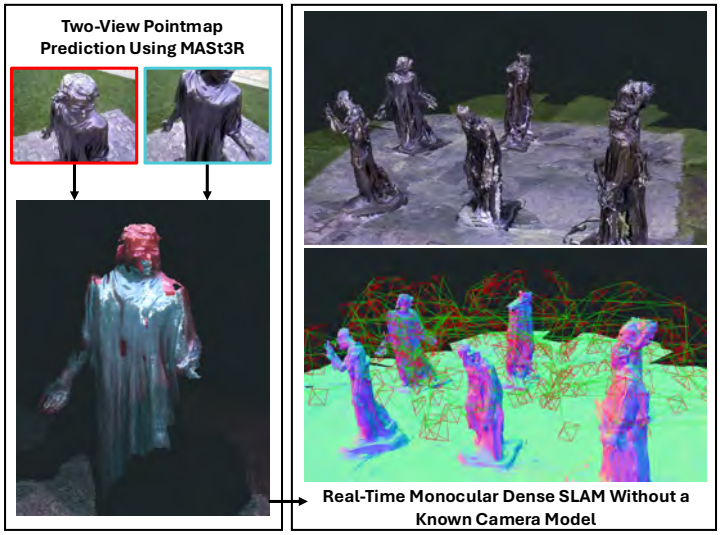


Figure 13.17 Reconstruction from the dense monocular SLAM system on the Burghers sequence [1291]. Using the two-view predictions from MASt3R shown on the left, the system achieves globally reconstructions in real-time without a known camera model. TODO: Reuse permission

through. This results in a real-time dense monocular SLAM system capable of reconstructing scenes with generic, time-varying camera models. Given calibration, MASt3R-SLAM demonstrated state-of-the-art performance in trajectory accuracy and dense geometry estimation.