Master Thesis

Two Step Correspondence Matching in Vegetation using Deep Learning

Spring Term 2019

Supervised by:
Tejaswi Digumarti
Dr. Cesar Cadena Lerma

Author:
Vasily Vitchevsky
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

TWO STEP CORRESPONDENCE MATCHING IN VEGETATION USING DEEP LEARNING

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
VITCHEUSKY

First name(s):
VASILY

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich 2/5/2019

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.
# Contents

Abstract v  
Symbols vii  
1 Introduction 1  
2 Theory 2  
  2.1 Related Work 2  
  2.2 Truncated Distance Fields 3  
  2.3 Triplet Loss 4  
3 Data Set 5  
  3.1 Data Acquisition from Synthetic Trees 5  
  3.2 Training Data 5  
    3.2.1 Choosing two nearby poses 7  
    3.2.2 Choosing the three points 7  
4 Methods and Preliminary Results 9  
  4.1 Network Architecture 9  
  4.2 Pose estimation 9  
  4.3 Training and Triplet Loss 9  
  4.4 Evaluation 10  
    4.4.1 Losses 10  
    4.4.2 Metrics 11  
  4.5 Baseline 13  
    4.5.1 Original network based on 3D Match 13  
    4.5.2 Ratio Test 16  
  4.6 Squared Loss 16  
    4.6.1 Squared Loss Results 17  
  4.7 Include Semantics: Same Class Test 20  
    4.7.1 Semantic Check Results 20  
  4.8 SegNet + 3DMatch 20  
    4.8.1 Label Loss 22  
    4.8.2 TDFs with color 22  
    4.8.3 Combining feature maps with SegNet 22  
  4.9 The Correct Neighbor 23  
    4.9.1 Generating Difficult Triplets 25  
    4.9.2 Squared loss with difficult test set 25  
  4.10 Mixed Training 25  
    4.10.1 Mixed Training Results 25  
  4.11 Hard Training 28  
    4.11.1 Hard Training Results 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12</td>
<td>Two Step Method</td>
<td>30</td>
</tr>
<tr>
<td>4.12.1</td>
<td>Two Step Method Results</td>
<td>31</td>
</tr>
<tr>
<td>4.13</td>
<td>Hybrid / Fork Structure</td>
<td>31</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Pre-trained branch</td>
<td>33</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Forked Network with Mixed Training</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Results and Discussion</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Inlier Ratio</td>
<td>41</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Loss function</td>
<td>41</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Two Step Method</td>
<td>41</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Forked Network with Mixed Training</td>
<td>41</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Semantic Class Test</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Pose Estimation</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Accumulated Trajectory Error</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Precision / Recall</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion and Future Work</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>52</td>
</tr>
</tbody>
</table>
Abstract

Correspondence matching in vegetation is interesting for 3D reconstructing trees, agriculture and pose estimation. It is a challenging problem due to occlusions and repeating pattern, and no previous work has directly tackled this problem. Previous approaches such as SIFT\cite{1}, are not tuned for vegetation, and thus fail. Recently, 3DMatch was proposed as a new method for 3D correspondence matching by \cite{2}. Instead of using hand crafted features, it constructs a local 3D representation of the point cloud using TDF (Truncated Distance Field) and trains a deep neural network to convert this representation into a descriptor which can be further used for correspondence matching. In this project, we apply 3DMatch to a synthetic tree RGB-D data set and introduce the Two Step approach where we: first, train a 3DMatch instance on an arbitrary set of matches and non matches from different viewpoints, and use this instance to find a \textit{set of candidate matches} in the other view pose for a given query point in the current view. As a second step, we train another 3DMatch instance on the difficult examples from the \textit{set of candidate matches}, and use it to select the best match. We also evaluate a forked version of 3DMatch, where these two networks train in collaboration. This considerably increases the matching precision. On our synthetic data set (which simulates Kinect v2) we reach precision of 0.43 overall and 0.36 and 0.57 for leaf and non-leaf points, respectively.
Symbols

Notation

\( A \) anchor / query point
\( P \) positive input (match, corresponding point)
\( N \) negative input (non-match, non-corresponding point)
\( f(A) \) descriptor of input \( A \)

\( T \) translation vector \( \in \mathbb{R}^3 \)
\( R \) rotation matrix \( \in \mathbb{R}^{3 \times 3} \)

Acronyms and Abbreviations

TDF Truncated Distance Field
TSDF Truncated Signed Distance Field
SIFT Scale-Invariant Feature Transform
RGB-D Color and Depth
RANSAC RANdom SAmple Consensus
Chapter 1

Introduction

Vegetation is part of our world. If we are lucky, we have some trees in our neighborhood, and there’s always plenty of different kinds of trees, bushes and flowers in less populated areas. Navigation of robots has been a hot topic for some time now, and it is even more important these days with the increasing popularity of flying drones, specifically autonomous ones. Drones can fly in places where traditional ground-moving robots can not pass. New environments for robots include dangerous areas (e.g. fire, earthquake, underwater) and natural environments such as forests, jungles, caves and more.

Traditional navigation methods (e.g. SLAM) were built and evaluated for the purpose of navigation in urban environments, and as part of their pipeline they use correspondence matching algorithms (e.g. SIFT) for localization that work well in those environments. With the new available flying environments, specifically jungles and forests with a lot of vegetation, these algorithms have to be re-evaluated and adapted.

Another area where correspondence matching in vegetation is important is media. Movie and game companies want to import real life trees into their movies/games, this is done by reconstructing the tree geometry, which usually involves correspondence matching for point cloud alignment.

Finally, in agriculture, automated analysis of crops, plantation and vegetation, such as estimating the Leaf area index\(^1\) are important for effective growing and good health of the vegetation. Correspondence matching in those areas could help with the 3D reconstruction of the vegetation and a more accurate estimation of their health.

In this work, we focus on correspondence matching in vegetation environments. The main challenge, compared to urban environments, is that there’s a lot of occlusion and similar colors and patterns, as well as shape deformation due to wind, with which the previous approaches don’t work well.

There has been no previous work that dealt with navigation within jungles, but other natural environments were tackled, such as navigation underwater \(^2\), or in a tree plantation \(^3\) (though our approach is very different). Otherwise, previous work usually assumed existence of artificial objects (buildings, roads).

\(^{1}\text{Leaf area index} = \text{leaf area} / \text{ground area}. \quad ^{2}\text{"It is used as a reference tool for crop growth"} \quad ^{3}\text{https://en.wikipedia.org/wiki/Leaf_area_index}\quad ^{2019-05-02}
Chapter 2

Theory

Correspondence matching is a well researched topic in the field of Computer Vision with applications in pose estimation, image-based localization, etc. In many of these applications, the overall performance strongly depends on the quality of the initial feature matching stage [5]. Consequently, determining which local feature descriptors offer the most discriminative representation and the best matching performance is of significant interest. Classical methods establish matches by building descriptors using the local neighborhood of salient keypoints and comparing them. There are two classes of approaches to building the descriptors: using hand crafted methods or using learnt descriptors. We look at related work developed in both areas in the following section.

2.1 Related Work

Hand-crafted local feature descriptors have been around for quite some time and have had considerable success in identification of correspondences. Many of these feature descriptors like SIFT [1], Fast Point Feature Histograms (FPFH) [6], Spin Image [7] and Signatures of Histograms [8] work well for 3D models with complete surfaces but are often unstable or inconsistent in real-world partial surfaces from 3D scanning data and difficult to adapt to new datasets. They struggle when dealing with noisy and highly repeating structures in data like vegetation. Moreover, hand crafted features inherently carry some human bias and while they work well for specific classes of object or pattern recognition, they often fail to generalize.

The other class of building descriptors relies on deep learning based methods which have attracted much attention recently because they are built to automatically learn features from raw input data, so as to avoid manually engineered features. For instance, some works [9] use 2D image patch descriptor trained on multi-view stereo datasets to do correspondence matching on RGB images. The drawback of multi-view stereo datasets is that they have a fixed image size and small view range. Thus we need 3D descriptors to learn the full surface representation. For example, 3D ShapeNets [10] introduced deep learning for modeling 3D shapes while other works [11] use 3D descriptors to recognize object category. However, instead of using the local features, they learn the object’s representative features on a global scale. In contrast, in the 3D Match paper by Zeng et al. [2], the focus is on learning geometric features for real-world RGB-D scanning data at a local level, to provide more robustness when dealing with partial data suffering from various occlusion patterns and viewpoint differences. In this work too, we intend to use a local 3D descriptor that directly learns from data to provide more robust and accurate geometric fea-
2.2 Truncated Distance Fields

As mentioned in the previous section, we want to use a local geometric 3D descriptor to do corresponding matching. We follow the methodology of Zeng et al. [2] to construct this descriptor by extracting a 3D volumetric representation of the local region surrounding every interest point. Each 3D region is converted from its original point cloud representation into a volumetric 30 × 30 × 30 voxel grid of Truncated Distance Function (TDF) values.

In our experiments, these local 3D patches spatially span a cube of size (30cm)³, where each voxel (mini cube) size is 1cm³. The voxel grid is aligned with respect to the camera view. The TDF value of each voxel indicates the distance between the center of that voxel to the nearest point in the 3D point cloud. These TDF values are truncated to [0cm, 5cm], normalized and then flipped to be between 1 (point directly at center of cube) and 0 (point distance ≥ 5cm from center of cube).

An alternative distance measure is Truncated Signed Distance Function (TSDF) which also makes a distinction between points inside and outside of a surface with respect to the voxel. So we would record negative values when the voxel is encapsulated by the object. However, in our case, this is not particularly useful since a significant part of the tree like leaves and branches are flat. Using TDFs, while we lose the distinction between free space and occluded space, we still retain information about the surfaces (in form of high second derivative of voxel values near the surface) which is crucial for the robustness of the descriptor on partial data. Hence, we use TDF as our distance measure.
2.3 Triplet Loss

Recent work on deep learning for learning feature embeddings in context of correspondence matching examines the use of triplets of samples instead of solely focusing on pairs of matches vs non-matches [13]. Learning with triplets involves training from samples of the form \( \{A, P, N\} \), where \( A \) is the anchor sample, \( P \) is a positive example which is a different sample of the same class as \( A \), and \( N \) is a negative example which is a sample belonging to a different class. In our case of correspondence in vegetation, \( A \) and \( P \) are different viewpoints of the same physical point on the tree, and \( N \) comes from a different class of the tree (for example, \( N \) belongs to leaf class if \( P \) was a trunk point and vice versa). Hence, training the network with these triplets brings \( A \) and \( P \) close in the embedding space, and pushes \( a \) and \( n \) far apart.

Borrowing notation from Balntas et al. [14] we can write the euclidean distance in embedding space of positive and negative example from the anchor as follows:

\[
\delta_+ = \|f(a) - f(p)\|_2 \\
\delta_- = \|f(a) - f(n)\|_2
\]

The loss functions proposed in the literature [15], [13] for learning convolutional embeddings with triplets can be grouped into two classes:

(i) **Margin-based losses**: In general, we want the anchor to be closer to the positive example than the negative example by at least some margin in the embedding space. These losses measure the violation of the ranking order of the embedded features inside in the triplet. Thus, the contrastive loss is defined as follows:

\[
\lambda(\delta_+, \delta_-) = \max(0, \delta_+ + \mu - \delta_-)
\]

Note that if this marginal distance difference is respected, the loss is 0, and thus the weights of the CNN are not updated. The loss remains 0 until the margin is violated, and after that, there is a linear increase. Thus the loss is not upper bounded, only lower bounded to 0.

(ii) **Ratio-based losses**: These losses investigated in [13] optimise the distance ratio within triplets. This loss is designed such that the ratio of \( \delta_- \) to \( \delta_+ \) is pushed to \( \infty \). A possible ratio based loss function investigated in [13] is:

\[
\tilde{\lambda}(\delta_+, \delta_-) = \left( \frac{e^{\delta_+}}{e^{\delta_+} + e^{\delta_-}} \right)^2 + \left( 1 - \frac{e^{\delta_-}}{e^{\delta_+} + e^{\delta_-}} \right)^2
\]

There is no margin associated with this loss, and by definition the loss is bound between \([0,1]\) for all possible values of \( \delta_- \) and \( \delta_+ \). Unlike the margin-based loss, where \( \lambda = 0 \) is possible, every training sample in this case is associated with some non-negative loss value.

In this project, we used the margin based loss for optimization following Zeng et al. [2]. The ratio based loss can be explored as future work.
Chapter 3

Data Set

3.1 Data Acquisition from Synthetic Trees

We worked with parts of the data set from [16] and [17]. The following explains how the data was generated. SpeedTree\(^1\), a suite for vegetation programming and modeling software that generates the 3D models of various varieties of synthetic trees, is used to generate Cherry tree type meshes. The synthetic Cherry trees are positioned in a scene and rendered with Unreal Engine\(^2\). A drone trajectory is then simulated using the AirSim plugin\(^3\), where the drone flies around a tree in a spiral motion, from bottom to top and always looks at the tree from a distance of about 5 meters. In this work, we focus on scenes with only a single tree. Images from the low part of the tree mainly contain the trunk, images from the top part mainly contain leaves and branches, while the middle part has a mix of everything. Distribution of leaf versus non-leaf points is depicted in Figure 3.2. A sample drone trajectory can be seen in Figure 3.1.

Along the trajectory, at approximately every 2 meters, we render a RGB, depth and semantic label (leaf, trunk, branch, background) images of size 424×512 pixels, which simulates the output of a Kinect V2\(^4\) sensor. In this work, however, we did not simulate Kinect’s noise over the depth image. In total, our data set contains 100 Cherry trees, for each tree we have about 70 poses, depending on the tree height, and each pose consist of the mentioned data.

3.2 Training Data

We split the 100 trees in the dataset to train (1-80) and test (81-100) sets. The training set consists of triplets of TDFs / volumetric patches of three points:

- A query point from the viewpoint of the first pose
- A matching point from the viewpoint of a second nearby pose
- A non-matching point from the view of the second pose

Patches around all these points are extracted using the relevant depth images and passed to the network.

\(^1\)SpeedTree - 3D Vegetation Modeling and Middleware - https://speedtree.com
\(^2\)Unreal Engine 4 is a suite of tools to design and build games, simulations, and visualizations - https://unrealengine.com
\(^3\)AirSim is a simulator for drones and more, built on Unreal Engine - https://github.com/Microsoft/AirSim
\(^4\)Kinect for Xbox One https://en.wikipedia.org/wiki/Kinect
Figure 3.1: A tree from our data set. The red spiral line is the drone trajectory. Cones (red, blue, yellow) represent the camera orientation along the trajectory (camera looks in the direction of the cone, towards the tree). Yellow arrow is a random first pose, and the blue arrows are its’ nearby poses, from which we select the second pose. From the viewpoint of the first pose we extract a patch around some query point. From the viewpoint of the second pose we extract a patch around the same query point (match), and in addition a patch around a different point (non-match). The selection of the query, match and non-match points is explained in 3.2.2.
3.2 Training Data

Figure 3.2: Distribution of number of leaf vs non-leaf points in the data set. For every tree pose/image we count the number of leaf/non-leaf points and put a green point on the plot. In red and blue are sample distributions of two trees, each of which has about 70 images, represented by the 70 red/blue points on the plot. We can see that both trees have images with different number of leaf/non-leaf points mostly covering all the possible cases (more non-leaf points, more leaf points, overall many points, etc.). A few outliers are not shown in the plot, where the trunk is very close to the camera and there are more than 100,000 trunk points.

3.2.1 Choosing two nearby poses

The triplets are taken from two adjacent poses that are up to 3 meters apart (maximum drone displacement over the two poses). In the case of our spiral drone trajectory, consecutive poses along the trajectory are acceptable since they are about 2 meters apart or poses that are adjacent vertically (separated by one revolution around the tree) which are about 2.5 meters apart. Both these distances can be calculated from frame rate of drone camera. Refer to Figure 3.1 for an example.

3.2.2 Choosing the three points

We first randomly sample two adjacent poses from some random tree as described above. Then, we choose three points in total from these two poses:

1. The first point is a random point on the tree from the first pose.

2. The second point is a matching point, the original point as viewed from the second pose. If the point is occluded, we try again from step 1. Occlusion can be checked using the depth map at the relevant pixel.

3. Finally, the third point is a non-matching point which we also take from the second pose with the criteria that it has to be at least 1 meter away from the matching point.

The matching point in the second view is computed using the ground-truth drone poses, camera transformation and synthetic images. Finally, we extract the TDFs at the three points to create one training triplet. We generate 10,000 such triplets where the first point is a leaf, and another 10,000 where the point is a trunk or a branch.
Figure 3.3: Choosing the training points. The first and the second poses are marked with a yellow and blue arrows respectively, they are the poses from Figure 3.1 and the corresponding views on the right. The first 3D patch is taken at the endpoint of the green line from the viewpoint of the first pose. The second patch is extracted also at the endpoint of the green line (a match) but from the viewpoint of the second pose. The third patch is extracted at the endpoint of the red line (non-match) also from the viewpoint of the second pose.
Chapter 4

Methods and Preliminary Results

4.1 Network Architecture

3DMatch\textsuperscript{[2]} network is used to compute a descriptor for a given point. The 3D patch around the point is converted to a TDF, which is then passed to the network. The network is a standard 3D convolution network. Given a 30x30x30 TDF grid, it is passed through eight 3D convolution layers (each with a ReLU activation) and a pooling layer to compute a 512-dimensional feature vector, which serves as the descriptor of the given point. Network parameters are shown in Figure \textsuperscript{4.1}.

4.2 Pose estimation

To estimate the relative pose from which two images were taken, we find matches between points of one image and the other. We pass these matches to RANSAC to estimate a 3D Euclidean transform. The matches are found using different methods, the basic one is to compute the descriptor for all of the tree points in both images, then for each point in the first image find the nearest neighbor in the second image (in descriptor space), and pass these matches to RANSAC. The quality of the result we get from RANSAC, as well as the number of iterations required, depends on the quality of the matches, specifically the inlier ratio, the fraction of good matches passed to RANSAC. This is one of the metrics we evaluate.

Possible improvements for pose estimation could be, for instance, an alternative algorithm to RANSAC, a method to filter away the bad matches, a different way to find matches, or better descriptors. We always use RANSAC, however, the methods described later discuss filtering the matches, finding better matches and also more complex training schemes. For the purpose of evaluation, we always use a constant number of iterations (2,000). In real life, however, one should toggle the number of iterations depending on the inlier ratio. In our work, we look at the inlier ratio as a metric to how good are the descriptors that the network computes.

4.3 Training and Triplet Loss

During training, the objective is to optimize the generated descriptors such that they are similar for 3D patches corresponding to the same point, and different otherwise. To train the network we use a triplet loss (recap from section \textsuperscript{2.3}). We
collect patches around the query point, a matching point from a second view, and a non matching point from the second view, and pass these patches through our network and optimize the following loss:

\[
\text{Loss} = d(f(A), f(P)) + \max(\alpha - d(f(A), f(N)), 0)
\]

Where \(d(a, b)\) is the Euclidean distance:

\[
d(a, b) = ||a - b||^2
\]

The full pipeline is depicted in Figure 4.1. The loss pushes the distance of matching point’s descriptors to zero, and the distance of non matching points to above some boundary \(\alpha\) (\(\alpha = 1\) in our work). Similarly to a Siamese network, the weights are shared across the three passes and are updated together. We evaluated different learning rates, and found 0.00005 to work the best across our experiments and different network structures.

### 4.4 Evaluation

To compare between the different methods, we compare losses during training and metrics when performing pose estimation.

#### 4.4.1 Losses

Losses are computed over the triplets test set taken from the trees test set (trees 81-100). Losses are shown from TensorBoard from all our experiments. The timestamp is measured in iterations, where each iteration is a batch of size 16. Since our input is 20,000 triplets, we go over the training set in 1250 iterations for one epoch. Following the notation from the Symbols chapter, \(A\), \(P\) and \(N\) are anchor point, positive (matching) point and negative (non-matching) point respectively, and \(f(\cdot)\) is the descriptor for a given input, we define the total loss as the sum of matching and non-matching loss as shown below:
4.4. Evaluation

The loss value over a validation set of matching pairs.

\[ L_{\text{matches}}(f(A), f(P)) \]  

Non-Matches Loss

The loss value over a validation set of non matching pairs.

\[ L_{\text{non-matches}}(f(A), f(N)) \]

Total Loss

The sum of the matches and non-matches losses.

\[ L = L_{\text{matches}}(f(A), f(P)) + L_{\text{non-matches}}(f(A), f(N)) \]

4.4.2 Metrics

Precision and camera pose errors are evaluated over a fixed set of 100 random pose pairs from the trees test set (trees 81-100). Ideally, a full evaluation should be performed to compare every combination of training method with all pose estimation methods, over the whole test data set, but due to computational time constraints, this was not performed. However, we found that evaluation of the training methods over a representative subset of the pose pairs is sufficient. A representative subset would include pairs of poses for the different scenarios: majority leaf points, majority non-leaf points, and cases where there are more points overall. The 100 random pose pairs cover these possible cases. In Figure 4.2, the chosen tree poses are marked with respect to the number of leaves/non-leaves in the first image of the pair. In Figure 4.3, some of the chosen tree poses are displayed.

Precision

Given a pair of poses, we find matches using the different methods. Precision is the fraction of correct matches over total matches. Throughout the thesis we sometimes
Figure 4.3: Sample pairs of poses in the test subset. Images from the first and third columns are evaluated with the image to their right for relative pose estimation.
call this the inlier ratio, the ratio of inlier matches out of all matches. We consider a match to be correct (inlier) if the matching point is within 10 centimeters from the correct location. Of course, higher precision is better, since we later pass these matches to RANSAC for pose estimation. The precision of matching process is defined as:

\[ P(\text{matches}) = \frac{|\text{matches} : \text{error(match)} < 0.1m|}{|\text{matches}|} \]  

(4.6)

Camera pose translation and rotation errors

Given a pair of camera poses, we compute the ground truth transformation \((T \text{ denoting translation}, R \text{ denoting rotation})\) between the two poses and use the matches (found by some method) to estimate the transformation \((T^*, R^*)\) between the two poses. The translation error, \(T_{\text{err}}\), and the translation error, \(R_{\text{err}}\) are then:

\[ T_{\text{err}} = ||T^* - T||_2 \]  

(4.7)

\[ R_{\text{err}} = ||R^* R^{-1} - I_{3 \times 3}||_2 \]  

(4.8)

In this work we are considering camera poses where the camera is looking at a tree from roughly five meters away. We evaluate transformation between two camera poses that are about two meters far apart, matching points from one view to points in the other one. Since the matches are all on the tree, the tree’s translation error between the two poses is usually good. However, the camera’s translation error is directly affected by the rotation error multiplied by the distance. For instance, if the relative translation of the tree’s center is estimated accurately, but we have 1 degree rotation error, the camera’s translation error (at five meters away from the tree) will be about nine centimeters, a non negligible error.

4.5 Baseline

4.5.1 Original network based on 3D Match

We use the original 3D match network as explained in 4.1 and estimate the pose in the straightforward way described in 4.2. As we said in 4.4.1 and 4.4.2, we look at the evolution of losses (match, non-match and total loss) of the triplets test set computed over roughly 100 iterations of training. The pose estimation results are computed after training, over the fixed set of 100 random pose pairs from the trees test set (trees 81-100).

Original Network Results

The learning curve is shown in Figure 4.4. The network reaches the lowest test loss for matches at step 12,000 and for non-matches around step 30,000, which we reach in an hour and two hours respectively. Upon further training, the network starts to overfit, which can be seen from the observation that the training losses keep going down, while the test loss for matches slightly goes up, and the test loss for non-matches stays at its minimum value.

Inlier ratio result is shown in Figure 4.5. Pose estimation result is shown in Figure 4.6.
Figure 4.4: Training with L2-norm loss. Matches loss is the loss of the matching pairs. Non matches loss is the loss of the non matching pairs. Total loss is the sum of matches and non matches losses.
Figure 4.5: Inlier ratio results for the original 3D Match method. This is a basic boxplot, where the box represents the two middle quartiles, the line in the box is the median, and the whiskers represent the range of the values. The median inlier ratio of the leaf points, non-leaf points and overall are 0.04, 0.15 and 0.07, respectively.

Figure 4.6: Pose estimation errors for the original 3D Match method. Median translation and rotation are 0.027 [m] and 0.344 [deg], respectively.
Chapter 4. Methods and Preliminary Results

4.5.2 Ratio Test

A basic improvement for the performance of pose estimation is to perform ratio test instead of taking all the matches. Using the following notation:

- $\gamma \in (0, 1)$ a constant ratio
- $f(q)$ descriptor of a 3D patch of point $q$
- $q$ query point
- $p_1$ best match of query point in the second pose
- $p_2$ second best match of query point in the second pose
- $d(a,b)$ distance measure between vectors $a$ and $b$

We only take matches such that the best match is closer the given query point than the second best match by at least a factor of $\gamma$ in descriptor space:

$$d(f(q), f(p_1)) \leq \gamma \cdot d(f(q), f(p_2))$$

We use $\gamma = 0.8$ for as default threshold for Ratio Test throughout the thesis. No additional training is performed while applying the Ratio Test.

**Ratio Test Results**

We apply ratio test on the matches and estimate the camera pose transformation. Compared to the baseline method, inlier ratios went up (Figure 4.7), and similarly translation and rotation errors decreased (Figure 4.8).

We henceforth use the 3D Match network with the ratio test applied ($\gamma = 0.8$) as our baseline method.

4.6 Squared Loss

Minimising the squared $L_2$ norm loss penalizes difficult inputs more by having a stronger gradient for inputs with a higher loss as compared to using just the $L_2$ norm loss. Squared loss function is also differentiable at zero and the derivative is more stable when approaching zero. Thus, instead of optimizing the Euclidean
distance, we optimize the squared Euclidean distance using \( \alpha \) squared (defined in triplet loss):

\[
\text{Squared Loss} = d(f(A), f(P)) + \max(\alpha^2 - d(f(A), f(N)), 0) \tag{4.9}
\]

Where

\[
d(a, b) = ||a - b||_2^2 \tag{4.10}
\]

### 4.6.1 Squared Loss Results

The learning curve is shown in Figure 4.9. The network reaches the lowest test loss for matches at step 20,000 and for non-matches around step 10,000. Upon further training, the network starts to overfit, the training losses keep going down, while the test losses for matches and non-matches slightly go up.

The loss of the non-matches converges to a lower value than when optimizing the normal Euclidean distance. On the other hand, the loss of the matches converges to a higher value. One reason for this is that the non-matches loss is on average greater than the match loss and thus it has a stronger gradient, so the network is encouraged to learn the non-matches better.

We observe that the loss metric (\( L_2 \) loss versus squared \( L_2 \) loss) attains a more optimal value depending on which the objective function we choose (\( L_2 \) loss vs squared \( L_2 \) loss). Thus, the \( L_2 \)-norm loss is lower using the baseline, while the total squared \( L_2 \)-norm is lower when optimizing with respect to the squared loss function. It's hard to say which learning result would perform better (with the pose estimation task) solely from the total loss. However, one may ask what is more important - lower loss of matches, or non matches?

We are performing dense matching: for each point, we have 10,000 – 50,000 non-matches, and only a single match. Since there are so many more non-matches, it is more important to have good accuracy at classifying them. Indeed, with the lower non-matches loss and higher match loss, we get a considerable 2-4 times improvement in precision compared to the baseline. Specifically, inlier ratio for leaves goes from 0.05 to 0.2 and for non-leaves from 0.19 to 0.46 (all with ratio test) (Figure 4.10).
Figure 4.9: Comparison of the learning process when optimizing $\ell_2$-norm loss versus squared $L_2$-norm loss. For both runs we also compute the other loss for comparison. The left column is the value of the normal $L_2$-norm during training, while the right column is the squared $L_2$-norm during training. In blue “$\ell$ train/test” is the run optimized for $L_2$-norm, while in red “$\ell^2$ train/test” is the run optimized for squared $L_2$-norm. Same as before, the rows are for matches loss, non-matches loss and total loss respectively.
4.6. Squared Loss

Figure 4.10: Inlier ratio results for the squared $L_2$-loss training with and without ratio test. The median inlier ratio of all the points, of leaf points and of non-leaf points is 0.19, 0.14 and 0.30, respectively without ratio test, and 0.27, 0.20 and 0.46, respectively with ratio test.

Figure 4.11: Pose estimation errors for the squared $L_2$-loss training with and without ratio test. Median rotation and translation errors are 0.053 [deg] and 0.004 [m], respectively without ratio test, and 0.052 [deg] and 0.004 [m], respectively with ratio test.
Another interesting result is that ratio test helps with the inlier ratio, but does not affect the pose estimation results (Figure 4.11). The underlying reason could be that RANSAC usually find the correct solution, and having a higher inlier ratio does not help anymore with 2000 RANSAC iterations. However with a higher inlier ratio, we can perform less iterations to reach the solution, which mean faster run time.

Henceforth, we always use the squared Euclidean distance as the loss function for optimisation.

4.7 Include Semantics: Same Class Test

To further improve the precision, in addition to ratio test, we can add a check that a match is between two points of the same semantic class. To evaluate the potential benefit, we use the ground truth label from our data set. We followed the methodology of Digumarti et al. [19] and trained SegNet HA/HHA on our data set to predict the label (leaf vs non-leaf) given a point. We use the predicted class label to filter out bad matches (matches belonging to a different class than that of the query point).

4.7.1 Semantic Check Results

The inlier ratio has increased in comparison to the the ratio test method for all the class check methods (Figure 4.12). For non leaves, we have a 12-14% improvement for ANHAW6/ANHHAW6 and 25% improvement for groundtruth class check. For leaves, about 13% improvement for ANHAW6/ANHHAW6 and only 8% improvement for groundtruth class check. Overall, 8-9% improvement for ANHAW6/ANHHAW6 and 12% improvement for groundtruth class.

Pose estimation error hasn’t changed much, for the same reason as before - RANSAC finds the correct solution.

4.8 SegNet + 3DMatch

Another way to include semantics is to train SegNet and 3DMatch networks together, such that they assist each other, yielding a better descriptor. One challenge with this approach is that SegNet works on the whole image, while 3DMatch only works on a patch around a point.

Before trying to also incorporate semantics, we spent some time analyzing why is the inlier ratio low. We found that although semantics can potentially help, as we saw from the results in 4.7, the main issue is actually the specification abilities of the descriptor, the ability to differentiate between similar points. See discussion in 4.9. Nevertheless, for completeness, we mention a few ideas for combining 3DMatch with semantics or combining the two networks.

1HA - SegNet trained on horizontal distance and surface normal angle. See [19]
2HHA - SegNet trained on horizontal distance, height and surface normal angle
3ANHAW6 - The HA network predicting 6 classes
4ANHHAW6 - The HHA network predicting 6 classes
Figure 4.12: Inlier ratio results for the squared $L_2$-loss training with different class equality checks.

Figure 4.13: Pose estimation errors for the squared $L_2$-loss training with different class equality checks.
4.8.1 Label Loss

A basic extension is to add an output that predicts the label. We appended a fully connected layer with two outputs to the resulting descriptor, two outputs which predict the probability that the input TDF is a leaf or a non-leaf. See Figure 4.14. We added a cross entropy loss for the labels, the label loss. Training the network with only label loss, the loss was not going down and the network could not learn to predict the labels. To train the network with both label loss and triplet loss, some fusion of the losses needs to be done, since they are of different sizes. However, this was not tried.

4.8.2 TDFs with color

In addition to passing the TDF values (a tensor of shape $30 \times 30 \times 30$), we may add more information for each of the points in the TDF, such as the color of the nearest neighbor, or a descriptor representing the point. We added the color information, but the network performed similarly to without it.

4.8.3 Combining feature maps with SegNet

See Figure 4.15 for schematics of SegNet. SegNet and 3DMatch can be fused in different ways. Since 3DMatch operates on local patches, while SegNet is working on the whole image, we could take a crop of some layer in SegNet of a relevant region around the point that we passed to 3DMatch. As for the fusing location, for instance, we can merge feature maps from SegNet with 3DMatch at some layer, then append the remaining layers of each of them to this merged point. For 3DMatch,
we would have to append SegNet’s features to the last dimension (if 3DMatch’s
tensor is of shape $w \times w \times w \times C$ then after appending we would have a shape of
$w \times w \times w \times (C + D)$), while for SegNet we simply flatten 3DMatch’s features and
append. The loss of each of them would then propagate through the layers before
the merging point of both networks.

4.9 The Correct Neighbor

In this section we analyze the nearest neighbors of the query points. Let us have a
look at a similarity heat map of a given leaf point in Pose 1 to all points in Pose 2.

We can see in Figure 4.16 that the leaves in the second image are indeed similar to
the query leaf in the first image, but the network fails to identify which of the po-
tential matches is the correct one. The network generally finds similar 3D patches,
but has a hard time finding the specific patch that is the correct match.

When looking at the distance between a query point to its ten closest neighbors in
descriptor space, in the top figure we see the frequency of inlier goes down as we
move along the x-axis (farther along in descriptor space). However in the bottom
plot, we see that the descriptor distance doesn’t show sharp variations even when
the distance in world coordinates varies quite a bit. Thus it is reasonable to use
the descriptor distance to find similar points as potential matches, but cannot use
the same descriptor to distinguish between the quality of similarity to pick out the
correct match. We observed this low level and high level trend in descriptor space
for all query points.

As a result, most of these potential matches (points close in descriptor space) are
incorrect - they are non-matches(since they are far away in world coordinates or
groundtruth distance). Why do we see this happening, if during training the non-
matches loss is almost zero? The reason is that during training the input non-
matches given are arbitrary pairs of non-matches, which the network easily learns.
Whereas, distinguishing between the non-matches present in the set of top 10 po-
tential matches is the most difficult example of non-matches, which the network
probably never saw.
Figure 4.16: Heat map depicting the similarity, in descriptor space, of a single point in pose 1 to all the points in pose 2. Pose 1 and 2 images are on the first line, where the red point in pose 1 represents the query point. Then follows the heat map, where the brighter a point is, the more similar it is to the query point in descriptor space. Circled in red is the correct matching point, circled in yellow are the 10 nearest neighbors according to our descriptor. Finally, we see two plots of the descriptor space distance and the corresponding distance error for the most similar points.
In the next couple of sections, we will use the difficult pairs to train the network to distinguish also the difficult matches and non-matches.

4.9.1 Generating Difficult Triplets

In order to improve the network’s accuracy of choosing the correct point from the set of potential matches (generated using nearest neighbors in descriptor space), we want to train the network using difficult examples of non-matches. Therefore, we iteratively sample difficult triplets from adjacent pairs of poses as follows:

Given a pair of poses, we make a set of eligible query points for hard training from all the points in the first image that satisfy the following criteria:

1. have a matching point (not occluded from the other view),
2. do not have the correct match in the top 10 candidate matches (computed with the first network) and
3. have a bad match in the top 10 candidate matches (i.e. a point far enough from the correct point).

We randomly sample points from this set of eligible query points for hard training and with each query point we make a triplet:

- the query point itself,
- one difficult match (correct match not present in the top 10 candidate matches according to the first network)
- one difficult non-match (non-match present in the top 10 candidate matches according to the first network)

We use this sets of these difficult triplets to further train the network.

4.9.2 Squared loss with difficult test set

We evaluate during training over the test set and the new difficult test set separately. In Figure 4.17 we can see that the loss over the difficult test set is much higher.

4.10 Mixed Training

The idea is to make the network learn both of the triplet types. The network is trained on both the normal and the difficult training sets. Every batch is sampled randomly from the training sets.

4.10.1 Mixed Training Results

In Figure 4.18 we see the training losses for normal and mixed inputs. The mixed losses are similar to the normal training scheme. Additionally passing the difficult examples for training did not help.

In Figure 4.19 we see that the inlier ratio is slightly worse when we feed both hard and normal triplets as the train set, compared to when we only feed normal triplets. In summary, the training with a mixed training set had no improvement over the baseline.
Figure 4.17: Training with squared loss, testing with the original and difficult test sets.
Figure 4.18: Training with squared $L_2$-norm loss, over the normal dataset versus a mixed dataset consisting of both hard and normal triplets.
4.11 Hard Training

We train the network only on the difficult training set. We expect it to be good at the difficult pairs, but less good than the original network on the arbitrary pairs.

4.11.1 Hard Training Results

In Figure 4.20 we see the training losses. For matches, the normally trained network performs better than the hard trained network across all sets: normal train, normal test and hard test. This is surprising, because it never saw the hard test set. One possible reason is that in order to make the new network distinguish better between the difficult non-matches, it has to increase the distances between pairs of non-matching descriptors, which inadvertently affects the distance of the matching pairs as well.

For non-matches, the network trained on difficult triplets is performing way better with the difficult test set and slightly worse on the normal test set when compared to the network trained normally. The sum of the two losses (matches and non-matches) show that the network trained on the difficult triplets performs better on the difficult test set, but worse on the normal test set, which is expected.

In Figure 4.21 we see that the inlier ratio is considerably worse when we train on hard triplets, compared to when we feed normal triplets. The reason it is worse is that most of the potential matches are easy, rather than hard, so the network that was not trained on those normal examples performs worse. This is also in line with the training loss, where the normal test loss is worse.
Figure 4.20: Training with squared $L_2$-norm loss, over the difficult dataset consisting of only the hard triplets. In the middle plot, the loss for hard test set for normal training is way too high at around 6,000 and is not displayed.
To summarize, hard training samples train the network worse with regards to normal/random matches, and therefore the inlier ratio is worse and the final pose estimation is also worse as seen in Figure 4.22.

### 4.12 Two Step Method

As we saw in Section 4.9, we are doing a fine job with finding the top candidate matches for a query point, but the top match in this candidate pool is not necessarily the correct match. We therefore introduce The Two Step method to tackle the problem of selecting the correct match out of the top candidates.

We train two instances of the network (both with squared $L_2$-norm):

1. We train the first instance with the original training set, consisting of triplets of arbitrary matching and non-matching points as described in section 3.2.2.

2. We then train another instance of the network with the difficult triplets, which we find using the first network as described in section 4.9.1.

We use the first network to find the top 10 candidates for each point, apply ratio test, then the second network to pick the best match out of the top 10 candidate matches.
4.12.1 Two Step Method Results

We re-use the normal training (section 4.6) and hard training (section 4.11) networks as the first and second steps, respectively. The increase in inlier ratio is huge, larger than the other improvements (ratio test, class test).

In Figure 4.23 we see that the inlier ratio is consistently higher when using the Two Step method. When using either only ratio or both ratio and class tests together, we get an improvement of 40% for leaf points, 10% for non-leaf points and 30% overall respectively. When not filtering the matches (neither ratio test nor class test performed), we get an improvement of 79% for leaf points, 29% for non-leaf points and 55% overall respectively.

Interestingly, the improvement for leaves is larger than the improvement for non-leaves. This could be because the original network is better for non-leaves vs leaves (higher inlier ratio) so the potential improvement of only using different training data is lower. Therefore, the Two Step method does not benefit non-leaves as much. The pose estimation results in Figure 4.24 show an overall improvement when using the Two Step method versus the normal training. However, within the results of the different filtering methods, the results are inconclusive. In any case, all the results are well below the 1 centimeter error over the pose estimation test set.

4.13 Hybrid / Fork Structure

A hybrid between a single network (the original) and two networks (the Two Step method) is a fork structure, depicted in Figure 4.25. The network is forked at some point to create two equivalent branches: one branch is trained with the normal triplets, while the other is trained with the difficult ones. Similarly, one branch is used to compute the descriptors for the first step, and the second branch for the second step (where we have hard triplets). The weights along the path from the
Figure 4.23: Inlier ratio results for the Two Step method. class=gt specifies whether we filtered matches by checking their groundtruth class.

Figure 4.24: Pose estimation results for the Two Step method.
input to the fork point are shared.

One benefit of such an architecture, is that it may be more computationally efficient than using two separate networks:

- Single network - We pass points from two poses through the network.
- Two Step - We pass points from two poses through the first network, then all their top 10 neighbors through the second network.
- Forked - We pass points from two poses through the two branches of the network.

Depending on the fork position along the network, the forked version is not much heavier than the single network approach, since the last layers are much faster to compute so having another branch/tail is almost for free. Indeed, in Figure 4.26 we see that a branch before the 3rd to last layer only adds 4.5% runtime.

Another benefit of the forked architecture, compared to the Two Step method, is that the shared part of the network (before the fork) may be trained more robustly, such that the difficult and normal training benefit each other and create better initial features. We suspect that having a fork before one of the last layers could have as good results as the Two Step method, but without the computational overhead.

### 4.13.1 Pre-trained branch

We evaluate a few options of a forked structure where the shared path + normal branch are pre-trained using the normal training set, the shared part weights are
Figure 4.26: Runtime of a forward pass of a batch of 16 TDFs (30x30x30). The three possible architectures: **normal** is the original 3DMatch network, **two step** is passing the TDF through two networks, and **forked** is passing the TDF all the way through both branches, where the fork is before the 3rd to last layer, like in Figure 4.25. Layers **siamese_normal** and **siamese_hard** are the two branches. Time measurements shown are the median of 10 runs, measured with Tensorflow timeline profiler.
then frozen, and we only train the second branch with the difficult training set. We tested the following forks:

- fork 1 - the fork is just before the last layer.
- fork 2 - the fork is just before the second to last layer.
- fork 3 - the fork is just before the third to last layer.

Pre-trained branch Results

Figure 4.27 shows the losses during training. We compare the hard test loss and the difficult training loss over the three forked methods, and for reference we also compare with the single network trained from scratch on the difficult training set from section 4.11 (difficult/hard network).

Looking at the Total loss, we can see that the test loss is in the following order (low to high): difficult, fork 3, fork 2, fork 1. For fork 3, the loss is nearly as low as for difficult, which suggests that we can use this single forked network instead of two networks - the normal branch is equivalent to the first network in the Two Step method, and the difficult branch is almost as good as the second network in the Two Step method. fork 1 does not allow the pre-trained network to adapt the branched part to the difficult set, and we can see that the loss does not go down.

See figure 4.28 for inlier ratio results with the different second step networks. If we fork just before the last layer, the network performs even worse than only using the normal network. As we introduce the fork earlier, the results are better, and the overall results are only slightly worse when using fork 2 or fork 3 versus when the second step is the separate hard network. Again, we see that the forked network can be an alternative for the Two Step method with a better run time and a small decrease in precision.

4.13.2 Forked Network with Mixed Training

The network is forked similarly to 4.13 with a fork before the third to last layer. However, instead of pre-training it on the normal triplets, the network is trained from scratch on batches where half of the samples are normal and half are difficult. The loss is the sum of the losses of the normal and hard branches. The idea is that the branches learn features specific to the appropriate task (normal / hard), while the shared path will learn general spatial features.

Forked Network with Mixed Training Results

Figure 4.29 shows the losses during training. As the network now learns both the normal and difficult branches, we compare the learning process with the normal (section 4.6) network and hard network (section 4.11) that were trained on normal/hard training sets, rather than both. The left column shows the normal training, we can see that the new forked network is slower to converge, but eventually reaches a lower total test loss. The right column shows the hard training, we can see that the new forked network is slower to converge, and eventually reaches slightly higher total hard test loss.

In Figure 4.30 we can see how the Fork Mixed network performs. Interestingly, using only the normal branch from the Fork Mixed network outperforms the Two Step method (normal then hard). The normal branch became more robust because it
Figure 4.27: Training losses of the pre-trained fork networks, over the difficult data set consisting of only the hard triplets.
Figure 4.28: Inlier ratio results for the three forked architectures. **normal** is the original single network. **normal then X** is the Two Step method where the second network is changed. We compare the results where the second network is a branch from the original network, versus the original Two Step method where the second network is a separate network.
shared the initial part with the difficult branch. Previously, in section 4.10 we tried feeding both hard and normal training samples for a single network, but it did not manage to learn any more than the normal trained one. The Two Step method with the Fork Mixed networks performs even better, but this is largely because the normal network became better at finding the top 10 candidates. It would be interesting to try generating a new difficult training set, and see if we manage to train the single branch network even better.
Figure 4.29: Training with squared $L_2$-norm loss, over the difficult dataset consisting of only the hard triplets.
Figure 4.30: Inlier ratio results for the mixed trained forked architecture. 

- **normal** is the original network.
- **mixed normal** is using only the normal branch of the Fork Mixed trained network.
- **normal then hard** is the Two Step method.
- **mixed normal then mixed hard** is the Two Step method where the first and second networks are the first and second branches of the Fork Mixed network.
Chapter 5

Results and Discussion

5.1 Inlier Ratio

In Figure 5.1 and Table 5.1 we show a summary of the results.

5.1.1 Loss function

The main improvement, at 227% over the baseline, was changing the loss function to squared $L_2$-norm. We saw that the network then performs much better (lower loss) with the non-matches, which helped get rid of the false positives, and this is clearly seen in the inlier ratio results.

5.1.2 Two Step Method

We then tried various Two Step methods, where we first find the top few candidates matches using one network, and then select the best one using another network. These methods also considerably helped, increasing the overall inlier ratio by a further 27%, 31% and 55% for the normal then fork 3, normal then hard and mixed normal then mixed hard methods, respectively. The reason this helps so much, is that the first network learns how to find somewhat relevant candidate matches, but the task of selecting the specific candidate is different, and the first network was not trained for it, while the second network is. The Two Step methods namely normal then fork 3, normal then hard and mixed normal then mixed hard methods, respectively had a larger effect over leaf points at 40%, 43% and 76% improvement respectively (compared to normal), which is crucial since the leaf is the more frequent class in the data set. Non-leaf points also had some increase in performance, but it was smaller at 6%, 13% and 22% for the aforementioned Two Step methods.

5.1.3 Forked Network with Mixed Training

In Section 4.13.2 we fork the original network, duplicating the last few layers, training the network over normal and difficult examples in one go, and finally either using the normal path alone (mixed normal) or both (normal and difficult) ends as part of the Two Step method (mixed normal then mixed hard). The resulting mixed normal method shows better performance with a single descriptor even compared to the other Two Step methods, while mixed normal then mixed hard Two Step method improves over that.
Chapter 5. Results and Discussion

Figure 5.1: Inlier ratio results for the different correspondence matching methods with ratio test.

<table>
<thead>
<tr>
<th>Method</th>
<th>Leaves</th>
<th>Non-leaves</th>
<th>Overall</th>
<th>Improvement over baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>l2norm (baseline) (4.3)</td>
<td>0.045</td>
<td>0.194</td>
<td>0.084</td>
<td>1.000</td>
</tr>
<tr>
<td>normal (l2norm^2) (4.6)</td>
<td>0.204</td>
<td>0.465</td>
<td>0.275</td>
<td>3.277</td>
</tr>
<tr>
<td>normal then fork 3 (4.13)</td>
<td>0.285</td>
<td>0.495</td>
<td>0.349</td>
<td>4.166</td>
</tr>
<tr>
<td>normal then hard (4.12)</td>
<td>0.291</td>
<td>0.524</td>
<td>0.360</td>
<td>4.287</td>
</tr>
<tr>
<td>mixed normal (4.13.2)</td>
<td>0.305</td>
<td>0.547</td>
<td>0.378</td>
<td>4.502</td>
</tr>
<tr>
<td>mixed normal then mixed hard (4.13.2)</td>
<td>0.360</td>
<td>0.566</td>
<td>0.427</td>
<td>5.094</td>
</tr>
</tbody>
</table>

Table 5.1: Median precision / inlier ratio for the different correspondence matching methods with ratio test.

5.1.4 Semantic Class Test

In Section 4.7 we described the class test as a semantic check. In Figure 5.2 and Table 5.2 we show the results with the ground-truth class test, which shows the potential gain. The class test is performed using the ground-truth label, and therefore can not be done in practice. What we can do, however, is to use SegNet to predict the class label. We saw in Section 4.7 that using the SegNet predicted class label, we can have about 50-80% of the possible improvement compared to using the ground truth class.

5.2 Pose Estimation

In Figure 5.1 and Table 5.1 we show a summary of the pose estimation results. For all the results, except for l2norm (baseline), the pose estimation is excellent, below 1 centimeter of translation error for more than 75% of the pose pairs from the test set. We still see some improvement with respect to the upper range of errors, where the methods which had better inlier ratio, also have a somewhat better pose estimation error, however it’s not consistent. A more thorough evaluation needs to be performed to understand what is the next limiting factor. For example, why are 75% of the translation errors above 2 millimeters, for all the methods?
5.2. Pose Estimation

Figure 5.2: Inlier ratio results for the different correspondence matching methods with ratio test with and without class test.

<table>
<thead>
<tr>
<th>Method</th>
<th>Leaves</th>
<th>Non-leaves</th>
<th>Overall</th>
<th>Improvement over baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>l2norm (baseline)</td>
<td>0.045</td>
<td>0.194</td>
<td>0.084</td>
<td>1.000</td>
</tr>
<tr>
<td>normal (l2norm^2)</td>
<td>0.204</td>
<td>0.465</td>
<td>0.275</td>
<td>3.277</td>
</tr>
<tr>
<td>normal+class</td>
<td>0.220</td>
<td>0.580</td>
<td>0.307</td>
<td>3.664</td>
</tr>
<tr>
<td>normal then fork 3</td>
<td>0.285</td>
<td>0.495</td>
<td>0.349</td>
<td>4.166</td>
</tr>
<tr>
<td>normal then hard</td>
<td>0.291</td>
<td>0.524</td>
<td>0.360</td>
<td>4.287</td>
</tr>
<tr>
<td>mixed normal</td>
<td>0.305</td>
<td>0.547</td>
<td>0.378</td>
<td>4.502</td>
</tr>
<tr>
<td>normal then fork 3+class</td>
<td>0.299</td>
<td>0.598</td>
<td>0.378</td>
<td>4.510</td>
</tr>
<tr>
<td>normal then hard+class</td>
<td>0.305</td>
<td>0.631</td>
<td>0.392</td>
<td>4.668</td>
</tr>
<tr>
<td>mixed normal+class</td>
<td>0.327</td>
<td>0.658</td>
<td>0.423</td>
<td>5.042</td>
</tr>
<tr>
<td>mixed normal then mixed hard</td>
<td>0.360</td>
<td>0.566</td>
<td>0.427</td>
<td>5.094</td>
</tr>
<tr>
<td>mixed normal then mixed hard+class</td>
<td>0.380</td>
<td>0.660</td>
<td>0.462</td>
<td>5.502</td>
</tr>
</tbody>
</table>

Table 5.2: Median precision / inlier ratio for the different correspondence matching methods with ratio test, sorted by overall inlier ratio. +class means that we also use same (ground-truth) class test.
Chapter 5. Results and Discussion

Figure 5.3: Pose estimation results for the different correspondence matching methods with ratio test.

Figure 5.4: Pose estimation results for the different correspondence matching methods with ratio test with and without class test.
5.2.1 Accumulated Trajectory Error

We estimated the transformation between every two consecutive poses along trajectory 82 and computed the accumulated error. The overall trajectory length is 227.86 [m], over 93 poses with a distance of 2.45 [m] between every two poses. The trajectory goes 5 times around the tree similar to Figure 3.1. The results are shown in Figure 5.5. The accumulated translation error is between 10-14 [cm] for the Two Step methods and 23-30 [cm] otherwise. The rotation error is 0.76-0.92 [deg] for the Two Step methods and 1.45-1.67 [deg] otherwise. This suggests that the Two Step method is consistently better than using a single descriptor.

One thing to keep in mind is that this is a single result of accumulated error along a trajectory, rather than an average accumulated error over multiple trajectories. A single bad result in one of the transitions could make the accumulated error bad.

5.3 Precision / Recall

To gain more insight into the performance of our methods, in Figure 5.6 we show the precision-recall plot for the main methods. We also show the precision (inlier ratio) for different top-k values, which measures the fraction of query points for which the correct match is in its top-k candidates. As we increase k, so do we increase the precision. However, for the purpose of pose estimation, we still choose the best match (k = 1) for each query point, since the next best match is usually not so good and decreases the inlier ratio.

The first thing we see in the plot is that we have three lines, corresponding to the different networks that we use to find the top matches for all the query points. The
worst performing is the \texttt{l2norm} (baseline, trained with Euclidean distance loss), then \texttt{normal} (squared distance loss) and the best one is \texttt{mixed} (forked network trained on both normal and hard examples).

We can also see the effect of the Two Step methods. They only improve along the lines of the networks used to find the top candidates, such that their best match ($k = 1$) is better (e.g. \texttt{normal then hard} versus \texttt{normal}, and \texttt{mixed normal then mixed hard} versus \texttt{mixed normal}). Since for the Two Step methods we only rank the top-10 candidates with the second descriptors, the results for $k = 10$ with and without the Two Step methods are equivalent, and their results coincide. It would be interesting to try the Two Step approach with a bigger pool (more than top 10) of candidate matches.

Lastly, we can see again that it is easier to find the matches for non-leaf points than leaf points.
Figure 5.6: Precision-recall plots for the main methods. The different k values specify for each query point whether we have a correct match in the top-k matches. The three plots are for overall/leaf/non-leaf scenarios.
Chapter 6

Conclusion and Future Work

In this work we tackle the challenge of pose estimation using correspondence matching in vegetation / trees. We implemented the 3DMatch [2] network and trained it on our trees data set in various ways. We presented a few methods to perform correspondence matching within vegetation and evaluate them over a synthetic tree test set as described in section 4.4.2. With our best approach, the Two Step method (section 4.12) in conjunction with the Fork Mixed method (section 4.13.2), we reach precision values of 0.36, 0.57, and 0.43 for leaf points, non-leaf points and overall respectively. If we incorporate semantic class check, we may increase the precision results up to 0.38, 0.66 and 0.46, respectively. All of our methods showed excellent pose estimation results of less than 1cm of translation error for more than 75% of the test set.

Future Work

Multiple directions need to be explored with respect to vegetation correspondence matching and pose estimation.

First of all, a real data set needs to be created covering many different tree kinds, with ground truth poses and preferably at least a small number of correspondence points for the different scenarios (leaf, trunk, branch, inside/outside the tree, etc.). The possibility to automatically annotate correspondence points should also be evaluated. For instance, we can physically color some leaves and identify them in the images corresponding to the different poses. Since we modify the color, these images not be used in methods that rely on color. However, the color has no effect on the depth value, an therefore we can use the depth images with all the 3DMatch based methods discussed in the thesis.

Second, a more thorough evaluation needs to be performed over:

- different kinds of synthetic trees
- real data
- different view point distances
- more complex scenes with multiple trees
- dynamic trees
All needs to be thoroughly analyzed, and specifically the order of the top neighbors (section 4.9) need to be analyzed to see why they are being voted. Also, evaluation of existing SLAM methods in vegetation areas, and incorporating the discussed methods into them.

Third, the run time of the methods needs to be improved. As of now, extracting TDFs + descriptors takes between 10 seconds to a minute per image pose. Depending on the number of points, the correspondence matching step takes a few seconds more. To make this useful for in real time applications, the total run time should be considerably decreased (hopefully below 1 second). Sub-sampling the points may be possible, since the number of points is large and the precision is also relatively high.

Fourth, additional training methods need to be researched:

- Combining RGB or semantics with our network, for example as discussed in section 4.8.
- A single network trained with the Fork Mixed method showed very good results, and more work needs to be done to get an even better network that finds the top candidates for the query points. One idea which we did not try, is to again generate a new difficult training set (like in section 4.9.1) with the better Fork Mixed network, then use it to train the difficult branch of the Fork Mixed network.
- The Two Step approach should be tested with ranking the top 100/1000 candidates, rather than only the top 10.
- A Three Step approach could be tried, where we use the second network to rank the top 100 candidates, then a third network to rank the top 10 candidates.
- Use a different network architecture to find the best match in the second step, such as a ranking network [20, 21].
- Change the TDF cube sizes, or perhaps a combination of different sizes, which may help for the second step of the Two Step method, since it deals with small changes of similar structures.

Fifth, instead of doing correspondence matching, we may look at the problem as retrieving the best matching location within a second point cloud, given a query point cloud. Specifically, the match does not have to lie on a matching point, e.g. if it’s occluded, but we could still estimate it’s position precisely using its neighborhood.

And lastly, throughout the work we considered the trees to be static. In real life, this is not the case, and the pose estimation method (3D Euclidean transform + RANSAC) would have to be adapted. Maybe we need to take the shape deformation into account. The correspondence matching should probably still work, since even with wind, the changes are mostly global and deform the shape of the whole tree, rather than the local features. This also needs to be evaluated, and initially can be done with dynamic synthetic trees. Speed Tree, the synthetic tree generator that we used, supports dynamic trees, so generating them should be relatively easy.
Bibliography


