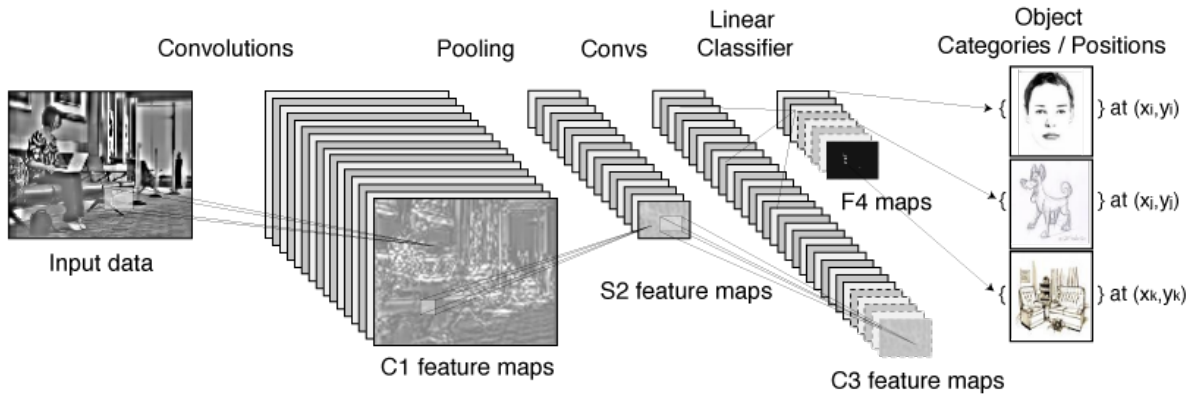
Polytech' Nice-Sophia
Sophia-Antipolis, France
September 01, 2014

Technische Universität München
Munich, Germany

Intermediate Internship Report

Learning similarity metrics for loop-closure detection using Convolutional Neural Networks

Arnaud TANGUY (SI5, VIM)



Supervisors :

Home University : Comport, Andrew

Hosting University : Sturm, Jürgen

Abstract

In the recent years, odometry algorithm have had sustained attention of the research community, which has led to great improvement in both accuracy and robusteness of the algorithms. However, as good as the algorithm might be, they are still only an iterative process approximately estimating the sensors location based on inaccurate sensor data. Over time, this leads to a drift of the estimated sensor location. To reduce this induced drift, a commonly used technique is loop-closure detection: by finding out whether the sensor is looking at a previously visited location, it is possible to use this knowledge to correct the estimated sensor locations over the whole trajectory.

This document explores the possibility of using *Convolutional Neural Networks* to recognise such loop-closures. The main idea is to learn small discriminative feature descriptors that can be efficiently used to determine whether a pair of images belongs to the same place.

Keywords: similarity metric learning, convolutional neural networks, loop-closure detection, SLAM

Contents

Introduction

We often don't realise how much we rely on our vision system to do everything in our lives. Of course, we are conscious of using our vision for very specific tasks, such as reading, looking for a specific object... What we often don't realise, is the extent to which we rely on vision to localize ourselves. Our brain has an incredible ability to remember information. As we walk around our environment, we remember in some way the essential information about where we've been. This allows us to get a sense of how we've moved with respect to our initial starting point, thus letting us to know where we are in the environment.

Since our vision system is helping us so efficiently to localize ourselves, why not do the same using computers and cameras? This is what scientist have been trying to do by developing a set of algorithm commonly referred to as *SLAM (Simultaneous Localization and Mapping)*. As the name indicates, these algorithms have the ability to create a map of the environment, and localize ourselves within this map.

SLAM

Simultaneous Localisation and Mapping uses only visual cues to determine both the camera's pose and the map of the environment. For simplicity of explanation, I will only describe 3D-SLAM, that uses cheap RGB-D sensors such as Microsoft Kinect or Asus Xtion to acquire visual data. These RGB-D cameras both provide low-resolution color and depth maps.

Visual Odometry

Visual Odometry refers to the process of determining the position and orientation of a visual system by analysing the associated camera images. Typically, this is done by tracking features in images, and estimating camera motion between them.

Figure 1 from Kerl et al. (2013a) show the basic principle behind the pose estimation. Image I_2 is obtained by warping image I_1 according to the transformation parameters $\xi \in \mathrm{SE}(3)$ (lie group of 3D rigid transformations). The goal is to find ξ such that the photometric error between I_1 and I_2 is minimal. The intuition behind that is easy to understand: we are trying to find the 3D transformation that maps the pixels of I_1 to those of I_2.

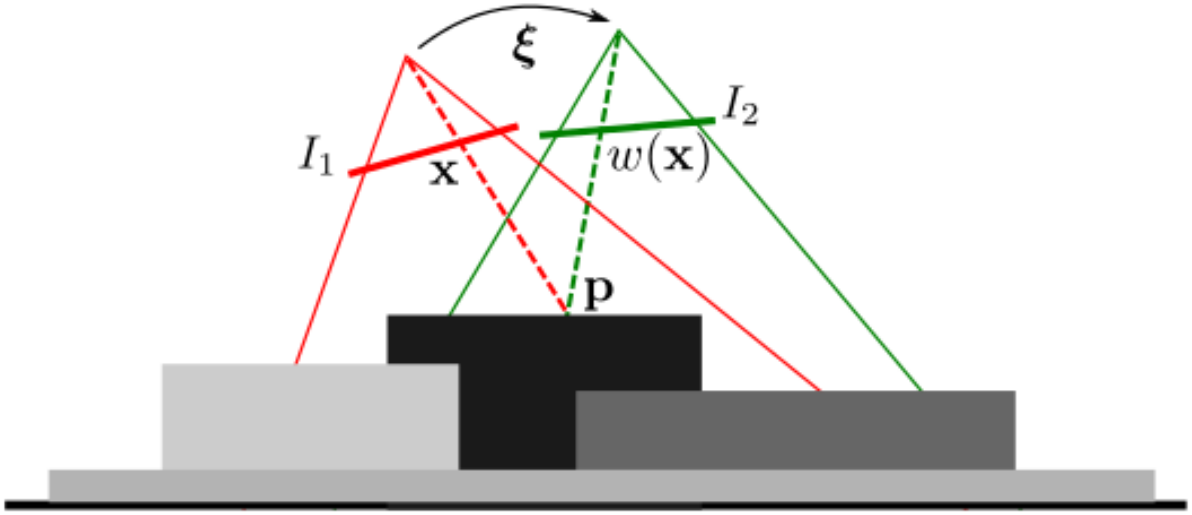


Figure 1 – The goal is to estimate the camera motion ξ such that the warped image I_2 matches the first image I_1.

Using this principle, only relative motion between one frame (the current frame in case of real-time applications) and a reference frame (commonly referred to as keyframe) can be estimated. Thus, estimated the odometry for a complete path is an iterative process. A keyframe is selected, and as long as the estimated transformation isn't too large for robust photometric match, the relative-motion is estimated with respect to the previous key-frame. When the camera has moved too far away from the last key-frame, a new one is inserted into a graph structure. The pose at any given moment can be computed by combining the pose determined for every key-frame.

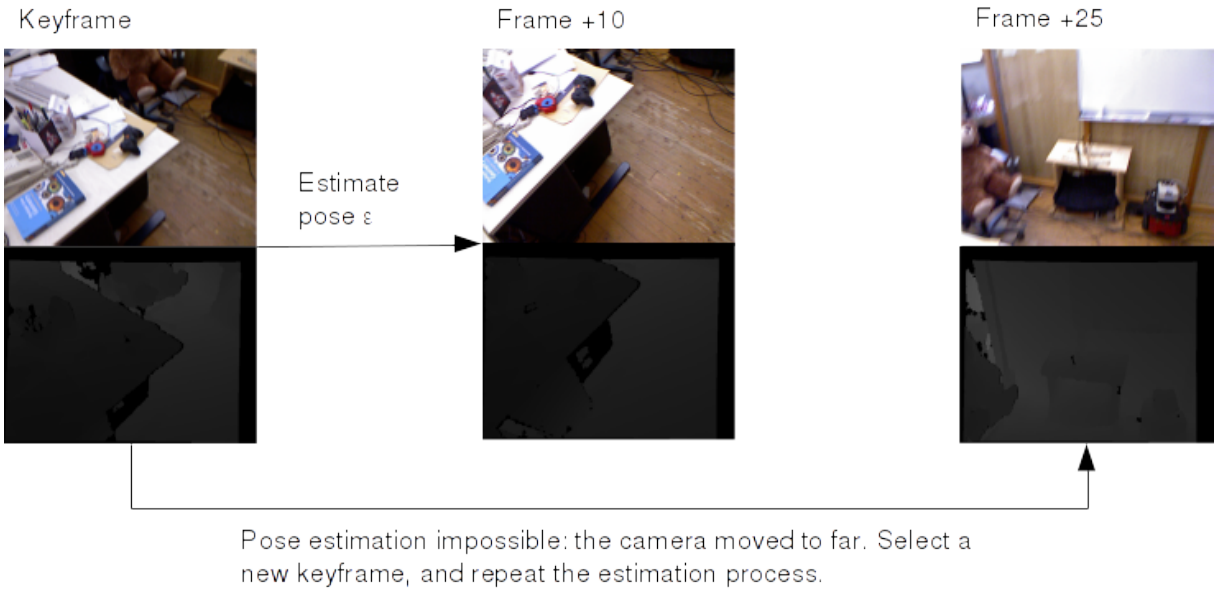


Figure 2 – Example of motion estimation process between a keyframe and 2 candidate frames. The first frame can be matched with the keyframe as the camera hasn't moved too far away yet. The second frame has moved too far away, and the photo-metric match fails. Thus a new keyframe must be selected to continue the process.

The pose estimation step isn't perfectly accurate, and thus, this can leads the pose estimation to slightly drift overtime. This estimation error is caused by sensor noise and inaccuracies of the error model, which does not capture all variations in the sensor data. Thus a critical part of the algorithm is to reduce the drift as much as possible, so that the trajectory doesn't keep getting worse over time. One way to achieve this is to use the loop-closures to correct the pose estimates. The goal is to detect when the camera is looking a previously seen key-point and use this knowledge to correct the current pose. Then, all the camera poses in the loop can in turn be more precisely estimated.

Figure 3 from (Kerl et al., 2013b) shows the effect of drift and how loop-closure detection helps improving the estimated trajectory.

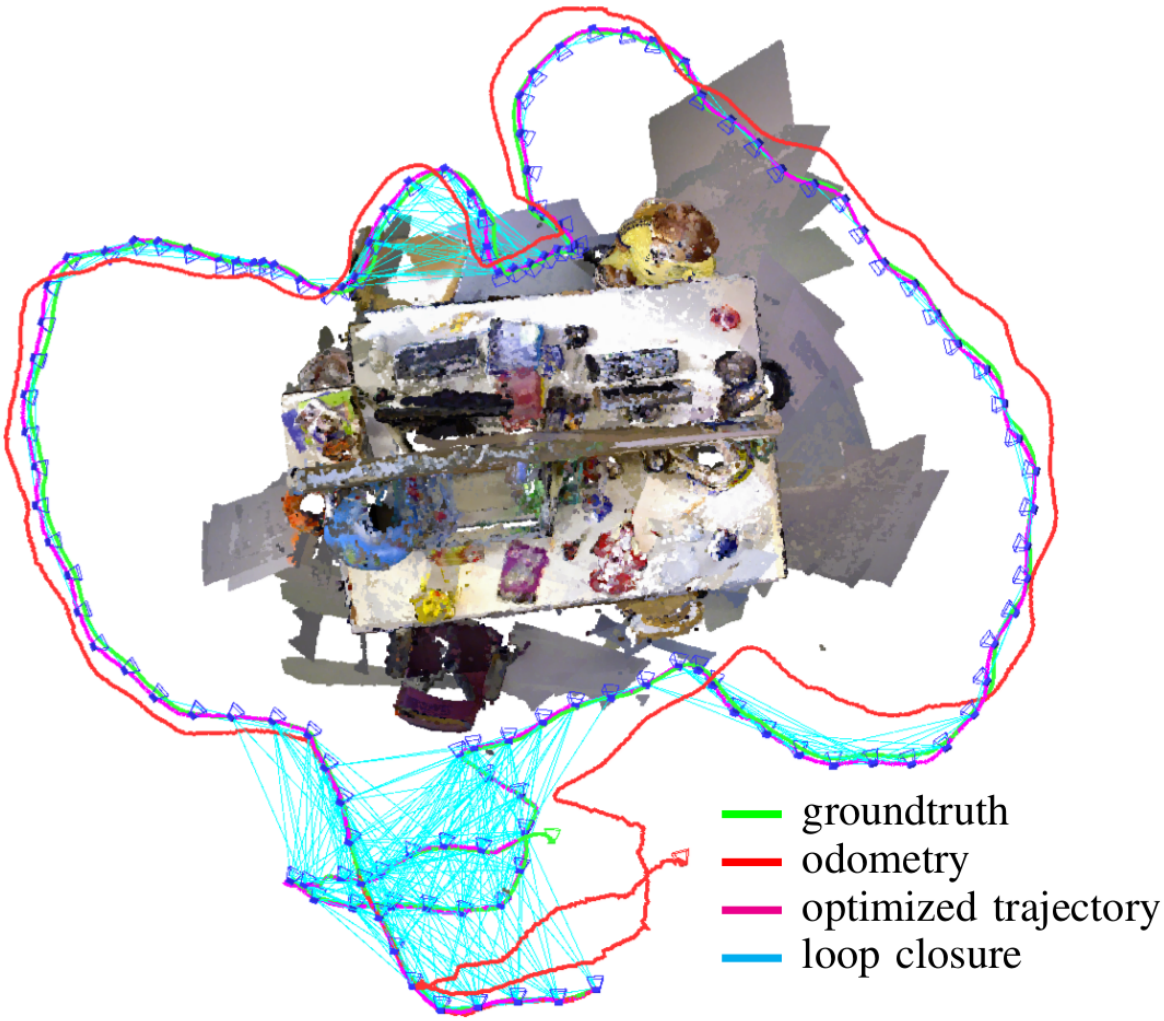


Figure 3 – Keyframe-based dense SLAM method for RGB-D cameras. This shows the advantage of using loop-closures to improve the pose estimation.

1 Background

Loop-closure detection is important in order to account for the drift accumulating over time. By finding known reference points, every pose along the trajectory can be improved. Typically a loop closure algorithm performs the following tasks

Loop-closure detection A lookup algorithm finds best matches between the current keyframe and all previously stored keyframes.

Map optimization Use the previously found loop-closures to reduce the drift.

1.1 Loop-Closure algorithms

Several approaches exists for the detection of loop-closures. The simplest strategy is a linear search over all existing key-frames, i.e, a new keyframe is matched against all others. For very small scenes, this method could eventually be applicable. However, with this method, the time required for loop-closure detection increases linearly with the number of keyframes, which is unacceptable for real-time applications. Therefore, it is important to only match against the most likely candidates. One common method is place recognition systems extracting feature descriptors and training efficient data structures (such as Bag of Words with SIFT-Features (Angeli et al., 2008)) for fast search and candidate retrieval (Cummins and Newman, 2010; Nister and Stewenius, 2006). Other common approaches use metrical (Henry et al., 2014) or probablilistic (Stuckler and Behnke, 2012) nearest neighbour search.

Some methods also consider doing continuous loop-closure detection. This is usually achieved by using a voting mechanism. I was asked to review an interesting paper that seems to have found a very promising method. As the paper hasn't been published yet, I won't tell too much about it. The key idea is to integrate the evidence over multiple frames (in contrast to most existing systems such as FABMAP which just compare two frames). This is done by matching BRISK features between the current frame and the frames in the database, and accumulating the resulting weights in a 2D image capturing the correlations between frames. The correlation image is then processed to find regions corresponding to loop-closures. The work published in Milford and Wyeth (2012) presents a similar technique.

Once the loop closures have been found, steps can be taken towards reducing the drift. In keyframe-based SLAM, one common technique is to represent the map as a graph of camera poses, where every vertex is the pose of a keyframe, and the edges represent relative transformations between the keyframes. Valid loop-closures become new edges in the graph, as illustrated by figure 4. The error can then be corrected by solving a non-linear least squares optimization problem, as described in Kerl et al. (2013b).

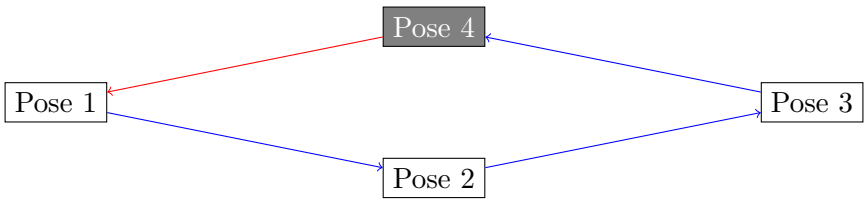


Figure 4 – Each vertex represents the pose of a keyframe. Here we assume that a loop closure was found between the 4th and 1st keyframe. Thus, they are linked together. An obvious result of optimisation will be to pull Pose 1 and 4 closer.

1.2 Convolutional Neural Networks

1.2.1 Description

In order to provide neural networks that are efficient at handling visual information, Yann LeCun and Toshua Benhio tried to capture the organization of neurons in the visual cortex of cats, which is known to consist of maps of local receptive fields decreasing in resolution as the information goes towards the back of the visual cortex. This lead to a new model of neural networks, commonly referred to as Convolutional Neural Networks. A CNN implementation can be described by the following process:

1. Convolve several learnt filters on the input image
2. Subsample this space of filter activation.
3. Repeat step 1 and 2 until an appropriate feature size is reached
4. Use the feature as input to some another learning algorithm to solve a particular task. The most commonly used method at this step is to use a standard feed-forward neural network.

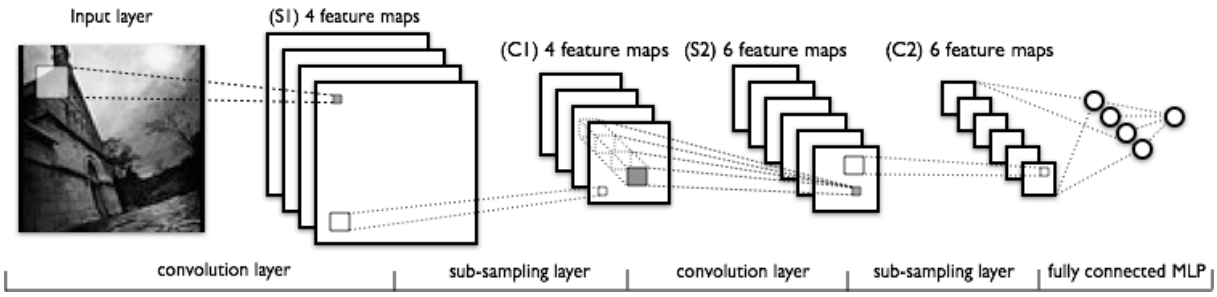


Figure 5 – Complete Structure of LeNet CNN.

The *convolutional layer* is responsible for the filtering of the input image. A filter consists of a set of connection weights, with the input being a small 2D image patch, and the output being a single unit. Since filter is applied repeatedly at each location in the input image, the connectivity looks like a series of overlapping receptive fields, as shown in figure 6. While there is plenty of connections between the input layer and filter output layer (due to the convolution being applied to every pixel of the input image), the weights are tied together, thus limiting the amount of parameters to adjust during backpropagation to a single instance of the filter. This tied-weight property is what makes backpropagation for CNNs much more efficient than the counterpart in standard feed-forward neural networks. Due to the nature of this layer, it is possible to apply it to any structured input, not necessarily only images. This property could be used to handle other information, such as depth with CNNs.

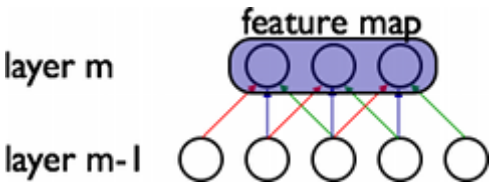


Figure 6 – Sparse connectivity of the convolutional layer. Weights of the same color are shared.

Now that the input has been convolved, the next step is to reduce its dimensionality. This is useful to both subsample the 2D signal, and also increase the positional invariance of the filters. The subsampling method used in LeNets is referred to as "max pooling". This involve

splitting up the output images of the convolutional layer into a grid of non-overlapping patches, and reducing each patch to it's maximal value. This steps helps the network to conceptualize the information by moving the processing from pixel information towards area information.

As we add more successions of convolutional and pooling layers, the network will "conceptualize" the information ever more, resulting in compact features providing abstract location-invariant feature descriptors, that are also somewhat robust to small local distortions.

Finally, the result is processed according to the goal of the network. The most classic use of CNNs is found in image classification tasks. For such tasks, fully-connected perceptrons are usually used to perform the actual classification task.

1.2.2 Description of the most common layers

Most Convolutional networks rely on the same type of layers, here are the most common:

Convolution layer Applies a convolution operation on the input data. The convolutional layers are used to generate some task-driven filtering, in which the filters are directly learnt from data (see figure 14 for an example of learnt weights).

Pooling layer Reduces the data dimensional by subsampling the data. The most commonly used type of sub-sampling used is max-pooling, as it is efficient at further abstracting translation invariance within the data.

ReLU layer A rectifier is an activation function defined as $f(x) = \max(0, x)$. This type of activation function has been argued to be more biologically plausible than the widely used logistic sigmoid function. The role of rectified linear units is to introduce non-linearities to the network.

Fully connected layers (inner-product) These layers are most commonly used prior to the classification task.

Loss layer Any minimization algorithm is trying to solve a specific problem. The role of the loss function is to best describe the minimization problem and drive the learning of weights towards the desired solution.

1.2.3 Interesting results

Convolutional Neural Networks have recently shown state of the art result in several object recognition challenges. In 2010, a deep convoutional network (CNN) achieved top-1 and top-5 error rates of 37.5% and 17.0% in the ImageNet LSVRC-2010 contest, which was considerably better than the state of the art. In 2012, another CNN obtained a top-5 test error rate of 15.3% in the ILSVRC-2012 challenge, considerably better than the 26.2% achieved by the second best entry (Krizhevsky et al., 2012).

CNNs have also shown an impressive ability at building high-level features from unsupervised training. In (Le, 2013), it was shown that a sufficiently big CNN (with 10 billion parameters in this case) can learn high-level class-specific neurons from unlabelled data. For instance, their network was notably sensible to the concept of human face, but also cat face, human body, and other notable high-level concepts.

Most importantly, in Chopra et al. (2005) CNNs were applied successfully to learning similarity metric between pairs of RGB images. In this work, they showed that it is possible to learn similarity metric from data that can later be used to compare or match new samples from previously unseen categories. All these success and interesting properties of CNNs have prompted the Computer Vision Group at *Technische Universität München* to try and use them in the context of loop-closure detection for SLAM.

2 Project Description

Loop-closure detection is often based on hand-crafted feature descriptors (such as SIFT, BRISK...), used together with an efficient pruning algorithm to provide fast lookup speeds.. While these features have been proven to be efficient for recognising similarities between images, there is still room for considerable improvements.

In this project, we propose to explore the possibility of training Convolutional Neural Networks to recognise loop-closures robustly and efficiently. The idea is similar to the one proposed in Chopra et al. (2005):

> The main idea is to find a function that maps input patterns into a target space such that a simple distance in the target space (say the Euclidean distance) approximates the "semantic" distance in the input space. More precisely, given a family of functions $G_W(X)$ parameterized by W, we seek to find a value of the parameter W such that the similarity metric $E_W(X_1, X_2) = ||G_W(X_1) - G_W(X_2)||$ is small if X_1 and X_2 belong to the same category, and large if they belong to different categories. The system is trained on pairs of patterns taken from a training set. The loss function minimized by training minimizes $E_W(X_1, X_2)$ when X_1 and X_2 are from the same category, and maximizes $E_W(X_1, X_2)$ when they belong to different categories.

This project aims to apply a similar method to loop-closure detection:

— Implement a Siamese CNN network.
— Create a dataset of loop-closures, or use a general purpose datasets of image similar images for training.
— Use the trained CNN to generate feature descriptors for each new keyframe.
— Create a compact representation of the feature descriptors. Research has shown that it is possible to map high-dimensional feature descriptor into a binary codes, without reducing their performance (Gong et al., 2013; Norouzi et al., 2012)
— If the learnt feature descriptors provide better loop-closure detection, integrate them in the metric-based loop-closure search algorithm (Kerl et al., 2013b).

This is a speculative research project, and it is impossible to guess whether the learnt feature descriptors will perform any better than the currently used methods.

3 Siamese Neural Network

In this section, we will focus on describing the Siamese architecture and appropriate loss functions that can be used for learning appropriate feature descriptors.

Siamese neural networks aim to accommodate problems which require two input elements to be processed together, such as for instance a pair of images. This can be used to train a network to recognize image-similarities (with for instance applications to face-recognition in Chopra et al. (2005)), or, in our case learn feature descriptors for loop-closure detection.

The principle of this architecture is showed in figure 7. Images X_1 and X_2 are set as input of two CNNs that are sharing the same weights W. The forward pass will compute the output $G_W(X_1)$ and $G_W(X_2)$. The purpose of weight sharing is to ensure that both network will process the images in the same way during the forward pass, while allowing backpropagation to adjust weights according to the output of each network. This will be further explained later.

A loss function must then be computed based on the output descriptors $G_W(X_1)$ and $G_W(X_2)$. This is what will define the goal of the minimization process.

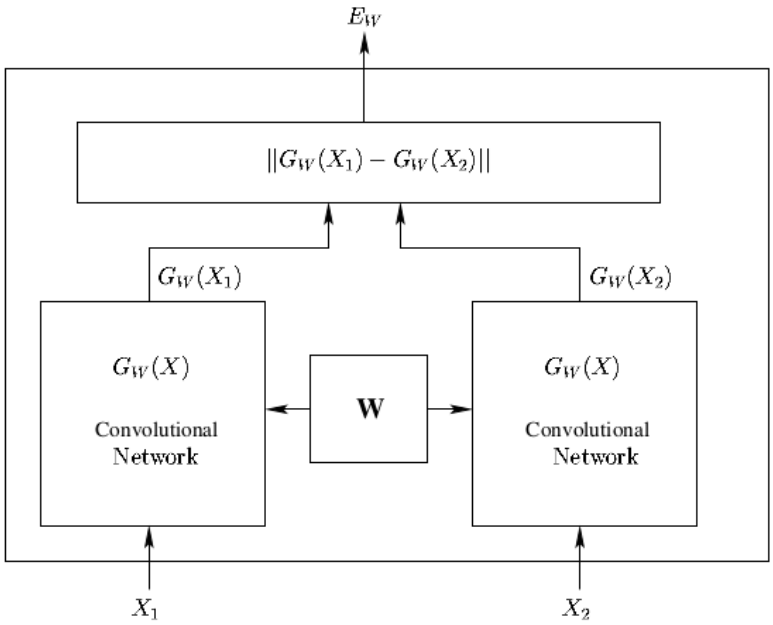


Figure 7 – Siamese Architecture

3.1 Backpropagation

Backward propagation of errors is a common method for training neural networks. The goal is to update the network parameters towards a state where the loss function has a minimal cost. Most minimization algorithm achieve this by using gradient descent. Neural networks are no different, and commonly used stochastic gradient descent to learn its trainable parameters.

Neural network are organized as a graph of interconnected layers, linked together through weights and biases (see figure 15). The role of gradient descent is to figure out in which direction to update each weight so that the networks results improves towards the correct solution. To do this, each weight must be updated in the direction of the error derivative with respect to itself. Let $E(W)$ be the layer's error for the current weights of the network W. Each individual weight w_i must be updated in the direction of the following derivative.

$$\frac{\partial E(W)}{w_i} \tag{1}$$

This can be easily achieved for the output layer (loss layer) of the network, as both the error for each neuron and the weights are known. However, the hidden layers don't have any error information about each neuron's error. That is where the backpropagation algorithm comes into play. It's role is to figure out each neuron's error (commonly referred to as δ_i) by propagating the final error back through the network. Once all the error values for each neuron are known, the derivative 1 must be computed to allow each weight to be updated.

The back propagation algorithm can be divided in two phases: propagation and weight update. The propagation pass consists of determining the error corresponding to each of the neuron's activation value. The weight update phase consists of computing the actual gradient of the error with respect to each weight, and update them accordingly.

Phase 1: Propagation

Forward propagation Forward propagation of the input data (X_1 and X_2) through the network in order to generate the propagation's output activation for each neuron. Despite sharing the same weights, the two Siamese networks will have different activation values for each neuron (due to having a different input).

Backward propagation Propagate the error from the last layer all the way back to the input layer (compute all the δ_i of the network).

Phase 2: Weight Update

Weight update (without shared weights) Thanks to the forward pass, each neuron's activation value is known, and thanks to the backwards pass, the error δ_i at each neuron is also known. The weights can thus be updated as such, where x_{ij} and w_{ij} are respectively the output activation value for the j^{th} neuron of the i^{th} layer.

$$w_{ij} = w_{ij} + \epsilon * w_{ij} * x_{ij}$$

Weight update (with shared weight) This is the only modification of the backpropagation algorithm needed to learn a Siamese network with shared weights. The forward and backward pass of the algorithm are done as usual. This leads to two distinct set of weights, one for each side of the network. However, the weights are shared, so they have to be updated together. Let the shared weights be respectively named w_{ij} and w'_{ij}, and the output named x_{ij} and x'_{ij}.

$$\begin{aligned} s_{ij} &= w_{ij} + w'_{ij} + \epsilon * (w_{ij} * x_{ij} + w'_{ij} * x'_{ij}) \\ w_{ij} &= s_{ij} \\ w'_{ij} &= s_{ij} \end{aligned}$$

3.2 Loss Functions

We chose to create a loss function that would drive the neural network to create descriptors that are closeby in euclidian space when a genuine pair is provided, and far away in case of an impostor. The choice of the euclidian space is somewhat arbitrary, but provides the advantage of being well studied, and providing a fast way to compare a pair of descriptors by simply using a norm.

To achieve this goal, several potential loss functions have been identified. The two most suitable ones will be presented here.

3.2.1 Hinge loss function

The *hinge loss* function is commonly used to train classifiers. It is notably very commonly used in SVMs. The hinge loss goal is to perform "maximum-margin" classification, which is a

great property for training Siamese networks. Indeed, the output of the siamese network can be seen as a classification problem, in which input pairs of images are either similar or dissimilar. For an intended output $t = \pm 1$ and a classifier score y, the hinge loss of the prediction y is defined as

$$L(y) = max(0, 1 - t.y)$$

To perform a classification that will lead to suitable descriptors for computing their relative distance in euclidean space, we use the following classifier score, where w is a weight vector and b a scalar bias, and $f(x) = x^2$

$$\begin{aligned} y &= b + \sum_i w_i f(X_{0i} - X_{1i}) \\ &= b + w.x \text{ where } x = (f(X_{00} - X_{10}), ..., f(X_{0n} - X_{1n})) \end{aligned}$$

The hinge loss is a convex function and as such can be used for the gradient descent. As it isn't differentiable we will use its subgradient with respect to the weights w of the network

$$\frac{\partial L}{\partial w_i} = \begin{cases} -t.x_i & \text{if } t.y < 1 \\ 0 & \text{otherwise} \end{cases}$$

3.2.2 Contrastive loss function

This loss function is designed specifically to minimize the Euclidian distance between features descriptors $G_W(X_1)$ and $G_W(X_2)$ when X_1 is similar to X_2 and to maximize this distance when they are dissimilar. The chosen loss function to achieve this goal is very similar to the one designed by Chopra et al. (2005), that tries to minimize the energy associated with the difference between the descriptors when they belong to a genuine pair, and maximize it when they belong to an impostor.

The energy and the loss function are expressed as such:

$$\begin{aligned} E_W(X_1, X_2) &= ||G_W(X1) - G_W(X2)||_1 \\ L(E_W) &= (1 - Y)L_g(E_W) + Y L_i(E_W) \end{aligned}$$

Where:
— (X_1, X_2) is a pair of input images
— $G_W(X)$ represents the output descriptor from the neural network for input X
— E_W is the energy associated with a pair of descriptors
— Y is the label associated with a genuine ($Y = 0$) or impostor ($Y = 1$) input pair.

The role of the L_g function is to reduce the cost associated to the energy for a genuine pair, while the role of L_i is to increase the cost for an impostor pair. Thus, L_g must be a monotically decreasing function, while L_i must be a monotically increasing function. The exact best properties of both functions are well defined in Chopra et al. (2005). The paper suggests using

$$\begin{aligned} L_g(E_W) &= \frac{2}{Q} E_W(X_1, X_2)^2 \\ L_i(E_W) &= 2Q\mathbf{e}^{\frac{-2.27}{Q} E_W} \end{aligned}$$

Where Q is a constant set to the upper bound of E_W.

Backpropagation Let's compute the needed derivatives for the backpropagation algorithm. We will consider the loss function to be fully connected to the previous layers. The following method can be applied to other types of connectivity with some minor adjustments.

$$f_w(X_1, X_2) = G_W(X_1) - G_W(X_2) \tag{2}$$

$$\frac{\partial L_g}{\partial W} = \frac{\partial L_g}{\partial E_W}\frac{\partial E_W}{\partial W} \tag{3}$$

$$\frac{\partial L_i}{\partial W} = \frac{\partial L_i}{\partial E_W}\frac{\partial E_W}{\partial W} \tag{4}$$

Let's solve 3 first. Let's call each neuron value in the layer linked to the loss layer (containing the descriptors values computed by the network) $I_1, ..., I_N$, where N is the descriptor dimensionality as defined by the network's structure.

$$\frac{\partial L_g}{\partial E_W} = \frac{4}{Q}E_W$$

Now, we need to compute $\frac{\partial E_W}{\partial W}$. However, E_W is an L1-norm, which is non differentiable. However, we can use its sub-gradient to derivate it.

Let f_W be the inner function of the norm:

$$f_W(X_1, X_2) = WI_1 - WI_2 \in \mathbf{R}^N$$

Let

$$g \in \mathbf{R}^N$$

$$g_i = \begin{cases} -1 & f_i < 0 \\ 0 & f_i = 0 \\ 1 & f_i > 0 \end{cases}$$

The subgradient of E_W is defined as

$$g^{\mathbf{T}}\frac{\partial f}{\partial W}$$

We have

$$Y_1(I) = W_1 I \qquad Y_2(I) = W_2(I)$$

$$f_W(X_1, X_2) = \begin{bmatrix} Y_1(I_1) - Y_1(I_2) \\ Y_2(I_1) - Y_2(I_2) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

We show that

$$\frac{\partial f_1}{\partial W} = (I_1 - I_2)$$

$$\frac{\partial f_2}{\partial W} = (I_1 - I_2)$$

Thus

$$\frac{\partial f}{\partial W} = \begin{bmatrix} I_1 - I_2 \\ I_1 - I_2 \end{bmatrix}$$

Finally we have

$$\frac{\partial L_g}{\partial W} = \frac{4}{Q} E_W(X_1, X_2) g^{\mathbf{T}} \begin{pmatrix} I_1 - I_2 \\ I_1 - I_2 \end{pmatrix} \tag{5}$$

A very similar derivative can be computed for the impostor function L_i.

4 Caffe Framework

Implementing such a complex fully-functional neural network is a daunting task. There is a lot a various building blocks to implement: all the individual layers (convolutions, relu, fully-connected...), a minimization algorithm such as stochastic gradient descent, saving and loading the network state... And all that must be really efficient, implemented on the GPU. This is just too much for a one-man job, especially as a 6 month project. That is without relying on the work shared by the researchers at the Berkeley Vision and Learning Center. They've developed and released Caffe (Jia, 2013), an open-source convolutional neural network framework that contains most of the features needed for our project. This is the fastest and most expansible framework we could find, and it being open-source gives us the possibility to contribute our improvement directly to the framework itself, so that it can be used by others in the future.

4.1 Caffe Development

Caffe's framework development process is a perfect example of good methodology, and as it finally made me understand the point of strict coding rules and unit tests, I will spend some time to describe the development process, and what I learnt from it.

4.2 Overview of Caffe

As most of the project revolves around the Caffe framework, I will briefly describe the basic working principles of the framework.

Caffe is a C++/CUDA neural network framework. It features

— A fast CPU/GPU implementation. A forward pass typically takes about 2.5ms on GPU, and about 20ms on CPU (on the alexnet architecture).
— A library of common layers (convolution, pooling, fully-connected...)
— A fast stochastic gradient descent solver
— A higly customizable framework through the use of Google's procolbuffer library.

Caffe's architecture revolves around the concept of blobs. A blob is an entity containing all required data. This can mean various things, depending on what you want to do (input data, weights, biases, temporary data to be saved for backpropagation, labels...).

Figure 8 shows how a convolutional layer would work. It takes a blob of data as input. This can be a batch of image extracted from the dataset (here images with 3 channels, and of size 227x227), or data coming from previous layers. It doesn't matter to the convolutional layer: all it does is compute data formatted in a predefined way, regardless of where it comes from or what it means. The convolutional layer will compute the convolution based on its weight and bias blobs, and write the result to the output blob, so that it can be used by other layers. The same goes for backpropagation, every layer will pass on the required information to the previous layer. This leads to a very flexible design, where every layer takes care of its own forward and backward propagation, and communicate with other layers through the use of blobs.

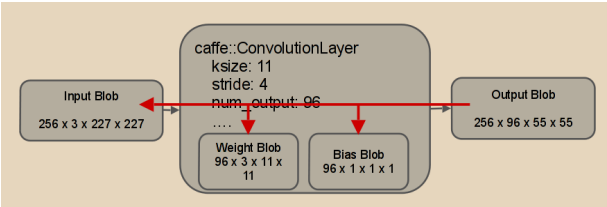


Figure 8 – Example of caffe's blob usage

This designs also allows for arbitrarily directly acyclic graph (DAG) designs, which as we will later see comes in handy for Siamese Networks.

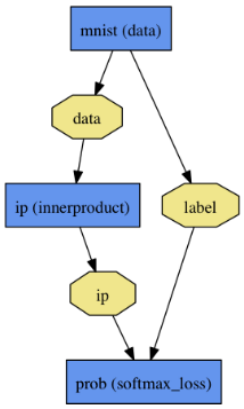


Figure 9 – Very simple neural network architecture defined with caffe. The blobs are represented by hexagons, the layers by squares. This example network consists of a feed-forward neural network (innerproduct), linked to a softmax classifier. Notice how the blob architecture allow to directly link the label data with the softmax layer, thanks to the blobs based architecture.

4.2.1 Developpement Process

Caffe's development, as any large-scale project, revolves around a version control system: Git. While I had used Git on many small to medium scale projects before, I never had the opportunity to use it on a large scale, highly active project. And I was lucky to end up contributing to a very rigorously maintained project. To give an example of scale, Caffe has about 30 active contributors. The 3 most active developers contributed well over 6 million lines of code to the project. There has been about 2000 commits contributed to the development branch to this day, and a dozen new commits are contributed every day. And all that while working on one piece of software architecture very well renowned for its difficulty of debugging. So how does one manage such a big project? The answer lies in their own "How to contribute" guide in the form of a quote from Ralph Waldo Emerson: *"a foolish consistency is the hobgoblin of little minds"*. For them, this entails:

Coding Convention The Google coding style is strictly enforced. Code will not be merged in the main development branch if it doesn't abide by the coding rules. That not only entails coding style (tab width, line width, bracket style), but also C++ coding rules.

Unit tests Any code provided to the Caffe framework must be provided along with extensive unit tests ensuring the good workings of the code, now, and at any future point in time. The development of unit tests is made simple by the use of the Google Testing Framework, providing all the features necessary for efficiently writing and running unit tests.

Efficient logging system By using Google's logging system, the network can be efficiently debugged on various levels at run-time through the use of log files. This is much more effective than the more commonly used temporary *"printf"* style of logging.

Automatic building process All code proposed to the mainstream repository is automatically checked for coding convention compliance, and validity of the unit tests. Unless both these steps are correctly passing, code won't be accepted.

Code review Before merging any code, it will be reviewed, discussed and improved as much as possible. Also, they agree with the "commit often, perfect later" practice, which helps fostering healthy communication between the contributors.

4.2.2 Unit tests

Before working on the Caffe framework, I never really saw the point of unit tests. After all, of what use could they possibly be on a school project led by 2 to 4 people developing in the same room half of the time? Sure, we all understood that it could prevent us from breaking some previously written code, but it never really seemed to make sense to spend so much time working on writing the test.

But it's all together very different for a project of such scale that isn't meant to last for a couple of weeks, but years and years. By having every part of the code carefully and automatically tested, the development can be made much easier, and much safer. If one wants to implement a new functionality, he can make sure in only a couple of minutes that he hasn't broken any of the previous functions. Better yet, one can use the unit tests as a checking tool when re-factoring becomes necessary. I was led to refactor the data layers, and found the unit tests to be incredibly useful in ensuring the validity of the refactored code.

This is especially true for this project. Here is an outline of what is being checked by the unit tests

— Consistency of the network definition and save files
— Loading of data in all possible settings.
— Forward/Backward propagation of all layers in all possible settings.
— Coherency check between the CPU and GPU implementation
— Various full network configurations

None of these could be trivially checked without a full set of unit tests.

This project opened my eye on how useful unit tests can be, and I intend to use them extensively in every future project.

4.2.3 The Documentation Problem

Despite all the care taken into producing the best of code, one thing, as is the case of many open-source, and especially research projects, has been greatly overlooked: the documentation. The only existing documentation is very succinct, and only concerns itself with explaining the high-level way of running the framework using the python interface. The C++ documentation was virtually nonexistent up until a couple of weeks ago, when we started setting automatic documentation tools such as Doxygen to generate documentation from the code comments. However the code-base now has some catching up to do to get a full documentation.

Also, the project didn't have a proper communication channel to discuss non-development driven issues, such as implementation problems, general questions about the framework... A mailing list has now been set up, allowing for easier communication and collaboration.

4.3 Dual CPU/GPU implementation

Another really interesting feature of this project, is the dual implementation on CPU and GPU. I didn't know much about CUDA before starting, and even less about how to structure a program using both CPU and GPU. The code-base has been a huge source of learning and inspiration, on how to organize such a big program to be run on either architecture. I am now confident that I could write a similar architecture if need be.

Interestingly, the current code runs with CUDA 5.0, but a lot of now core-features of CUDA 6 had already been implemented within the framework. The most notable one is the transparent shared memory management between CPU and GPU (named unified memory in CUDA), that is a true wealth of information on how to share resources between devices.

4.4 Personal experience

Developing for such a big project has certainly been eye opening, and I hope to take the experience gained from it into account for my further projects. In particular, I would stress the importance of:

Documentation Starting developing for the Caffe framework was very painful and time consuming due to the lack of documentation. While the main contributors have been developing it for years, and as such have a fair idea of the project intricacies, a new developer has no such back and will struggle without documentation. It took me a couple of months to fully grasp most of the code concepts, and a lot of time was wasted from misunderstanding of the inner workings of the layer system. My future project will be documented with that in mind, so that myself and any other potential contributor can easily keep track of the project.

Unit test It is certainly no waste of time to rigorously write them, and can help the development process a lot, especially in bigger projects. I will do my best to write them where appropriate.

Enforcing a coding style While I do not personally agree with every rule and convention of the Google coding style, this project made it clear to me that it is best to have everyone follow a clear well-defined set of rules, than to code arbitrarily according to one's belief. Thus I will strive to enforce a set of rules to my future projects, whether it'd be Google coding style or any other agreed upon style.

I would also add a personal advice to this set of goals:

Fast merging process The longer a pull request is left opened and unmerged, the most likely it is to considerably diverge from the rest of the project development, thus requiring considerable management from the pull-request submitter. A clear and efficient reviewing process should be enforced to avoid needless maintaining of code.

5 Development of the Siamese Network

5.1 Generation of a Loop-closure Dataset

In order to teach the network how to detect loop-closure detection, it is paramount to create a dataset as close as possible to the situations that the network is meant to encounter after training. Thus, we decided to create a loop-closure dataset by using some of the most common RGBD datasets available: the Freiburg SLAM Dataset (Sturm et al., 2012), MSRC RGBD scenes Glocker et al. (2013), Washington's RGBD Scenes Dataset Lai et al..

Each of these datasets contains sequences of RGBD images, along with ground-truth pose. I wrote a set of scripts to convert all datasets to a format compatible with the Freiburg dataset, so as to be able to use the tools and programs developed here at TUM.

Generate loop-closures based on groundtruth only To compute the loop-closure correspondences, the first version only used the groundtruth information, and for each possible image pair in the dataset, it was kept as a loop close only if both images were viewed from a similar viewpoint, which in effect corresponds to a small enough translation and rotation of the camera computed from the groundtruth.

Refine the loop-closures The first version presented above is too simplistic for generating a dataset containing only robust loop-closure image pairs. Only using the groundtruth information doesn't provide reliable guarantee on data similarity. It indeed doesn't account for any sensor defects such as motion blur, or more importantly it doesn't take into account occlusions. Thus, the dataset is further refined to select only the loop-closures with most similarities.

To do so, we performed a warping operation to try and transform one image of the pair into the other by using the groundtruth information. Depending on how well the wrapped image matched with the reference image, I chose or not to keep the pair as a genuine loop-closure. The correlation coefficient is based on the percentage of pixels that were correctly wrapped. This prevents big occlusions, and important sensor noise.

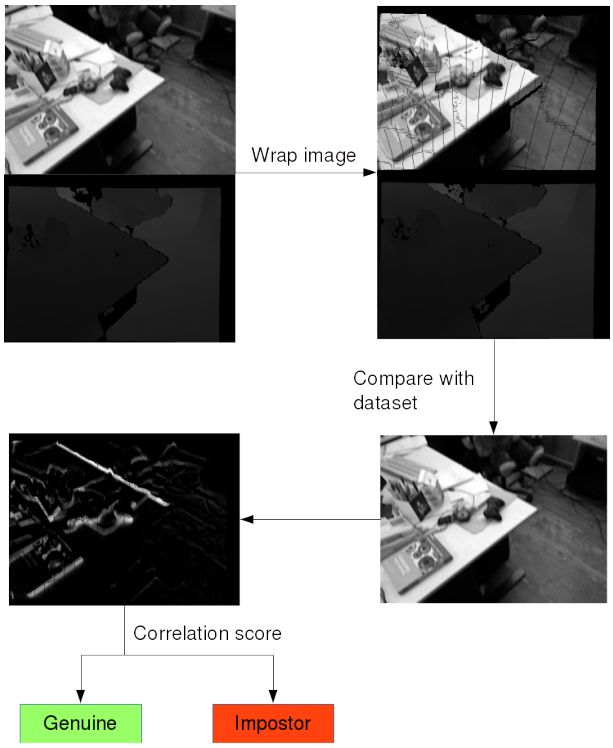


Figure 10 – Dataset refinement

Improve performance by packing images inside a database All datasets are composed of a huge amount of images stored in folders. As Caffe's architecture is meant to work on batch of images, reading them independently from disk is too inefficient. Indeed, in addition to the file system access time, one needs to decompress the images, convert the data to an acceptable format for Caffe's framework. This is too slow to be done on the fly.

Instead, we chose to pack all images together inside a lightweight efficient database: Symas Lightning MDB (LMDB). Images are read and converted from interleaved images (R,G,B,D - R,G,B,D...) to a contiguous array of image values (R,R,R,R.... - G,G,G,G... - B,B,B,B... - D,D,D,D...) in order to take full advantage of Caffe's architecture, particularly on GPU.

5.2 Data Layers

The data layers role is to feed data to the neural network. Typically, this means reading images, and eventually labels from a database and copying them inside a blob that can be used by any layer in the network.

5.3 Improving the existing data layer architecture

Most of the existing data layers needed to be greatly cleaned up and refactored. They contained a lot of code duplication, and a poor prefetching architecture. My main contribution was to introduce a clean data prefetching architecture, allowing to start fetching data in a separate thread on the CPU while the network is running. This comes from the simple observation that

while the network is computing the forward/backward pass, it doesn't need to read data from disk. Thus I implemented a generic way of starting a prefetching thread, and implemented it within all relevant data layers.

Also, each data layer can apply on the fly transformation to the data. However, it was previously done on a per-layer basis, and thus this led to a lot of code duplication. I abstracted the data transformations to allow any data layer to make use of it.

5.4 Development of the input data layer

The existing input data layers weren't designed for use with Siamese networks. Thus, I had to write my own data layer. It works by reading image pairs from a file (generated as described in 5.1), and reading the corresponding data from a prefetch thread. The data is read in two channels, thus allowing to independently create two blobs containing batches of paired data that can be used by the Siamese network.

This is where Caffe's ability to understand arbitrary DAG (directly acyclic graphs) models comes into play. This allows us to create to branch out from the data layer to the two separate sides of the Siamese network, which will be later recombined into one network to compare the descriptors.

5.5 Weight Sharing Implementation

One of the main development hurdles was to implement the weight sharing feature needed for Siamese networks. As I wasn't familiar enough with Caffe's framework, I first implemented a temporary solution, and then moved on to a more reliable solution integrated within the framework itself.

Temporary solution This solution consisted of independently doing the forward and backward computation of two independent networks A and B. The weights of the two networks were then updated by

— Adding net's B gradients to net A
— Updating net A
— Copying net's A parameters to net B to keep them synchronized

While this approach worked, it was far from optimal, and the needless copying of weights affected performance during training.

Final solution The final solution was made possible by changes in Caffe's architecture. I won't go into details about how it is done, as it reaches too deep within the framework to explain in a couple of lines. The basic principle is that the layers are now able to have "shared-blobs" that are updated together.

5.6 Loss Layer

The implementation of the loss layers gave me some trouble, mostly for theoretical reasons, as I had to figure out how the backpropagation was working, both theoretically, but also in Caffe. The implemented loss layers are a straightforward implementation of the theoretical derivations made in section 3.2.

One has to be careful when using the contrastive loss layer, as it can very easily diverge. Indeed, its role is to drive the data apart. As long as the learning rate is low, it does this in a controlled way. However, if the learning rate is set too high, it will tend to diverge.

6 Experiments

6.1 Extracting and testing feature descriptors using ImageNet CNN

The idea of this experiment was to see empirically how well descriptors extracted using the ImageNet pre-trained network would perform for the task of recognising image similarities. This network was trained to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into a 1000 different classes. Due to the high number of training classes, we expected to be able to extract feature descriptors that represent any scene fairly accurately at object level.

Thus, I developed a tool using Caffe and a public RGB-D dataset (Sturm et al., 2012) to extract feature descriptors from every image of the dataset, and perform simple distance comparison between a reference image and a sequence containing loop-closures (multiple views of the reference image). The features are extracted by using the neuron values from the last fully connected layer as a feature descriptor.

This tool allows to quickly visualize whether the feature descriptors extracted from the ImageNet CNN compared with a manually defined distance function (we tried L1 and L2 norm, and dot product) are efficient at detecting image similarity. As expected, we found that some degree of similarity was visible (see figure 12 for an example), yet not accurate enough to be used by itself.

This first experiment shows the flexibility of CNNs. This network was originally trained on an image-classification problem, on a completely different dataset. Yet, it was able to provide sufficiently accurate feature descriptors to notice loop-closures with a reasonable distance margin. This encourages us to think that a network specifically trained to look for image-similarity could provide state-of the art results for detecting loop-closures.

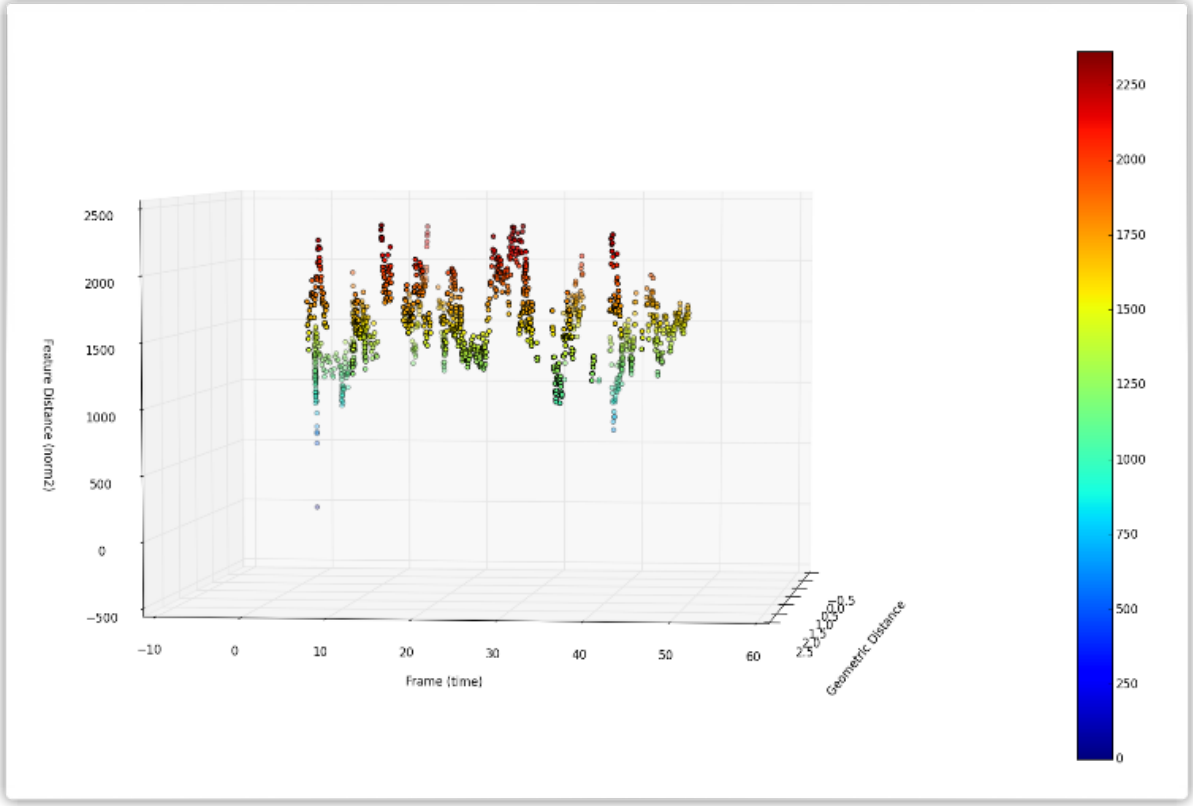


Figure 11 – Feature distance from the 1st frame of the freiburg1_room to all the other frames of the sequence, which feature loop-closures at 3s and 36s. The color coding represents the Euclidian distance between the feature vector and the reference vector. As expected, it can be seen that loop-closures coincide clearly with smaller Euclidian distances (displayed in blue).

6.2 Training a Siamese network from scratch

We created a Siamese network roughly based on the AlexNet design used for classification of the ImageNet challenge, as its structure is well suited to process and reduce image information. We tried training the network from scratch on our dataset. While the training results were better than random, it wasn't even close to the quality of the previous experiments done using only the network trained on the ImageNet classification challenge.

This was a big disappointment, as we expected the network to manage to learn some sense of similarity directly from the data. Apparently, we were too ambitious and overestimated the abilities of convolutional networks.

6.3 Training a Siamese network initialized from ImageNet's weights

Considering the relative quality of our initial experiment on ImageNet, and taking into account the apparent difficulty of training a Siamese network directly from data, we decided to simplify the learning task by pre-initializing the first layers of a Siamese network with weights from the already trained ImageNet network. The intuition is that having learnt to classify images quite reliably, the ImageNet network has acquired some ability to abstract and map high level information to a more compact and meaningful representation. The hope is to manage to train a Siamese network to fine-tune the weights and train additional layers to perform the comparison task at hand.

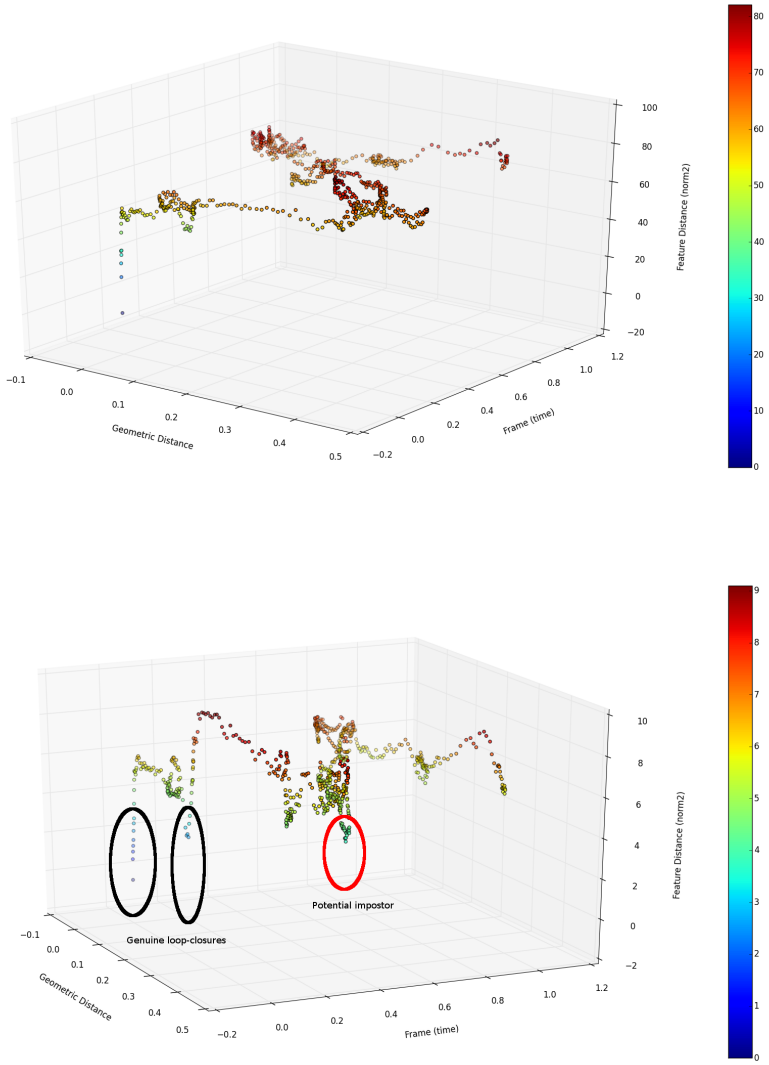


Figure 12 – Side by side comparison between distances obtained from ImageNet (top) descriptors and descriptors fine tuned by the Siamese network after 2000 iterations with a batch size of 64 images (bottom). Distances represent the euclidean distance between an image sampled at a place visited multiple times, and all other images of the freiburg1_360 sequence from the Freiburg dataset.

From Figure 12, we can empirically see a clear improvement over the imagenet version. The two main loop closures corresponding to the distances circled on the left. The only part where there could be a doubt about whether we found a loop-closure or not corresponds to the "potential impostor" circled on the right side. Unfortunately, observing the images (figure 13) leading to the bad loop closure detection, there doesn't seem to be any obvious reason for this misbehavior.

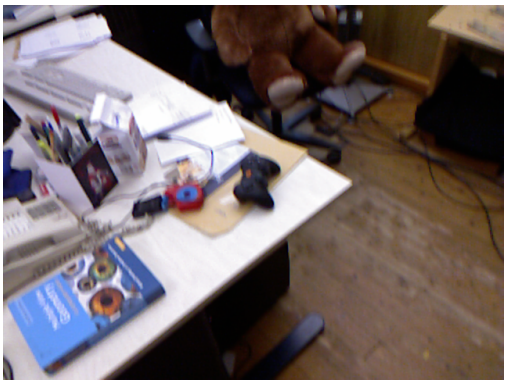
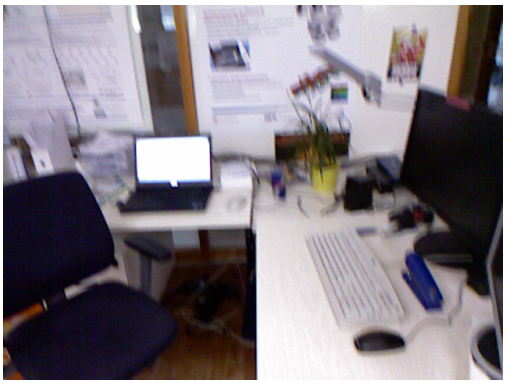



Figure 13 – Sample of images leading to bad loop closure detection

6.4 Conclusion and work in progress

Unfortunately, the experiment performed here didn't work as well as we hoped for. It seems that learning similarities between arbitrary images reliably is a rather complex task, and cannot be achieved in a straightforward manner. Some experiments with a bigger training set and various network parameters are currently running and thus aren't part of this report. However, we do not expect a significant improvement of the results.

Previous successful application of Siamese networks were concerned about more specific task. One of the most notable ones is the work on face recognition Chopra et al. (2005) that achieved good accuracy by training a Siamese neural networks on face-recognition datasets.

It seems that our use case have too much variability in the training data to reach a highly discriminate state as necessary for efficient loop-closure detection.

7 About the Internship

I chose to do my Internship in Munich for two main reasons: one was the opportunity to once again further my experience abroad, the other was the quality of the Computer Vision Group at the Technische Universität München.

7.1 Computer Vision Group at TUM

This group is working on a wide range of topics, all related to computer vision. Projects include: image denoising, shape analysis, 3D reconstruction from multiple cameras, 3D SLAM, scene flow, autonomous navigation, flying quadcopters... This is such a thriving place to discover what is currently being done in the field of computer vision, and have first-hand experience with all of these projects. The frequent seminars are also an interesting way to learn about what is being done elsewhere or within the group.

They also have very interesting online courses, one of which, taugh by Prof. Daniel Cremers concerns variational methods, and resumes quite nicely the math behind most of the project led by his group. Some other online courses on machine learning and flying quadcopters are also available.

7.2 Choice of the project

After hearing about my interest in exploring the capabilities of neural networks, Prof. Daniel Cremers suggested exploring the use of convolutional neural networks for which there seems to be a growing interest within the lab.

Thus, I was charged with the project of trying to use neural networks to explore a new way to achieve loop-closure detection. Unfortunately, nobody in the chair knew much about the topic, so I was led to do a lot of preliminary research to find out the possible ways to go about it.

7.3 Supervision

The project started with great supervision from Jürgen Sturm. Unfortunately he had to leave the chair, leaving me to work mostly on my own. I could occasionally ask for the advice of Christian Kerl, but due to the nature of the project, I was mostly fending for myself. I think the project would have progressed better with better supervision, and it would have been very helpful to have someone knowledgeable about the topic around.

7.4 Difficulties encountered

Coming to grasp with Caffe's framework was no easy task. The existing code-base is consequent, and is currently lacking good documentation. Thus, I had to figure out mostly by myself how it all fitted together, and how I could go about extending it for my own purposes. In the end, I managed to overcome these difficulties, and greatly contribute to the framework. Also, working in the context of neural networks with millions of interconnected weights makes it really hard to debug work. Hopefully, the unit test framework was of great help to ensure the correctness of the code.

But my main struggle came from training and testing the network itself. I have always been used to developing algorithm for which you can see the result straight away, and figure out the problems in a direct way. Training a neural network takes a considerable amount of time, and figuring out the reasons for failure is a complex task.

7.5 Learnt skills

This project led me to learn a lot of new technologies, skills and methodology.

CUDA The intensive use of CUDA in Caffe's framework gave me a very good outlook on how to structure a big program to extensively use GPU computing. My previous projects had only been limited to have some very specific functions on the GPU, but never had the whole architecture built around it. I knew nothing of CUDA (though I knew OpenCL) before starting the internship, so I was led to learn CUDA programming, and discover the main computation libraries that one can use (BLAS).

Coding rigor Caffe's framework enforced a strict set of coding rules. I grew to like the clear coding style and guidelines, and will definitely apply it on my later projects.

Git for large projects I had never had the opportunity to use git on such a large scale. It was definitely interesting to see how everything was organized to move the project forward as a team effort.

Paper reviewing I was included by my supervisor in the reviewing process of a paper on loop-closure detection. This was an interesting experience, and will be useful for future paper reviews that I will have to do during my future academic work.

Neural Networks Through all the preliminary research and hands on development of neural networks, I know have a good overview of how it works. From having to modify the backpropagation algorithm for weight sharing, I now have a much clearer understanding of backpropagation.

Conclusion

This internship has led me to do a lot of background research on various aspects of computer vision and machine learning, mainly SLAM and Convolutional Neural Networks. I got the opportunity to define everything about my project, from it's subject to finding out ways of solving it, and choosing how to implement them. Thus, I was truly led to experience all phases of a research project.

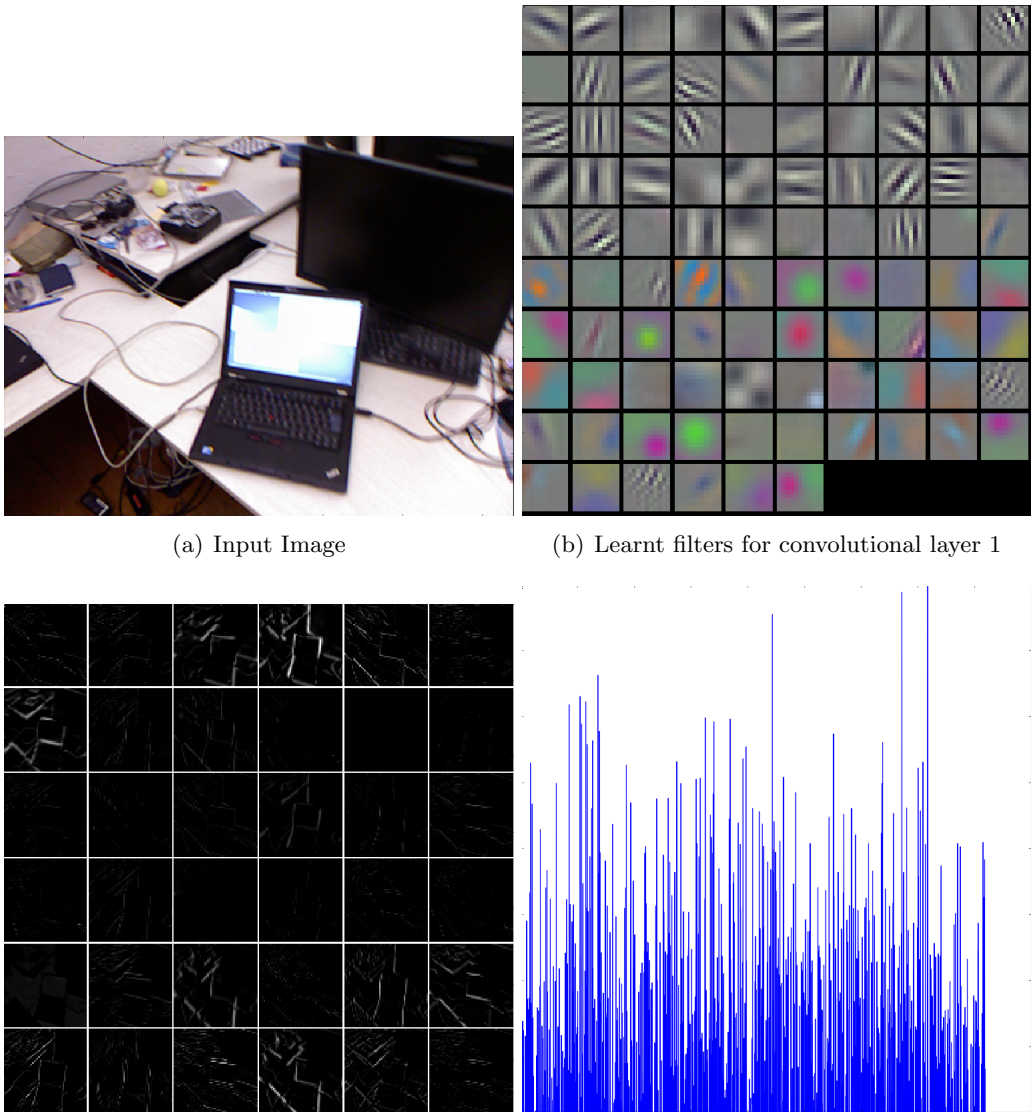
Unfortunately the results achieved by the Siamese network architecture didn't live up to our expectations. As such, this work cannot be directly used in the original context of loop-closure detection. However, the improvements made to the Caffe architecture will make it much easier for future projects to experiment with Siamese networks. Instead of spending time on implementing the network architecture, it is now possible to simply define and configure Siamese network. This should encourage there use and foster experimentation, which could lead to interesting uses of Siamese networks in the future.

8 Annex

8.1 Visualizing the internal state of CNNs

By using Caffe, I developped a simple python tool to visualize the internal state of the network. This is based on Yangqing et al. (2013); Zeiler and Fergus (2013).

Here are some images generated from the internal state of the ImageNet CNN, applied on an image of the RGBD dataset.



(a) Input Image

(b) Learnt filters for convolutional layer 1

(c) Neuron activation of convolutional layer 1

(d) Feature descriptor extracted from a fully-connected layer

Figure 14 – Visualisation of the internal state of the first layer of ImageNet CNN, along with a feature vector generated by the second to last fully connected layer

8.2 Complete Siamese Network

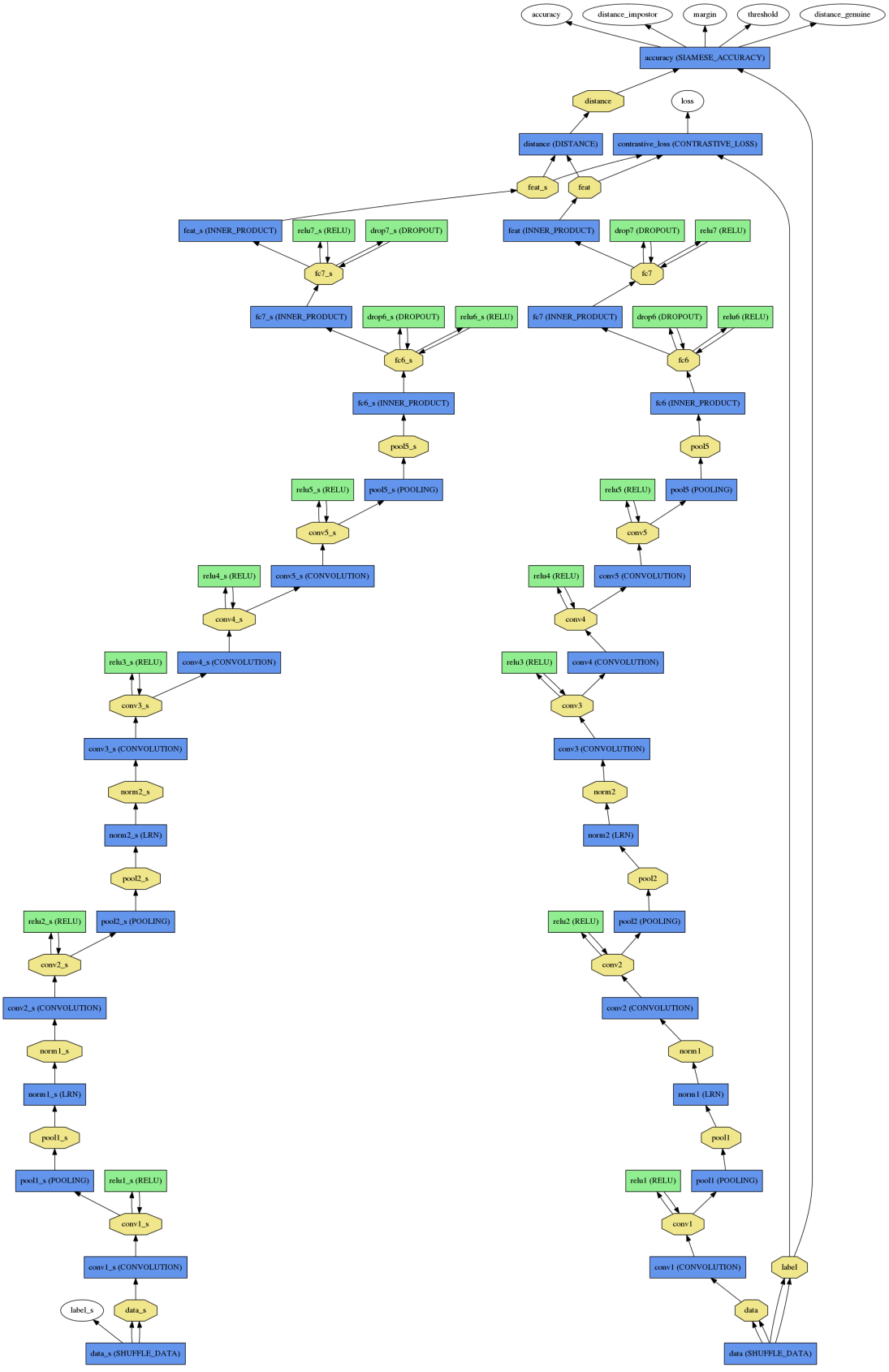


Figure 15 – Directly Acyclic Graph of the entire Siamese Network used for training.

Figure 15 represents the complete Siamese network used for our experiments. Most layers are identical to the Imagnet network, as it allows us to use the pretrained weights to initialize the network to a less random state and drive the siamese optimization from here.

The data layer feeds data to two sides of the network: the network and its Siamese duplicate. During training, either the contrastive loss function or the hinge loss function is used to drive the minimization process. The distance and accuracy layers are only used for the test phase to compute the average accuracy of the network. The shared weights aren't represented on this graph for reading clarity. If they were, they would link all the yellow hexagons named "layer" with their siamese counterpart "layer". The green boxes (ReLU and Dropout layers) are operations that can be done in place, without any memory overhead. Thus they are very fast and do not notably affect the computation time of the network.

The role of all layers from the bottom to the fifth convolutional layer is to visually process the data through learnt filter in order to extract higher level information, allowing for the rest of the network to perform its comparison task through the use of fully connected layers.

This might seem like a daunting task to generate such an impressive layer graph, but it is in fact made easy by Caffe's network definition system. All that one has to do (after the non-trivial task of implementing the missing layers), is to define a configuration file describing the layers and their parameters.

References

C. Kerl, J. Sturm, and D. Cremers. Robust odometry estimation for rgb-d cameras. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 3748–3754, May 2013a. doi: 10.1109/ICRA.2013.6631104.

C. Kerl, J. Sturm, and D. Cremers. Dense visual slam for rgb-d cameras. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 2100–2106, Nov 2013b. doi: 10.1109/IROS.2013.6696650.

A. Angeli, D. Filliat, S. Doncieux, and J.-A. Meyer. Fast and incremental method for loop-closure detection using bags of visual words. *Robotics, IEEE Transactions on*, 24(5):1027–1037, Oct 2008. ISSN 1552-3098. doi: 10.1109/TRO.2008.2004514.

M. Cummins and P. Newman. Invited Applications Paper FAB-MAP: Appearance-Based Place Recognition and Mapping using a Learned Visual Vocabulary Model. In *27th Intl Conf. on Machine Learning (ICML2010)*, 2010.

D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2161–2168, 2006. doi: 10.1109/CVPR.2006.264.

Peter Henry, Michael Krainin, Evan Herbst, Xiaofeng Ren, and Dieter Fox. Rgb-d mapping: Using depth cameras for dense 3d modeling of indoor environments. In Oussama Khatib, Vijay Kumar, and Gaurav Sukhatme, editors, *Experimental Robotics*, volume 79 of *Springer Tracts in Advanced Robotics*, pages 477–491. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-28571-4. doi: 10.1007/978-3-642-28572-1_33. URL `http://dx.doi.org/10.1007/978-3-642-28572-1_33`.

J. Stuckler and S. Behnke. Integrating depth and color cues for dense multi-resolution scene mapping using rgb-d cameras. In *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2012 IEEE Conference on*, pages 162–167, Sept 2012. doi: 10.1109/MFI.2012.6343050.

Michael J Milford and Gordon Fraser Wyeth. Seqslam: Visual route-based navigation for sunny summer days and stormy winter nights. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1643–1649. IEEE, 2012.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, volume 1, page 4, 2012.

Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 539–546 vol. 1, June 2005. doi: 10.1109/CVPR.2005.202.

Yunchao Gong, Sanjiv Kumar, Henry A Rowley, and Svetlana Lazebnik. Learning binary codes for high-dimensional data using bilinear projections. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 484–491. IEEE, 2013.

Mohammad Norouzi, David J Fleet, and Ruslan Salakhutdinov. Hamming distance metric learning. In *NIPS*, pages 1070–1078, 2012.

Yangqing Jia. Caffe: An open source convolutional architecture for fast feature embedding. `http://caffe.berkeleyvision.org/`, 2013.

J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

Ben Glocker, Shahram Izadi, Jamie Shotton, and Antonio Criminisi. Real-time rgb-d camera relocalization. In *International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE, October 2013. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=196003`.

Kevin Lai, Liefeng Bo, and Dieter Fox. Unsupervised feature learning for 3d scene labeling.

Yangqing, Jeff Vinyals, Donahue, Oriol Jia, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*, 2013.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional neural networks. *arXiv preprint arXiv:1311.2901*, 2013.