Technische Universität München
Fakultät für Informatik
PD Dr. Slobodan Ilic

WS 17/18
Exercise Sheet 3
January 18, 2018

Exercises in Tracking & Detection

3D pose estimation and object instance recognition are very well known problems in computer vision. They have various applications in the fields of robotics and augmented reality. Despite of their popularity, the most common pose estimation methods use a single classifier per object, making their complexity grow linearly with the number of objects for which the pose has to be estimated. For real-world purposes, however, scalable methods that work with many different objects are often desirable. In this exercise, you will implement a full pipeline for instance classification and pose estimation using convolutional neural networks or CNNs. The implementation will closely follow the approach of Wohlhart et al. [1] The authors tackle both pose estimation and object instance recognition of already-detected objects simultaneously by learning a discriminative feature space using CNNs. Particularly, given a single image patch containing an already-detected object in the center surrounded with the cluttered background, the descriptor CNN is used to map this patch to a lower-dimensional manifold of the computed descriptors. This manifold preserves two important properties: the large Euclidean distance between the descriptors of dissimilar objects, and the distance between the descriptors of the objects from the same class is relative to their poses. Once the mapping is learned, efficient and scalable nearest neighbor search methods can be applied on the descriptor space to retrieve the closest neighbors for which the poses and identities are known. This allows us to efficiently handle a large number of objects together with their view poses, resolving the scalability issue. We will use Python as a programming language and TensorFlow for setting up the network.

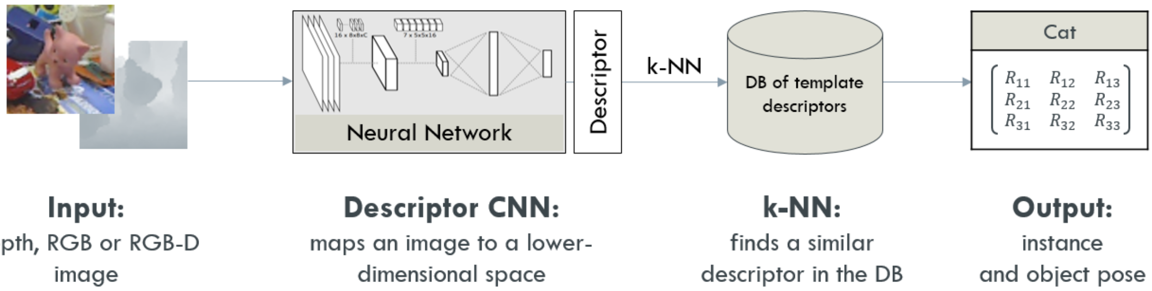

Neural Network
Descriptor
k-NN
DB of template descriptors
Cat
R_{11} R_{12} R_{13}
R_{21} R_{22} R_{23}
R_{31} R_{32} R_{33}
Input:
depth, RGB or RGB-D image
Descriptor CNN:
maps an image to a lower-dimensional space
k-NN:
finds a similar descriptor in the DB
Output:
instance and object pose

Figure 1: Working Principle

<u>Exercise 0</u> **Setting Up The Environment** As a first step, you will need to set up the environment. We will use conda package manager for easier and more convenient usage and setting up virtual environments. To acomplish this project you will only require the OpenCV and TensorFlow packages.

- Install anaconda or miniconda: `https://anaconda.org`
- Set up a virtual environment with all the needed dependencies using: *conda env create -n ex3 -f env.yml*. Alternatively you can create your own environment.

[1] P. Wohlhart and V. Lepetit. Learning descriptors for object recognition and 3d pose estimation.

- In order to activate the newly created environment use: *activate ex3* for Windows or *source activate ex3* for Linux.

In order to have better debugging capabilities, we suggest you to use the PyCharm IDE while working on your project: `https://www.jetbrains.com/pycharm/download/` As a student you can also get a professional distribution of it: `https://www.jetbrains.com/student/`. To configure PyCharm to work with your newly created conda environment, simply specify the appropriate interpreter under project settings, e.g. `<Userpath>\AppData\Local\Continuum\Anaconda3\envs\ex3\python.exe` or `<usepath>/anaconda3/envs/ex3/bin/python`.

<u>Exercise 1</u> Data Preparation

As a first task, you will need to prepare the data for training and testing. The data consists of 64x64 sized RGB patches of 5 different objects taken under different camera poses. Each image, therefore, has a corresponding pose stored next to it (saved in `poses.txt`). You need to read the images together with the poses (stored in quaternions) and construct 3 datasets: the training set S_{train} (test subset of the `real` folder and `fine` folder), test set S_{test} (test subset of the `real` folder) and database set S_{db} ("coarse" folder). The training subset indices of the real data is stored in the `training_split.txt` file, the remaining indices belong to the test subset. RGB channels have to be normalized to the zero mean and unit variance.

(a) Template set sampling

(b) Training set sampling

Figure 2: Different sampling types: each vertex represents a camera position from which the object is rendered.

The template set S_{db} contains only synthetic samples with the renderings coming from the coarse sampling (Fig. 2a). It is used in both training (to form triplets) and test (as a database for the nearest neighbor search) phases. The samples of S_{db} define a search database on which the nearest neighbor search is later performed. This is the main reason for coarse sampling: We want to minimize the size of the search database for faster retrieval.

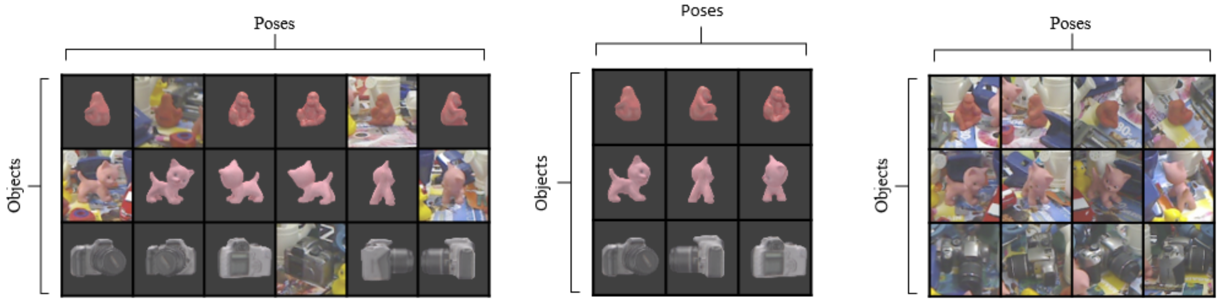
The training set S_{train} (Fig. 3a) consists of a mix of synthetic and real data. The synthetic data represent samples coming from the renderings defined by the fine sampling (Fig. 3b). Approximately 40% of the real data is added to the training set. The rest of the real samples are stored in the test set S_{test} (Fig. 3c), which is used to estimate the performance of the algorithm.

Once datasets are constructed, you can proceed with the implementation of the batch generator. The batch generator should be able to construct batches of triplets that are later fed to the network. Each triplet consists of 3 samples (see Fig. 5a: anchor, puller, and pusher. Anchor is chosen randomly from the training set S_{train}. Puller is the most similar (quaternion-wise) to anchor sample of the same object taken from the db set S_{db}. In order to find the puller for each sample of S_{train} use the quaternion angular metric given in Eq. 1. Finally, there are 2 types of pushers: it can either be the same object but a different from puller pose or a randomly chosen different object. Pushers are also drawn from S_{db}.

Therefore, a batch should have the following structure: $x_a^1, x_+^1, x_-^1, x_a^2, x_+^2, x_-^2, ..., x_a^n, x_+^n, x_-^n$, and should be divisible by 3. When implemented, you can either use it to generate the batches online, i.e. during the training, or offline, i.e. by storing them prior to the training.

$$\theta(q_a, q_+) = 2 \arccos(|q_a \cdot q_+|), \tag{1}$$

where q_a, q_+ are quaternions of anchor and puller respectively.



(a) Training set S_{train} (b) Template set S_{db} (c) Test set S_{test}

Figure 3: Datasets: The training set S_{train} consists of both real and synthetic (fine sampling); the test set S_{test} consists of the real data not used for the training set S_{train}.

TLDR:

- Download **dataset.zip** from the course webpage. Load the images next to their poses and store them as S_{train}, S_{db}, and S_{test}, as shown in Fig 3.
- Implement a batch generator forming triplet batches using the quaternion similarity metric given in Eq. 1.

Exercise 2 Convolutional Neural Network and Loss Function

When the S_{train} and S_{db} sets are generated, we have all the data needed to start the training. The next step is to construct the actual neural network and define the loss function. The network architecture (Fig. 4) is fairly simple and closely follows that one of the famous LeNet. it consists of 2 convolutional layers, each followed by ReLu activation functions and a 2x2 max pooling layer, and 2 fully connected layers. The output descriptor size set to be 16 for all the experiments. An example of constructing a similar network in TensorFlow as well as the layer descriptions can be found here: `https://www.tensorflow.org/tutorials/layers`. When you are done with constructing the network, you can use TensorBoard (`https://www.tensorflow.org/get_started/graph_viz`) to ensure that your network fits to the underlined specification.

The loss function is defined as a sum of two separate loss terms $L_{triplets}$ and L_{pairs}:

$$L = L_{triplets} + L_{pairs}. \tag{2}$$

The first addend $L_{triplets}$ is a loss defined over a set T of triplets, where a triplet is a group of samples (s_a, s_+, s_-) selected such that s_a and s_+ always come from the same object under a similar pose, and s_- comes from either a different object or the same object under a less similar pose (Fig. 5a). By minimizing $L_{triplets}$, one enforces two important properties that we are trying to achieve, namely: maximizing the Euclidean distance between descriptors from two different objects and setting the Euclidean distance between descriptors from the same object so that it is representative of the similarity between their poses. In our terminology,

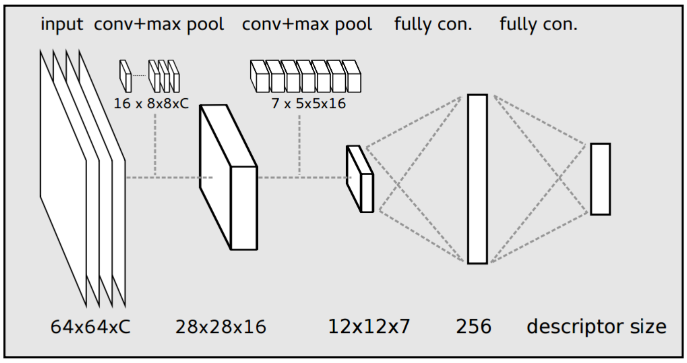
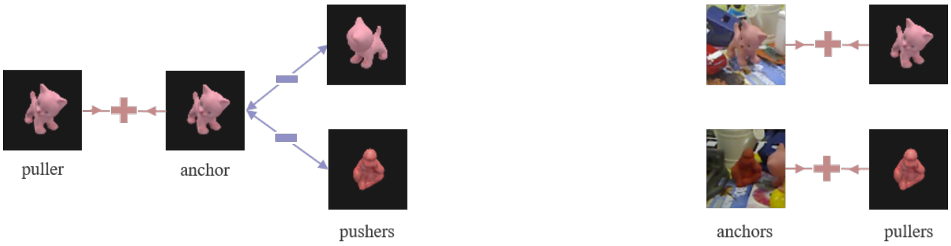


Figure 4: CNN Architecture

we call s_a an anchor, s_+ a positive sample or a puller, and s_- a negative sample or a pusher. The triplet loss component has the following form:

$$L_{triplets} = \sum_{(s_a,s_+,s_-)\in T} max\left(0, 1 - \frac{||f(x_a) - f(x_-)||_2^2}{||f(x_a) - f(x_+)||_2^2 + m}\right), \tag{3}$$

where x is the input image of a certain sample, $f(x)$ is the output of the neural network given the input image, and m is the margin for classification and,which sets the minimum ratio for the Euclidean distance of the similar and dissimilar pairs of samples. The term m should be set to be 0.01 by default for all the experiments, but you can also try to tune it.



(a) Triplet-wise term

(b) Pair-wise term

Figure 5: CNN input format: triplets are used to learn a well-separated manifold, whereas pairs make the mapping invariant to various imaging conditions.

Hint: For the prescribed batch structure $||f(x_a) - f(x_+)||$ and $||f(x_a) - f(x_-)||$ can be computed as follows:

```
diff_pos = feats[0:batch_size:3] - feats[1:batch_size:3]
diff_neg = feats[0:batch_size:3] - feats[2:batch_size:3]
```

The second addend L_{pairs} is a pair-wise term. It is defined over a set P of sample pairs (s_a, s_+). Samples within a single pair come from the same object under either a very similar pose or the same pose but with different imaging conditions. The pair-wise loss has a function of domain adaptation, i.e. we map descriptors of real and synthetic representations to be as close as possible to each other:

$$L_{pairs} = \sum_{(s_a,s_+)\in P} ||f(x_a) - f(x_+)||_2^2. \tag{4}$$

To make it simpler, the pair-wise term is computed using the same pairs that are used for the triplet term.

The last component we need to start the training is the actual solver to update the network parameters. The solver we are going to use is the Adaptive Moment Estimation (Adam), which computes the learning rates adaptively for each parameter.

TLDR:

- Construct a CNN following the architecture shown in Fig. 4
- Implement a pair-wise term as shown in Eq. 4
- Implement a triplet-wise term as shown in Eq. 3
- Combine 2 loss terms to get the final loss as shown in Eq. 2
- Set an Adam solver for the network parameter update.

Exercise 3 Evaluation and Nearest Neighbor Search

Now we have everything ready to start the process of training. However, we cannot evaluate how good our results are. Once the mapping is learned, efficient and scalable nearest neighbor search methods can be applied on the descriptor space for each sample of S_{test} to retrieve the closest neighbors from S_{db} for which the poses and identities are known. This allows us to efficiently handle a large number of objects together with their view poses, resolving the scalability issue. In this case, we are going to use the brute force matcher provided by OpenCV (however, you are welcome to use the implementation of your choice). Use Euclidean distance as a metric.

Use the network to get the descriptors of S_{db}. Next, do the same for S_{test}. Once done, find the nearest neighbor in the S_{db} for each test descriptor. If the found descriptor belongs to the same class, then calculate the angular difference and store it. If the class is wrong, do not store the angle. Based on the stored values, plot a histogram with the following tolerance angles: <10°, <20°, <40°, and <180°, e.g. if the actual angular difference is equal to 15°, it should go to bins 20°, 40°, and 180°. Since 180° is the maximal possible angular distance between the objects, it also corresponds to the classification rate.

(a) PCA

(b) t-SNE

Figure 6: Example Feature Visualizations

Now you are all set to start the training. Output the loss value at each 10th iteration and store it (preferably use TensorBoard). At each 1000th iteration compute the histogram and store the results.

TLDR:

- Implement (or use a ready implementation of) the nearest neighbor search using Euclidean distance as a metric.

- Find the nearest neighbor in the db set for each test descriptor. If the found descriptor belongs to the same class, then calculate the angular difference and store it. If the class is wrong, do not store the angle. Based on the stored values, plot a histogram with the following tolerance angles: <10°, <20°, <40°, and <180°.

- Perform the training. Store intermediate results: loss - each 10th iteration, histogram - each 1000th iteration.

Bonus Task Additional Visualizations

In order to get bonus points for this project you can do 2 following things:

- Visualize the feature descriptors of S_{test} in 3D as shown in Fig. 6. PCA or t-SNE algorithms are used to map 16D descriptors to 3D. You do not have to implement them. Instead, you should use the embedding visualization feature provided by TensorBoard, which gives these projections for granted.

- Plot confusion matrix using the seaborn package `https://seaborn.pydata.org/generated/seaborn.heatmap.html`. Confusion matrix is a good way to visualize the classification accuracy of our algorithm. It is not only going to show the actual classification rate for each object, but also explicitly demonstrate which objects cause the most confusion.