Robot Positioning Using an Omnidirectional Camera

Lab Exercise 5 for Beeldverwerken

Informatics Institute
University of Amsterdam
The Netherlands

May 14, 2018

1 Robot Positioning

The apparent goal of this lab exercise is to develop a positioning system for a robot with an omnidirectional camera. It is going to do so by matching its present view with a library of pre-recorded views at known locations, in an efficient manner. The actual goal of the exercise is to become familiar with Principal Component Analysis (PCA), a based technique for characterizing large data sets by their essence (in a linear algebra manner).

In Figure 1, a picture from an omnidirectional camera is shown. This picture is a 360 degree image (or panorama image), with the robot 'centered' in the image. Looking closely one can identify floor, walls doors and other pieces (of the first floor of Kruislaan 403, our previous building).

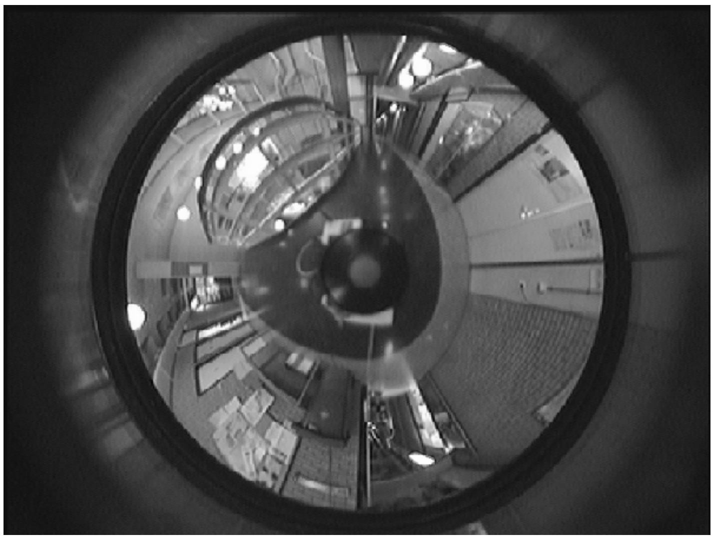

Figure 1: **Omnidirectional image.** Image from our omnidirectional robot with position $(616, 7)$.

The dataset consists of 550 images of the first floor of Kruislaan 403 taken with an omnidirectional camera. All images have been annotated with a 2D position.

To prevent lengthy processing, we have prepared the dataset for you by converting the images to grayscale, and we have reduced width and height of the images by a factor 4. All images have a resolution of 150 by 112 and are oriented in the same direction, giving a total of 16800 pixels per image. The processed images are available in the file `omni.mat`. Using the command `load omni` you will create a variable named `images` in your Matlab workspace. With `images{i}.img` you will access the image data and with `images{i}.position` you will access the 2D position of the robot while capturing this image. Clearly, `i` ranges from 1 through 550.

2 Principal Component Analysis

We are going to identify the location of an unknown image by finding the best match among the known images. That requires a distance meausre between images. On BB a Chapter by Trucco introduces a similarity measure between images (*image correlation*) on page 264.

Use the theory questions below as pointers to write a theoretical section devoted to image correlation. **(10 points)**

Theory Questions

2.1. Show under what conditions the sum of squared differences between pixels of two images I_1 and I_2 reduces to a correlation-based similarity measure.

2.2. Now check some of the images to see whether those conditions hold in our data set.

2.3. By representating images as vectors, simply concatenating their pixels, the correlation becomes a dot product. Do you see that?

2.4. To apply this trick, we first need to convert our 150 by 112 input images into one long feature vector with 16800 elements - the Matlab command `reshape` comes in handy here.

Computing the dot product of such large vectors whenever we want to do a comparison is costly. But we do not have to compute it *exactly*: it is possible to preprocess the data set to home in on the most important parts of the distance computation, and to then perform it by a dot product between much smaller vectors (of 20 dimensions or so, you will have to decide later).

Read Trucco section 10.4 for the technique of Appearance Based Identification through once, to get the flavor of how the SVD helps when used as *Principal Component Analysis* (PCA), to reduce the essential dimensionality of determining the similarity of images. Ignore the Learning aspects for now.

A more extended tutorial explanation of the PCA technique, by Shlens, can be found on BB. It has a rather lame motivating example, Trucco's example of appearance based identification is much more *a propos* for us. But Shlens builds up the story rather well, so we'll use it.[1]

Use the theory questions below as pointers to write a theoretical section on data representation as vectors and matrices for the PCA algorithm. **(10 points)**

Theory Questions

2.1. Note the notation in Shlens, for instance in (3): he sometimes uses row vectors rather than column vectors, and therefore get transposes in unusual places. I find it more

[1]For once, the Wikipedia page on PCA is unfortunately not very well written or structured in its mathematical parts, though some of the texts are OK.

convenient to use column vectors consistently. Check what he does for X: is it the same as Trucco, or transposed relative to it?

2.2. The introduction of the covariance as a 'straight-forward generalization' of variance is a bit lazy, and it does not give any intuition. In our Lecture Notes Section 4.1.2, we show the geometrical meaning: the covariance matrix C allows you to measure the covariance in a direction $\mathbf{r}$ as $\mathbf{r} \cdot (C\mathbf{r})$, and that definition determines what the formula should be. Make sure you understand that derivation.

2.3. In the narrative, Shlens pretends for a while that there are some choices to be made for reasonable definitions of 'principal'. He later finds out that some of these choices are not independent: as soon as you talk about the main directions with extreme variance, the symmetry of the covariance matrix *automatically* makes them orthogonal. You learned about that in linear algebra, and Section 3 in Shlens' Appendix gives the pointer to the crucial terminology. Look it up in Bretscher Chapter 8 (or some other linear algebra text) if you forgot!

2.4. If Shlens explanation of the SVD works for you, enjoy it. What I miss is the geometrical meaning, sketched in Figure 2: *any linear transformation can be written as a rotation/reflection back (V^T) to a standard coordinate frame, then a stretch along each of the coordinate axes (D or Σ), then a rotation/reflection (U) to the 'output frame'.* See also Bretscher Section 8.3 Figure 4.

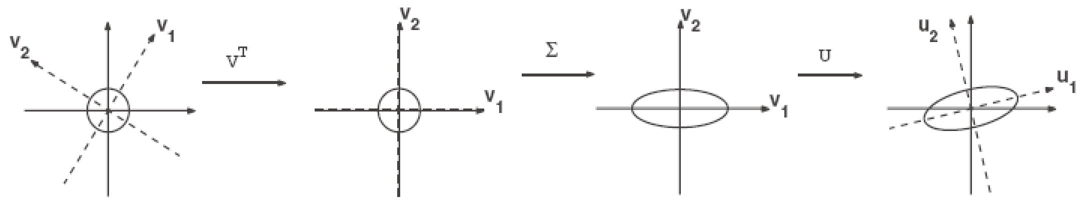

Figure 2: *The geometrical meaning of the terms of the SVD.*

With all of that basically understood, let us start the implementation.

⋆ Split the data into two parts. A 'training' set of 300 images and a test set.

⋆ **(30 points)** Apply PCA to our dataset to reduce dimensionality of the data to d, making d a configureable parameter.

Hint Be sure to apply PCA only to the training set, otherwise you are learning on your test set.

Hint Be sure to subtract the mean from your data when applying PCA: PCA should be applied to data with zero mean. This is a common implementation mistake.

Hint As a suggestion, place your data in a matrix X with size n by d with n the number of datapoints (300 for the training set) and d the number of data dimensions (16800 for our vectorized images).

Hint Using the Matlab function `eigs` can be used to extract the first m eigenvectors from a matrix (see Matlab help). However, be sure to understand how PCA works before blindly applying this function to something!

Hint Property 5 in Trucco, Section A.6, is often important to make the PCA approach efficient. In this case, what are the relevant dimensions, and which method are you going to use?

Hint The Matlab function `repmat` can be used to tile and replicate vectors.

- ⋆ **(5 points)** Plot the first 9 PCA vectors as images.
- **(10 points)** Plot the eigenvalues of the first 50 PCA components. How many PCA components will you use?
- Using the PCA basis, compute the coordinates of all images on this basis.
- Take an image from the set, and find the next best match using the image similarity (the best match is of course the image itself!).
- **(10 points)** Compare the speedup relative to the naive implementation of the image differences without the PCA.

Remember the following: using PCA will yield '*the most expressive features*', not necessarily '*the most discriminating features*'.

3 Positioning with Nearest Neighbour

Now that we have reduced the dimensionality of our data to something tractable, we can apply supervised learning approaches[2]. We will use one of the simplest approaches available: Nearest Neighbour. You should use images 1 through 300 as your training set and the others as your test set. The order of the images has been sufficiently randomized.

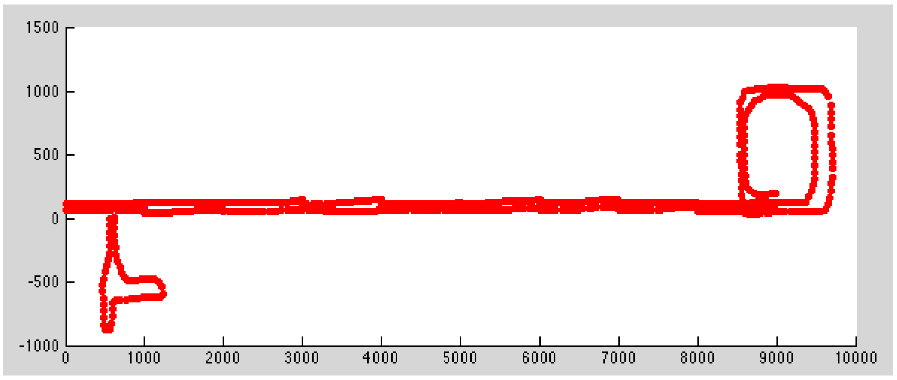

Figure 3: The location of each of the images.

Matlab's **`struct`** and **`cell`** syntax is not always intuitive (quite the opposite), to make your life a little easier, use the following code:

```
imagestruct = [images{:}];
positions = vertcat(imagestruct.position)
```

[2]Note that our problem is *not* a classification problem (because we do not have a fixed set of classes), so some learning algorithms are not useable in their standard form.

- ⋆ **(40 points)** Project both the training and test set onto the PCA components, yielding the reduced representation of the images.
- ⋆ For each image in the test set, find the closest image in the training set.
- ⋆ Evaluate your accuracy. Here we have to take care as to what to accept as a 'correct' position for an image in the test set. The images in the test set were not in the training set, and as such the positions can never be exactly correct because we use the labels of the closest neighbour in the training set. Knowing that the original robot took images about every 50 units, we will say that the position is 'correct' if the Euclidean distance is less than 150 units.

- **(5 points)** Would results improve if we were to remove the nearly black borders of our images? Why?
- **(5 points)** Experiment with the number of PCA components used when positioning, and report results.
- **(5 points)** What if we were to leave out the PCA step? The Nearest Neighbour algorithm is able to handle all 16800 dimensions computationally. But will it work on the raw images, or suffer from *the curse of dimensionality*? Try it and report positioning accuracy for this naive algorithm.

MAX POINTS: 130