Image Mosaicking
Study Week Fascinating Informatics

Adrian Wälchli
Computer Vision Group
University of Bern

August 30, 2018

Contents

Figure 1: Example of a image mosaic.

1 Introduction

1.1 Welcome to the Institute of Computer Science

The Institute of Computer Science, Prof. Paolo Favaro and I welcome all participants of the study week "Fascinating Informatics". As every year, we prepare a unique student project that promotes computer science with a glimpse into the research at the University of Bern. Our institute consists of five research groups: Computer Vision Group (CVG), Communications and Distributed Systems (CDS), Logic and Theory Group (LTG), Software Composition Group (SCG) and Computer Graphics Group (CGG). I am a first-year PhD student from the Computer Vision Group supervised by Prof. Paolo Favaro. I will be the tutor for your group project and it is my job to guide you and answer any questions you have or help with problems. I am very much looking forward getting to know you and work with you. It is the first time I design a project like this for the study week, so I am curious to hear your feedback!

1.2 Image Mosaicking

In this project, you will develop an algorithm that intelligently selects and arranges small image patches to form a mosaic of a given target image. An example of this is shown in figure 1. The target image in this example is the lion, and the image patches are a random collection of images from the internet down-scaled to a smaller resolution. In terms of content, these small patches have nothing in common with the target image. In order to create a mosaic of an image, one has to define a similarity measure to be able to compare an arbitrary image patch with a region in the target image. Although humans are very good at judging images by their content and appearance, you can imagine that it would still take a lot of time for a human to manually assemble a mosaic like the one shown in figure 1. In this project, you will learn how a computer can "learn" to compare image patches and automatically create a mosaic for any image within seconds.

1.3 Prerequisites

You will use Python as a programming language. It is of course beneficial to have little experience with Python, or any other language, but it is not a requirement. Since Python is very beginner friendly, you will be able to pick it up very fast. And you will like it! And in any case, your tutor will help you to get prepared for the programming.

We will provide at least one computer for you to work with, but it is good to have your personal notebook with you for researching, taking notes etc.

1.4 Learning Outcome

After successful completion of this one-week project, you will

- have a basic understanding of Machine Learning and Computer Vision,
- know how to implement and/or apply algorithms in Machine Learning and Computer Vision,
- have acquired skills for scientific Python programming,
- know how to write a scientific report and create a poster summarizing your research and results,
- get a better picture of what academic research looks like,

and hopefully be proud of your work!

2 Preparation

2.1 Working with the Terminal

2.2 Python

2.3 Tips and Tricks

3 Theory

3.1 Understanding the Computer Vision Problem

In the context of computer science, an image is simply an array of numbers where each number represents the intensity of light shown in one pixel. Such an image is usually taken with a digital camera that – for a short period of time – exposes its sensor to the light passing through the aperture. Depending on the camera, a small amount of low-level processing (brightness adjustments, sharpening, white balance, etc.) is applied to the pixel values read from the sensor. However, this type of processing does not require any knowledge of the contents in the image. What if we want to do more? Here are some examples of Computer Vision problems that are too complex to solve just by "looking" at the raw pixel data:

- Detecting a face in an image
- Solving a jigsaw puzzle
- Generating a caption for an image
- Removing the background in an image (semantic segmentation)
- Identifying and classifying a tumor in an x-ray image

There are many more examples of problems like these that are not that hard for a human to solve, but humans are slow and cannot do it reliably sometimes.

Figure 2: Left: Target image. Middle: Algorithm selects patches randomly. Right: Algorithm uses similarity measure to find good patches.

3.2 Datasets

3.3 Image Mosaicking

The problem of Image Mosaicking can be described as follows. We are given a large dataset of small image patches (say 32×32 pixels) and an arbitrary user-supplied target image (say 1024×1024 pixels). The Mosaicking algorithm must select patches from the dataset and arrange them in a grid to form an output image of the same size as the target image, depicting the same object or scene. Figure 2 shows an example of a failing algorithm that randomly chooses image patches, and a working algorithm that produces the desired result. Hence, we can see that Image Mosaicking requires a somewhat abstract understanding of the image patches in order to find patches that are visually similar to the target image's patches. In the next section, you will learn how exactly we define similarity and why it is hard measure the "distance" between two images.

3.4 Metrics

You are probably familiar with the Euclidean distance (even if you have never heard of Euclid) and know how to measure distances on a 2D map or even in 3D space. The concept of a distance or *metric* (as mathematicians call it) extends to other, high-dimensional spaces. In general, a metric d is defined by these four properties:

$$d(x, y) \geq 0 \tag{1}$$
$$d(x, y) = 0 \iff x = y \tag{2}$$
$$d(x, y) = d(y, x) \tag{3}$$
$$d(x, z) \leq (x, y) + d(y, z) \tag{4}$$

These properties need to hold for all x, y, z. Don't be afraid of these formulas, they are very intuitive! The first inequality says that a distance is non-negative. This certainly makes sense. The second equation simply means that the distance between two objects can only be zero if and only if the two objects are one and the same. This also makes sense (for example think of distances between cities in Switzerland). The third property says that a distance should be symmetric, that is, the distance measured from Bern to Zürich is the same as measured from Zürich to Bern. Finally, the third property is called the *triangle inequality*. It simply means that the distance is longer or equal if you take a detour, e.g. d(Bern, Zürich) $\leq d$(Bern, Basel) + d(Basel, Zürich).

Can we define a metric on images? Say we have one, then we would be able to quantitatively measure the difference between two images A and B. If $d(A, B)$ is a small number, we would say A and B are similar, and if the distance

Algorithm 1 Nearest Neighbor Search

```
for x in image do
  smallest ← ∞
  nearest ← ∅
  for y in dataset do
    if d(x, y) < smallest then
      smallest ← d(x, y)
      nearest ← y
    end if
  end for
  replace x in image with nearest
end for
```

is large, we say they are not similar. But how does one compute such a number for *any* pair of images (or patches in our project)? Is it even possible to do that?

3.5 A Simple Mosaicking Algorithm

As an example, let's look at a naive way of comparing two patches: we compute the average color of each patch and look at the difference.

$$d(x, y) = \|\bar{x} - \bar{y}\| = \left\| \frac{1}{N} \sum_{i=1}^{N} x_i - y_i \right\| \tag{5}$$

Here, $\bar{x}$ and $\bar{y}$ denote the average/mean of patch x and y respectively, and N is the number of pixels in each patch. Note that this is *not* a metric. Can you figure out which of the four properties of a metric are violated?

Having this function $d(x, y)$ defined, we can now formulate a simple algorithm to search through the dataset and find a good patch for every patch in the target image (see algorithm 1). This algorithm is called a *nearest neighbor* search. It finds the patch in the dataset that is closest to the target patch (according to the distance $d(x, y)$).

As explained later in the assignments, you will use nearest neighbor search and implement the average distance for your first Mosaicking algorithm in Assignment 1.

3.6 Clustering

Clustering in Machine Learning is used to understand structure in data. It can be easily explained with a real-life example: A supermarket. Let's say you want to go to Migros and buy pasta, but the brand or type of pasta does not matter. Will you go through every shelf and look at every product individually to check if it is pasta? No. You probably know that there is one shelf that has all the pasta, and only pasta. In data science, this is called a cluster. A cluster is a part

of the space (a shelf) that contains a certain type of data (a product category). Note the following important remarks about clustering.

- Every data point belongs to a cluster, and to one cluster only.
- Clustering only make sense if there is structure in the data (what would an "unstructured" supermarket look like?)
- Often it is unknown how many clusters there are in the data.

How could we use clustering for Image Mosaicking? The space of all images (or image patches in our case) has structure in it, and we could reason that there must be clusters for specific types of images. For example, it is reasonable to assume that it is possible to draw a boundary between daylight- and nighttime images (yielding two clusters), but it is not trivial to describe this boundary exactly. Another challenge is that we don't know exactly how many clusters there are or how many we *need*.

In the second assignment, you will get to use the **Kmeans** clustering algorithm. This algorithm can find clusters in our large collection of image patches. It returns the following two quantities.

- A list of *cluster centers*. The center is an image that represents the average for the images in that cluster.
- A *label* for each image that tells us to which cluster it belongs (a number from 1 to N).

4 Assignments

4.1 Preparation

Take note of the following remarks before starting with the programming assignments.

- Try to understand how the code skeleton we supply is structured. You certainly don't have to study it in detail, because there are some technical parts that are not necessary for understanding the project. Start with the assignment files and see what other code is used or imported.
- If you get stuck on a problem (error message, coding, theoretical, ...) for more than 15 minutes, you should ask your tutor for help.
- Ideally, the student with the weakest programming skills should be the one coding the most, and the more experienced students should offer help and advice or do other work. But feel free to discuss other options. The most important thing is that everyone can contribute to the team in a meaningful way.

4.2 Assignment 1a: Average Color Distance

Utility functions We will use a fixed square size of 32×32 for the patches. Therefore, the height and width will not necessarily be divisible by 32. Implement the two functions `number_of_patches` and `output_image_size` in the file `utils.py`. These helper functions are used to define the grid dimensions and to truncate the image to a size that fits the grid.

Average feature Implement the method `feature` in `assignment1.py`. Hint: Have a look at `numpy.mean` how to compute the average of a vector. Note that we are dealing with colored patches (red, green, and blue component). This means that your average will be a vector of size 3, not be a scalar number. Ask your tutor for advice on this.

Distance Finally, implement the method `distance` in `assignment1.py`. Here you should call the `feature` method to compute the features first, and then compute the difference between those features as presented in equation 5. Your method should return a single scalar number, the distance.

Testing Run the code in `assignment1_test.py` to see if your computed features and distances make sense.

4.3 Assignment 1b: Nearest Neighbor Search

What you have so far is a way of measuring the distance between patches. To find the closest match between the target patch and patches in the dataset, we will use the nearest neighbor search as discussed in section 3.5. Instead of coding it yourself, you should use an existing implementation such as the one from the Python `scikit-learn` package. In the `get_model` method from `assignment1.py` you should construct the NearestNeighbor object with two parameters:

- `n_neighbor`: This is the number of closest neighbors you want to retrieve.
- `metric`: This is a pointer to the function that computes the distance. Set it to the method you implemented.

Testing You can test your nearest neighbor search by running `assignment1_test.py`. It will show you a few examples of nearest neighbors for a given patch. If it looks good, run the main program in `assignment1.py` with your own images. You will notice that the program takes a long time to finish. **What do you think causes the long computation time?** Measure the time it takes to process a single patch (directly in the code, or with a stopwatch).

4.4 Assignment 2: Clustering

Implement the method `get_model` in the `assignment2.py` file. Similar to