UNIVERSITÀ
DEGLI STUDI
DI PADOVA

UNIVERSITY OF PADUA

COMPUTER VISION COURSE

PROJECT REPORT

PARKING SLOT DETECTOR

Student
Enrico Gottardis

Student ID
2122138

Contents

1 Introduction **3**

2 Parking detection **4**

3 Parking slot occupancy **5**

3.1 First approach 6

3.2 Second approach 7

4 Conclusion **11**

Chapter 1

Introduction

The aim of this project is to study the occupancy of a parking slot. I decided to split the project in two tasks, the former regarding the parking slots detection and the latter regarding the parking slot occupancy.
I chose the dataset offered by the website http://cnrpark.it/, which provides both the full images and the single slot patches. There are two main folders I made use of in my project: *CNR-EXT_FULL_IMAGE_1000x750* and *CNRPark-Patches-150x150*. The first folder contains many subfolders with the images of the parking slots as well as 9 csv files containing the coordinates regarding the parking slots seen from each camera. The second folder contains instead numerous patches of the images, cropped on the parking slots, and based on the name of the subfolders (the patches are either contained in a subfolder called *'Busy'* or *'Unoccupied'*) it was of great use for implementing machine learning approaches.
The project has been developed on the platform Jupyter Notebook and all the results, included the ones reported on this document, can be found there.

Chapter 2

Parking detection

In the first part of the project, the main focus was on detecting the parking slots. I used two methods to do so, *img_list* and *drawLines*. The latter exploits the information contained in the csv files regarding the coordinates of the parking slots and draws a white rectangle for every set of coordinates on each image, highlighting the parking slots. The former, instead, has the purpose of collecting all the images in a folder inside of an array. Moreover, *img_list* incorporates *drawLines* so that the images are saved inside of an array with the parking slots already drawn on them.

Even though this approach is the one I chose for the project, it is not the only viable solution. Another interesting approach, yet more elaborate, would have been to not make use of the information contained in the csv files regarding the coordinates of the parking slots and instead find them by scratch. In order to do so, I would first apply edge detection on a blurred image, so that I would reduce the noise of the image, and then apply the Hough transform. However, the results obtained by doing so are not optimal, so I would have needed to manually correct the results obtained with such approach for all the 9 positions of the cameras. Anyway, the methods to pursue such approach have been implemented in the code along with their results on the first image contained in the dataset.

Also, I implemented another method, *show_img*, taken from a precedent lab, which is useful to print an image on the output of the jupyter notebook.
In the end, after the first part of the project I managed to save all the images, with parking slots already drawn, into three arrays: *overcast_images*, *rainy_images* and *sunny_images*.

Chapter 3

Parking slot occupancy

The second part was trickier than the first. This was due to the fact there are multiple ways to tackle the problem and there needs to be preprocessing of the data with some approaches. So, the first thing when dealing with this part was a brainstorming on how to tackle the task. I opted for two different approaches: both exploiting machine learning but with different methods. It is important to notice that the results obtained are also bound to the limits of the hardware: as it will be explained later, in some instances this prevented the use of machine learning models. Anyway, before delving into the approaches I will explain the general method *img_list_without_lines* implemented for this part.

The method *img_list_without_lines* is similar to *img_list*, with the difference that now it saves the patches contained in the coordinates defined in the csv files, all in one array. By exploiting this method, I was able to store all the patches in one array. Before developing multiple approaches to find the park occupancy, I applied *img_list_without_lines* on the folders contained in *CNRPark-Patches-150x150*, thus saving all the patches in one array X and by looking at the name of the folder the images were being fetched (*'Busy'* or *'Unoccupied'*), I assigned to an array y the value of the parking slot: 1 for the busy parking slots and -1 for the unoccupied ones. This was fundamental for the implementation of the machine learning models.

An important remark is that by following the aforementioned approach, I am highly bound to the labelling of the images. This means that any errors in labelling the patches of the parking slot, for example stating that a busy parking slot is unoccupied or the opposite, will be fed into the machine learning models, thus decresing their accuracy. In order to prevent this from happening, other methods should be investigated, such as an approach involving deep learning with neural network where

the images are fed into the model without any regards to their labelling and the system is able to develop a model on its own.

3.1 First approach

The first approach I used to solve the task was to apply machine learning on the histograms of images. After saving all the patches and their respective labelling, I defined a method to calculate the histogrms of the images. To do so, I exploited the method *calc_hists* developed for the first laboratory of the course, which takes in input an image and outputs an array containing a triplet (b, g, r) for each pixel, where the letter i, with $i = b, g, r$, represents the value of the corresponding histogram. Then, in order to reduce the dimensionality of the input into the model, I took the mean of the histograms for each patch, saving these values as a dataframe into *df_histograms*. Working with 12585 patches, the previous operation meant that the dataframe *df_histograms* had 12585 and 3 columns, M_Blue, M_Green, M_Red, each referring to the mean of the histograms of the color component inside of the image.

I used the method *train_test_split* from *sklearn.model_selection* to divide my dataset into a training set and a test set, with the respective value of y. As regards the proportions, I chose to take 80% of the total dataset as the training set and the remaining 20% as the test set.

I also defined the method *evaluate* to better investigate the behaviour of the machine learning models. *evaluate* returns the accuracy and the area under the ROC curve of a given model. It was applied only to the *LogisticRegression* model.

As regards all the machine learning models, I checked the results of the following ones, described with the names they are seen with on the code:

- LogisticRegression, *LR*
- LinearDiscriminantAnalysis, *LDA*
- KNeighborsClassifier, *KNN*
- DecisionTreeClassifier, *CART*
- GaussianNB, *NB*
- SVC, *SVM*

To obtain the results, I applied *cross_ val_ score* from *sklearn.model_ selection*, using StratifiedKFold with $n\_splits = 10$. From the results, I chose *KNN* as it was the most performing one, with an accuracy of 0.729937 and a standard deviation of 0.014184. As mentioned in the code, these accuracies are not ideal, as I get an accuracy of almost 73%. This means that there exist better approach to find the parking slots occupancy.

After training the models and having chosen the most performing one, I had to exploit it to check the number of busy and unoccupied parking slots for each image. In order to do so, I developed 3 methods: *patchesFinder*, *patches_ list* and *histogram_ occupancy*. The first method was used to find the patches inside of the images: it is similar to *drawLines*, but the purpose is different, as *patchesFinder* returns all the patches (found exploiting the information contained in the csv files) inside of an image into an array. The method *patches_ list* made use of *patchesFinder* to find the patches for every image and save them in an array, iterating this procedure for all the images. In this way, I was able to store all the patches in one multi-dimensional array, where if I accessed the array in position i I could retrieve all the patches contained in the image i. This method was extremely useful to determine the number of busy and unoccupied slots per image.

In the end, I checked the number of busy and unoccupied parking slots inside of the images contained in the folder *OVERCAST*, saving the results of the prediction made by the model into the dataframe *res_ df*: for each image there is the number of busy parking slots, the number of unoccupied ones and the total of the parking slots inside of that image. I checked only the images contained in the folder *OVERCAST*, as the computation is a bit slow (considering there are 1307 images in that folder with an average of about 30 parking slots per image, thus bringing the total patches to about 39210, without considering the computation needed to retrieve the patches from each image), but the same procedure can be applied to find the parking slot occupancy in the images contained in the folders *RAINY* and *SUNNY*, there would just be the need to change the path to the folders to retrieve the patches from the respective images and then feed these patches to *histogram_ occupancy*.

3.2 Second approach

For the second approach I opted to exploit the computer vision algorithm *SIFT*, which stands for *Scale-invariant feature transform*, and perform scene recognition using *bag of visual 'words'* models. By applying *SIFT* on each image of the array

X mentioned before, so the array containing all the patches inside of the folder *CNRPark-Patches-150x150*, I was able to find the descriptors of every image with the method *detectAndCompute* offered by OpenCV. I stored all the descriptors in one array, called *all_ descriptors* which was later used for retrieving the visual 'words'.

Afterwards, I used *kmeans* clustering to perform a fit of *all_ descriptors*: having choosen a size of $k = 20$ clusters, the *fit()* method assigns each descriptor to a corresponding cluster among the 20. Finally, the cluster centers are extracted from the trained KMeans object and stored in the *words* variable. Cluster centers represent the centroids of the clusters, effectively encapsulating the typical descriptors of each cluster.

The *all_ bows* list was used to store the bag of words (BoW) representations for all image patches. To effectively fill *all_ bows*, I used the following *for* loop:

```
all_bows = []
for i in range(len(X)):
    bows = []
    try:
        for j in range(len(descriptors[i])):
            best_match = -1
            best_distance = np.inf
            for k in range(len(words)):
                distance = np.linalg.norm(descriptors[i][j] - words[k])
                if distance < best_distance:
                    best_match = k
                    best_distance = distance
            bows.append(best_match)
    except:
        ''
    all_bows.append(bows)
```

Figure 3.1: Snippet of the code used to obtain the BoW representations

In the figure 3.1 it can be seen that I iterated the *for* loop over all the patches in input X. For each patch, corresponding to the image of a parking slot, I initialized a list *bows* to store the BoW representations for that patch. Inside of a *try/except* construct, I found the best match for each descriptor by iterating through the descriptors of each patch. In particular, for each descriptor the code:

- Calculates the Euclidean distance between the current descriptor and each word descriptor.
- Keeps track of the best-matching word descriptor and its distance.
- Appends the index of the best-matching word descriptor to the *bows* list.

Whenever an error was encountered during descriptor processing, the code simply skipped the current descriptor and went onto the next one. Finally, I appended *bows* to *all_bows*: in position i, with $i = 0, 1, ..., len(all\_bows)$, $all\_bows(i)$ contained all the processed descriptors of that patch.

To even the shape of *all_bows* I opted to take only 40 columns for each row, where each row corresponds to a patch, and I filled the *NaN* values in the dataframe with 0.0.

The dataframe *all_bows* and the array y were then fed to the same method as in the first approach, *train_test_split()*, and the same structure was used to find the accuracies of various machine learning models. However, different than in the first case I now did not manage to implement the *KNeighborsClassifier* model due to errors most probably linked to the memory limits of my system. Anyway, the most performing model achieved with the SIFT and BoW approach was higher than the result achieved in the first method. With an accuracy of 0.760529 and a corresponding error of 0.012832, the *LinearDiscriminantAnalysis* was the most performing model. Recalling that with the first approach the accuracy was of almost 73%, using SIFT the accuracy of the system (approximately 76%) has increased of slightly more than 3%.

In order to implement also the *KNeighborsClassifier* model, I tried redefining the number of clusters and the batch size of *KMeans*, but unfortunately I was unsuccessful and I encountered the same error as before.

Having found the model, I only needed to preprocess the images of the parking slots (not just the patches of the images) and feed them to the model.In order to do so, I developed another method, *SIFT_occupancy*, which takes in input the data, the chosen model, and the training arrays. I exploited the methods *patchesFinder* and *patches_list* defined for the first approach to have a list containing in position i all the patches corresponding to the parking slots of image i. As regards the preprocessing inside of *SIFT_occupancy*, I followed the same pattern used to preprocess the images for the machine learning models. I added only 2 checks: one as regards the number of clusters, as there were patches with a number of descriptors inferior to the number of clusters, and one as regards the columns of the *all_bows* dataframe, as there were patches with less than 40 columns. Finally, the method returns an array where for each position i, with i being an image, there are two values: the number of busy parking slots and the number of unoccupied parking slots.

In the end I saved the output of *SIFT_occupancy* in *sift_predict* and, as done in the first approach, I translated everything into a dataframe to have a better repre-

sentation of the number of busy and unoccupied parking slots for each image. As in the first approach, I checked only the images contained in the folder *OVERCAST*, as the computation is extremely slow (in order to check the park occupancy of the *OVERCAST* images the code took approximately 1600 seconds), but the same procedure can be applied to find the parking slot occupancy in the images contained in the folders *RAINY* and *SUNNY*, there would just be the need to change the path to the folders to retrieve the patches from the respective images and then feed these patches to *SIFT_ occupancy*.

Chapter 4

Conclusion

In the last part of the document, I would like to make some comments on the results obtained. As it can be seen from the results, by pursuing the first approach I am able to obtain an accuracy of almost 73%, whereas by choosing the second approach I am able to get an accuracy of approximately 76%. This shows two main things:

1. The second approach is better

2. The accuracy is not high, even if I follow the second approach

The first point is due to the fact that the first approach, involving the (mean of the) histograms of the patches, is more rudimental with respect to the second approach. It is easy to notice a flaw in the reasoning behind this approach: the model will learn that a parking slot is unoccupied if it has a certain color, however if there is a car on that parking slot which has the same (or similar) shade of color as the ground, then the model will label it as an unoccupied parking slot. The same can be said as regards the opposite: if there are some obstacles between the camera and the parking slot, it could happen that the model labels the parking slot as occupied, even if it is not. The second approach, instead, is more reliable. This is due to it looking for features inside of every image and looking for those same features when determining if a parking slot is busy or unoccupied.

As regards the second point, the coordinates of the parking slots play a huge role in the issue. Even though many times *SIFT* is computed on the grayscale of an image, I decided to perform it onto the colored images to find and retrieve as much features from them as possible. However, in the end I achieve a maximum accuracy of 76%, which could be higher. As said before, I exploited the information in the dataset to draw the parking slots and obtain the patches of the images on those

coordinates. By looking at the images with the parking slots drawn onto them, it is easy to notice that the coordinates for some parking slots overlap or are not well-defined: this obviously brings noise into the model, which is then perpetuated through every image, thus leading to the accuracy shown.