Robotics Practical 1: Assignment 4

Laura M Quirós (s4776380)
Adriana A. Haralambieva (s4286103)

March 24, 2024

Q1. Describe the method you used to decide whether an ROI should be considered. Which criteria of the ROI did you take into account? What did you do to test your criteria?

To retrieve the ROI, we first pass the dilated and eroded binary image through the `find_contours()` function. Based on that, it will extract the coordinate points for what was left as signs after the filtering. After that, we obtain the rectangular bounding box and crop it from the actual image. We first check whether the bounding box's height and width are more than 50 pixels. If there are multiple signs on the image, we only want to take into account the one that is closest to the robot (thus putting this requirement). Afterward, we check whether the absolute difference between the width and height is less than 15 pixels. Since the signs appear as circles, ideally the cropped image should be a square. However, we allowed for some margin of error due to the filtering. Finally, if the sign becomes too close to the robot - width and height larger than 100 pixels, we would ignore it. Finally, we resize the extracted ROI to side lengths. As for testing, we placed the robot in different positions compared to the signs. The scenarios included the robot being too far from 1 sign, too close to it, and having multiple signs in the frame.

Q2. Describe the method you used to collect your data set. Focus on reproducibility and your reasoning. How did you acquire the data? How many images did you collect per class and in total? How did you decide on this?

In order to collect enough instances of each of the signs classes, we manually added each sign class after another, each 2m apart from each other. By running the simulation with the `data_collect` set to True, which was the default value, we make each of the detected ROIs be added to the list `collectedROIs`. This list is saved to the home folder as a numpy file upon shutdown of the program. We collected a total of 2996 images, classified as follows in table 1.

Class	Number
left_arrow	263
right_arrow	215
up_arrow	310
square	386
triangle	1335
smiley	278
background	209

Table 1: Classes and Corresponding Numbers

Q3. Describe the method you used to design your CNN. Focus on reproducibility and your reasoning. How many layers did you choose, why? How many filters, why? What did you do to come to these numbers?

We did not change the number of epochs nor the batch size, leaving them at 15 epochs and batch of 20. This decision stems from the sufficiently accurate results seen in the plot below (Figure 1. Had we found evidence of lack of convergence due to short training time, this would have been incremented, and viceversa, had we had reasons to believe the training time was too long. In terms of the model layers, the initial convolutional layer had 32 filters, similarly to why the final dense layer had a default value of 64 neurons, it's because the exponents of 2 are usually quite efficient. We decided to go for a big enough number of filters to ensure that the details of each of the features were captured. With regards to filter size, we attended to the lecture slides which suggest using two different filters, one with size (3,3) and one with size (2,2) as a way to replace a (5,5) filter but with more layers. In our first attempts, the model only had one dropout layer before the maxpooling, flattening and dense layer, but this lead to sub-optimal results. We argued that it was because the dropout layer was not helping that much in the generalisation by itself, so we decided to add another dropout layer, still with the same dropout rate and after it, another convolutional layer, to guarantee that the dropout and maxpooling layers would cause some learning.

This finally lead to training results always between 0.8 and 1 (Figure 1), which upon further testing in gazebo, proved to be quite good.

```
    # Create a model
    model = models.Sequential()

    # Add CNN layer(s) to the model
    # image size is 40x40x3 so we want to use a kernel size of 3x3
    n_filters = 32
    filter_size = (3, 3)
    model.add(layers.Conv2D(n_filters, filter_size, activation="relu", input_shape=(IMAGE_SIZE,
        IMAGE_SIZE,3)))
    # Possibly add dropout, and/or max pooling
    filter_size = (2, 2)
    dropout_rate = 0.2
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.MaxPooling2D(filter_size))
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.Conv2D(n_filters, filter_size, activation="relu"))

    # Add a flatten layer
    model.add(layers.Flatten())
    # Add Dense layer(s) to the model
    n_neurons = 64
    model.add(layers.Dense(n_neurons, activation= "relu"))
    # The final layer needs to be a softmax
    model.add(layers.Dense(NR_OF_CLASSES, activation="softmax"))
```

Q4. Include at least a graph/plot of the model accuracy over epoch time.

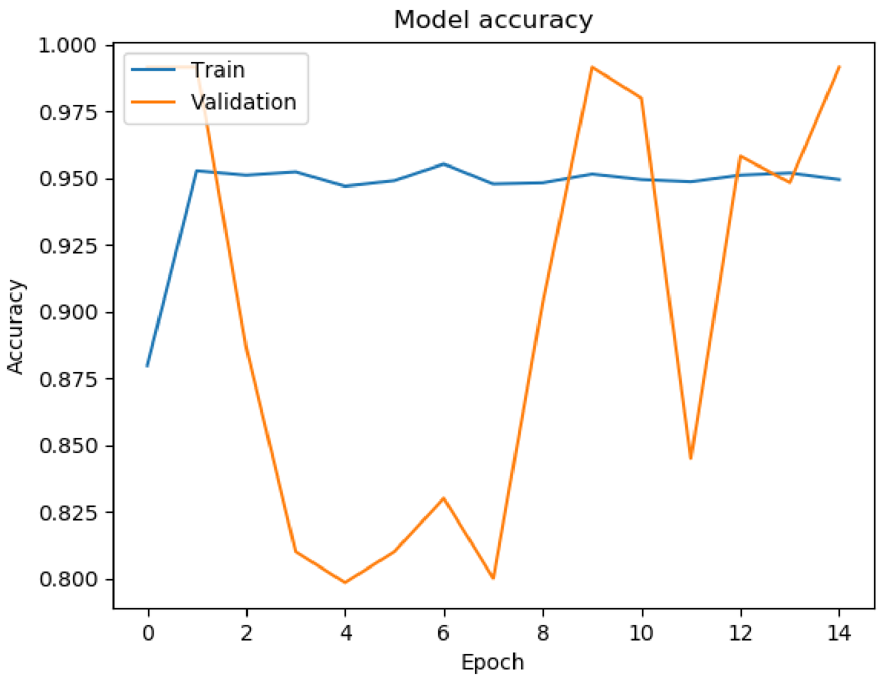

Figure 1: Caption

Q5. Describe how you treat incoming recognized labels to make sure each sign is only printed once.

We created a history array as one of the fields in the controller class. We would only add a label to this array if the history is empty, or if the last item in the history is the same label. When we have added 10 labels (by definition, all equal), there are two possible ways to go. First, if the history label is not the background class (we only want to print the signs) and that it is not the same as the last encountered sign (the variable `last_sign`), we print and store the last sign.

Secondly, the variable `last_sign` can also become background if the background has been encountered 10 consecutive times, we just don't print it. This way we guarantee that if a sign appears twice, it will be identified as two separate instances of the same class of a sign.

```
    check_threshold = 10

    if label is not None:
      if len(self.history)>0 and label == self.history[-1]:
        self.history.append(label)
      else:
        self.history = [label]

      if len(self.history) > check_threshold:
        self.history.remove(self.history[0])

      if len(self.history) == check_threshold:

        if self.history[0] != "background" and self.history[0] != self.last_sign:
          # if 3 are the same but not background, we print and set new last_sign
          self.last_sign = self.history[0]
          print(self.last_sign)

        elif self.history[0] == "background":
          self.last_sign = self.history[0]

    self.update_robot()
```