CMPUT 412: Exercise 5 - ML for Robotics

Justin Valentine (jvalenti), Sergey Khlynovskiy (khlynovs)

1.1 MNIST Dataset and ML basic terminologies:

Deliverable 1.

After the the first pass of backpropagation we have the following weights:

$$w_1 = 0.12,\ w_2 = 0.23,\ x_3 = 0.13,\ w_4 = 0.1,\ w_5 = 0.17,\ w_6 = 0.17$$

Plugging this back into the NN gives a new predicted value of 0.26 that means we have an error of 0.2738 which is lower than the first prediction!

Next we will do the next pass of backpropagation:

$\Delta = \textit{prediction - actual} = 0.26 - 1 =- 0.74$

$h_1 = i_1 w_1 + i_2 w_2 = (2)(0.12) + (3)(0.23) = 0.93$

$h_2 = i_1 w_3 + i_2 w_4 = (2)(0.13) + (3)(0.1) = 0.56$

Therefore using the backward pass equations derived here, we get the following updated weights:

$$\begin{pmatrix} w_5 \\ w_6 \end{pmatrix} = \begin{pmatrix} 0.17 \\ 0.17 \end{pmatrix} - (0.05)(-0.74)\begin{pmatrix} 0.93 \\ 0.56 \end{pmatrix} = \begin{pmatrix} 0.2044 \\ 0.1907 \end{pmatrix}$$

$$\begin{pmatrix} w_1 & w_3 \\ w_2 & w_4 \end{pmatrix} = \begin{pmatrix} .12 & .13 \\ .23 & .1 \end{pmatrix} - (.05)(-0.74)\begin{pmatrix} 2 \\ 3 \end{pmatrix}\begin{pmatrix} .17 & .17 \end{pmatrix} = \begin{pmatrix} 0.13258 & 0.14258 \\ 0.24887 & 0.11887 \end{pmatrix}$$

That is the new weights are (to 2 digit precision):

$$w_1 = 0.13,\ w_2 = 0.25,\ x_3 = 0.14,\ w_4 = 0.12,\ w_5 = 0.20,\ w_6 = 0.19$$

With these new weights we get a new predicted value of 0.329, which gives us an error of 0.225. Note that after the second pass of backpropagation the loss is closer to 0.

Deliverable 2.

1. During training, random rotation and random crop are used. This allows the model to be more accepting of distorted images. With distortion the model finishes with a test loss of 0.061 and test accuracy of 98.12%. Removing the transformations we finished the training with test loss of 0.061 and test accuracy of 97.75%. So clearly having the transformations improves number detection. We introduce transformations back into the code in the following tests.

2. In the code the batch size is 64. This represents how many training samples we will train on each epoch. Lowering this to 16 we get a test loss of 0.064 and a test accuracy of 98.13%. Increasing this to 1024 we get a test loss of 0.078 and test accuracy of 97.54%. So it appears that a lower batch size is better for this model and dataset, however, this could also be explained by the random seed we have chosen as these models are stochastic in nature. We reset batch size to 64 in the following tests.

3. The model uses a ReLU activation function which takes the max of 0 and x so the output can never be negative. With this activation function the model achieves a test loss of 0.061 and a test accuracy of 98.12%. Changing this to a linear function, which is equivalent tojust removing the ReLU function, we get a test loss of 0.463 and a test accuracy of 87.43%. This is significantly worse because any linear function can be simplified into a straight line which is not complex enough to detect digits. ReLU on the other hand is non-linear.

4. The model uses the ADAM optimizer. The role of an optimization function in machine learning is to minimize the loss which represents how far our estimated detections are from the true value of the digits.

5. Neural networks can overfit quite early. Dropout allows you to avoid this by ignoring random layer outputs which helps the model not get used to a certain architecture during the training process. Adding dropout before each linear layer we get a test loss of 0.087 and test accuracy of 97.25%, so in our case adding dropout did not seem to change the results that much.

1.2 Number Detection Node:

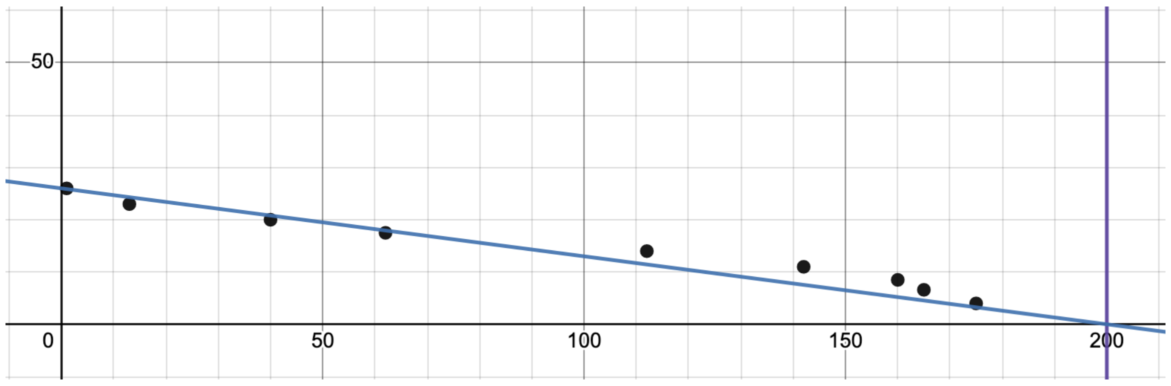
Robust Traversal of Duckie Town:

A primary focus on robustness guided the development of our Duckie Town traversal architecture. We aimed to achieve this by incorporating sensory feedback to inform the bot about its optimal state.

Lane following:

We optimized lane following by building upon Justin's Solution 3 code and implementing minor adjustments. By decreasing the offset from 220 to 200, we improved the controller's reliability, particularly when navigating tight corners. This modification contributed to a more robust and efficient navigation system in Duckie Town.

Stopline approach:

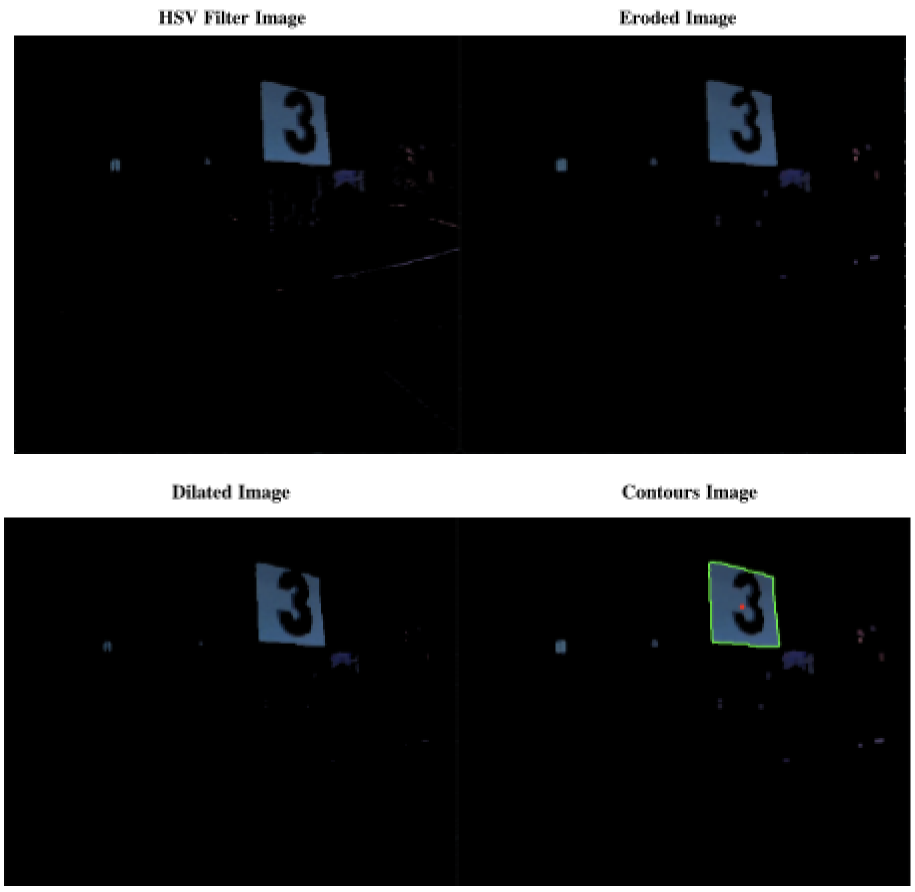


The accompanying plot displays the relationship between the y-coordinate of the centroid (cy) for the red stopping line (horizontal axis) and the distance in meters (vertical axis). As anticipated, a linear relationship exists between the two variables. Leveraging this linear relationship, we designed a strategy to decelerate the robot as it approaches the stop line. The implementation details can be found in the code snippet below.

```python
self.proportional_stopline = (cy/170)*0.14
```

This adaptive deceleration technique allows the robot to respond effectively to varying conditions and distances, minimizing the risk of overshooting or stopping too far from the line. As a result, the overall performance and dependability of the stopping mechanism are greatly improved, contributing to a more robust and reliable navigation system. However there are occasional instances where the speed reduction may be overly conservative, causing the DuckieBot to become stuck. Fortunately, a gentle nudge from a nearby human can quickly get the bot back on track and resume its journey.

Stopping at Signs:



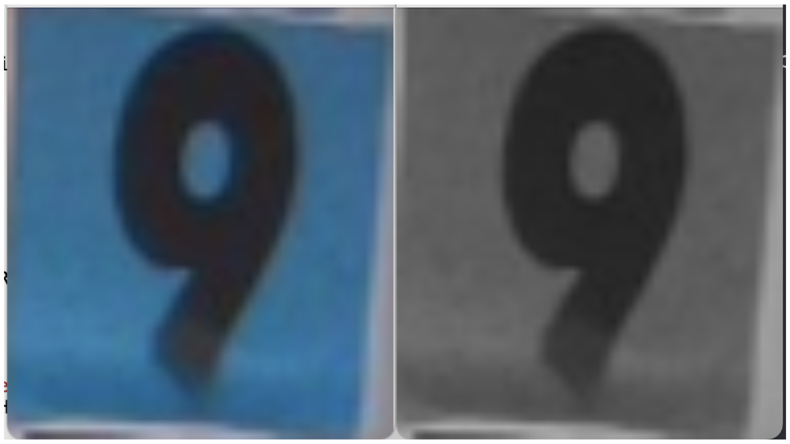
We investigated and implemented a sign detection algorithm to enhance our DuckieBot's ability to reliably stop at and read numbers from road signs. The primary goal was to detect and highlight the largest contour within an HSV-filtered image.

The process began with loading and cropping the input image to concentrate on the region of interest. Subsequently, we defined a color mask to isolate the specific color range of the signs. The image was converted from BGR to HSV color space, enabling us to apply the HSV filter effectively. The color mask was then applied to the HSV image, resulting in an output where only the regions of interest had non-zero pixel values.

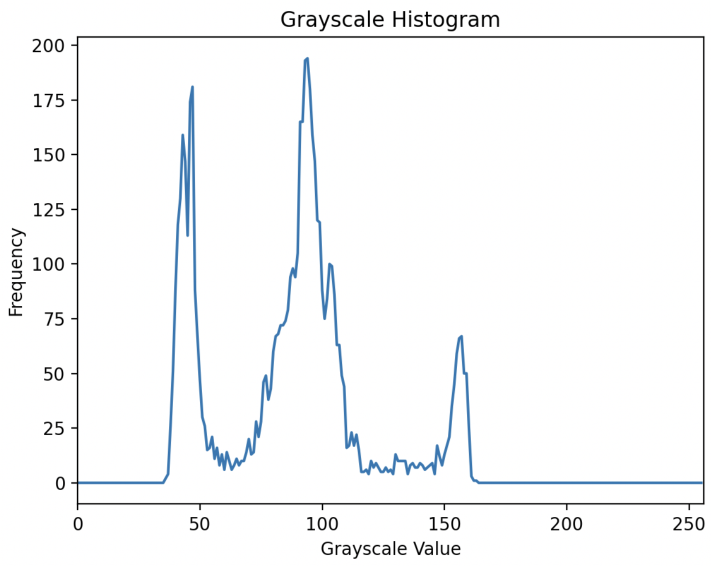
To further refine the output image, we utilized morphological operations, including erosion and dilation, to minimize noise and emphasize the signs' boundaries. We extracted contours from the processed image using OpenCV's cv2.findContours() function and identified the largest contour by comparing the areas of all detected contours. This largest contour area also served as a stopping condition for the DuckieBot, ensuring reliable navigation and sign recognition.

Filtering the Digits:
We developed and implemented a method to extract and process the digits from road signs for accurate number recognition using a machine learning model. The primary goal was to create a binary image with the digits represented by white pixels and the background by black pixels.

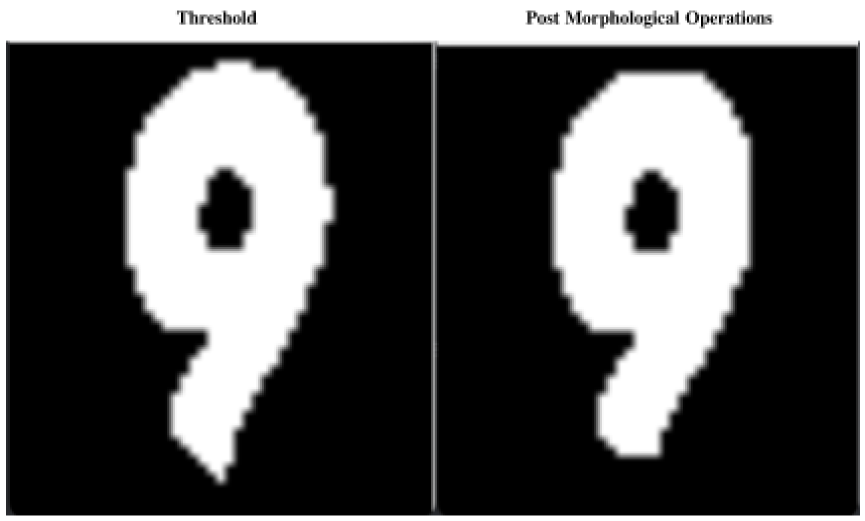
The sign area was then cropped and converted to grayscale for subsequent processing. This can be seen below:



Below is a histogram of the grayscale image, which visualizes the distribution of pixel values. This histogram aids in determining an appropriate thresholding intensity value to apply an adaptive Gaussian threshold for creating a binary image of the digit. The rationale behind this approach is to identify a threshold value that effectively separates the image at the two peaks, corresponding to the background and the digit.



Next, we removed noise from the binary image using morphological operations, specifically by applying an opening operation with a small kernel.

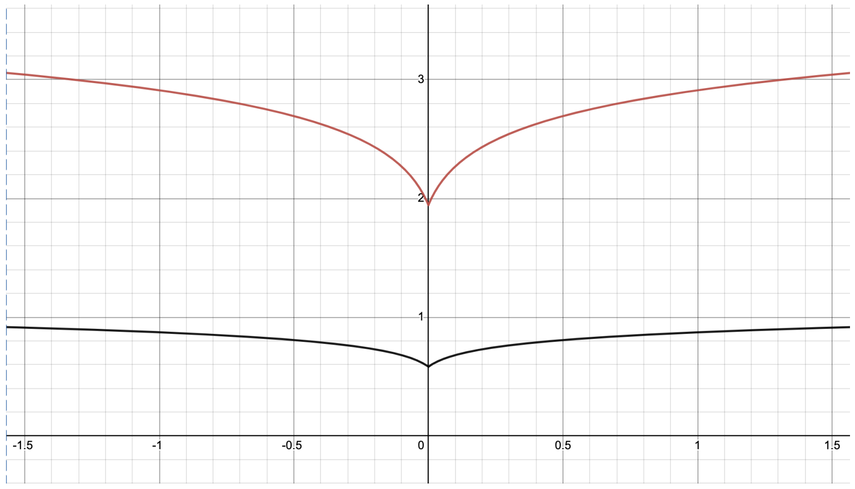


However, we found that using this approach was less efficient than using an adaptive threshold, which we ultimately utilized in our final solution instead of the histogram method. Despite this, the overall process resulted in a clean, binary representation of the digits on road signs, which can be fed to a machine learning model for accurate number recognition

Turning:

In Lab 4, we identified turning as a weak point in our implementation. Previously, we controlled turns by specifying particular angular and linear velocities and maintaining them for a predetermined duration. This approach proved to be rather unreliable. To address this issue, we have now incorporated odometry data, derived from wheel encoders, into our turning strategy. By utilizing this feedback from the robot as it moves, we can more accurately determine if the robot should continue turning, resulting in a more precise and dependable turning mechanism.

We attempted to improve the turning mechanism by using the error in the robot's orientation to adjust its angular velocity. However, we discovered that this approach was overcomplicated and unnecessary. For the sake of completeness, we have included the function we initially intended to use below:

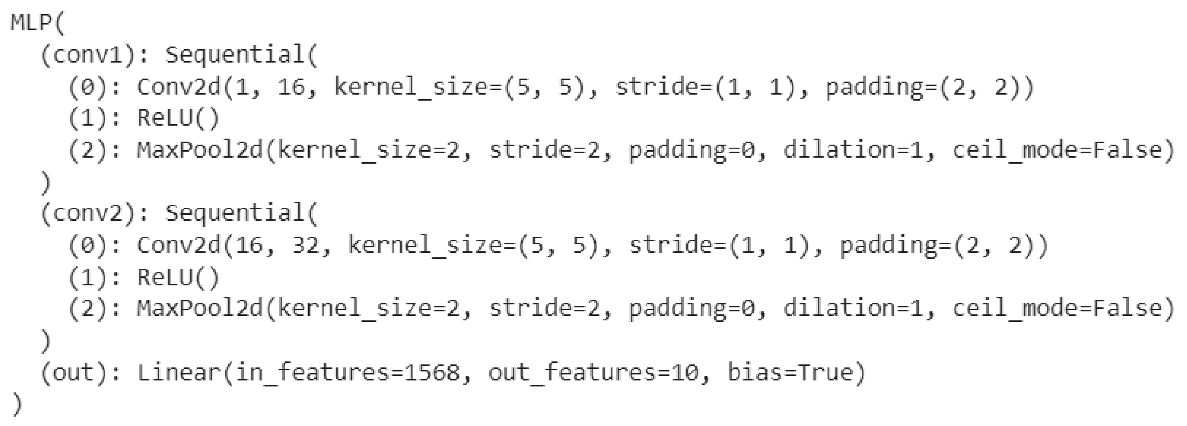


In the graph presented here, the horizontal axis represents the error in the robot's orientation. The red curve corresponds to the robot's angular velocity as a function of orientation error, while the black line represents the associated linear velocity. The relationship between linear and angular velocities can be described using the equation $V = r\omega$, where V is the linear velocity, r is the radius of the turn, and ω is the angular velocity.

Custom Data Set:

We used Steven Tang's custom dataset and appended some of our own images by moving the bot around the town and saving images using cv2.imwrite('/data/rospy.Time.now().png', img) that appropriately show the number after processing. This proved a bit tricky as this function does not throw an error if an image was not saved and we could only get this to work when running on the bot locally. This roughly doubled the dataset. On the left you can see a 0 from the original dataset and on the right is a 7 from our appended images.

We then trained a model on all 488 images, having 351 images for training and 39 for validation. We initially tried training a linear model but found that this overfits on our small dataset so we needed to improve generalization by creating a convolutional model.



```
MLP(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

Even with the improved model some numbers are still mixed up especially when the filtering of the numbers does not capture the number properly.

Connecting Everything:

We made our april tag detection code into a service as well as the machine learning node which detects numbers. This was done so that we could wait until everything was ready before running. Making services sped up computation time as we send images on each service call so we only have to subscribe and preprocess an image once. To do this we created a custom service which sends images and receives an integer. In april tag detection we denoted 0 as not detected and 1 a detected, and in the machine learning service this was used to indicate which digit was detected or -1 if we are not confident enough in our detection. The connection to the services was done in a node separate from lane following in order to get these to run in parallel. This way the color detection and lane following works properly. Service calls are done once per second and we send a boolean in a topic to the lane following once all digits are found indicating that we should shutdown.

Deliverable 3.

References

https://github.com/Aizuko/duckietown/tree/main/lab5/pytorch - dataset
https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118 - CNN model