Group R: Learning to drive by crashing

1st Brian Pulfer
Faculty of Informatics
University of Southern Switzerland
Lugano, Switzerland
pulfeb@usi.ch

2nd Rwiddhi Chakraborty
Faculty of Informatics
University of Southern Switzerland
Lugano, Switzerland
chakrr@usi.ch

3rd Shubhayu Das
Faculty of Informatics
University of Southern Switzerland
Lugano, Switzerland
dass@usi.ch

***Abstract*—Through this document, we present our studies of how a Mighty Thymio, an extension developed at IDSIA (Istituto dalle Molle di studi sull'intelligenza artificiale) of the more known Thymio robot, can be programmed to avoid obstacles and walls in a virtual environment through the use of Convolutional Neural Networks and a camera sensor.**

***Index Terms*—robotics, self-driving robot, thymio, gazebo, ros**

I. Introduction

This document represents the final write-up of the final project for the course of Robotics held by the University of Southern Switzerland (USI - *Università della Svizzera Italiana*) during the spring semester 2020.

During the first part of the course, students were introduced to theoretical topics as well as practical ones. By the start of the project, students were familiar with ROS (*Robots Operating System*) and with writing scripts using the package *rospy* that could send messages through the ROS nodes.

Projects would need to use this theoretical and practical knowledge to implement some complex program that manipulates a robot and makes it achieve some goals.

The project repository is available at the following link: *www.github.com/BrianPulfer/Learning-to-drive-by-Crashing*.

A. Project

The fundamental idea is to exploit the camera sensor mounted on the Mighty Thymio robot to teach the robot, through the use of Convolutional Neural Networks (CNNs), to not crash in a virtual environment. The goal is to make the robot able to autonomously and freely move in the environment without ever making contact against walls and obstacles.

Simulations are run on *Gazebo*.

B. System Environment

The code present in the repository is meant to run on the virtual machines that were provided at the beginning of the course. To replicate results, here's a list of requirements:

- Ubuntu Operating System
- ROS
- Python 2
- Tensorflow 2
- Gazebo
- Virtual model of the robot (Mighty Thymio)

C. Pipeline

To achieve the mentioned results, we have been iterating multiple times on the following pipeline: creating a world to collect a training corpus, create the training corpus with adequate labels, fit a CNN on the labelled data and finally build a controller that uses the CNN predictions to steer the robot away from obstacles and walls.

The pipeline was executed for a total of 5 times: 2 times for a classification task and 3 times for a regression task, yielding to increasingly better results.

II. Creating the Gazebo World

Gazebo allows for the creation of virtual environments, or worlds, through its editor mode. We first built a big 10 meters by 10 meters environment with few big obstacles and a wall surrounding the robot.

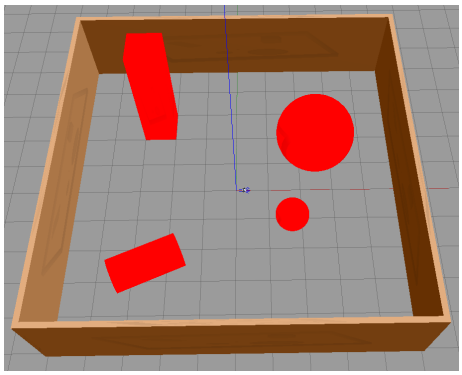


Fig. 1: First world

On this world, we collected 2 data sets for classification and 1 data set for regression.

After running the whole pipeline using this world three times, we opted for a different world. The reason for this decision is that in such a big environment, the right thing to do is usually to move straight, which makes the

labels distributed in an unbalanced way. Also, since a CNN prediction is expensive to make at run-time and making them as much as possible in short intervals of space is important, a smaller world is better suited for this purpose.

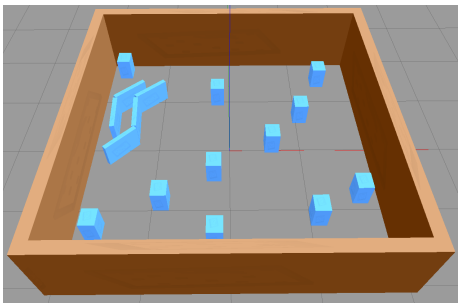


Fig. 2: Second world

The second world is a 4x4 grid filled with more tiny obstacles with respect to the first world. This allowed for more balanced labels in the data set and more evident effects of the machine learning model.

Using this world, the rest of the pipeline was run twice for solving regression task.

III. Collecting the dataset

Collecting the dataset was done in two fashions: by labelling images manually and automatically.

A. Manual labelling

The first approach was to manually label camera readings that were visible from *rqt* and having a script automatically storing the labels and images in a folder and then teleoperating the robot inside the environment with a script.

This approach was better suited for classification tasks, since it was fairly easy for a human to distinguish between 4 cases (nothing is in front, obstacle on the left, obstacle on the center, obstacle on the right). The problem with this approach is that collecting 1'000 images or more was a tedious task. Also, when the dataset was collected for solving the regression version of the task, guessing the labels was much harder.

B. Automated labelling

This approach is the one used for the final dataset that was used to train the CNN. We used a script of a previous assignment that would let the mighty thymio move around an environment without crashing thanks to its infrared sensors that would sense obstacles. When an obstacle was sensed, the robot would put itself orthogonal to the obstacle and start moving straight again until the next obstacle was faced.

We adapted this script such that images were stored as well as labels. To each image, 2 labels are associated. Labels were defined as:

$$L_1 = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix} \cdot S \tag{1}$$

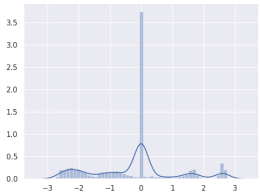
$$L_2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot S \tag{2}$$

Where S is the five-dimensional row vector of the five front infrared sensors readings of the thymio. Each component of the vector ranges from 0 to 1. The higher the value, the more the sensor is sensing some obstacle or wall.

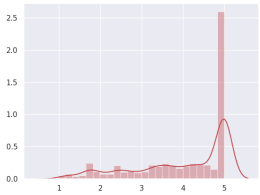
The first label L_1 is in the range [-3, 3] and tells how much the robot should steer in a direction or the other. The second label L_2 instead is in the range [0, 6] and is used to distinguish between the case when the robot is facing no obstacle and when instead it is facing an obstacle from way too close (note that in both cases $L_1 = 0$ and thus is not enough to make a correct decision). Labels were then normalized ($L_1 \in [-1, 1]$ and $L_2 \in [0, 1]$) when training the CNN model.

In total, 8'056 labelled images of the second world were collected.

By analyzing how the data is now distributed in terms of labels, we see that the first labels have a mean of -0.23 and a standard deviation of 1.34, which suggests that instances when the robot should have turned left or right are overall pretty balanced, while the second labels have a mean of 4.01 and a standard deviation of 1.15, which tells that during training, the robot has been facing obstacles and walls from close pretty often.



(a) Distribution of L_1



(b) Distribution of L_2

Fig. 3: Distribution of the labels

In figure 3 we see how the labels are distributed. In particular, on the x-axis we see the labels and on the y-axis the relative density. By the subfigure a, we see that the dataset is

still unbalanced towards cases where no steering is needed, but also has a good variety position of the obstacles with respect to the robot. Subfigure b instead tells that the robot, during the collection of the dataset, has been facing walls and obstacles from close many times, and that is also why in subfigure a we have so many cases where the predicted steering is close to 0.

IV. Fitting a model

As anticipated, the machine learning model used is a custom relatively shallow convolutional neural network. The architecture of the network is presented here in figure 4.

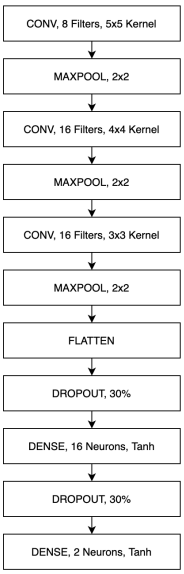


Fig. 4: Architecture of the Convolutional Neural Network (CNN)

Starting from this architecture, we trained the model using 70% of the dataset as training set, while the remaining 30% was used as validation set to apply early stopping with a patience of 30 epochs.

Images were pre-processed in two steps:

- Reshaped from 640x400 to 160x100 pixels
- Each pixel value was normalized to stay in range [0, 1]

Of course, this pre-processing is also carried out when the CNN is used in the final controller before feeding the current camera frame to the model.

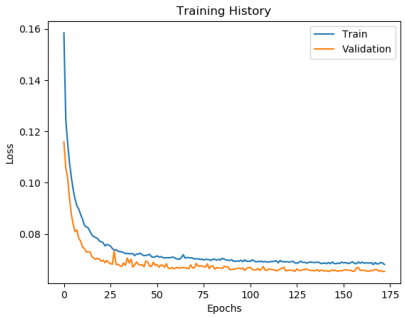


Fig. 5: Training and Validation losses through epochs

In figure 5 we provide the values of the training and validation loss through epochs. The network finished its training after around 175 epochs using batches of 32 images each. The model achieves a final loss function, the Mean-Squared-Error (MSE), of 0.07 circa for both the training and validation data. Surprisingly, the performance on the validation data is better than in the training data for the whole training procedure, suggesting that even better results are easily achievable.
However, a MSE of less than 0.1 on the validation data is enough to obtain good results. Furthermore, a more complex model would make the prediction time even longer.

V. Building the final controller

The final controller is a script that obtains the camera readings and uses the trained model to predict in which direction the robot should steer and by what measure. The robot is then moved accordingly for a little time interval and the process is then repeated. If the model predicts that the robot is close to an obstacle, we decide to make a 45° turn counterclockwise and resume from there.

parameter: proximity threshold: *prox_thr*
parameter: steering constant: *steer_c*
parameter: linear velocity: *linear*
parameter: driving time: *time*
Result: Drives the robot without crashing

```
do
    current_frame = Robot.get_frame()
    current_frame = preprocess(current_frame)

    label_1, label_2 = cnn.predict([current_frame])
    if label_2 ≤ prox_thr then
        angular_speed = label_1 * steer_c
        Robot.move(linear, angular_speed)
    else
        Robot.do_45_degree_turn()
    end
    Robot.stop()
while Program runs
```

Algorithm 1: Driving the Robot in the environment without crashing using the CNN

By Algorithm 1, we see that 4 parameters had to be tuned:

- The **proximity threshold** tells when to consider the robot too close to a wall. A too high value might make the robot crash, while a too low value will make the robot rotate on itself too much
- The **steering constant** converts the model prediction to an angle. A too low value might make the robot crash, while a too high value might make it do violent turns
- The **linear velocity** and the **driving time** together define how many meters is the robot going to follow the defined trajectory. The lower the intervals between a model prediction and another, the better the control of the CNN on the robot

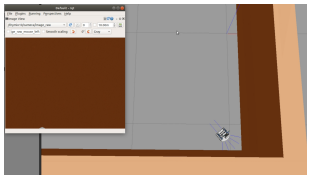
VI. Results

After building the final controller, we obtained a self-driving robot that can successfully avoid every wall on its way and rarely ever touches the obstacles.

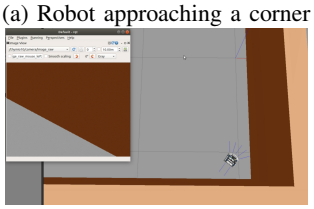
In assessing the results, we checked what the robot reactions were to dynamically placed walls and obstacles. Finally, we let the robot run on the environment in which the dataset was gathered.

A. Walls Avoidance

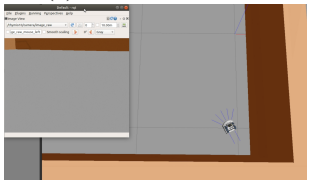
By putting the robot on the second presented world, and after deleting the obstacles leaving the robot surrounded by walls, we could see that the mighty thymio successfully avoids every wall no matter by which angle / position it faces it by doing the 45° turns.



(a) Robot approaching a corner



(b) After one 45° turn



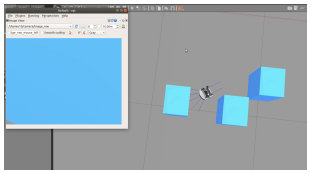
(c) After two 45° turns

Fig. 6: Robot reaction to a corner

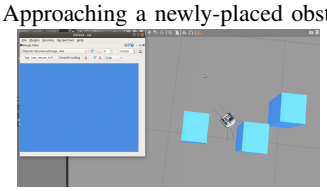
By figure 6 we can see how the robot reacts when put close to a corner. It goes straight until the wall is almost approached, at that point does a 45° turn twice to avoid both walls forming the corner.

B. Obstacles Avoidance

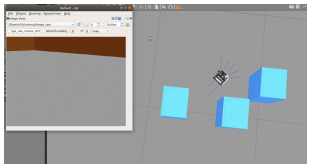
We also tested how the robot would reacts to obstacles placed dynamically at run-time. To do that, again we removed obstacles from the world and re-placed them as the robot moved around.



(a) Approaching a newly-placed obstacle



(b) The robot avoids the first obstacle



(c) The robot avoids the second obstacle

Fig. 7: Robot's reaction to newly placed obstacles

In figure 7 we see how the robot successfully avoids newly placed obstacles by sensing that they are too close and doing a 45 degree turn twice. Also, we have shown that the robot successfully avoids obstacles also by just making little deviations and more interestingly, before the obstacles could even be sensed by the infrared sensors (which are unused).

However, the robot still has a hard time avoiding obstacles that are not visible (robot has a dead zone) or just barely visible. In this cases, the robot might slightly hit the obstacles with one side of its body.

C. Performance on the world

The robot was let run on the second world, the world from which the data was collected. The robot successfully managed to avoid every wall and every obstacle except for one that was close to one wall. From the video, it can be seen that the robot decided to go straight in a situation where on the left there was a wall and on the right an obstacle. The robot slightly hit the obstacle and then did a 45 degree turn twice to avoid both the obstacle and the wall.

VII. Conclusions

The results achieved are satisfactory, since the robot never crashes against a wall and only occasionally slightly hits obstacles. These can however be improved in different ways.

A. Possible improvements

Here, a list of possible improvements that could be applied to the current state of the project:

- Using a pre-trained and more powerful model could lead to better predictions of the steering angle, seen that the current model seems to underfit
- Extend the dataset to have more cases where the obstacles are just slightly visible
- It should be possible to have the model making predictions faster. At the current state, the robot has to be stopped each time and it takes very long (circa 10 seconds) to obtain a prediction from the CNN

Note that the fact that the throughput of predictions is low is probably due to the fact that such computations are not done on dedicated hardware (GPUs), and is thus just a minor issue.

B. Project Extensions

Here we list a couple of extensions that could be implemented in the project in the future:

- Train the model to generalize to differently colored and shaped obstacles and walls
- Cases where obstacles are not visible could be handled implementing some memory (map) of what has been seen previously (localization)

References

[1] J. Guzzi, A. Giusti, G. di Caro, L. M. Gambardella, “Mighty Thymio for University-Level Educational Robotics”

[2] N. Koenig and A. Howard, ”Design and use paradigms for Gazebo, an open-source multi-robot simulator,” 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566), Sendai, 2004, pp. 2149-2154 vol.3, doi: 10.1109/IROS.2004.1389727.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Van Rossum, G. & Drake Jr, F.L., 1995. Python reference manual, Centrum voor Wiskunde en Informatica Amsterdam.