CS230

Using Deep Learning to Estimate the Velocity of Robots in Simulation

Avidesh Marajh
Department of Computer Science
Stanford University
avidesh@stanford.edu

Ian Chang
Department of Computer Science
Stanford University
ianchang@stanford.edu

Zahra Albasri
Department of Computer Science
Stanford University
zalbasri@stanford.edu

Abstract

Velocity estimation is becoming increasingly important in autonomous vehicles and robotics. This paper explores a method for velocity estimation using only low fidelity camera inputs. We simulate our environment using Gazebo, an open source simulation tool, with an off-the-shelf TurtleBot3 as our simulated robot. Our convolutional neural network model estimates the relative velocity of TurtleBot3 with respect to different environments using a sequence of images. The main novelty in our approach is its reliance on vision only and its use of low fidelity simulated data. Our approach performs well when using images from one simulated world, but does not perform as well when testing on images from several simulated worlds.

1 Introduction

Velocity estimation is an important challenge in robotics and autonomous vehicles. Current systems use LiDAR which provides great accuracy, though it can be very costly. LiDAR also does not perform well in extreme weather conditions. Methods of measuring speed using vision that are widely employed are stationary cameras in streets. However, there is little research about vision-only velocity estimation using a non-stationary camera on the vehicle. Using vision to estimate velocity can provide additional data that can help a robot move more smoothly and in accordance to its desired behavior. Moreover, cameras are inexpensive and small compared to LiDAR or radar. Implementing a system that can effectively estimate velocity requires a large amount of data which can be expensive to obtain. In this paper, we present a novel approach to vision-only velocity estimation. We exploit images gathered from a simulated robot in different environments to train a convolutional neural network (CNN) model. Our CNN model takes 3 frames as an input and outputs a velocity estimate in meters per second.

2 Related work

Current approaches utilize LiDAR and radar techniques for velocity estimation and robot localization. While LiDAR can provide high accuracy of metrics like velocity, it is very expensive and prone

CS230: Deep Learning, Fall 2022, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

to malfunctioning under certain weather conditions [4][5]. On the other hand, radar is cheap but inaccurate and susceptible to noise.

Rotating Radio Frequency for Localization An attempt to improve upon radar methods uses rotating radio frequency identification. While more accurate than a simple radar system, this method is computationally expensive and not yet able to perform in real time [6].

Velocity Estimation with Camera and Radars An alternative approach uses a monocular camera along with a mmW radar to estimate velocity and improve performance [5]. This system uses a video matched with different radar readings; the data is passed through two convolutional layers and each iteration uses the same set of data. This system performs well without any need for calibration, though it could still be more costly than just using a camera alone for velocity estimation.

Vision-only Velocity Estimation There has been some interest in camera based velocity estimation. One method uses RGB video data that is passed through a DepthNet, a FlowNet and a Tracker that is then passed through an MLP. This approach is capable of performing in real time, but is less accurate with distances greater than 20 meters [4]. Another successful method is vision-only RatSLAM, a biologically inspired SLAM method [2]. This method uses the difference between images to estimate speed. However, RatSLAM has only been tested in simulation. Another vision-only SLAM uses low-cost cameras to estimate speed. This, however, method does not achieve real time performance [1].

The novelty of our approach is that it is vision-only and uses simulation generated data only. Unlike other approaches, our approach has a very modest computational cost, using a simple convolutional neural network and only three frames to estimate velocity.

3 Dataset and Features

We did not utilize an imported dataset. Rather, we generated our data using a simulated robot in a simulated environment. Our environment consists of a g4dn AWS Linux VM set up with a X11 Virtual Networking Copmuting server. We installed ROS Noetic and Gazebo. ROS is the Robot Operating System which defines a useful set of packages as well as a helpful pub/sub messaging system to operate our simulated robot with. Gazebo is a fully fledged robotic simulator that allows for the creation of simulated world environments and plugs in to ROS which allows us to operate the robot within our simulated world environments. We use a simulated robot called a turtlebot. It is a two wheeled robot that contains a virtual camera which can record the virtual environment as we travel through it. Our simulated robot travels down a one-dimensional path in a variety of virtual worlds. We have 5 virtual worlds varying in length between 10 m and 20 m. Each world contains a patterned path as well as a variety of objects off the side of the path for the robot to recognize. Below we display the 5 worlds in Fig. 1 that we created and ran our robot within. These are all original and specifically designed for our project.

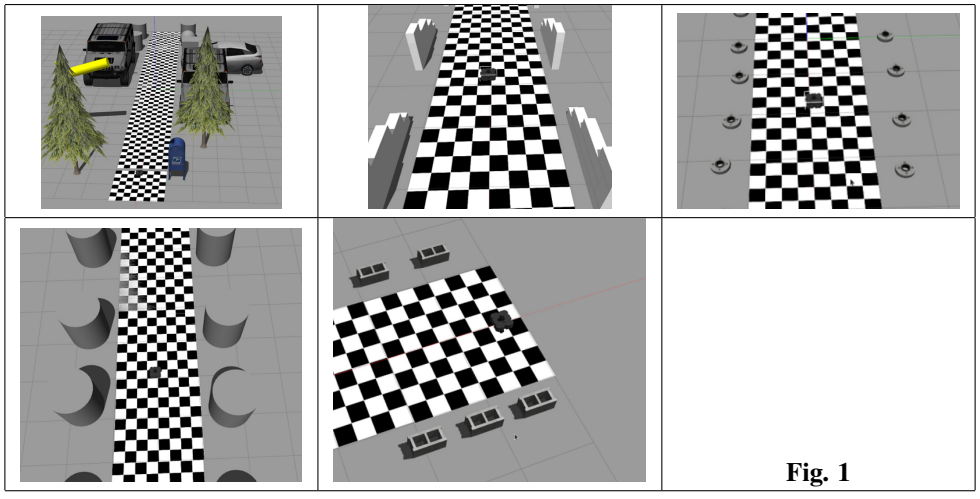

Fig. 1

The robot travels down the track at a speed of 0.04 m/s. Using ROS, we record the images coming out of the virtual simulated camera, and that is what creates our dataset. Below in Fig. 2 we display a sample images from the simulated virtual camera on the robot. As we see, these display the image the camera sees in the virtual world. We have an example image here from each of five worlds.

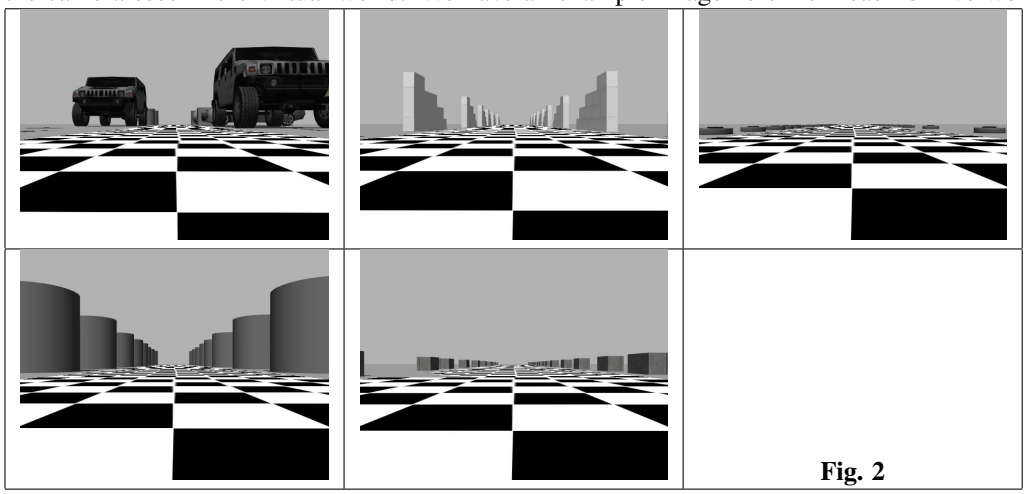

Fig. 2

Finally, because we know the total distance of the track we travel, we can speed up or slow down the speed we travel at using the captured images. Our dataset totalled around 30,000 images total, all generated through simulation with around 6,000 images for each world. Each image resolution is 480x640. This data is then used to generate artificial problems for our model by first sampling a speed between 0 and 5 m/s and then sampling a series of n images such that the 'distance traveled' between each image simulates the camera inputs over 1 second capture intervals if the robot were moving at the sampled speed. We experimented with various values of n ranging from 3-5. We also experimented with various capture intervals before settling on 1 second. Thus, our dataset in actuality gives us much more data then the 30,000 images would imply due to the sampling and artificial problem generation. We created an extremely large effective dataset as it involved sampling over a continuous problem space and gives a large number of possible problems. One of the five worlds is reserved for testing our model, while the other four were used for training giving us an 80/20 split. In order to create samples problems for training and testing we used the following methods. For a desired velocity v from a dataset with N images collected at a speed V over t seconds we find the number of frames,n, that would be elapsed in a single second travelling at speed v using: $n = \frac{Nv}{tV}$.
A starting frame, f_0, is then randomly sampled from the dataset and the frames that followed are taken as $f_i = f_0 + i * n$.

4 Methods

For this project, we performed initial experimentation on 2 supervised learning models with data collected from a single world. The fully shared model joins the n input images along their channel axis before feeding them into 2 convolutional layers (each with 'same' padding and stride 1). We experimented with n input images ranging from 3 to 5 input images before concluding that 3 worked best and was computationally most efficient. The first layer has a kernel size of 5 and doubles the input channels, the second layer has a kernel size of 3 and produces a single output channel. The outputs of these are then flattened and passed through 3 linear layers. All layers save for the final linear layer include a relu activation. Our second model is identical to the previous except that each of the n input images are first passed through a single individual convolutional layer before being joined to each other along the channel axis. They are then passed through a single shared convolutional layer before flattening and passed through a similar set of linear layers as in the previous model.

All experiments were done with a sampling rate of 1 frame per second and a 3 frame input to the model at every call. These models were trained using data from the simulation method described in the dataset section with an L1 loss between the model outputs and the speed label. L2 loss was also used for experimentation but converged to near identical points, as expected. All training was done with a random seed(23) over 10000 training problems. The outputs of the models were then evaluated

over 1000 testing problems, taking the mean absolute error. All testing was done with a random seed(212). Our initial experimentation concluded that the the fully shared model was superior and was used for all further tests. We expanded our model by including data from 3 additional worlds in training as well as 1 additional world to be used for testing. Additional improvements were made to the model including the use of dropout layers between linear layers, resulting in the final architecture as show in Fig 3 and the use of the Adam optimizer instead of SGD. Our findings immediately showed that the model was not adapting well to the new data distribution and was simply relying on a large bias and was therefore invariant to the input data. In order to reduce the load of the task, experiments were run where the range of data used was reduced to [0,1]. We also experimented with training the model in the [0,5] but with discrete samples rather than continuous.

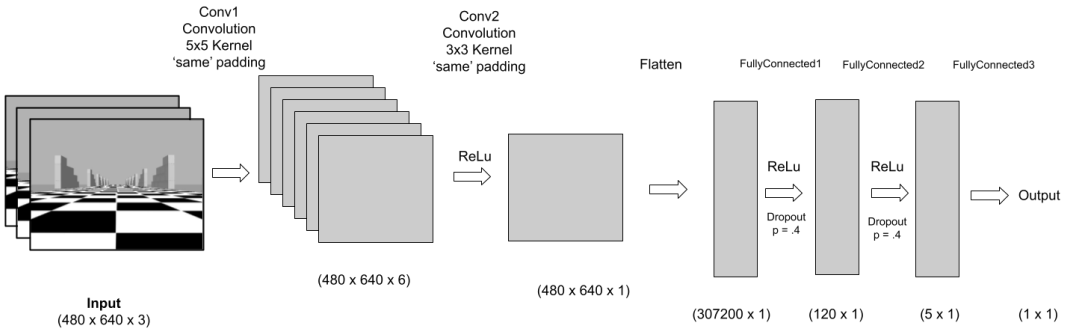

Fig. 3

5 Experiments/Results/Discussion

When testing on the first world, results show that the fully shared model outperformed with a testing error of 0.1155 and a training error of 0.2347. The other model showed a testing error of 0.2248 and training error of 0.2480. However, when using data from four worlds for training and using the data from the fifth world for testing, our model did not perform as well. When testing on the full range of velocities (0-5 m/s), our model shows high bias and poorly fits on both familiar and unfamiliar data. The model predicts 2.3966 m/s as the speed regardless of the actual speed. We see that when tasting on velocity ranges 0-1, there is a general trend that the higher the simulated velocity is, the larger the mean of the outputted velocity by our model is. However, our model still poorly fits and shows high bias for the 0-1 case. This could be attributed to having a relatively small dataset, imprecise data and simulating the ground truth. We chose to include 3 frames only to reduce computational load and overfitting. However, this choice of using only 3 frames for predicting the speed likely contributed to the low accuracy.

Model performance on training data with velocity range 0-1		
Simulated velocity	Model mean output velocity	Mean L1 loss
0	0.3546	0.3546
0.04	0.3931	0.3531
0.1	0.4005	0.3005
0.2	0.4093	0.2104
0.25	0.4070	0.1617
0.5	0.4896	0.0434
0.75	0.5449	0.2115
1	0.5471	0.4529

Model performance on testing data with velocity range 0-1		
Simulated velocity	Model mean output velocity	Mean L1 loss
0	0.4665	0.4665
0.04	0.4692	0.4292
0.1	0.4693	0.3693
0.2	0.4697	0.2697
0.25	0.4694	0.2194
0.5	0.4689	0.0311
0.75	0.4697	0.2803
1	0.4692	0.5308

Model performance on training data with velocity range 0-5 (using discrete samples)		
Simulated velocity	Model mean output velocity	Mean L1 loss
0	-0.1398	0.1467
0.04	0.5081	0.4819
1	1.1021	0.4749
2.5	1.4170	1.0830
3	1.3367	1.6633
4	1.3310	2.6690
5	1.3489	3.6511

Model performance on testing data with velocity range 0-5 (using discrete samples)		
Simulated velocity	Model mean output velocity	Mean L1 loss
0	-0.0557	0.1468
0.04	0.6859	0.6960
1	1.0846	0.4127
2.5	2.1395	0.6017
3	2.1981	0.9624
4	2.7337	1.3275
5	2.3462	2.6538

6 Conclusion/Future Work

We clearly had mixed results. We believe that this is due to a combination of factors. The simulation data was low fidelity and we probably needed to generate more. In addition, this is generally a difficult problem and though we tried a variety of data sampling methods none gave stellar results. In addition to varying the velocity sampling range, we also varied the number of frames sampled. We did not observe better results for sampling more frames and sampling more frames became computationally expensive due to how the data was encoded so we did not continue down that path. The most interesting observation we had is that the discrete data gave better results than the continuous data. We think the discrete sampling gave better results because it created a smaller data space that the model could learn better. If had more time and team members, we would create new simulation data with much higher fidelity and much greater quantity. We would also experiment more with sampling different velocity rates, different number of frames for the input, and different ways of generating our artifical problems. If these problems are able to be worked out and the model performs well for simulated images, performing transfer learning on real images would be the next step we would take.

7 Contributions

We all contributed equally. Avidesh Marajh worked on model architecture design, model coding, project ideation, and report writing. Ian Chang worked on AWS set up, robot simulation infrastructure creation, simulated images generation and collection, helped with model architecture design, project ideation, and report writing. Zahra Albasri worked on project ideation, simulated world generation, research on related work, research on novelty, and report writing.

References

[1] Burgard, W., Brock, O., & Stachniss, C. (2008). Mapping large loops with a single hand-held camera.

[2] Dall'Osto, Dominic, Hausler, Stephen, Jacobson, Adam, & Milford, Michael (2018) Automatic calibration of a biologically inspired neural network for robot SLAM. In Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2018). Australian Robotics and Automation Association (ARAA), Australia, pp. 1-10.

[3] M. J. Milford and G. F. Wyeth, "Mapping a Suburb With a Single Camera Using a Biologically Inspired SLAM System," in IEEE Transactions on Robotics, vol. 24, no. 5, pp. 1038-1053, Oct. 2008, doi: 10.1109/TRO.2008.2004520.

[4] Moritz Kampelmühler, Michael G. Müller, Christoph Feichtenhofer. Camera-based vehicle velocity estimation from monocular video. (2018).

[5] Pandya, A.; Jha, A.; Cenkeramaddi, L.R. A Velocity Estimation Technique for a Monocular Camera Using mmWave FMCW Radars. Electronics 2021, 10, 2397. https://doi.org/10.3390/ electronics10192397

[6] R. Zhao, Y. Zhang, G. Wang and D. Wang, "Mobile Robot Localization using Rotating Synthetic Aperture RFID," 2018 IEEE CSAA Guidance, Navigation and Control Conference (CGNCC), 2018, pp. 1-6, doi: 10.1109/GNCC42960.2018.9019177.

[7] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[8] Clark, A. (2015). Pillow (PIL Fork) Documentation. readthedocs. Retrieved from https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf

[9] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

[10] Stanford Artificial Intelligence Laboratory et al. (2018). Robotic Operating System. Retrieved from https://www.ros.org

[11] 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), 2004, pp. 2149-2154 vol.3, doi: 10.1109/IROS.2004.1389727.

[12] T. Foote, M. Wise, Robotis. (2019). Turtlebot3. Documentation Manual. Retreived from https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/