A Quick-Response Dynamic Obstacle Avoidance Strategy in Unknown Environments

1st Shreya Nahta
Department of Electronics and
Communication Engineering,
Nirma University,
Ahmedabad , India
21bec117@nirmauni.ac.in

2nd Jay Vadodariya
Department of Electronics and
Communication Engineering,
Nirma University,
Ahmedabad , India
21bec130@nirmauni.ac.in

3rd Vyom Mistry
Department of Electronics and
Communication Engineering,
Nirma University,
Ahmedabad , India
21bec139@nirmauni.ac.in

4th Sagar Ramani
Department of Electronics and
Communication Engineering,
Nirma University,
Ahmedabad , India
21bec102@nirmauni.ac.in

Abstract—One of the fundamental skills of intelligent mobile robots is obstacle avoidance. Due to the increasing diversity of the application environment, mobile robots must steer clear of barriers that are more broadly based. In order to satisfy this increased demand for generality, we devise a few-shot dynamic obstacle avoidance technique, taking advantage of the advancements in deep learning algorithms and mobile platforms in recent years. Finding the best Convolutional Neural Network (CNN) for a Deep Learning-based Self-Driving robot (JetBot) is the goal of this paper. The JetBot plans to utilise a fully trained CNN to drive over a fictitious road without assistance from a human. The CNNs utilised in this work are imported for training in a Reinforcement-Learning fashion via PyTorch. To reduce the error margin of the outcomes, the pre-trained models are imported. We apply this method to train a model, deploy it on the mobile robot, and conduct several obstacle avoidance recognition tests in an actual setting to confirm its efficacy. The outcomes of tests conducted on the mobile robot platform show good performance and support the plan we suggested.Apart from examining the experimental outcomes, the pros, cons, and prospective applications of the suggested approach as a tool for decision-making are also talked about.

Index Terms—Convolutional Neural Networks, NVIDIA Jet-Bot, Deep Learning, Self-Driving, few-shot learning, obstacle avoidance, mobile robot, unknown environment

I. INTRODUCTION

The fundamental need for mobile robots to navigate autonomously is the ability to avoid obstacles. As application situations become more complex, the dependability and universality of mobile robots' ability to avoid obstacles is rising steadily. Conventional obstacle avoidance algorithms, including the visibility graph and grid technique, perform well when dealing with known barriers, but they frequently fall short when dealing with unknown or dynamic obstacles. As a result, a number of sophisticated obstacle avoidance algorithms are created; the effectiveness of these algorithms is dependent on the geometry model's design. With the processor's upgrade and the use of numerous sophisticated algorithms in recent years, it becomes easier to carry out complex operations on the mobile robot platform.

Artificial Intelligence (AI) research has been a source of both excitement and concern since its founding in the 1950s. A widely discussed area of artificial intelligence

Fig. 1. The JetBot, fully built on top of its box

machine learning. This paper, however, delves deeper into a subfield called deep learning. Although the phrase "deep learning" was first used in the early 2000s, the principle has been around for decades. To put it simply, Deep Learning is the process of teaching a computerised brain how to function independently, enabling robots to complete challenging tasks. The idea of using deep learning to teach a robot how to drive itself is explored in this research.

We used a JetBot artificial intelligence (AI) kit with a NVIDIA Jetson Nano-based rearranged system. The fundamental components for these specifications are the Jetson Nano microcomputer board and the NVIDIA Jetson Nano Developer Kit, which is the packaged, two-wheeled car-shaped experimental system kit set developed and distributed by NVIDIA Inc. The aforementioned board is used in the

areas of executing diverse deep learnings, such as image data classifications, detecting indicated objects, automatic segmentation computing, recognizing diverse sound data including real human voice data, and parallel processing for some of the above.

Jetson Nano runs on a specific operating system (OS) called Linux4Tegra OS, and the program development environment is a Jetpack software development kit (SDK). The SDK contains the aforementioned OS, general library, application programming interface (API) files, developer tools, documents, demo-samples, Compute Unified Device Architecture (CUDA), and Robot Operating System (ROS). The CUDA Toolkit is a development environment to develop high-performance graphical processing unit (GPU)-accelerated applications. Developers can create and optimize diverse applications on GPU-accelerated systems. It can be used flexibly for desktop workstations, enterprise data centers, cloud-based platforms, and high-performance computing (HPC) supercomputers. CUDA sets include libraries, debugging and optimization tools, a compiler, and a runtime library to deploy applications. ROS is a set of software libraries to help developers build robot-based applications. It includes drivers, state-of-the-art algorithms, and powerful developer tools, and is open source.

We describe other specifications of the components of the JetBot-based system below. A small-sized, 8 MP 160 FOV, camera was installed with a resolution of 3280×2464 pixels. For the object-detecting sensor, we used a Sony IMX 219 sensor. For the microcomputer board, we used the widely available Jetson Nano board, an alternating current (AC) electrical resource set, an HDMI cable, and jumper pins, all distributed by NVIDIA Inc.

For data storage, we used a Sundisk 64 Gigabyte micro SD card. An organic light-emitting diode (OLED: small-sized monitor board) was mounted with a monitor size of 0.91 inch, 128×32 pixels, so the IP address, RAM, and battery life could be displayed. Two dual-mode wireless NIC AC8265 antennas, which are relatively high speed, were installed to connect the kit with Wi-Fi. This relatively compact and simple composition allows flexible redesign (e.g., attaching other sensors or IoT modules).

The NVIDIA Jetson Nano CPU, which enables effective neural network computation with low power consumption, is in charge of the self-driving robot, or JetBot. JetBots exhibit remarkable performance and are extensively employed in the development of self-driving systems. The JetBot is a well-liked robot for AI-based research since it runs on an open-source platform supported by NVIDIA. While the idea of employing deep learning to create self-driving automobiles is not new, this study aims to provide guidance for future research in this area.

II. BACKGROUND

The Jetson Nano board serves as the foundation and control system for the JetBot. Figure 1 shows the completed robot from a side view. Its two antennas poking out are its most distinctive features.

its chassis-sized wheels and the "eye" (a single front-facing camera) protrude from the back. The JetBot's two separate motors that power its wheels allow it to move around. The fully trained CNN will operate the motors to show off its ability to drive itself using the camera as input.

A diverse set of training data is essential for effectively training the JetBot, as well as any other robot that utilises Deep Learning. In order to meet this need, a dataset including 281 images and a collection of data points indicating the JetBot's intended path were generated using the camera of the robot. Figure 11 shows the view from the JetBot's camera and how it was used to collect the data. The open-source example code from NVIDIA served as the basis for the Python code utilised in this work.

```python
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)

image = widgets.Image(format='jpeg', width=224, height=224)

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)

display(image)
```
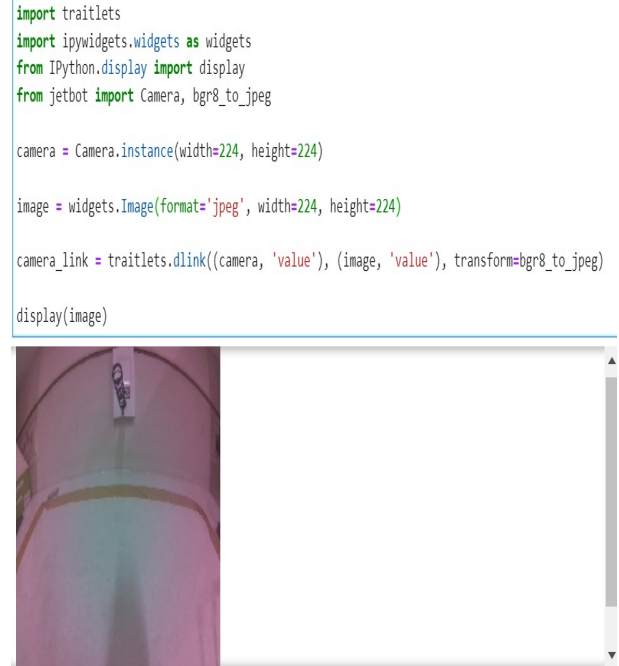

Fig. 2. The Camera Instance

Now that the motors are turned on and the software has been initialised, the JetBot is ready to use deep learning to avoid collisions while navigating. We train a model that can recognise and avoid barriers by taking pictures and using transfer learning techniques. The JetBot then uses this trained model to put it through a rigorous testing process in actual situations. Our goal is to improve the JetBot's collision avoidance skills and guarantee safe and effective navigation in a variety of contexts by means of this iterative training and testing procedure.

III. Introduction to components of jetbot

It is made up of various essential parts that come together to build a flexible and strong robot platform.

NVIDIA Jetson Nano: Designed for AI applications, the Jetson Nano is a compact, potent computer. It has a 128-core NVIDIA Maxwell GPU, 4GB of RAM, and a quad-core ARM Cortex-A57 CPU. The JetBot's brain is the Jetson Nano, which manages its motors, sensors, and AI algorithms.

DC Motors: DC motors are commonly used by JetBots for propulsion. The robot moves around thanks to these motors, which are attached to wheels or tracks. JetBot can traverse its surroundings by adjusting the motors' speed and direction.

Motor Driver: A motor driver is an electrical part that regulates a DC motor's speed and direction. It decodes the Jetson Nano's signals and transforms them into motor commands. Precise control over the robot's motion is guaranteed by the motor driver.

Camera: JetBot has a camera that takes pictures of its surroundings. Typically, it is a Raspberry Pi Camera Module or something similar. The Jetson Nano receives visual input from the camera to enable it to carry out functions including object detection, navigation, and surveillance.

Micro SD Card: The NVIDIA Jetson Nano's user programs, software libraries, configuration data, and operating system—typically NVIDIA JetPack OS—are all stored on the micro SD card. It functions as the Jetson Nano platform's main storage device. An SD card with a minimum size of 16GB is needed for the Jetson Nano; greater capacities can be used for more storage.

OLED (Organic Light-Emitting Diode) Display: An OLED display is a kind of display technology that, in response to an electric current, emits light using organic molecules. An OLED display can be utilized by JetBot to show real-time feedback, status updates, or user interface components. Being able to show text, images, or basic animations makes it an adaptable tool for communicating with the robot.

WiFi Dongle: TP-Link Archer T2U Nano: This USB WiFi dongle connects the Jetson Nano to a wireless network. One of the smallest, fastest WiFi adapters available is the TP-Link Archer T2U Nano, which also supports 802.11ac. For purposes like data transfer, cloud connectivity, and remote control, it permits the JetBot to establish connections with WiFi networks. The JetBot's WiFi functionality enables it to interact with other devices, browse the internet, and use cloud services for AI processing and data storage.

Sensors: In order to learn more about its surroundings, JetBot may be equipped with a number of sensors. Infrared sensors are commonly used for obstacle detection, ultrasonic sensors for measuring distance, and IMUs (Inertial Measurement Units) for detecting acceleration and direction.

Chassis: Robot's physical framework, or chassis, supports the motors, electronics, and other parts of the machine. JetBot chassis are available in a variety of sizes and shapes, from straightforward acrylic frames to intricate models with several attachment points and suspension systems.

Battery: JetBot normally draws its power from a rechargeable battery pack, which powers the robot's motors, sensors, and internal computer. The robot's runtime is determined by its battery capacity, which varies according on the particular use.

IV. Proposed System

Fig. 3. The Experimental Field

Our method for preventing collisions is enclosing the robot in a virtual "safety bubble". The robot can rotate in a circle inside this safety bubble without running into any obstacles or potentially hazardous scenarios like falling off a precipice.

Naturally, the robot's capabilities are restricted by what it can see, and we are powerless to stop objects from being

positioned behind the robot, among other things. However, we can stop the robot itself from going into these situations.

This is really easy for us to do:

The robot will first be physically placed in situations where its "safety bubble" is breached, marking those situations as prohibited. We store this label together with a picture of what the robot sees. Secondly, we will manually position the robot in situations where it is safe to advance a little and mark these as free. Similarly, we store a picture with this label.

have a large number of images and labels, we will send this data to a machine with a GPU so that we can train a neural network to determine whether an image the robot sees would cause its safety bubble to be breached. Ultimately, we'll use this to construct a basic collision avoidance behaviour.

Following are the steps to deployed :

Step 1: Gather Training Data

Step 2: Train Model

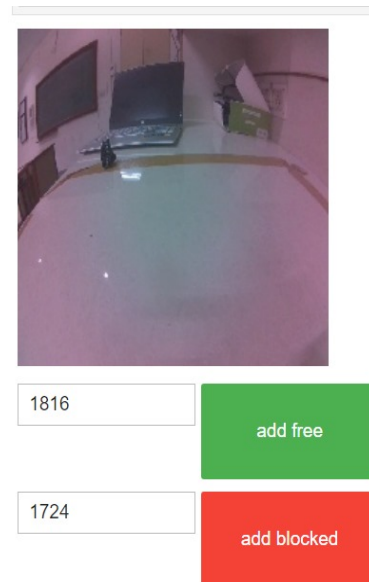Step 3: Model Deployment on JetBot

Fig. 5. Number of Free and Blocked Samples of the Trained Model

V. ALGORITHM

A. Gather Training Data

1) Initialize camera with specified width and height.
2) Create an image widget with JPEG format and specified dimensions.
3) Link the camera's value (image data) to the image widget using traitlets.
4) Display the image widget.
5) Import necessary libraries.
6) Define directories for storing images for "blocked" and "free" classes.
7) Try creating the directories, handle the case if they already exist.
8) Define button layout and create buttons for adding "free" and "blocked" images.
9) Display widgets for showing counts and buttons.
10) Define functions for saving snapshots:
    - Concatenate directory path with a unique filename generated using `uuid1()`.
    - Write the image data to the file in binary mode.
11) Define functions for saving "free" and "blocked" images, updating counts.

Fig. 4. Flowchart for part of the code used in the training of the CNNs

We will just be collecting data in this notebook. After we

12) Attach callbacks to buttons to trigger saving functions.
13) Display the image widget and button widgets again.
14) Stop the camera.


```
[*]:  NUM_EPOCHS = 30
      BEST_MODEL_PATH = 'best_model_resnet18.pth'
      best_accuracy = 0.0

      optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

      for epoch in range(NUM_EPOCHS):

          for images, labels in iter(train_loader):
              images = images.to(device)
              labels = labels.to(device)
              optimizer.zero_grad()
              outputs = model(images)
              loss = F.cross_entropy(outputs, labels)
              loss.backward()
              optimizer.step()
```
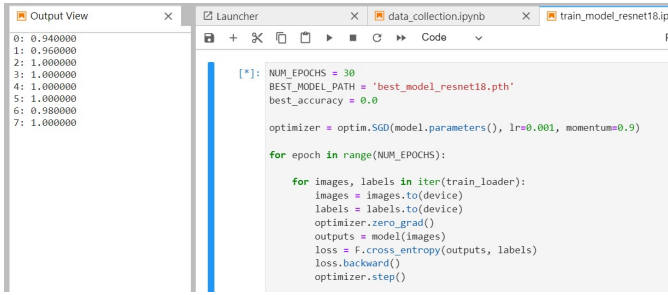
Fig. 6. The Deployment Method

B. Train Model

This algorithm outlines the key steps involved in training a ResNet-18 model for an image classification task using PyTorch.

1) Import necessary libraries including PyTorch for building and training neural networks.
2) Define a series of transformations to be applied to the images in the dataset. These transformations include color jitter, resizing, converting to tensor, and normalization.
3) Load the dataset using ImageFolder from torchvision.datasets, applying the defined transformations.
   - Split the dataset into training and testing sets using torch.utils.data.randomsplit.
   - Create data loaders for both training and testing datasets using torch.utils.data.DataLoader.
   - Load a pre-trained ResNet-18 model and modify the fully connected layer to output 2 classes.
4) Set the device to GPU if available, otherwise use CPU.
5) Define the optimizer (SGD) with specified learning rate and momentum.
6) Loop over each epoch. For each epoch, iterate over batches of training data, perform forward pass, compute loss, perform backward pass, and update model parameters.
7) After each epoch, evaluate the model on the test dataset. If the current model achieves better accuracy than the previous best model, save the current model parameters.

C. Deploy Method

1) Initialize device to CUDA.
2) Import necessary libraries.
3) Load the TensorRT optimized model.
4) Define preprocessing steps for input images.
5) Initialize camera and widgets for displaying images and controlling sliders.
6) Link camera input to image widget.
7) Display the camera image and sliders.
8) Initialize the robot.

9) Define an update function to process camera image changes: a. Preprocess the camera image. b. Pass the preprocessed image through the model to get predictions. c. Apply softmax function to normalize output. d. Extract probability of being blocked. e. Update the blocked_slider value with the probability. f. Control the robot's movement based on the probability of being blocked.
10) Call the update function with the initial camera image.
11) Observe changes in the camera image and call the update function accordingly.

VI. CONCLUSION

To carry out complex operations on the mobile robot platform This work presents a few-shot dynamic obstacle avoidance technique that allows mobile robots to swiftly adapt to unfamiliar environments. The strategy is based on monocular visual information.

You successfully completed the obstacle avoidance exercises in the actual world. The few-shot obstacle avoidance challenge can be successfully completed by our scheme, according to experimental data. In theory, this enables mobile robots to acquire human-like skills and learn much like people. However, the approach has drawbacks, like the incapacity to retain information acquired in various testing contexts, therefore it might not be the best choice for real-world scenarios. In the next research, we will examine how to get experience in every walking setting so that the phase of gathering support sets in different surroundings can be skipped.

REFERENCES

[1] S. Zhao, Z. Zhang, D. Xiao, and K. Xiao, "A Turning Model of Agricultural Robot Based on Acceleration Sensor," IFAC-Papers On Line, vol. 49, no. 16, pp. 445-450, January 2016.
[2] N. Dawar, and N. Kehtarnavaz, "Action detection and recognition in continuous action streams by deep learning-based sensing fusion," IEEE Sensors Journal, vol. 18, no. 23, pp. 9660-9668, September 2018.
[3] Jon Morss, Official website, "NVIDIA Jetson Nano: Collision Avoidance," https://www.element14.com/community/people/jomoenginer/blog/2019/05/14/nvidia-jetson-nano-collision-avoidance, May 2019.
[4] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 7263-7271, July 2017.
[5] Wu, S., Li, X., and Wang, X. "IoU-aware single-stage object detector for accurate localization," Image and Vision Computing, arXiv:1912.05992v4, April 2020.
[6] P. Shi and Y. Cui, "Dynamic path planning for mobile robot based on genetic algorithm in unknown environment," in 2010 Chinese control and decision conference. IEEE, 2010, pp. 4325–4329.
[7] L. Fei-Fei, R. Fergus, and P. Perona, "One-shot learning of object cate gories," IEEE transactions on pattern analysis and machine intelligence, vol. 28, no. 4, pp. 594–611, 2006.
[8] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017, pp. 1126–1135.
[9] W.-G. Han, S.-M. Baek, and T.-Y. Kuc, "Genetic algorithm based path planning and dynamic obstacle avoidance of mobile robots," in 1997 IEEE International Conference on Systems, Man, and Cybernetics, vol. 3. IEEE, 1997, pp. 2747–2751.
[10] B. Patle, G. B. L, A. Pandey, D. Parhi, and A. Jagadeesh, "A review: On path planning strategies for navigation of mobile robot," Defence Technology, vol. 15, no. 4, pp. 582– 606, 2019.