ABSTRACT

Machine learning is a subcategory of artificial intelligence involving the use of algorithmic retraining to analyze data. In the scope of this paper, machine learning was used to recognize humans using a NVIDIA Jetson TX1 and its built-in camera, with the overarching goal of simulating a self-driving car. Arguably, the most crucial function of a self-driving car is its ability to stop when detecting a pedestrian in its vicinity. Therefore, this project centered around using machine learning to program a robot, named Pedestrian Detecting Rover (PDR), to stop and beep once it recognizes a person in its path. The programmed rover stops at 306 centimeters, on average, in front of a person, but the stopping distance varies depending on the speed of the robot and the size and orientation of the person. For example, the robot stopped closer to the subject when they stood with their side facing the robot, as well as closer to the shorter subject than to the taller subject. The speed of the robot did not seem significant; while there was variation between the 300 mm/s trials and the 500 mm/s trials for each subject and orientation, this variation was irregular and the error bars overlapped often. The robot was found to be able to detect and stop for a suddenly appearing subject with considerable accuracy. Unexpectedly, it stopped farther away for the shorter subject than for the taller subject. Ultimately, while PDR worked as a basic model, many improvements must be made before the robot can be said to have the functionalities necessary for a fully autonomous self-driving car.

INTRODUCTION

From the Curiosity rover exploring the terrain on Mars to Marty the Robot alerting Stop & Shop customers of juice spills, the applications of artificial intelligence (AI) are endless (Figure 1). Along with self-driving vehicles, AI algorithms are used regularly for advertising, credit scoring, local policing, and various other purposes (1). On the road, driverless cars can increase traffic safety, potentially without the need for human intervention (2). In hospitals, computers trained to recognize magnetic resonance imagery (MRI) and optical coherence tomography (OCT) scans assist pathologists, helping to diagnose cancer and eye diseases (3–4). In space, robots act as “human proxies,” capable of studying geography, environmental conditions, and other parameters useful to scientists’ understanding of extraterrestrial worlds (5). While these applications may at first seem unrelated to each other, they all, like many modern AI technologies, implement machine learning.
Machine Learning

Used for the classification of inputs, machine learning refers to the computerized processing and analysis of data based on continual algorithmic retraining. In a world of abundant data, the large processing power of machines allows features of information, such as images or sounds, to be extracted and mathematically modeled by machine learning algorithms. The process of machine learning first involves a training period where labeled data is processed by the algorithm, which creates, adjusts, and fine-tunes a function to map the inputs to an output (e.g. a category or class). Supervised machine learning requires the algorithm to be provided with a pre-labeled training data set, while unsupervised machine learning does not require pre-labeled training data (6). In either case, the machine mimics the cognitive procedures that human learners perform, such as categorizing foreign objects, recognizing voices as belonging to familiar people, and learning languages.

Deep Learning

Deep learning is a subset of machine learning that involves layers of nodes, or neurons, connected by edges (analogous to synapses) to form a neural network, capable of processing unstructured data. Each input into the neural network triggers a series of activations in the following layer, a process that repeats through each layer in the network until the computer determines an output (Figure 2). This technique allows the computer to learn through its own errors, independent of human intervention to retrain the neural networks. The ultimate aim is for the computer to be able to extrapolate from training data and correctly classify new information.
Within a neural network, every node is associated with a bias and every edge is associated with a weight. To organize data from thousands of nodes and edges, matrices are employed for their usefulness in locating and iterating values. To visualize this process, for each layer of nodes $l$, the nodes are numbered from top to bottom. The bias of the $j$th node on the $l$th layer is assigned $b^l_j$. An entry in the weight matrix is similarly assigned to $w^l_{jk}$, where $k$ represents the other end of the edge, a node from the previous layer (7). The activation of a node can then be assigned by the following function, called the ReLU function (8):

$$ a_j^l = \max \left( 0, \sum_k w^l_{jk} a_{k}^{l-1} + b^l_j \right) $$

Given an input, each layer of a neural network recognizes certain patterns from the stimulus, increasing its specificity based on prior layers. To ensure that only certain neurons are activated by a stimulus, weights and biases (relative importance and threshold for activation, respectively) are assigned to each neural connection, allowing the computer to classify new data. The difference between the computerized values and the desired outputs is factored into the network’s loss, a function that takes into account the average disparity across all training examples. To improve classification accuracy and minimize the loss function, the neural network tweaks various parameters such as weights, biases, and activations from the previous layer. This facilitates the network with efficiently reducing loss without having to rewrite its entire structure, a significant advantage for a system with massive amounts of data (6).

**Backpropagation and Gradient Descent**

The process by which neural networks corrects their errors to improve accuracy is known as backpropagation. From the last layer to the first layer, weights and biases are adjusted to improve its performance. The objective function is the loss function, which needs to be minimized in order to maximize the rate at which the machine learning algorithm outputs the correct response (Figure 3). The loss function can be applied to any node as such:
Although both weights and biases factor into the value at each node, this example captures the underlying logic of backpropagation. Using the Chain Rule, which decomposes the relationship between two functions into multiple steps in between, the loss at node $e$ can be differentiated with respect to the loss at node $d$, which can then be differentiated with respect to the loss at node $b$ and $c$, all the way back to node $a$, which is the input layer.

The major mechanism of backpropagation is gradient descent. Gradient vectors generated through differentiation at a point allow for the locating of the local minima of the loss function at the output node. The graph below provides a visualization as to how this process works (Figure 4). As shown, there are many local minima in the graph and they are not equal to each other. By starting at an arbitrary point, or loss value, on the graph, knowing the gradient at that point would guide the algorithm to take a “step” downward, towards a local minimum. However, it is impossible to find the global minimum (which would be the ideal objective) without traversing through an uncountable number of points on the function, and the starting point(s) becomes critical to the performance. There are improvements to this issue, such as random starting points and stochastic gradient descent.

Convolutions

A Convolutional Neural Network (CNN) is a form of deep learning that is used for image recognition. CNNs utilize the loss function, backpropagation, and gradient descent just like
standard neural networks. However, they differ from regular neural networks in that they deal with processing images. A CNN treats a video as a stream of images. Each of these images is then treated as a matrix composed of pixel values. The neural network then applies a mathematical function to the matrix in order to detect features. This mathematical function is known as a convolution. In order to do a convolution, CNNs use a matrix called a kernel or a convolutional matrix to detect features in images. This kernel is compared to the matrix of the pixel values of an image (Figure 5). For example, assume the following kernel and pixel values of an image:

$$
\begin{bmatrix}
1 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 \\
\end{bmatrix}
$$

Kernel | Image

*Figure 5: Kernel and Image Matrices.* Image patch 1 is outlined in red. Image patch 2 is outlined in blue. The kernel will produce a high value for patch 1 and a lower value for patch 2, since patch 1 more closely resembles the kernel.

In order to apply convolution to the 3x3 kernel and the 3x4 image, a 3x3 patch must be extracted from the image. Then, each corresponding value of the kernel and the 3x3 patch will be multiplied. The products are then summed into one value. This process is also known as a dot product. If the patch of the image is very positive where the kernel is very positive, then the sum of the convolution will be larger. On the other hand, if the patch is very negative where the kernel is very positive, the sum of the convolution will be much lower. Therefore, a kernel’s values will correspond to the shape of the feature it is detecting. For example, the convolution between the kernel and patch 1 in *Figure 5* will return a high value of 9. On the other hand, the convolution between the kernel and the patch two in Figure 5 will be 5, because the patch has a weaker resemblance to the kernel.

The first convolution produces a higher value because the first patch was identical to the kernel, whereas the second patch produced a lower value because it is quite different from the kernel. When the kernel is applied to every patch on the image, the resulting values can be placed into a new matrix, which is called a feature map. This feature map highlights which parts of the original image contain the features that the kernel is searching for.

In order to apply convolutions to a neural network, every layer of the network will contain many kernels. In between convolution layers, CNNs typically utilize pooling in order to streamline the calculations. There are two common pooling operations: average pooling and maximum pooling. The pooling operator filter is usually a filter with 2x2 pixels that moves across a feature map, two pixels at a time. In average pooling, the filter outputs a value that is the average value of all the pixels within the filter. On the other hand, maximum pooling outputs a
value that is equal to the maximum value of the pixels within the filter (9). After pooling has been completed, the image maps from one layer of kernels will be inputted into the next layer of kernels. After the final layer, the features that were identified are used to determine what is in the original image.

**SELF-DRIVING IROBOT OVERVIEW**

Artificial intelligence is the driving force behind many cutting-edge technologies in today’s society, especially in areas concerning self-driving vehicles. These intelligent rovers are able to navigate seamlessly, reduce traffic-related injuries and fatalities, and provide newfound mobility for the disabled. Despite this, it seems that people still hesitate to surrender full control to artificial intelligence. A Gallup poll showed that only 9% of American adults were willing to utilize self-driving cars as soon as the safety concerns were resolved (10). What might not be clear to the average citizen are the implications of traffic control reaching even further than the apparent safety benefits from autonomous vehicles. By using reinforcement learning, researchers were able to improve traffic flow, even doubling the car’s average speed in certain instances. These benefits were especially highlighted when just a tenth of regular cars were replaced with self-driving ones in merging situations that usually cause serious traffic congestion (11).

Undoubtedly, self-driving cars still err, but machine learning can improve autonomous driving. This piqued the team members’ interests to simulate self-driving rovers using the iRobot® Create 2, an autonomous and hackable robotic vacuum cleaner.

The overall objective of this project was to program the Create 2 so that when it “sees” a person using the webcam, it beeps and stops immediately. The Create 2 received commands from the Jetson TX-1, which learned to detect humans using object detection algorithms that implement a pre-trained model capable of recognizing people. When there were no human beings in the Create 2’s line of sight, the robot was expected to advance straight, but stop and make a beeping noise as soon as it identified a person at an unsafe distance, determined by the programmer. The Jetson TX-1 and Jetson Nano computers were set up using software from GitHub, a website which provides software development and open source projects on the web. Once the Jetson TX-1 was trained to recognize humans using one of the models from GitHub, PDR was programmed using Python to take input from the Jetson TX-1 and adjust movement as needed. To analogize this situation with the actual connection between the human brain and body, the Jetson TX-1 acted as the visual processing component while the Create 2 served as the motor component (Figure 6).
Figure 6: PDR Constituent Dynamic. Overall, the Jetson TX1 as part of the PDR acts as the visual cortex and the Roomba acts as the motor cortex. The Jetson TX1 interprets information from the webcam. Through serial communication, the information from the visual cortex is communicated to the roomba.

METHODS

Jetson Nano

The Jetson Nano is a computer that runs neural networks for object detection. To be able to run the Nano, the Jetson Nano Developer Kit SD Card Image needed to be flashed onto the Nano’s SD card. This process was not necessary for the Jetson TX-1 as its SD card had already been flashed. The Nano was set up as a backup to the TX-1 in case newer image recognition software was incompatible with the older TX-1. It was also a backup in case the hardware of the TX-1 was not compatible with the Create 2 or Python 3 (which is available on the Nano and not the TX-1) needed to be used.

Jetson Nano vs. TX-1
Figure 7: Jetson TX-1 (left) and Jetson Nano (right). a. Ethernet port. b. HDMI port. c. USB port. d. Wifi antenna. e. Power jack. The TX-1 developer kit has approximate dimensions of 22 cm X 21 cm, and the Nano has dimensions of 7 cm X 4.5 cm. The Nano’s smaller size can make it very useful for some applications, but ultimately the additional processing power provided by the TX-1 was better able to accommodate real-time video processing.

It was important to have both the Nano and the TX-1 set up for redundancy in case of any unforeseen problems or hardware issues. However, only one system was necessary for the final product. Both the Nano and the TX-1 are relatively small supercomputers; however, the differences in their size and processing power makes them more suited to different applications (Figure 7). The TX-1’s greater processing power and larger size makes it more appropriate for computer vision, integrated deep learning, and graphics applications, although the size and cost may be prohibitive for some projects. By contrast, the Nano’s smaller size and lower cost makes it more appropriate for Internet of Things projects or for use by hobbyists. It can also run less resource intensive pre-trained machine learning models. The availability of these increasingly low-cost supercomputers has enabled many new and innovative machine learning projects to take place, since even with a low budget, people can afford the hardware necessary to run machine learning algorithms (12). However, it was decided that since this project involved handling real-time video processing, the Jetson TX-1 would be the best hardware to use with the Create 2 because of its more powerful processor.

Capabilities of the Jetson TX-1

The Jetson TX-1 is a powerful, while still fairly compact supercomputer. It runs on a Linux operating system, running Ubuntu, which comes with the Jetson Jetpack package provided by Nvidia. It also comes wifi enabled, and with a carrier board and wifi antenna, a TX-1 can be turned into a wireless hotspot, which other devices can connect to. Using this capability, it was possible to use SSH (Secure Shell) to run Bash commands on the Jetson from other devices without having to connect an ethernet wire, making the testing process much quicker and easier.

Jetson Inference

Jetson Inference is a Python/C++ package and guide to running deep-learning inference networks using the Jetson and NVIDIA TensorRT, a platform for deep learning inference. “Inference” refers to the phase in which an already-trained network uses its training to make inferences about new data. It comes pre-trained with three vision primitives--an image recognition network (ImageNet), object detection network (DetectNet), and semantic segmentation network (SegNet) (12).

Jetson Inference was downloaded and installed onto Jetson TX-1 and Jetson Nano. It was determined that DetectNet, an object detection algorithm, was most suitable for this program compared to ImageNet (image recognition), and SegNet (semantic segmentation--breaking an image down into component objects and detecting free space). Due to a bug in the DetectNet’s underlying C++ files (a misplaced line of code), the initial run of detectnet-camera.py was unsuccessful (error message: network not found) despite having downloaded every network available from the menu. After an update from the uploader, the package was successfully re-downloaded and installed. DetectNet had already been pre-trained with pedestrian pictures for human recognition. After detectnet-camera.py ran, the parameters of the detection boxes from
the terminal were located and were able to call and utilize them as parameters for the Create 2 movement. Then the Create 2 code and the detection code were combined, as they are both in Python 2.7. Most of the work was done on Jetson TX-1 and backed up on Nano.

The camera attached to the Jetson TX-1 takes in a video, which is then converted into a stream of images in RGBA format. These images are ran through the algorithm that stores recognized human figures in a list named detections where each item is a detection object. For example, if the length of the list detections is 0, then nothing has been detected. However, if the length of detections is ≥1, then 1 or more objects have been detected. Each detection object has many parameters, including the x-coordinates of the detection object in the image, the y-coordinates, the height, and the width. This information is used to create a blue box surrounding the detected person that the user can see around the detection of the human in a live-stream video representation. In order to decide whether PDR was close enough to the test subject to stop from moving towards the test subject, the area of the box with a constant threshold (80,000 pixels) was analyzed; if it is greater than 80,000 pixels, PDR stops moving because it is too close to the test subject.

This threshold was determined using the area instead of the height or width. The height of the box cannot provide a valid threshold because the height varies greatly. Once the test subject get closer to the camera, the box around the detection object significantly decreases in height. Oftentimes, if an individual’s entire body is not included in the camera’s frame, the algorithm recognizes only a part of the individual’s body as a human. Consequently, the length cannot be a reliable indicator of how close a subject is. The width, although it does vary consistently, also cannot provide a valid threshold on its own because the width of the test subject does not vary as significantly as height as the camera gets closer to the test subject. As a result, the area of the bounding box was used as a measure for the closeness of the subject, as area involves both height and width. Area still provides some difficulty if the person is too close to the camera since the detection box is very small. Therefore, the robot must stop within a certain range or the box will start to shrink and then become too small again.

**PDR Code**

To initially understand how to control the Create 2 remotely, a sample code titled the ‘Create® 2 Tethered Drive Using Python’ was used. This program permitted for control of the Create 2 using the arrow keys while it was connected to a computer running that program. This was a start, albeit far removed from the final product of this project. Through analyzing this program, a better understanding of how to connect to the Create 2 and how to send commands was obtained. The program allowed for the creation of a framework to simply “turn on” the Create 2 and have it drive or perform actions based on visual stimuli.

Since the Create 2’s coding was done in Python, pySerial, an application programming interface (API), was used to send commands to the Create 2 from the Jetson TX-1 via a communication cable. Using pySerial, the port that the Create 2 was connected to was specified in the code to establish a connection between the Create 2 and Jetson, enabling the robot to begin movement.
Controlling the Create 2 required using commands found in the Create 2 Open Interface. All commands in the Open Interface are associated with specific values, many of which require a set of parameters. After using pySerial to connect the Jetson TX-1 to the Create 2, a string was sent to the Create 2 with the command and other parameters. For example, to execute the command “Drive-Direct,” the command number (145) must be specified along with the velocities for the left and right wheels. These velocities need to be converted into 4 bytes which are later interpreted as 16-bit signed values using two’s complement. Thus, to move the Create 2 forward at a velocity of 300 mm/s, the program ran the following command:

```python
forwards300 = sendCommandASCII("145 145 01 44 01 44")
connection.write(forwards300)
```

To facilitate testing, the Create 2 was set in safe mode, meaning that the device would stop moving upon colliding into an object or being lifted off the ground. When the display window is open, the Jetson analyzes DetectNet input, searching for any human presence; if no person is detected within the camera’s frame, the Create 2 moves forward. If it detects a person and the area of the pixelated box surrounding the person is greater than 80,000 pixels, the Create 2 stops moving.

### Connecting Create 2 and Jetson TX-1

Next, the vision system needed to be connected to the Create 2 (Figure 8). The Jetson needed to be attached to the Create 2 so that the camera feed would be accurate in relation to the Create 2’s position and line of sight. However, the camera needed to be at a certain height in order to function; otherwise, the person would appear too tall for the screen and the camera would only detect the floor and the person’s shoes. Initial plans included connecting the Jetson TX-1 and a webcam onto a stand on the Create 2 with a USB cable. This would be the safest method of implementation, as the Jetson would not have to be elevated and thus had a much lower risk of a drop. However, the camera quality of the webcam was far inferior to the camera mounted on the Jetson TX-1. Perhaps more crucially, it had a much smaller field of view, which caused many errors; for example, people standing even slightly too close to the Create 2 would not be detected because their upper body and head would be outside the field of view. Since this would be incredibly dangerous for a self-driving vehicle whose purpose is to avoid pedestrians, the team decided to change tactics. The Jetson TX-1 itself was securely strapped to a stand made of cardboard boxes, which was then firmly attached to the Create 2 by tape. While slightly more risky, this method elevated the Jetson TX-1, allowing for the use of the higher quality Jetson camera.
The webcam takes in video that is then converted into a stream of RGBA images. Each image is then run through the machine learning neural network DetectNet. In DetectNet, objects in the image are classified. Detected objects are stored as detection objects in an array, and the contents of the array are analyzed by Python code. The code makes a decision based on the array contents, and using PySerial, it elicits a motor response in the rover.

**Testing Procedure**

Gaining a better understanding of PDR’s abilities in different circumstances required varying the robot’s speed, pedestrians’ height, and pedestrians’ orientation. Two test subjects of different heights (157 and 185 centimeters tall) initially stood at a distance of approximately 4.575 meters away from PDR. The robot, once released, would move forward at a speed of either 300 or 500 mm/s until it detected a person close enough to the camera, upon which PDR would stop and beep. For each test subject at each speed, three trials were conducted with PDR viewing the test subjects’ front, back, and side profiles. A control in which no one stood in front of PDR was added to check if the robot would continue moving forward in the case of no human presence (Figure 9).

Data was also collected to verify if PDR could detect a person suddenly appearing in its frame (Figure 13). To test this, the same two subjects held up a sheet of aluminum foil large enough to cover themselves and stood 458 centimeters away from PDR. Unable to recognize the person covered by foil at first (as desired), PDR moved forward for 153 centimeters until the test subject was instructed to remove the covering. The distance between when the subject removed the foil and when PDR stopped moving was measured in addition to the final distance between the subject and the motionless PDR. Altogether, these combinations of varying speeds, heights, and orientations resulted in a total of 46 trials.

<table>
<thead>
<tr>
<th>Subject Number</th>
<th>Height (cm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>157</td>
</tr>
<tr>
<td>2</td>
<td>185</td>
</tr>
</tbody>
</table>

*Table 1: Heights of the subjects.*
The straight track on which the testing occurred was a hallway with consistent lighting conditions and without any humans except for the test subject in the frame. During trials, the test subjects did not move from their initial distance away from PDR. The test subject had to be far enough from the camera in order to properly test the accuracy of the detection. It was possible that PDR would stop long before it was reasonably close to the test subject.

**Testing Modifications**

Actually testing the robot’s capabilities revealed that the network’s detection was not consistent, meaning that the x- and y-coordinates of the detection on the live feed would frequently change. This inconsistency affected the area of the detected person, causing PDR’s movement to falter, since the size of the detection box would only occasionally become large enough to reach the threshold. A re-evaluation of the code uncovered that the main issue was that the threshold area of the detection box, originally 130,000 pixels, was too large to make consistent detections. To reduce error, the threshold area was consequently lowered to 80,000 pixels. Another issue was that the test subject was not continuously detected. Not only did the detection box around the test subject flicker, but it also significantly decreased in length as PDR moved closer to the subject as parts of the body become only partially included in the detection. To address the problem, the code was adjusted so that PDR stopped moving as soon as the threshold area for the detection was reached.

Moreover, during testing, the Create 2 was observed to veer to its left while moving forward, insinuating that the wires attached to the robot for its functionality, including the power cord and the cable connecting the Jetson TX-1 and Create 2, were creating an unequal distribution of mass. Additionally, dragging the Jetson power cord on the left side was a force against the direction of movement. To counter this imbalance, extra weights were added to the right side of the Create 2, and the cords were picked up from the ground; these actions seemed to improve its ability to drive straight. When other problems with using the Create 2 were revealed during testing (such as the robot not being charged or not running the code), a second Create 2 of the same make and model was used to continue with the remaining trials.
RESULTS

Testing Phase 1

Figure 9: Diagram of Testing Phase 1. The rover is positioned at the starting line a fixed distance away from the test subject. The rover is allowed to move unguided in a straight line until it detects that the size of the test subject is larger than the threshold value and stops at (A). The stopping distance from the test subject to (A) is measured and recorded (B). This is repeated for 2 different subjects at 3 different orientations and 2 different rover speeds.
<table>
<thead>
<tr>
<th>Speed (mm/s)</th>
<th>Test Subject</th>
<th>Orientation</th>
<th>Average Stopping Distance (cm)</th>
<th>2x Standard Error of the Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Control</td>
<td>-</td>
<td>no stop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subject 1</td>
<td>Front</td>
<td>318.2</td>
<td>18.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Back</td>
<td>312.1</td>
<td>23.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Side</td>
<td>219.6</td>
<td>25.40</td>
</tr>
<tr>
<td></td>
<td>Subject 2</td>
<td>Front</td>
<td>373.1</td>
<td>27.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Back</td>
<td>374.1</td>
<td>8.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Side</td>
<td>282.6</td>
<td>127.83</td>
</tr>
<tr>
<td>500</td>
<td>Control</td>
<td>-</td>
<td>no stop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subject 1</td>
<td>Front</td>
<td>379.2</td>
<td>14.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Back</td>
<td>286.7</td>
<td>26.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Side</td>
<td>255.2</td>
<td>38.95</td>
</tr>
<tr>
<td></td>
<td>Subject 2</td>
<td>Front</td>
<td>353.8</td>
<td>25.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Back</td>
<td>367.0</td>
<td>15.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Side</td>
<td>153.5</td>
<td>26.67</td>
</tr>
</tbody>
</table>

*Table 2: Data for Phase 1 trials.* Average stopping distance from the subject as well as error (±2 SEM) is shown for each orientation and speed.
Figure 10: Average Stopping Distance from Subject 1 at Each Orientation and Speed. Error bars represent ±2 SEM. While the car stopped closer to the subject at 500 mm/s than at 300 mm/s for front and side orientations and farther for the back orientation, the differences are statistically significant only for the front orientation.

Figure 11: Average Distance from Subject 2 at Each Orientation and Speed. Error bars represent ±2 SEM. While the robot on average stopped closer from the subject at speed 500 mm/s, the error bars demonstrate that none of these differences are statistically significant. Notably, the trial at 300 mm/s with the subject oriented to the side had especially large error bars (SEM=41.06).
Figure 12: Average Distance for Each Subject Across All Speeds. Error bars represent ±2 SEM. While the car stopped closer to Subject 1 than Subject 2 for the front and back orientations and closer to Subject 2 than Subject 1 for the side orientation, this difference was only statistically significant for the back orientation.

Testing Phase 2

Figure 13: Testing Phase 2 Diagram. The rover is positioned at the starting line a fixed distance away from the test subject who is obstructed from view by a sheet. The rover is allowed to move unguided in a straight line; when it hits a certain point, the subject is told to drop the sheet. When the rover detects the subject and stops, the stopping distance (C) and distance from subject (D) are recorded.

“Average Stopping Distance” refers to how far PDR traveled from the point where the subject removed the covering to the point where the robot actually stopped. “Average Distance from Subject” refers to how far PDR was from the subject when it stopped. During this phase, the speed of PDR was kept constant at 500 mm/s.
<table>
<thead>
<tr>
<th>Subject</th>
<th>Average Stopping Distance (cm)</th>
<th>Average Distance from Subject (cm)</th>
<th>2x Standard Error of the Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject 1</td>
<td>19.32</td>
<td>285.7</td>
<td>5.379694332</td>
</tr>
<tr>
<td>Subject 2</td>
<td>40.67</td>
<td>264.3</td>
<td>12.36828381</td>
</tr>
</tbody>
</table>

*Table 3: Data for Phase 2 Trials.* Average stopping distance and distance from the subject as well as error (±2 SEM) for each subject.

![Average Stopping Distance and Distance from Suddenly Appearing Subject](image)

*Figure 14: Average Stopping Distance and Distance from Suddenly Appearing Subject.* Error bars represent ±2 SEM. Unexpectedly, the robot stopped farther from Subject 1 than Subject 2.

**ANALYSIS**

When set up correctly according to the testing guidelines, PDR successfully detected humans and stopped for all trials. While PDR was successful in stopping when detecting a human, there were some variations in the distance at which it stopped depending on the speed of the robot and the orientation and height of the subject. In application with driverless cars, this is very important as a pedestrian may be standing in any orientation, and the car must still be able to detect that it is a human and stop.

**Average Distance from Subject 1/Subject 2 (Figures 10-11)**

For the front and side trials for Subject 1, the stopping distance was shorter at 300 mm/s than at 500 mm/s, while the back trial had a shorter stopping distance at 500 mm/s. However, the error bars overlapped for all but the front orientation, suggesting that the difference is not statistically significant (Figure 11). For Subject 2, the stopping distance was shorter at 500 mm/s every trial, but for each orientation, the error bars overlapped (Figure 12). Since ±2 SEM indicates the variance between the experimental mean and the true population mean with about 95% certainty, this means that even though the experimental data may suggest that Subject 1 is detected sooner at 500 mm/s and Subject 2 was detected sooner at 300 mm/s, this may not hold true in reality. Therefore, one cannot conclude that speed significantly affects detection distance.
The large degree of variation seen can be explained by the small sample size as well as the inconsistency of the DetectNet system. When running DetectNet, the boundary boxes often flickered greatly from frame to frame, likely due to errors in the network’s ability to detect test subjects. This flickering meant it was very likely that the area of the boundary box was not exact to the dimensions of the person. When the person’s size is nearing the threshold value, this difference could result in premature or delayed stops causing increases in variability.

Comparison of Average Distance for Subject 1 and Subject 2 (Figure 12)

Subject 2's stopping distance was greater than Subject 1's because Subject 2 is the taller test subject. This most likely caused the detection box around Subject 2 to be greater than Subject 1’s, resulting in a greater stopping distance because Subject 2 fulfills the threshold at an earlier distance.

In the front and back orientations for each test subject, the area would be the same to the PDR. However, the stopping distances were different. This could have been because the neural network used on the TX-1, DetectNet, was exposed most often to front-facing human figures during its training. This also could be caused by the number of trials; more trials could decrease this disparity. The stopping distance for the side orientation is significantly lower than the other stopping distances most likely because the PDR’s camera perceives the test subjects as narrower on their side than when facing the PDR.

On the side orientation, Subject 2’s lower-than-expected stopping distance and large standard error are distinct. This makes sense because when the subject is oriented sideways, the area of the bounding box is much smaller due to the decreased width. Therefore, the robot must be much closer to the subject before the bounding box reaches the threshold area value. The irregularities could have appeared because the side view of humans are more prone to confusion with objects that the algorithm is trained to differentiate, such as poles. Additionally, the test subjects are not perfectly still. Likely, they are not in the same configuration perceived by the camera pixel-by-pixel. In every frame of the live video frame, the bounding box is changing. This all may have contributed to the large error. Furthermore, the lack of trials could have resulted in this error.

Average Stopping Distance and Distance from Suddenly Appearing Subject (Figure 14)

For these trials, PDR was relatively far from the test subject before it stopped. For both Subject 1 and 2, PDR traveled about less than 50 cm from the position it was at when the subject “appeared”. This demonstrates that PDR was able to distinguish between whether a human was present or not relatively quickly and stop. The robot stopped closer to Subject 2 than Subject 1, which is unexpected because Subject 2 was taller and thus should logically have been detected more quickly. This may be due to experimental error, which was likely since there were many moving parts—the command had to be given for the subject to drop the sheet when the robot reached a certain point, and the subject had to respond and drop the sheet. Notably, Subject 2 may have had increased difficulty removing the sheet compared to Subject 1 which may have resulted in a delayed response time and thus a longer period before the robot had an unobstructed view of the subject.
DISCUSSION

Before testing, it became apparent that there were many issues with the recognition software. Specifically, the software’s detections were not consistent, and the x- and y-coordinates of the detection would flicker and move. This was most likely because the live camera feed was converted to a stream of RGBA images, allowing the software to recognize any humans in each image created from the camera feed rather than a continuous recognition over time. The lack of continuity in recognition suggests the limitations of the image recognition software, DetectNet. For example, the software cannot continuously recognize a new object as the same object it last saw as the shape—or size—changes. This required that modifications be made to the code to start the testing over, such as changing the code so that PDR stops as soon as it recognizes a person of a certain size.

Additionally, the need to decrease the threshold area to recognize a person and stop PDR is indicative of the detection software’s inconsistency in recognizing test subjects between trials. When the threshold area was 80,000 pixels, the software, when recognizing a person, would only occasionally reach that threshold in subsequent tests using the same test subject under the exact same conditions. This demonstrates that DetectNet is not a reliable network to consistently recognize images in real-time. It especially would not be effective in self-driving cars.

Simulating a type of pedestrian recognition program on PDR was somewhat effective. Although PDR consistently stopped before hitting the test subject in every test trial, it did so by terminating the driving program every time it saw a human at a certain size. This is not an intrinsically successful model. Should a test subject move back and increase the distance between himself/herself and PDR, PDR will not respond under the current programming. Ideally, PDR should respond to this change in distance and move forward until the detection threshold is reached again. Instead of just stopping PDR, the code stops running. Additionally, the detections are not continuous and do not have a stable size. This is not ideal; the network should be able to consistently recognize humans if it is to be implemented in a safe and usable self-driving vehicle.

CONCLUSIONS

By running DetectNet on a Jetson TX1, interpreting the results using a Python program, and sending commands accordingly to a Create 2, a model “self driving car” can be constructed that stops upon detection of a human of a certain size. The car consistently stops under test conditions, and alterations to variables such as a test subject’s height or the car’s speed result in mostly insignificant differences in stopping distance. However, there are still many issues that need to be addressed before this rover can be considered a fully-functional model. In particular, if the subject moves too close to the machine and portions of their body fall outside the camera’s field of view, the boundary box decreases in size and ultimately disappears, preventing the robot from stopping. Ultimately, though many barriers remain to ensure continued performance across a wider range of situations, the robot was successful in simulating the basic safety functions required of a self-driving car.
REFERENCES


doi: 10.1126/sciCreate 2ics.aan5074


doi:10.1126/science.aaw0958

Appendix A: final version of the code implemented during testing

#!/usr/bin/python
# Copyright (c) 2019, NVIDIA CORPORATION. All rights reserved.
# Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation files
(the "Software"), to deal in the Software without restriction,
including without limitation the rights to use, copy, modify, merge,
publish, distribute, sublicense, and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:
# The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

import serial
import time
import jetson.inference
import jetson.utils
import argparse

# parse the command line
parser = argparse.ArgumentParser(description="Locate objects in a live
camera stream using an object detection DNN.",
formatter_class=argparse.RawTextHelpFormatter,
epilog=jetson.inference.detectNet.Usage())

parser.add_argument("--network", type=str, default="pednet",
help="pre-trained model to load, see below for options")
parser.add_argument("--threshold", type=float, default=0.5,
help="minimum detection threshold to use")
parser.add_argument("--camera", type=str, default="0",
help="index of the MIPI CSI camera to use (NULL for CSI camera 0)
or for VL42 cameras the /dev/video node to use.\nby default, MIPI CSI camera 0 will be used.")
parser.add_argument("--width", type=int, default=1280,
help="desired width of camera stream (default is 1280 pixels)")
parser.add_argument("--height", type=int, default=720,
help="desired height of camera stream (default is 720 pixels)")

opt, argv = parser.parse_known_args()
#make connection
port = '/dev/ttyUSB0'
connection = serial.Serial(port, baudrate=115200, timeout=1)
time.sleep(3)

# send commands
def sendCommandASCII(command):
    cmd = ""
    for v in command.split():
        cmd += chr(int(v))
    return cmd

# speeds
forwards100 = "145 00 100 00 100"
backwards100 = "145 255 156 255 156"
forwards300 = "145 01 44 01 44"
backwards300 = "145 254 212 254 212"
forwards500 = "145 01 244 01 244"
backwards500 = "145 254 12 254 12"

# commands
start = sendCommandASCII("128")
safe = sendCommandASCII("131")
beep = sendCommandASCII("140 3 1 64 16 141 3")
drive = sendCommandASCII(forwards500)
stop = sendCommandASCII("145 0 0 0 0")

# load the object detection network
net = jetson.inference.detectNet(opt.network, argv, opt.threshold)

# create the camera and display
camera = jetson.utils.gstCamera(opt.width, opt.height, opt.camera)
display = jetson.utils.glDisplay()

# process frames until user exits

connection.write(start)
connection.write(safe)
while display.IsOpen():
    # capture the image
    img, width, height = camera.CaptureRGBA()
    # detect objects in the image (with overlay)
    detections = net.Detect(img, width, height)

    # people in frame
    if len(detections) != 0:
        okayToDrive = True
        for detection in detections:
            area = float(detection.Area)
            # set threshold
            if area >= 80000:
                okayToDrive = False
                connection.write(stop)
                connection.write(beep)
connection.close()
raise SystemExit
if okayToDrive:
    connection.write(drive)
elif len(detections) == 0:
    connection.write(drive)

    # render the image
    display.RenderOnce(img, width, height)

    # update the title bar
    display.SetTitle("{:s} | Network {:0f} FPS".format(opt.network, 1000.0 / net.GetNetworkTime()))

    # synchronize with the GPU
    if len(detections) > 0:
        jetson.utils.cudaDeviceSynchronize()

    # print out performance info
    net.PrintProfilerTimes()
connection.write(stop)
connection.close()

**Appendix B: commands used from Create 2 Open Interface**

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Command Number</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>128</td>
<td>None</td>
</tr>
<tr>
<td>Safe</td>
<td>131</td>
<td>None</td>
</tr>
<tr>
<td>Drive Direct</td>
<td>145</td>
<td>Left and Right Wheel Velocity (as four bytes)</td>
</tr>
</tbody>
</table>

[9-23]