Real-Time Freespace Segmentation on Autonomous Robots for Detection of Obstacles and Drop-Offs

Anish Singhani
research@anishsinghani.com

***Abstract*—Mobile robots navigating in indoor and outdoor environments must be able to identify and avoid unsafe terrain. Although a significant amount of work has been done on the detection of standing obstacles (solid obstructions), not much work has been done on the detection of negative obstacles (e.g. dropoffs, ledges, downward stairs). We propose a method of terrain safety segmentation using deep convolutional networks. Our custom semantic segmentation architecture uses a single camera as input and creates a freespace map distinguishing safe terrain and obstacles. We then show how this freespace map can be used for real-time navigation on an indoor robot. The results show that our system generalizes well, is suitable for real-time operation, and runs at around 55 fps on a small indoor robot powered by the NVIDIA Jetson TX2 and its low-power embedded GPU.**

I. Introduction

Many small unmanned ground robots are being developed to perform tasks such as delivery, surveillance, household tasks, etc. These robots must navigate difficult terrain, requiring advanced perception capabilities. These robots use various sensors, including 2D and 3D LIDAR, RGB cameras, radar, ultrasonic, infrared sensors, and depth-sensing cameras. Most of these sensors, excluding LIDAR and RGB cameras, suffer from low resolution and/or short range. For this reason, most robots use a 2D or 3D LIDAR sensor for mapping and real-time obstacle avoidance.

LIDAR sensors use a scanning laser time-of-flight sensor to estimate distance extremely precisely. 2D (planar) LIDAR has high resolution and range, but can only detect opaque surfaces that intersect the plane made by the sweeping LIDAR beam. A lot of the work in the robotics field [1] [2] has focused on detection of obstacles using laser and visual sensors, but less work has been done on detecting drop-offs and other hazards, frequently referred to as negative obstacles [3]. These include ledges, staircases, and raised curbs, which are all examples of the absence of traversable ground. Most methods of detecting negative obstacles [4] require 3D LIDAR sensor. Although it can successfully can detect these obstacles, it is extremely costly. When used on outdoor robots, LIDAR sensors are also very sensitive to light refraction due to rainwater.

In this paper we use RGB cameras instead of laser-based sensors. These cameras have several advantages. They are cheap and small enough that they can be used on small, low-cost robots. For a much lower cost than a single LIDAR, it is possible to mount several cameras at different angles, providing a wider field of view [5] and redundancy, both of which are important for safety. Additionally, cameras are much more versatile. They can be used for many purposes, including object detection [6] and remote monitoring.

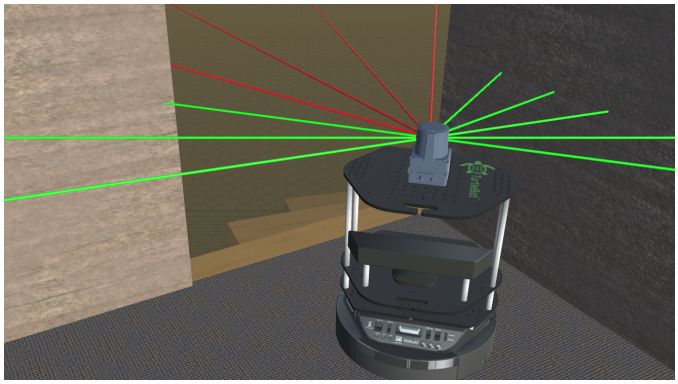


Fig. 1: While 2D laser-based sensors can detect standing obstacles (such as walls) very accurately (green laser beams), they fail to detect negative obstacles such as the downward staircase pictured behind the robot (red laser beams).

Using Convolutional Neural Networks, it is possible to detect and segment obstacles from free space in an image, to allow the robot to safely avoid obstacles. We define free space as regions in a camera image corresponding to terrain which a robot can safely drive on. Our approach aims to detect all types of obstacles, including those which cannot be detected by LIDAR: negative obstacles, ledges, overhangs, glass surfaces, etc.

II. Related Work

In the context of indoor and outdoor robot navigation, most development has been based on the use of LIDAR or depth-sensing cameras for localization and avoidance [7] [8]. These approaches create a pointcloud from data captured by the LIDAR or depth camera. This pointcloud is then used to build a costmap [9], which stores locations of obstacles and free space. Pathfinding algorithms can be used on this costmap to find a safe path for a robot to travel.

Segmentation has conventionally been approached in two different ways: instance segmentation [10] and pixel-wise semantic segmentation [11]. In instance segmentation, each object in an image is isolated and both a segmentation overlay and a bounding box is calculated. In pixel-wise segmentation, every pixel in an image is labeled with a classification. For our use-case of free space detection, we focus on pixel-wise semantic segmentation.

There are two main approaches to pixel-wise segmentation: encoder-decoder CNNs [11] and atrous convolution with bilinear upsampling [12]. The former uses convolutions and pooling layers to downsample an input image, then uses learned deconvolutions [13] and/or unpooling layers, along with skip connections to create an output image with the same resolution as the input and each pixel labeled with a classification. The latter uses atrous (dilated) convolutions to increase the effective receptive field of the convolutions while downsampling less than the encoder-decoder architecture, and uses simple bilinear upsampling on the output classifications to resize the output image to the same size as the input. Due to the atrous convolution's larger receptive field and ability to parse features at different scales, it has been much more successful at general-purpose semantic segmentation tasks than encoder-decoder architectures.

III. Approach

The main goal of our work is to allow an indoor robot to accurately detect free space using an RGB camera. We apply an end-to-end semantic segmentation model developed to be accurate while running in real-time on low-power embedded hardware. We then integrate the output of the segmentation into a real-time autonomous navigation stack running on an indoor robot.

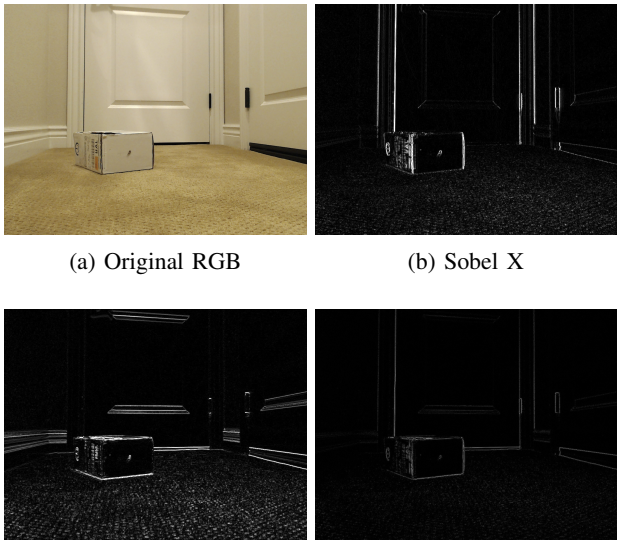
Our pipeline is separated into three steps. First, a frame is captured from the camera and simple edge detection techniques are applied. Next, the frame and detected edges are processed by our semantic segmentation neural network to produce a freespace map. Finally, this freespace map is converted to a 3D pointcloud, which is added to a navigation map to allow real-time navigation.

A. Input Preprocessing

Segmentation neural networks are generally trained with minimal domain-specific data preprocessing, to allow the optimizer to learn the best way to interpret the data by analyzing large amounts of labeled training data. However, this requires a large amount of labeled training data to ensure the learned parameters are robust to variations in the environment. In the case of semantic segmentation, the cost of labeling more images is high, a result of the precise nature of the necessary annotations.

To alleviate this problem and speed up the neural network's convergence, we add additional preprocessed channels to the network's input. These allow the network to more easily learn to refine edges of obstacles in the segmentation output by adding information about the target domain (i.e. physical edges of obstacles are generally accompanied by variations in color).

In order to calculate regions that are likely to be physical edges of obstacles, we use a form of edge detection based on computing the gradient of the image [14]. We compute the discrete Sobel [15] (1st order derivative) and Laplacian (2nd order derivative) gradients of the greyscale of the input image. These three gradients (Sobel X, Sobel Y, and Laplacian) are then combined into an image tensor, downsampled to 112x112 (half the input size), and used as an auxiliary input to the segmentation neural network.



(a) Original RGB (b) Sobel X

(c) Sobel Y (d) Laplacian

Fig. 2: The four inputs to the neural network. The main input contains the three colors channels in (a), while the auxiliary input contains (b), (c), and (d)

B. Network Architecture

The goal of semantic segmentation is to classify every pixel in an image into one of a set number of discrete classes [16]. For our use case, there are only two classes: free space and obstacles. Current state-of-the-art models, such as [17] [18], have millions of parameters that must be trained end-to-end, requiring massive amounts of labeled data and lengthy training time. Another problem with these larger architectures is that they often cannot run fast enough for real-time robot navigation.

Our specific task of segmenting free space (binary classification) requires much less contextual information than segmenting the multiclass [19] [20] datasets used as benchmarks for development of standard semantic segmentation models. As a result, we can design an architecture much simpler than the state-of-the-art, with the goal of achieving real-time performance on an embedded GPU and efficient training with a small dataset.

We propose an end-to-end semantic segmentation network that can accurately perform pixel-level segmentation of free space. We combine proven techniques from [17] [18] [21] [11] with our own research to create a robust neural network that can run at 55 fps on an embedded NVIDIA Jetson TX2's GPU and can be trained quickly with a only a small labeled dataset. Following is a detailed description of our model architecture.

Note: All convolutional layers use a 3x3 kernel, stride of 1, dilation rate of 1, and "same" padding (input size divided by stride is same as output size) unless otherwise noted.

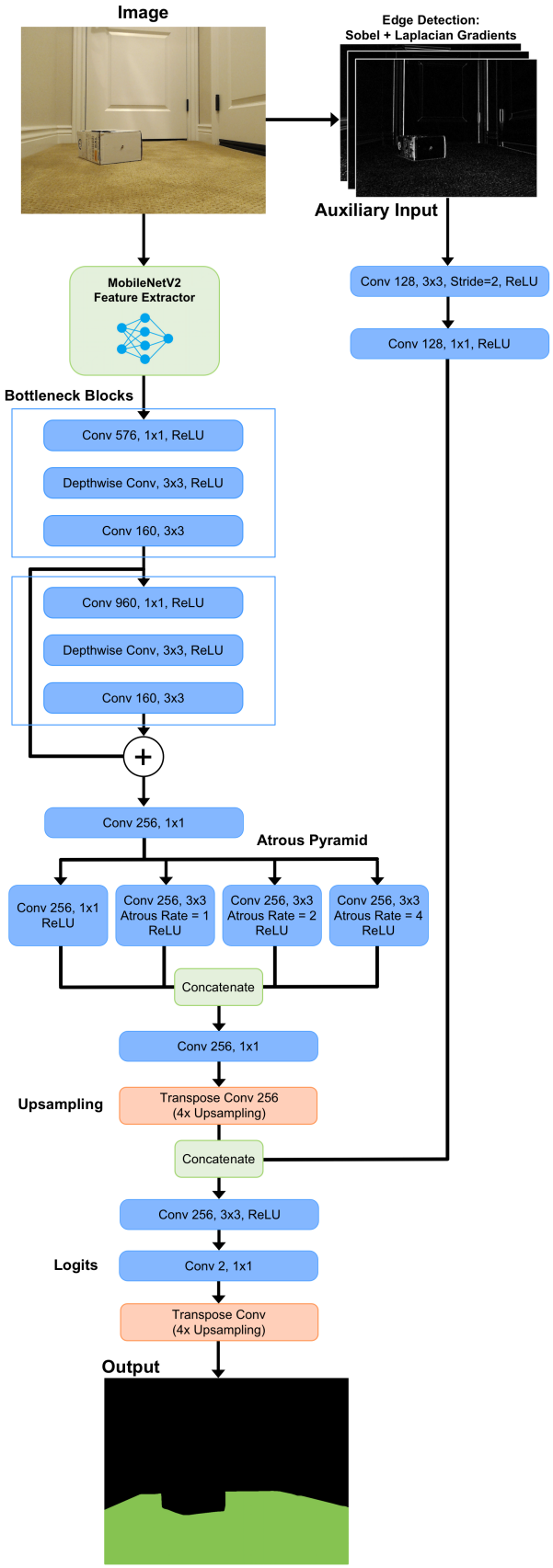
1) Input: The main input to the network is a 224x224 RGB image. The frame captured by the camera is undistorted, cropped (only the bottom 2/3 of the image is used because the upper 1/3 will never contain space above the horizon), resized to 224x224, then scaled linearly from values of [0, 255] to [-1.0, 1.0] to prepare it to be used as input to the neural network.

2) Feature Extractor: To improve training times, a pretrained classification model of MobileNetV2 [21] (width multiplier = 1.0) is used as the feature extractor for this network. Only the first 13 inverted residual bottleneck blocks from MobileNetV2 are used, with weights initialized by pretraining as on the ImageNet challenge dataset [20]. This is used as a method of transfer learning [22], which allows the feature extraction capabilities of the model to be learned in a controlled setting, and improves convergence when the full segmentation model is trained. The output of the feature extractor is a 14x14x96 tensor (1/16 of the input size)

3) Bottleneck Layers: Using the remaining bottleneck blocks from MobileNetV2 would reduce the output resolution further, so we replace them with two bottleneck blocks similar to those in [21], each having an expansion factor of 6 and 160 output filters. The addition of these blocks makes the network deeper and compensates for the layers removed from the end of the feature extractor. Finally, a 1x1 convolution with 256 filters is applied to the output of the second bottleneck layer.

4) Atrous Convolution Pyramid: To achieve more precise predictions while reducing the number of operations done by the feature extractor, we use an atrous pyramidal module inspired by [18] [23] to effectively capture information at multiple scales in the feature map. We use four convolutions: one 1x1 (with dilation rate 1) and three 3x3 (with dilation rates 1, 2, 4), each followed with by a ReLU activation. Batch Normalization [24] was not used here, as the batch size was small, and experiments showed that the network was able to converge successfully without use of Batch Norm. The outputs of these four convolutions are then concatenated along the channel axis, yielding a 14x14x1024 feature map. This is followed by a 1x1 convolution with 256 filters and a ReLU activation, to reduce the depth of the feature map back to 256.

5) Upsampling: We use the deconvolution (transpose convolution) layer to upsample the output of the atrous pyramid. The deconvolution has been shown to be extremely successful in upsampling for semantic segmentation in applications such as [11] We use a deconvolution layer with a kernel size of 8x8 and a stride of 4 to upsample the 14x14x256 output of the Atrous Convolution Pyramid to 56x56x256. We find that using deconvolution achieves better accuracy and faster convergence than the nonotrainable bilinear upsampling used in [17], at the cost of being slightly more computationally intensive.



6) Auxiliary Input: At this point in the network, the auxiliary input described in the previous section (Input Preprocessing) is added into the network. This input is in the form of a 112x112x3 feature map, scaled to [-1.0, 1.0], and includes the Sobel and Laplacian gradients of the image. The purpose of adding this input is to allow the network to fine-tune its predictions to known edges of objects. The image gradient has been shown [14] [15] to be highly effective in detecting edges of objects in an image. Because the edges of the freespace map (segmentation output) generally correspond with edges of objects, explicitly adding the image gradient at this point helps refine segmentation results at object boundaries. The image gradient is fed through a 3x3 (128 filters) convolution with stride=2, and then a 1x1 (128 filters) convolution with stride=1 to downsample it to 56x56x128. Each convolution includes a ReLU activation. The output of these convolutions is then concatenated to the output of the first part of the network, yielding a 56x56x384 feature map.

7) Logits and Output: Finally, to create the output logits, this feature map is fed through a 3x3 convolution with 256 filters and ReLU activation, followed by a 1x1 convolution with 2 filters, which forms the output logits layer for the network. Finally, similarly to [17], the logits are upsampled, again using a deconvolution layer. The output of this upsampling has a shape of 224x224x2, and is fed through a softmax activation to obtain a probability vector at each point on the image. In this vector, channel 0 represents obstacles and background while channel 1 represents free space. To create a single freespace map, the channel with the higher probability at each point is used as the segmentation output, yielding a 224x224 image of binary values (0 = background, 1 = free space).

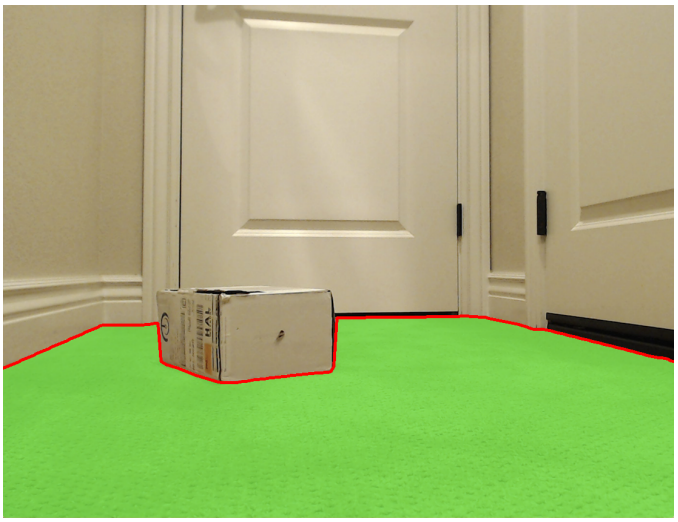


Fig. 3: The freespace map produced by the neural network, superimposed on the input image.

C. Inference Techniques

Freespace segmentation must be run in real-time on a low-cost embedded GPU to be useful for indoor robot pathfinding.

Real-time Inference: We use an NVIDIA Jetson TX2 SoC as our reference platform. It is low-cost, has a low power draw (15W), and has a small size and rugged carrier board, making it an ideal processing platform for the types of indoor and outdoor robots that our research targets. The goal of this neural network is to be able to run in real-time, which we define as at least 15fps, on a 200x200 or greater resolution input image.

TensorRT Optimization: The Jetson TX2, optimized for deep learning, includes hardware-level support for TensorRT, which can massively accelerate neural network performance. We are also able to use FP16 (16-bit floating point) compute, which halves the memory bandwidth required for inference, improving the runtime significantly. We take advantage of TensorRT to accelerate our network to maximum possible performance. In order to support TensorRT, all Relu6 operations (ReLU layers with a maximum value of 6.0) are replaced with the following expression:

$$\text{ReLU}(x) - \text{ReLU}(x - 6)$$

which is equivalent but supported natively by TensorRT. The use of TensorRT nearly triples our inference performance, which is a critical improvement for real-time use on a robot.

D. Navigation Stack Integration

To effectively use the freespace map outputted by the semantic segmentation network, it must be converted to a format usable by pathfinding and navigation algorithms to control a robot. We use ROS (Robot Operating System) [7] because it has SLAM mapping and localization capabilities built-in, as well as the ability to use arbitrary 3D pointclouds as input to the navigation algorithms [9]. We set up ROS and use its navigation stack to enable basic LIDAR-driven navigation capabilities on the robot. We then integrate our segmentation output into the costmap, which tracks obstacles and free space. In order to convert the 2D camera-perspective freespace map into a 3D pointcloud, we must use the following process:

1) Freespace Map Border Extraction: The output freespace map is a full-resolution image with many sharp edges and a large number of border points (border points are defined as points which lie along the edge between free space and obstacles). We propose three methods of extracting the most important border points from the image, and demonstrate optimal and sub-optimal cases for each:

(i) Vertical Line Projection: Vertical lines are drawn uniformly across the image. The lowest (closest to the camera) border point which intersects with each line is added to the pointcloud. The pointcloud density can be varied by the number of values of n to change the number of lines. This projection works well in many cases, but produces less precise results when an obstacle border is

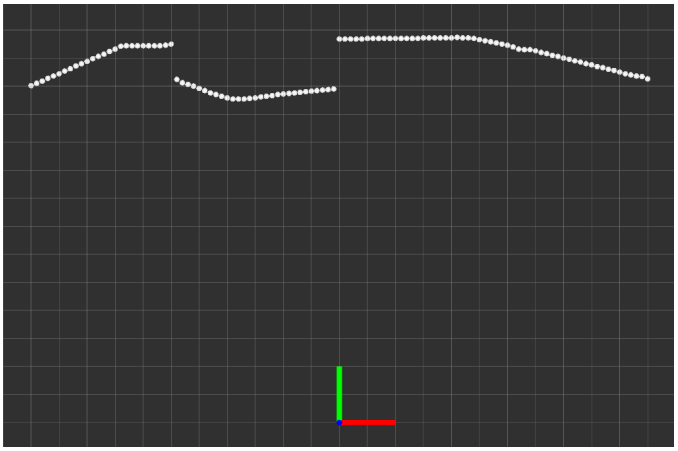
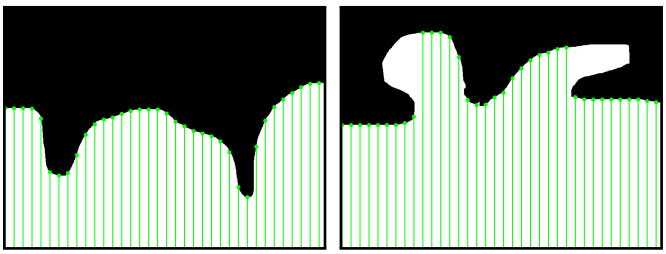


Fig. 4: The 3D pointcloud produced from the freespace map and used for real-time navigation.

between two vertical lines, which causes the line to miss most of the points on the border.
For integer values n distributed uniformly in the interval $[0, w)$, we define the projected line as:

$$a_n = \begin{cases} x = n \\ y = t \end{cases} \quad t \in [0, h) \cap \mathbb{Z}$$

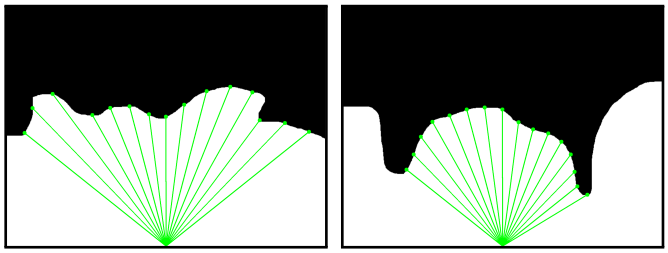


(a) Optimal Case (b) Sub-Optimal Case

Fig. 5: Examples of vertical line projection. Note how the selected points accurately represent the border in figure (a), while they do not capture two regions in figure (b).

(ii) Polar Projection: Lines are drawn from the point at the bottom-center of the image ($\frac{w}{2}$) extending outward. These lines are drawn at uniform angle increments, based on the equation below. The lowest border point that intersects each line is added to the pointcloud. The pointcloud density can be varied by changing the number of values n. This projection works well in many cases, including some of the vertical line projection's failure cases, but fails when an obstacle (not close to the center or edge of the image) is extending towards the robot.
For integer values n distributed uniformly in the interval $[0, \pi])$, we define the projected line as:

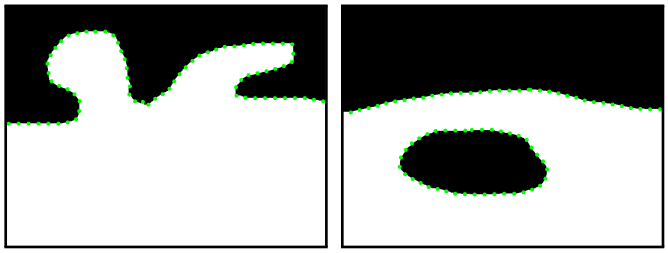
$$a_n = \begin{cases} x = \lfloor \frac{w}{2} - t \cos n \rfloor \\ y = \lfloor t \sin n \rfloor \end{cases} \quad t \in [0, \min(\frac{w}{2 \cos n}, \frac{h}{\sin n}))$$



(a) Optimal Case (b) Sub-Optimal Case

Fig. 6: Examples of polar projection. Note how the selected points cover the entire border in figure (a), but are cut off by the lowest obstacle points in figure (b).

(iii) Contour Extraction: Instead of using lines drawn from the bottom of the image, contours are extracted in the freespace map using methods from 17. Points along these contours are then used for the pointcloud. The density of the pointcloud can be controlled by selecting less points from the contours. This projection works well in most cases, but may produce improper pointclouds in cases where free space is visible both in front of and behind an obstacle.



(a) Optimal Case (b) Sub-Optimal Case

Fig. 7: Examples of contour extraction. Note how the contour points accurately match the borders in both images, but they may create a self-overlapping pointcloud in situations such as (b).

We find that each of these methods is useful in different scenarios, with the polar projection being the most successful in our trials with a navigation stack, most likely as a result of how it replicates the structure of commonly-used LIDAR data.

2) 3D Pointcloud Creation: After extracting the border points (where free space ends) on the 2D freespace map, these points must be projected to a 3D pointcloud in order to use them for robot navigation. We use the pinhole camera model to transform all 2D image-space points into 3D world-space points. By assuming that all border points are at ground level, we are able to determine the positions of all points in 3D and create a 3D pointcloud. These 3D points are then added to a ROS pointcloud data packet and added to the navigation costmap. This costmap, created by combining pointclouds and

robot odometry, can be used for safe robot navigation and obstacle avoidance.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

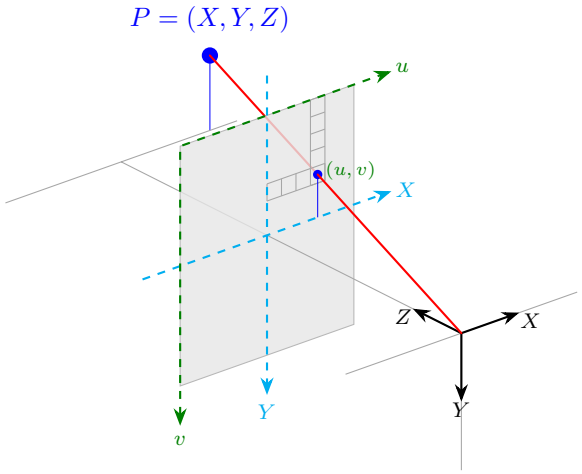


Fig. 8: A visualization of the pinhole camera model used to convert 2D image points to 3D pointcloud points.

3) Blur Detection: When a robot is moving at a higher velocity, the image produced by the camera is frequently blurred. We find that the output of the semantic segmentation is less accurate when using a blurred image. In order to prioritize less blurred (more accurate segmentation) images, we calculate the blur factor of the image and add this value to the pointcloud, such that the less blurred images have more effect on the final navigation costmap. To calculate the blur of an image we use the method proposed in [25]: the variance of the discrete Laplacian (second derivative gradient) of the image. The following equation shows how the blur factor β is calculated, given that the function $L(x, y)$ is a discrete convolution of the input image $I(x, y)$ where x and y are image coordinates:

$$L(x,y) = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} * I$$

$$\overline{L} = \frac{1}{XY} \sum_{x}^{X} \sum_{y}^{Y} |L(x,y)|$$

$$\beta = \sigma^2 = \sum_{x}^{X} \sum_{y}^{Y} (|L(x,y)| - \overline{L})^2$$

The scaling of β can be determined experimentally based on the camera intrinsics and image resolution. This β value can then be added as an intensity parameter in the pointcloud data packet.

IV. Experimental Evaluation

We implemented the entire approach described in this paper using the Google TensorFlow library, and OpenCV for preprocessing input data. Additionally, we use NVIDIA TensorRT to accelerate neural network inference on embedded platforms. We then test our approach using an iRobot Create 2 indoor robot with an RGB camera and NVIDIA Jetson TX2.

A. Training

1) Data Augmentation: In order to train the neural network to generalize from only a small training dataset, we apply data augmentation on the training data. We apply two types of augmentations to improve the generalization of the network. First, we randomly apply a small blur and/or brightness change to some of the images. This improves the network's performance on blurry images (which are often captured if the robot is moving at a high velocity), as well as making it more robust to different lighting conditions. Second, we apply affine transforms to the images. We apply random scaling, horizontal flip, translation, rotation, and shear transforms to the image. We fill edges of the transformed images by reflecting the original image, such that the image appears continuous instead of having sharp edges. When training, we use a generator-based approach. For each batch, images are randomly selected from the training dataset. These images (and their corresponding labels) are augmented randomly using our augmentation pipeline. This allows the generation of an infinite amount of randomly-augmented data, preventing the network from overfitting to the small dataset.

2) Training Setup: We train our neural network using Stochastic Gradient Descent using the Adam optimizer with binary crossentropy as a loss function. We use a variation of the "poly" learning rate decay policy proposed in [12]:

$$LR(\text{epoch}) = 0.0006 \cdot \left(\frac{1000 - \text{epoch}}{1000}\right)^{0.9}$$

The network is trained for 1000 epochs, after which we find no significant improvement in accuracy. Training takes about 15 hours on an NVIDIA GTX 1080Ti. To evaluate accuracy of the network output, we use the mIoU metric (mean intersection-over-union) averaged across our validation dataset.

B. Performance Evaluation

We evaluate performance using the mIOU metric with various configurations of our network and compare. We find that, although removing the atrous pyramid and/or auxiliary input reduces the number of FLOPS, it also lowers the accuracy. Our optimal performance is about 94.9% mIOU with all layers of the neural network enabled. Table I shows the performance of our network in a few different configurations (AP = Atrous Pyramid, Aux = Auxiliary Inputs) We find that this is enough accuracy for real-time navigation. To see examples of the network output, see the example images section at the end of this paper.

Resolution	AP	Aux	mIOU
224x224	✓	✓	**94.9%**
	✓		94.4%
		✓	93.7%
			93.5%

TABLE I: Performance of our neural network in different configurations

C. Rumtime Evaluation

We evaluate our neural network performance on a variety of computing platforms that can be used on embedded robots. We use the NVIDIA Jetson TX2 as our reference platform, and an NVIDIA GTX 1080Ti as a second testing platform (which can be used on larger robots such as those used outdoors). Note: all tests on the TX2 are done after setting it to MAX-N mode, which increases its max TDP to 15W and increases core clock speeds.

Our model runs at 18.9 fps on the Jetson TX2's GPU (standard TensorFlow). Optimizing with TensorRT nearly triples the network's performance to 55.6 fps. If the GPU is being used for other purposes (stereo matching, path generation, other neural networks, etc.), our network can run on the TX2's CPU at 3.6 fps, which is still viable for real-time navigation on slower-moving robots. If more precise segmentation is desired, our network can run with a 512x512 input size at 13.0 fps on the TX2's GPU. We also tested at higher resolutions using a GTX 1080Ti GPU, which can be used on larger robots.

Inference Platform	FP32		FP16	
	ms	fps	ms	fps
Jetson TX2 (TensorRT)	24	41.7	18	55.6
Jetson TX2 (GPU)	63	15.9	57	17.5
Jetson TX2 (CPU)	277	3.6	—	—
GTX 1080Ti	8	125.0	12	83.3

TABLE II: Runtime speed of the neural network at 224x224 resolution on different platforms

Inference Platform	224x224		512x512		768x768	
	ms	fps	ms	fps	ms	fps
Jetson TX2 (TensorRT) FP16	18	55.6	77	13.0	170	5.9
GTX 1080Ti FP32	8	125.0	21	47.6	38	26.3

TABLE III: Runtime speed of the neural network at different resolutions

V. Conclusion

In this paper, we presented an approach for detection of safe terrain on a robot solely from RGB camera data. We build and train a deep CNN that takes advantage of residual blocks, atrous convolutions, and specialized preprocessing to perform pixel-wise semantic segmentation of freespace in an image. The addition of auxiliary edge-detection inputs allows the network to perform better and makes it easier to train. We also successfully integrate the output of the neural network into a real-time navigation stack, allowing it to be used for robot pathfinding and obstacle avoidance. Our results show that the system generalizes well, is suitable for real-time operation, and runs at around 55fps on a low-power embedded GPU.

References

[1] I. Ulrich and I. R. Nourbakhsh, "Appearance-based obstacle detection with monocular color vision," in *AAAI/IAAI*, 2000.

[2] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. M. Oakley, M. Palatucci, V. R. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. R. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. V. Nefian, and P. Mahoney, "Stanley: The robot that won the darpa grand challenge," *AI Magazine*, vol. 27, pp. 69–82, 2006.

[3] N. Heckman, J.-F. Lalonde, N. Vandapel, and M. Hebert, "Potential negative obstacle detection by occlusion labeling," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, oct 2007.

[4] J. Larson and M. M. Trivedi, "Lidar based off-road negative obstacle detection and analysis," *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pp. 192–197, 2011.

[5] M. Brown and D. G. Lowe, "Automatic panoramic image stitching using invariant features," *International Journal of Computer Vision*, vol. 74, pp. 59–73, 2006.

[6] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[7] "Robot operating system." [Online]. Available: www.ros.org

[8] C. Rösmann, F. Hoffmann, and T. Bertram, "Planning of multiple robot trajectories in distinctive topologies," *2015 European Conference on Mobile Robots (ECMR)*, pp. 1–6, 2015.

[9] J. Fabro, R. Guimarães, A. Schneider de Oliveira, T. Becker, and V. Amilgar Brenner, "Ros navigation: Concepts and tutorial," pp. 121–160, 02 2016.

[10] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, "Mask r-cnn," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2980–2988, 2017.

[11] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440, 2015.

[12] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, pp. 834–848, 2018.

[13] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1520–1528, 2015.

[14] S. Gupta and S. G. Mazumdar, "Sobel edge detection algorithm," 2013.

[15] I. Sobel, "An isotropic 3x3 image gradient operator," *Presentation at Stanford A.I. Project 1968*, 02 2014.

[16] X. Liu, Z. Deng, and Y. Yang, "Recent progress in semantic image segmentation," *Artificial Intelligence Review*, jun 2018.

[17] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *ECCV*, 2018.

[18] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6230–6239, 2017.

[19] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3213–3223, 2016.

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, 2015.

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2018.
[22] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition," in *ICML*, 2014.
[23] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *CoRR*, vol. abs/1706.05587, 2017.
[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.
[25] J. L. Pech-Pacheco, G. Cristóbal, J. Chamorro-Martínez, and J. Fernández-Valdivia, "Diatom autofocusing in brightfield microscopy: a comparative study," in *ICPR*, 2000.

APPENDIX A: QUALITATIVE EVALUATION