AUTOMATED UNMAPPED FOREST PATH NAVIGATION OF MOBILE ROVER USING NEURAL NETWORKS

Nikola Jovičić

Union University, School of Computing
Knez Mihailova 6/VI, 11000 Belgrade, Serbia
E-mail: jovicicnikola@outlook.com

Abstract: In this paper, a system for autonomous navigation of unmapped forest paths in a simulation, along with a new simulator for testing and training it is presented. For navigation, it uses a combination of path planning and deep neural networks trained with imitation learning.
Keywords: imitation learning, forest, deep learning, simulation, path planning.

1. Introduction

Traversing forests and mapping them is a slow and arduous process for people. They would have to use and update a detailed map manually or with the help of a GPS device, and if they had to additionally collect some kind of data (humidity, temperature, road condition) it would make the effort even greater.

Although there are already systems that use neural networks to navigate forest paths, they are usually focused on drones [1]. Most drones do not have a flight autonomy of more than 30 minutes, so this system is designed to work with a mobile rover that runs on wheels or caterpillars. The mentioned drone systems do not have planning capabilities, so it is not possible to use them without human supervision. The system in this paper is fully automated. It can go to a given destination if it already knows the path to it, if it doesn't know the path it will try to find the fastest one. If a known path leads to an obstacle, the system will update the internal map with new information, stop using that path and try to find a new one.

2. Related Work

There are many approaches tackling forest trail following, but most of them are based on low flying aerial vehicles [1, 2]. There are not many complete systems published on forest trail exploration [3, 8]. Most don't do any autonomous exploration or planning and only deal with mapping [4], or mapping tree diameter [5] or forest restoration [6]. Ground vehicle forest path following is similar to the task of self-driving, therefore a neural network trained similar to [7] is used in this paper. Compared to similar solutions [8], this system does end-to-end path following without any global map, trajectory or previous knowledge of the environment, it uses data from only a single RGB camera for training and inference, and is easy to train, requiring only human driving data.

3. Simulator Overview

The first step in implementing this system is to find a suitable simulator. Required simulator features are:

- Realistic depiction of plants and trees
- Detailed camera setting
- Rover model and its manual control
- Ability to record location, control and camera
- Easy change of terrain, and driving routes

At the time of writing, no simulator meets at least half of these criteria, so it was necessary to create a new one. The simulator for this work was created in the "Unity3D" video game editor. Wherever possible, free plugins were used, for models of trees and plants, earth and sky textures, as well as for the rover itself, and the "ML-agents" package was used to communicate with the python. (Fig. 1.)

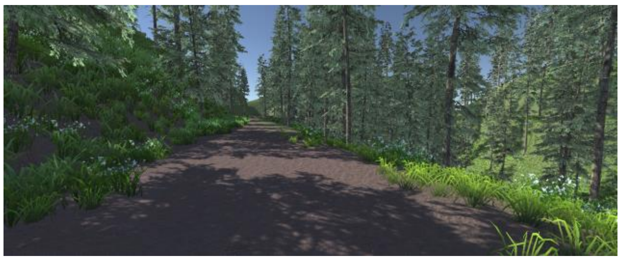

Fig. 1. First person view inside the simulator.

To build the terrain, the built-in terrain tool was used, in which two training maps for the rover were made (Fig. 2.), one smaller with narrow paths and one larger with spacious paths, along with a test map that had a mix of both. Obstacles can be spawned on the test map in the form of giant rocks at will.

Then the free "Vegetation Spawner" add-on was used, which allows trees and plants to be spawned on the map, but only in those parts that are not marked as a path. Then a simple vehicle that would move around the map is needed. The free add-on “Realistic Buggy Kit” was used for this, which offers several models of buggy vehicles. A camera was added to the vehicle, then it was scaled and inserted on the map as well as set so that it could be controlled outside the engine from the python with the help of the "ML-agents" package.

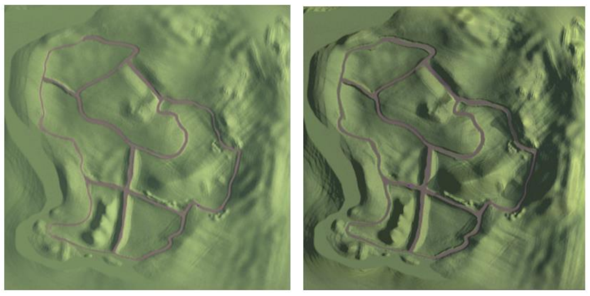

Fig. 2. Training maps, left is the large map and right is small, roads are brown, vegetation is green.

A game controller was used for control, through which it is possible to send commands for turning left and right, as well as for throttle. A controller was used instead of a keyboard because with it it’s possible to send analog commands, which enables more precise vehicle control and therefore better training data. Data collection was performed using a python script that was attached to the simulator. It sent control commands from the controller and in turn received information about the current location and image of what the vehicle's front camera sees, then it stored that information on the hard drive (Fig. 3.)

Around 1 hour of human driving was collected for training purposes. Driving data was collected randomly on the map, the vehicle was driven through each intersection at least several times so that each direction of the intersection was explored at least once in both directions. A lot of effort went into keeping the vehicle in the middle of the road while driving because the quality of the collected data will greatly affect the quality of the model.

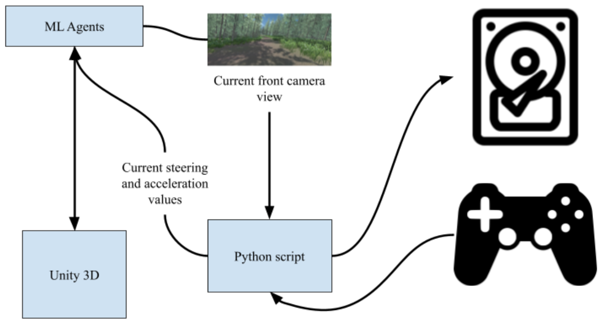
ML Agents
Current front camera view
Current steering and acceleration values
Unity 3D
Python script

Fig. 3. Dataset generation system.

4. Dataset Preprocessing

For the neural network to be trained, the data must first be processed into an appropriate format.

Steering wheel control is what the network should predict from the input image. It is saved as a number from -1 to 1, where -1 is the maximum left and +1 is the maximum right.

What distinguishes imitation learning [9] from supervised learning is the fact that the currently chosen action affects the next action of the policy, leading to a lack of adaptation to the test domain.

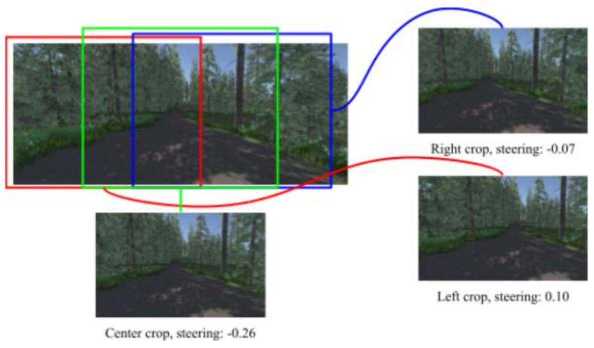
Right crop, steering: -0.07
Left crop, steering: 0.10
Center crop, steering: -0.26

Fig. 4. Image cropping with example image crops from the dataset.

To solve this problem, the collected images that are 1024px wide and 432px high need to be cut into three 700px wide images: left, right and central, and the steering wheel control for these images must be moved slightly to the right for the left image and slightly to the left for the right image (Fig. 4.). This way, if the camera is angled to the left or right of the path, the neural network will learn to correct for this behavior [7]. So that the network couldn’t learn to distinguish which is the left, right, and center image, the left, and right clippings are taken at random. The left image is cropped randomly from 0-137 pixels on the right while the right image is cropped by randomly from 0-137 pixels on the left, the steering wheel control is adjusted accordingly. Then the left, right, and center images (along with their steering values) are treated as separate elements of the dataset. They are then resized to 350x216px and normalized with (1).

$$I = I / 255 \qquad (1)$$

To control the neural network, it is necessary to tell it where it needs to turn when it comes to an intersection, left, right, or continue straight. The turn command is constantly sent to the network, even when it is not at the intersection, that way the system does not have to be careful when issuing turn commands, it is enough to command the network to turn at the next intersection and the network will turn when it reaches the intersection.

To make the network easier to train and later control, it is also trained to predict whether it is currently at an intersection, and that data is stored as 0 if not in an intersection and 1 if it is in an intersection. To increase the presence of intersections in the dataset, the data collected inside intersections is duplicated. This was done because intersections were only present in around 10% of the data.

5. Neural Network Architecture

The neural network was designed using the standard Resnet18 [10] architecture, which is commonly used to classify images (Fig. 5.).

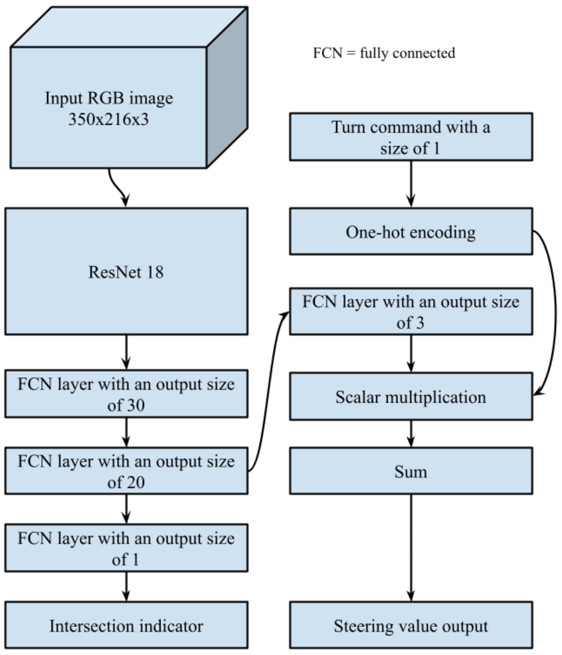
FCN = fully connected
Input RGB image 350x216x3
ResNet 18
FCN layer with an output size of 30
FCN layer with an output size of 20
FCN layer with an output size of 1
Intersection indicator
Turn command with a size of 1
One-hot encoding
FCN layer with an output size of 3
Scalar multiplication
Sum
Steering value output

Fig. 5. Neural network architecture.

The head has been replaced with a new head that supports an additional input for directional control and two outputs, one for steering control and the other for indicating whether it is currently at an intersection. Specifically, the last fully connected network layer is replaced with four new fully connected layers, the first has an output size of 30 numbers, the second is has an output size of 10 numbers, and the third and fourth are connected in parallel to the output of the second layer, where the third layer has an output size of 3 numbers and the fourth outputs 1 number. Thus, the third layer predicts the steering value and the fourth layer predicts whether the robot is currently at an intersection. The input for selecting the direction of the turn is one-hot encoded into zeros and ones of length 3 and scalar multiplied by the output of the third layer, the resulting sequence is summed and thus we get the steering value.

All fully connected layers use ReLu [11] activations except the last layers. The intersection indication layer uses the sigmoid and the pre-multiplication layer uses linear activations. The PyTorch [12] library was used for network definition and training.

An Adam [13] optimizer with a 0.001 initial learning rate was used for training, and a learning rate scheduler was also used, which halves the learning rate if the loss does not progress for 10 epochs. The batch size was 64. The best model was trained for 26 epochs on an NVIDIA 1080TI graphics card and it took around 2 hours.

6. Using the neural network for navigation

If such a network were to be used for navigation, the user would have to manually tell it where to turn at each intersection. Also, since the network does not have an internal state, it would not be able to distinguish whether it has already been at a location or if it has already discovered a shorter way to get there. Therefore, it is necessary to have a control system that allows the user to command the system with a target point, remember which locations have already been visited, as well as calculate the best way to get to a location. For the purpose of this paper, we used precise location data provided by the simulator, but in the real world, a visual odometry system, GPS, or a combination of these could be used [14]. The control system consists of three states: neural driving, automated driving, and turn around state (Fig. 6.).

6.1. Neural driving state

In this state, the neural network is used to control the vehicle. In order to reach the destination, turn commands are automatically given to the neural network. The command given depends on the angle to the goal, already visited locations, and vehicle orientation. To discourage the vehicle from going to already visited locations, the angles and distances between the vehicle and all visited points are calculated, the inverse distance values are saved in a histogram according to the calculated angles. Then a valley is found in the smoothed-out histogram that is closest in angle to the angle between the vehicle and the destination point. Thus, the vehicle will avoid already visited paths. Since the neural network needs to be told where to turn at the intersection, the obtained value angle of rotation is discretized to the values -1, 0, and 1 which corresponds to left, straight, and right.

This simple control scheme will not always find the fastest route, but it will take the vehicle to its destination at some point in time. While the vehicle is moving, the system saves the location and creates a graph of all the visited points. If the vehicle is in a location that is already saved in the graph, the current path will be connected to the already visited one. If it is in front of an obstacle, the vehicle enters the turn around state.

The algorhitm for this state can be described as (Alg. 1.)

```
Algorithm 1 Neural driving state
Input: goal coordinate G, current map graph M,
current image view from simulator I, current position
P, current rotation R, neural network N, simulator S
While P!=G: // While goal not reached
   rot_to_goal = rotation_between((P, R), G)
   all_distances = L2 norm(P-coordinates(M))
   inverse_distances = log2(map width – all_distances)
   all_rots = rotation_between((P, R), coordinates(M))
   hist = bincount(all_rots, weight=inverse_distances)
   smooth_hist = gaussian_filter1d(hist, sigma=6)
   extreme_locs = find_extreme_minima(smooth_hist)
   rot_to_goal = nearest(extreme_locs, rot_to_goal)
   command = 1 If rot_to_goal > 0.3,
2 If rot_to_goal < -3 Else 0
   I = preprocess_img(I)
   M.add_and_connect_node(P)
   predicted_steering = N(I, command)
   P, R, I, obstacle = S(predicted_steering)
   If obstacle: change_state(turn_around_state)
   End
End
```

6.2. Automated Driving State

Each time a new goal is issued, the system starts in this state. It will create a path between the current vehicle location and the point in the graph closest to the target point. It will move along the path until it reaches the end of the path or an obstacle. If it reaches the end of the path and fails to reach the goal point, it will go into the neural driving state. If it reaches an obstacle, it will break the graph at that point and go into the turn around state. The algorighm for this state can be described as (Alg. 2.)

```
Algorithm 2 Automated driving state
Input: goal coordinate G, current map graph M,
current position P, current rotation R, simulator S
While P!=G: // While goal not reached
   start_point = argmin(L2 norm(coordinates(M) – P))
   end_point = argmin(L2 norm(coordinates(M) – G))
   path = M.shortest_path(start_point, end_point)
   For point in path: // Follow saved path
      rot_to_target = rotation_between((P, R), point)
      P, R, I, obstacle = S(rot_to_target)
      If obstacle:
         change_state(turn_around_state)
   If P!=G: // Reached end of explored path
      change_state(neural_network_drive)
   End
End
```

6.3. Turn Around State

This state gives commands to turn the vehicle 180 degrees from the current direction of travel. When the U-turn is completed, the vehicle switches to the state it wasn't in before, if it was in the automated driving state it will switch to neural driving state, and vice-versa. This way the system will use the previously unused state to reach the obstacle.

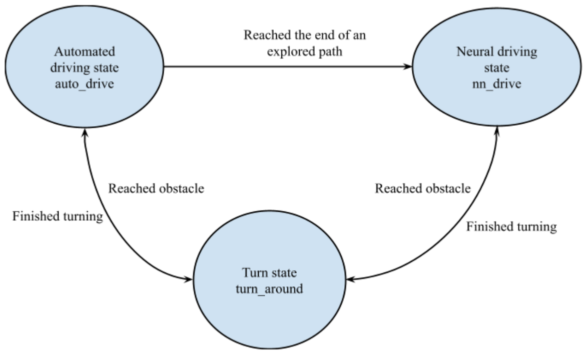
Automated driving state auto_drive
Reached the end of an explored path
Neural driving state nn_drive
Reached obstacle
Finished turning
Reached obstacle
Finished turning
Turn state turn_around

Fig. 6. Vehicle driving states

7. Experimental results

The rover was tested on a completely new map (Fig. 7.) that it had never seen. It has been tested on various configurations of obstacles and target points. The test cases are divided into three difficulties, easy, medium, and hard.

Easy cases include the rover going to a point on the map without any obstacles, medium cases involve going to a point with obstacles on the way, and difficult cases involve going to multiple points one after the other with obstacles that change their location to simulate forest paths changing over time. All test case results can be viewed at [15].

7.1. Easy Test Cases

For each of the cases from the figure (Fig. 7.), the vehicle managed to get to the desired location. Each case was run 10 times due to the randomness of the simulator and the neural network. Only for case number 2, the vehicle never managed to find the optimal route, for the others it either always followed the optimal route or more than half of the runs.

7.2. Medium Test Cases

As soon as obstacles are introduced the problem becomes more complicated, the vehicle can no longer use the fastest way to reach the destination but needs to explore the map to find the right solution (Fig. 8.). In a quarter of the test cases (16 out of 60), the vehicle failed to find its way to the destination, either due to moving around the map for too long or due to falling off the road. Only for cases 3 and 5 did the model manage to find its way to the goal each time. The vehicle has been tested 10 times for each case (Tab. 1.).

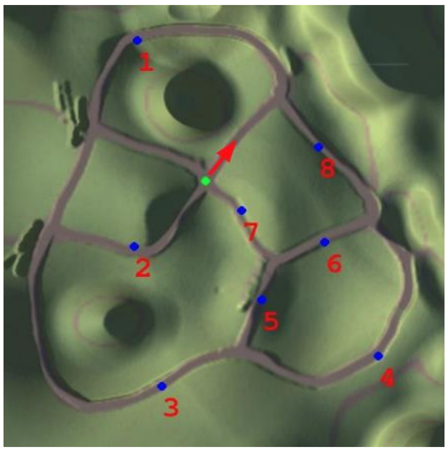

Fig. 7. Test map for easy test cases, starting position is the red arrow, forests are denoted in green and roads in brown, blue dots are target points, numbers indicate the test case number

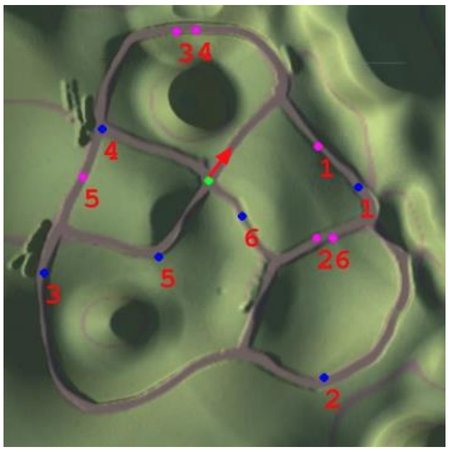

Fig. 8. Test map for medium test cases, pink dots are obstacles

Tab. 1.

Case number	1	2	3	4	5	6
Succes rate	30%	80%	100%	70%	100%	60%

7.3. Hard Test Cases

These cases test the entire system (Fig. 9.). From its ability to navigate with the help of a neural network to automated navigation when the map graph is created, as well as its behavior when obstacles appear on already explored parts of the map. Again, blue dots are targets and pink dots are obstacles, each test case is executed in two phases, "X.1" and "X.2". The system is run 20 times for each test case (Tab. 2.).

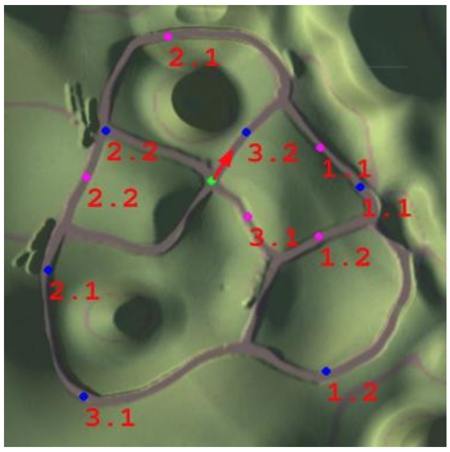

Fig. 9. Test map for hard test cases

Tab. 2.

Case number	1	2	3
Success rate of 1st phase	40%	85%	100%
Success rate of 2nd phase	40%	70%	100%

8. System Issues Discovered

In this chapter, we will address the most common system issues and possible ways to correct them.

8.1. Falling Off the Path

If the neural network fails to keep the vehicle on the path, then it can have bad consequences. The vehicle may collide with trees or rocks along the path or fall completely off of it.

One way to avoid this issue is to train a better neural network, using more data and/or a larger network. Another way is to use a better sensor, such as a depth camera, to give the network more information. With the current network, this issue is not common, but using the network in the real world would require a lot more training data.

8.2. Turn Around State

Since the vehicle cannot turn in place, a U-turn is implemented manually, this state doesn't observe obstacles or where the edge of the path is. Therefore, if the road is narrow, the vehicle may fall off or hit an obstacle or a tree next to the road.

The easiest way to solve this would be to equip the vehicle with a single-channel LIDAR sensor with a 360-degree view around the vehicle, so the vehicle could turn around in safely.

8.3. Missing a Possible Turn

Sometimes it happens that the network has an instruction to turn, but does not notice that a turn is possible and therefore does not execute the instruction. This issue could be avoided by using a camera that has a wider angle, making it easier for the network to see that turning is possible.

9. Conclusion

In the field of research of robotic vehicles on four wheels, this paper has contributed a realistic simulator of a forest environment, as well as a system that successfully solves the scenarios of the simulator. The system was tested in detail in the simulator on different obstacle and target point scenarios. With the help of the created system, it was shown how it is possible to achieve simple mapping of forest roads with a combination of classical methods of route planning and methods of neural networks. The paper describes the problems of the system as well as their potential solutions. Future work would include using this system in the real world. In order to do that, it would be necessary to train the existing neural network with images from the real world and add an odometry system. The rest of the system would remain the same.

10. References

[1] Giusti A. et al.: "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots," in IEEE Robotics and Automation Letters, vol. 1, no. 2, pp. 661-667, July 2016.
[2] Smolyanskiy N. et al.: “Toward Low-Flying Autonomous MAV Trail Navigation using Deep Neural Networks for Environmental Awareness”, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4241-4247, 2017.
[3] Ghamry, et al.: “Cooperative Forest Monitoring and Fire Detection Using a Team of UAVs-UGVs”, in ICUAS, 2016.
[4] Pierzchała M. et al.: “Mapping forests using an unmanned ground vehicle with 3D LiDAR and graph-SLAM”, in Computers and Electronics in Agriculture, Volume 145, 2018.
[5] Chisholm R. et al.: “UAV LiDAR for below-canopy forest surveys”, in Journal of Unmanned Vehicle Systems 01(01):61-68, December 2013.
[6] Mohan M. et al.: “UAV-Supported Forest Regeneration: Current Trends, Challenges and Implications”, in Remote Sensing, 2021.
[7] Bojarski M. et al “End to End Learning for Self-Driving Cars”, in arXiv preprint arXiv:1604.07316, 2016.
[8] Grigorescu S. “Embedded Vision for Self-Driving on Forest Roads”. on CVPR, Workshop on Embedded Vision, 2021.
[9] Osa T. et al “An algorithmic perspective on imitation learning”, in Foundations and Trends in Robotics, 2018.
[10] He K et al “Deep residual learning for image recognition”, 2015.
[11] Agarap, A. F. “Deep Learning using Rectified Linear Units (ReLU)”, 2018.
[12] Paszk A. et al “PyTorch: An Imperative Style, High-Performance Deep Learning Library”, 2019.
[13] Diederik P. K., Ba J. “Adam: A Method for Stochastic Optimization”, in 3rd International Conference for Learning Representations, 2015.
[14] Wang K. et al “Approaches, Challenges, and Applications for Deep Visual Odometry: Toward to Complicated and Emerging Areas”, in IEEE Transactions on Cognitive and Developmental Systems, 2020.
[15] Jovičić N.: “Automated unmapped forest path navigation of mobile rover using neural networks”, video file, November 2021, retrieved from https://youtu.be/nr6jNAzg1TQ