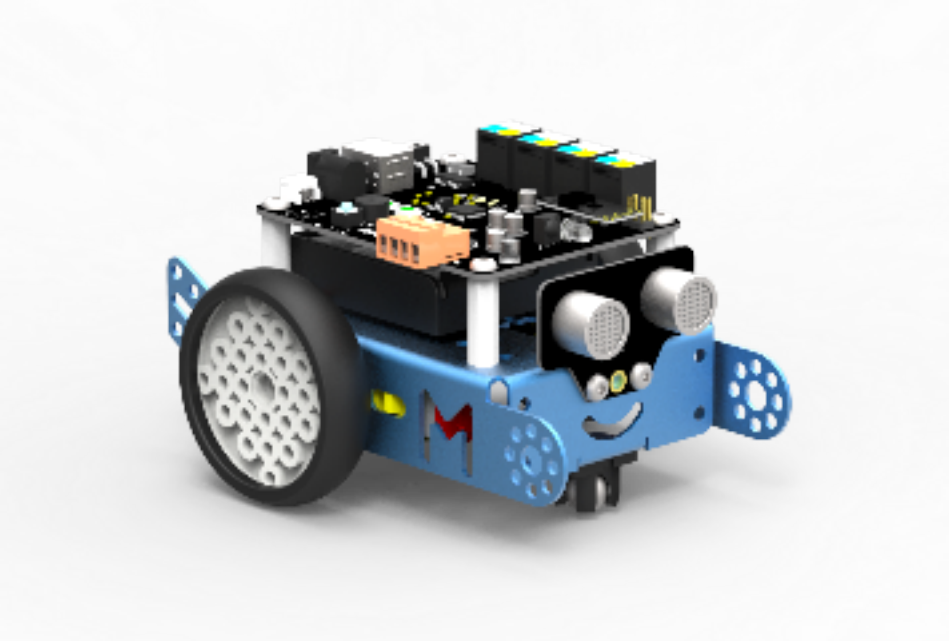
Robot Final Demonstration

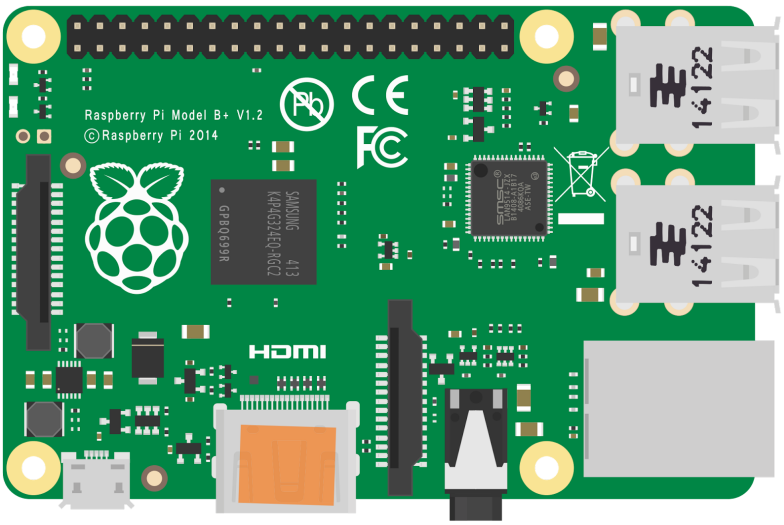
05.05.2017

Team Name: Raspberry Bot

Team Members: Stanislav Minev, Ivaylo Nenovski

Code and documentation presented by Ivaylo Nenovski





Executive Summary

The goal of the project was to classify different shapes with different colors that the Raspberry bot encounters while roaming and to sort them into different locations in the environment. In order to do that we constructed our own prototype of the Raspberry bot by installing a Raspberry Pi due to its high extensibility and a Raspberry Pi camera. The camera allowed us to take pictures of the objects that we encountered, while the Raspberry Pi allowed us to run a neural network on our robot that would classify those images and determine where they should be put.

On the other hand, the Raspberry bot came in with Arduino, which was used to control the movement of the robot and to interact with its manipulators. The Raspberry bot also came in with an ultrasonic sensor, which was used to detect how far objects were so that the robot can stop accordingly and take pictures. The communication between the Arduino and the Raspberry Pi was done via USB. In simplest terms, whenever an object was closer than 20 cm the arduino would stop the motors and send a signal to the Raspberry pi. Upon receival, the Pi would take a picture of the object and feed it to the neural network. Once the neural network was done classifying the object, it would send a message back to the Pi indicating what the object was.

Of course, some electrical modifications on the robot were required as well. In order to be able to fit all of that onto a robot of this size we had to make some design changes to the Raspberry bot. We decided to place the batteries on top of the Arduino shell and place the Raspberry pi underneath the Arduino, where the batteries were initially located. Moreover, we used a second battery for the Raspberry Pi in order to power up the Pi. The camera was placed on the front of the robot, sitting on the line detection sensor, but placed in such a way so that it would not interfere with the ultrasonic sensor.

Due to the lack of sensor options in the Raspberry bot, we extensively used the line following sensor for our control laws. We also had to create an environment in which the line following sensor could be used as it would not work in normal room conditions (the floor had to be black). For that purpose we printed black pieces of paper and stitched them together using tape. Furthermore, we handcrafted our shapes by printing them, cutting them and building them together.

Approach

Assembling the robot

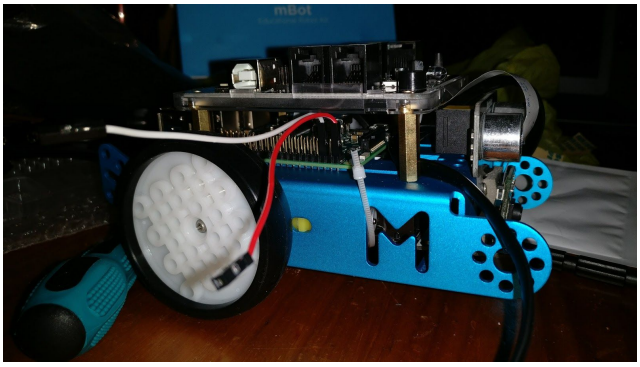
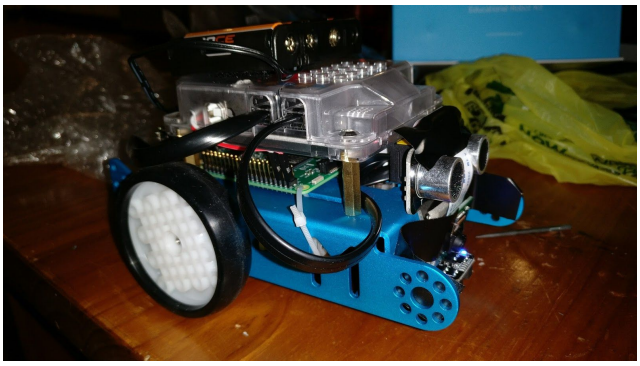
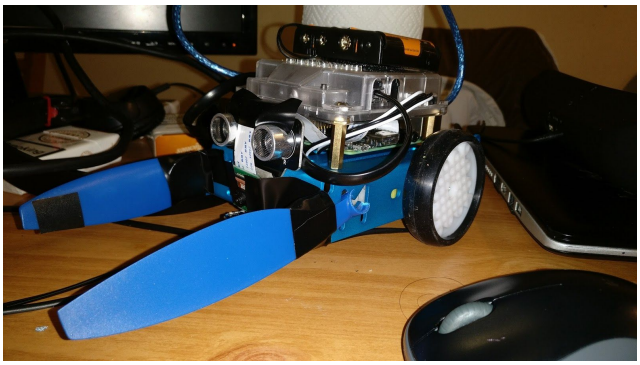
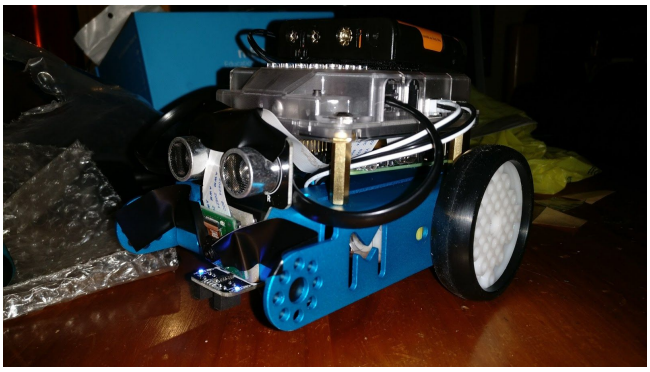
Assembling the robot in and of itself was a mini project and a challenge that we had to face before even starting the actual project. The size of the Raspberry bot made it extremely hard to include anything in it whatsoever apart from the hardware that it came with. The first problem that we had was powering up the Raspberry Pi. We could not afford putting an extra battery on the Raspberry bot as we have already used much of the space on it. After disassembling the Arduino and fiddling around with it for a little, we discovered that there are two 5V pins on the board that would help us do exactly what we wanted. We used those two pins to connect to Raspberry Pi's pins 1 and 3 via jumper wires, which was sufficient to give our Pi enough electrical power to operate. Doing this was a major breakthrough in our project. We placed the batteries on top of the Arduino and secured them with velcro so that they do not swivel or fall as the Raspberry bot is turning and doing sharp moves.

Although that worked at first, we decided to use the RGB LED sensor of the Raspberry bot. All of a sudden the Raspberry Pi started shutting down, or even not turning on in the first place. After a little troubleshooting it came clear that the RGB LEDs were using a lot of power and that there was just not enough electrical power to power up all of the components that we were using from a single battery source. That is why we used a second battery placed right next to the Raspberry bot batteries.

The Raspberry Pi itself was placed on the initial location of the Raspberry bot batteries. The only issue with that was that the jumper wires were a little too tall, so they would usually get a little bent by the Arduino on top. However, eventually as we tested we figured that this would not pose any problems to overall workings of the system. Next up, we used a short USB type A to type B cable in order to connect the Raspberry Pi to the Arduino. For communicating between the two devices we used the Serial class on the Arduino side, and the pySerial module on the Raspberry Pi side. We would send different messages to the same port, and wait for each device to be told what to do by the other one.

The Pi Camera was positioned on the front of the robot, right on top of the line following sensor. We taped it to the chassis of the Raspberry bot so that it would be secure to the robot. Also, we moved the interface cable of the camera to go through the ultrasonic sensor and taped it to the chassis so that it would not interfere with the sensor's readings.

Also, we got a damaged propeller from a quadcopter, that was not good for flying anymore, broke it into two and taped it to both sides of the Raspberry bot. We used the propellers as arms for when moving the objects later on. Here are some pictures of the building process:

The Neural Network

As with the design with all neural networks, we had our "try and error" phase to see which set of parameters is going to give us the best possible result. We started out by dividing our training data into separate folders and using One-Hot vectors to represent the labels of those folders and what kind of object are in there. Then we preprocessed the data, using opencv to resize the images to 32x32(later we used image augmentation library to enlarge the size of the training data, thus improve the accuracy of the model). Afterwards we divided our train data into train and test data so we can test the accuracy of the model. Choosing the right set of parameters for the model was cumbersome. We started out with two convolutional layers and one fully-connected layer and one output layer. In each layer we applied 5x5 convolution moving one pixel at a time and then applying 2x2 pooling and moving the window two pixels at a time. We used the **ReLU** activation function and **softmax_cross_entropy_with_logits** for helping us with the classification and the **AdamOptimizer** for the backpropagation algorithm to minimize the cost all of which are provided in TensorFlow.

Afterwards we divided the training data into batches and fed it through the neural net in a TensorFlow session. Tweaking the parameters we got the best results, using three convolution layers with a fully connected layer and an output layer. The following structure was used: First

layer takes the input and produces 32 outputs. The second layer takes the 32 outputs of the first and produces 64 while applying convolution and pooling. The next layer takes those 64 outputs as inputs and produces 128 outputs. The fully connected layer takes those 128 inputs and produces 1024. Finally the output layer takes the 1024 inputs and produces a vector of the number of classes (12). We also applied a keep rate of 90%, which means that at any time 10%, randomly chosen, neurons are not going to be active. We also used a learning rate of 0.0005. This slightly improves the performance of bigger neural nets. When the model is trained after N number of epochs, we saved the model into a TensorFlow .ckpt object and export it to the Raspberry bot. On the bot, we used the same model, while restoring the checkpoint object into a Tf.Session only once.

Afterwards, when the raspberry receives a signal to start the camera, we take a picture, save it in a folder as foo,jpg, apply Histogram Equalization on the picture (for sharper colors) and then run it through the network. After the prediction is made, we apply **softmax** algorithm to normalize the values in the vector and then use **argmax** to produce the index with the highest value, which is the prediction of the object that the neural network made (from 0 to b), which is then send to the Arduino.

Gathering Data and Training the Model

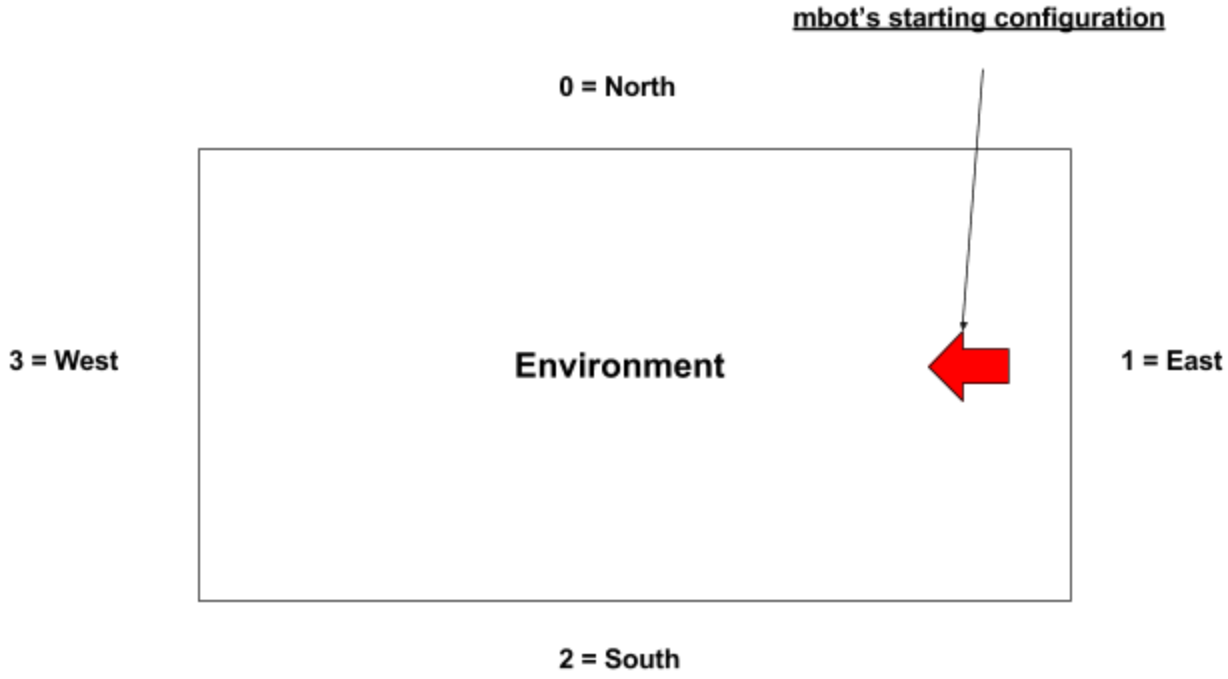
Gathering the data was tedious and long at first. We had to put the Raspberry bot on the floor, let it roam, and every time it detects an object make a few pictures of that object. It would then move a little bit at a different angle and take new photos. Soon, we realised that this was not a good approach. What we did instead is write a separate control law for gathering our training data. The Raspberry bot would roam around the environment, however, every time it would detect an obstacle it would stop and wait for a signal from the remote controller. Now it was our task to go and identify what the object was - whether it was a red sphere, or a green pyramid for example. We had different codes for each object. For example a green pyramid would be 5 on the remote controller. When 5 is pressed, the Arduino would give a specific message to the Raspberry Pi to go and take a picture of the object after which it would save it in a separate folder that contains pictures of only that one object. There were separate folders for each different color/shape combination. This way we would make the distinction between the different classes of data and train the model to classify them properly later on.

Moreover, we used a python library for image augmentation for machine learning purposes called imgaug. The library is used to make modifications to an existing dataset of images and therefore extend the dataset. The library makes various operations on images such as cropping, zooming, flipping, changing brightness, changing opacity and more. Not only did this enhance our dataset, but also helped the neural network to make better predictions as it had more corner cases to learn from.

The neural network was trained on a Windows laptop with an Nvidia GTX 730 graphics card. For the purpose of taking advantage of the many computing cores on a graphics card, the GPU version of TensorFlow® was used.

The Control Law

Because there was no way for us to know where our current location is at any given moment, we had to simulate a compass in our software. We used a static variable called *direction* which had four possible values - 0 for North, 1 for East, 2 for South and 3 for West (clockwise from North). We calibrated the Raspberry bot so that it would always turn at exactly 90 degrees when it does turns. Also our initial location would always point west, therefore using the above mentioned strategy, we would also know where our location is.



Each object in our environment would be associated with a placement variable. For example all red sphere must be sorted to the south. Therefore, there would be a variable in our code with the following declaration: int red_sphere_placement = 2; 2 meaning south. Also, we had a *move_object* function that would take the current facing direction of the Raspberry bot plus the location to where the currently detected object needs to go as arguments. The function is smart enough to make enough turns so that the Raspberry bot faces the direction in which the object needs to be sorted. After that the Raspberry bot would push the object until it cannot detect black terrain underneath it anymore. Once the terrain becomes white again the Raspberry bot would go backwards and then make two left turns in order to return to the environment.

Furthermore, we implemented a function called *change_direction* that takes two arguments - the current direction as an integer, and either a zero or a one, zero meaning a left

turn and one meaning a right turn. Depending on those arguments, the function's job is to change the direction variable of the Raspberry bot so that we always have an idea of where exactly in the environment we are.

In order to make sure that the neural network made a correct prediction of the detected object, we used the RGB LED sensor and the buzzer of the Raspberry bot. Whenever we got an output prediction from the neural network, we would light the RGB in either red, green, blue, or yellow signaling what the color of the object was. Moreover, the buzzer would give a short buzz signaling the shape of the object. One buzz would mean a cube, two buzzes would mean a pyramid and finally, 3 buzzes would mean a sphere.

Future Work

During the project we had a couple of good ideas for which we simply did not have either the time or the computational power to do. One of those ideas was instead of using a rectangle as our environment, to use a hexagon or even an octagon. We would then calibrate the robot to make smaller turns so that at each turn it would face the next side of the octagon. Also we would put different shapes at the different sides of the octagon, thus having a more graphical representation of our classification.

Another interesting idea that we had was to simulate a Cozmo using the Raspberry bot. The Raspberry bot would roam in its environment, taking about 30 pictures each second. It would store those in a folder and run a neural network as a background process. The network would learn its environment from interacting with it, and it would constantly keep getting better. The biggest obstacle in this project would be the sheer computational power of the Raspberry pi. It would simply not be enough to run a neural net of that scale.

Lessons Learnt

One of the biggest lessons we learned doing this project was that troubleshooting hardware issues is a completely different animal, than troubleshooting software. Unlike software, where one can run a debugger and see where one's mistake was relatively quickly, troubleshooting hardware issues is all about trial and error, and brainstorming new ideas about where a particular issue could occur and why.

Another lesson that we learned was that sometimes code is not documented really well, however, one has to learn how to work with a particular API again by trial and error. There was not a written documentation for using the mbot's API, so we had to learn how to use it by simply reading the API's source code and keep trying until something works.

Approximate time spent on the project: 80 man hours.