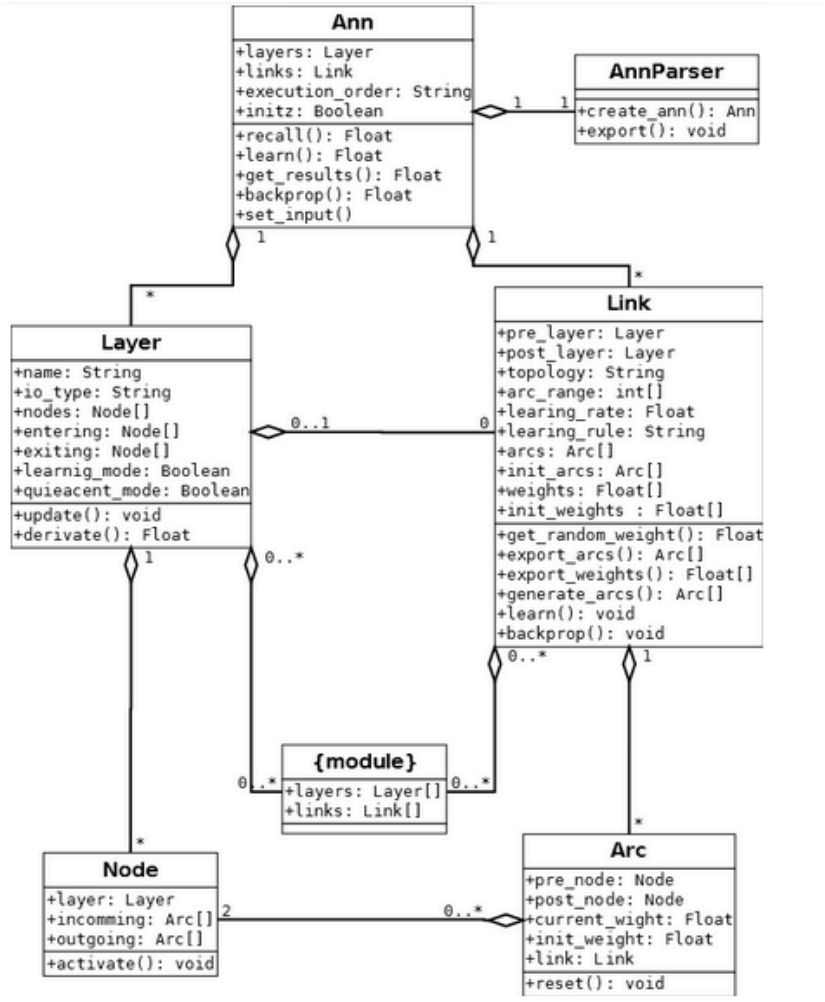
IT3708 Project 3

A Neural Network Controller for Webots

By: Mikael Brevik, Pål Moen Møst and Øyvind Selmer

Table of Contents

1. Code and Structure Overview



Ann

(Found in the file ann/ann.py)

The ANN class stores all layers and links, even the intra-layers from ANN modules. The ANN class finds all encoders and decoders (input and output layers), order the layers after execution order from script, and finds what order the links should have while doing back propagation.

The ANN class has three notable methods:

1. recall() – Sets the input to the encoders, updates all layers and returns the output result from decoders.
2. learn() – Uses recall(), and runs learning on all links. Returns the output results.
3. backprop() – Uses recall(), and uses back propagation learning on all links. Returns the output results.

It also contains the method test(), which calls recall() and returns the error factor.

Layer

(Class found in the file ann/layer.py)

The layer class stores information about layer name (used by the script files), nodes, and the links entering and exiting the layer. It also has information if the layer is a encoder- or decoder-type layer, which is used by the ANN to get results and set input, as well as determine the order of iteration on the links in back propagation.

Layer has a method update(), iterating over all contained nodes, with or without quiescent mode, and updates the activation level in the nodes using it's activate() method.

The layer file also have a class Activation, storing all static methods for activation functions (sigmoid log, tanh, step, linear and positive linear).

Node

(Class found in the file ann/node.py)

The Node class, stores values for membrane potential, current activation level and previous activation level. It also stores references to incoming and outgoing arcs. Every time the activation level is updated, the current value get set as the previous value, and the current value gets updated.

The function activate() calculates the membrane potential (weighted input) and passes that value up to the nodes layer, using the activation function to calculate the updated activation level. The weighted input is calculated differently based on if the incoming arc stems from a node within the same layer or not.

Link

(Class found in the file ann/link.py)

The Link class stores references to the pre- and post-synaptic layers, the link topology, learning rule and rates, initial weights and arc weight range. The link class has the method generate_arcs(), which is used to create arcs based on topology, or it can create arcs based of array of tuple values (with indexes for pre and post nodes) sent in by the constructor.

The link class has two notable methods; learn() and backprop(). Learn() iterates through all arcs and uses the learning rule to update current weight. The backprop() method is used to do back propagation, following the steps described in the "Generic ANN.pdf" document attached to the assignment description. It sets the delta value for both post and pre layer and uses this delta value to update the weight.

The link file also contains an ActivationRule class, with static methods for all learning rules.

Arc

(Class found in the file ann/arc.py)

Arc is a simple class, only storing values for pre- and post-synaptic nodes, current and initial weight and what link the arc is a part of. The only other method it has, is a reset() method used to set the current weight to the initial value.

The Arc class is used by the link class and is referenced from nodes.

Module

(Found in file ann/ann_modules.py)

The Standard Generic ANN modules, can be found in this file. On the class diagram all of these are shown as <Module>, but they are really four different classes. The implementations are done by extending the base layer class, and adding simple functionality. There are one extension per generic module; Competitive, Inhibitory, Associative and Transformer. Each one only implements the steps described for modules in the "Generic ANN.pdf" document, given as attachment in the project description.

Parser

(Class found in file ann/parser.py)

The parser module/class have both static and instance methods. The object methods are used to read input ANN file and create an ANN object, the static methods are used to export an ANN to a file.

The parser uses .INI files, and the python module "ConfigParser" to read and output data to the configuration files.

Each layer and link, are defined in the script file as a "section", with properties attached.

To export a ANN, the following code can be used:

```
AnnParser.export(ann_object, "path/to/filename.ini" [,
use_updated_values = False])
```

The last argument sets if the learned weights should be saved as initial weights in the script file.

To import an ANN, the following code can be used:

```
ann = AnnParser("path/to/filename.ini").create_ann()
```

After that line, the "ann" variable contains the functioning ANN object. The parser-class controls if the script file tries to connect two layers with a link, and one of the layers don't exist. I.e "No valid layer selected in Link.LINK_NAME".

You can see a script file in section 3.

Webot Controller and Execution

We've used the epuck_basic.py code given as an attachment for the project, we've also used the webann.py file as an extension of this e-puck code. The WebAnn class has one extended class BackProp, using the same code for running the robot, but runs N epochs of back propagation before hand to train the ANN.

The WebAnn class is fairly simple. The constructor set up the EpuckBasic class, and inits the ANN. As for the BackProp, it does the same thing, but the constructor also runs the training.

To initiate the robot a method run() is used. This method has an infinite loop doing three things:

1. Gets sensory data from both the IR-sensor and camera
2. Inputs this to the ANN and gets the resulting output (using recall()).
3. Set the drive speed of both wheels according the the results from running recall.

For running the project, the Webots simulator is used. This application automatically locates the WebAnn controller file, and executes it. Therefor, to select if you want the hardwired robot, or the learning robot, you must define what class to use on the ANN object; The basic WebAnn class, or the extended BackProp.

To get the ANN object you can use the Parser:

```
ann = AnnParser("ann/scripts/ann.ini").create_ann()
```

To initiate the controller you then do:

```
controller = WebAnn(ann) # For hard wired robot
controller = BackProp(ann) # For learning robot using back prop.
```

To start the robot you must run the `WebAnn#run()` method.

The way this structure is defined, it corresponds directly to the structure as defined in the assignment description.

2. Task Description

In our project, the robot tries to accomplish two things:

- Exploring while avoiding obstacles.
- Finding a green box.

We chose to find the green box, as that's the most distinct color in the world. There's a lot of red values in both the ground and walls, which makes the box harder to distinguish from the rest of the environment.

The robot will only be controlled by an ANN, both hard wired and trained by the back propagation method. The training is done off-line, by semi-hand made data, generated by a script following some constraints and traits.

3. ANN Script

See section 1, under Parser, for a more in detail description of the script. In essence, each component is defined as a section in a configuration script. There are four different kinds of section. Layer, Link, Execution order and one used for ANN modules. Each of these section types have properties specific for the type.

I.e. Layer has activation for activation function, and step if activation function is the step function, nodes for the number of nodes and definition if it is an encoder or decoder.

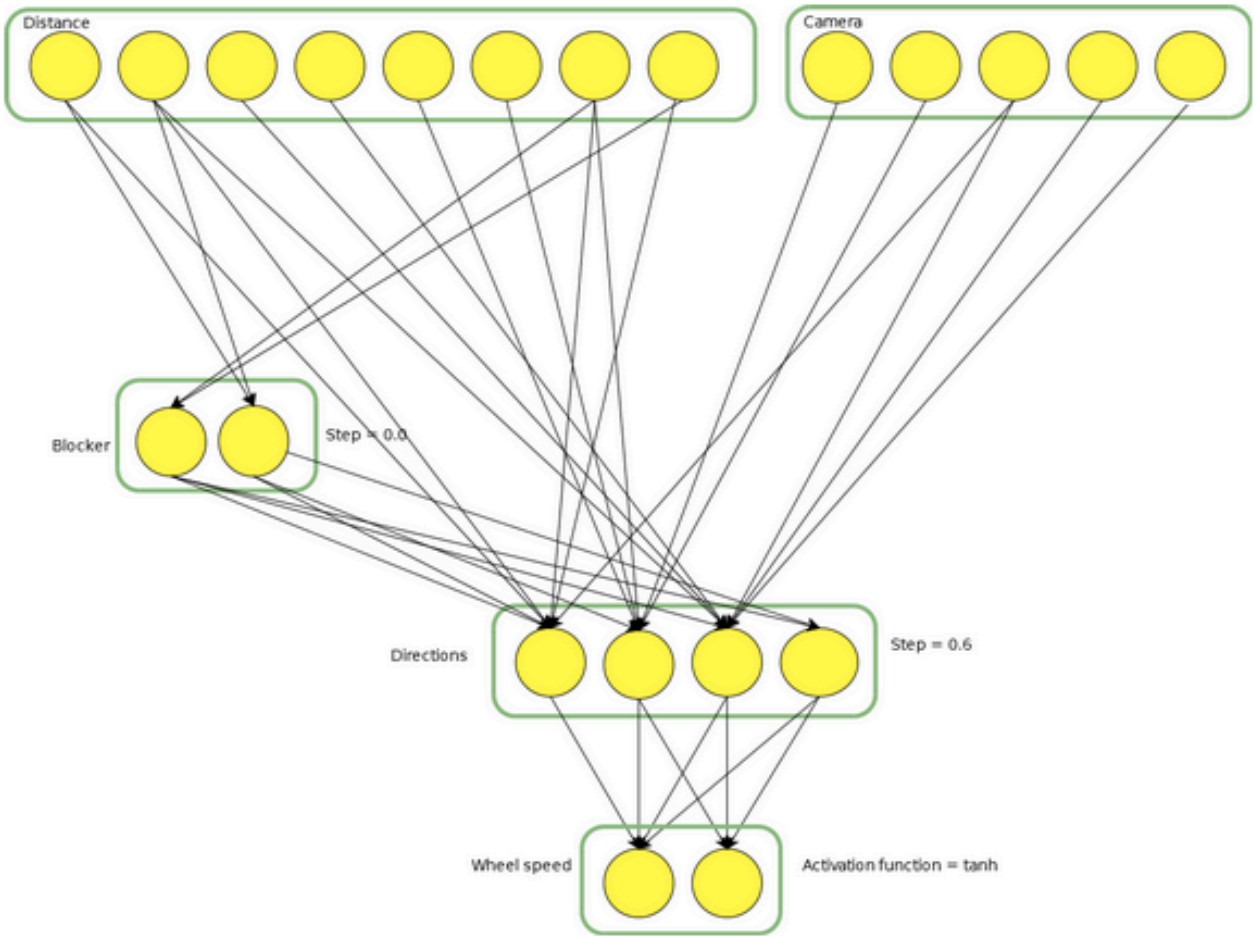
The Link type must have definition of pre and post synaptic layers. All containing arcs (both weights and direction) can also be defined. Other properties is initial weight range for arcs (used for random initialization), learning_rate and learning_rule.

The execution order is simply an array containing string representation of layers in the order they should be executed.

```
[Layer Distance]
activation = None
nodes = 8
io_type = encoder

[Layer Camera]
activation = None
nodes = 5
io_type = encoder

[Layer Blocker]
activation = step
step = 0.0
nodes = 2

[Layer Directions]
activation = step
step = 0.6
nodes = 4

[Layer WheelSpeed]
activation = tanh
nodes = 2
io_type = decoder

[Link 1]
arc_range = [-0.1, 0.1]
```

```
learning_rate = 0.2
weights = (-1, -1, -1, -1)
arcs = [(6, 0), (7, 0), (0, 1), (1, 1)]
post = Blocker
pre = Distance

[Link 2]
arc_range = [-0.1, 0.1]
learning_rate = 0.2
weights = (0.5, 0.5, -1, -1, 0.4, 0.4)
arcs = [(0, 2), (1, 1), (0, 0), (1, 0), (0, 3), (1, 3)]
post = Directions
pre = Blocker

[Link 3]
arc_range = [-0.1, 0.1]
learning_rate = 0.2
weights = (0.5, 0.8, 0.8, 0.5, 0.1, 0.25, 0.25, 0.1, 0.25, 0.25)
arcs = [(1, 0), (0, 0), (7, 0), (6, 0), (6, 1), (5, 1), (4, 1), (1, 2), (2, 2),
(3, 2)]
post = Directions
pre = Distance

[Link 4]
arc_range = [-0.1, 0.1]
learning_rate = 0.2
weights = (0.5, 0.5, 0.1, 0.5, 0.5, -1)
arcs = [(0, 1), (1, 1), (2, 0), (3, 2), (4, 2), (2, 3)]
post = Directions
pre = Camera

[Link 5]
arc_range = [-0.1, 0.1]
learning_rate = 0.2
topology = full
weights = (1.3, 1.3, -0.5, 0.5, 0.6, -0.6, -0.3, -0.5)
post = WheelSpeed
pre = Directions

[Execution Order]
order = ['Distance', 'Camera', 'Blocker', 'Directions', 'WheelSpeed']
```

4. Description of the ANN

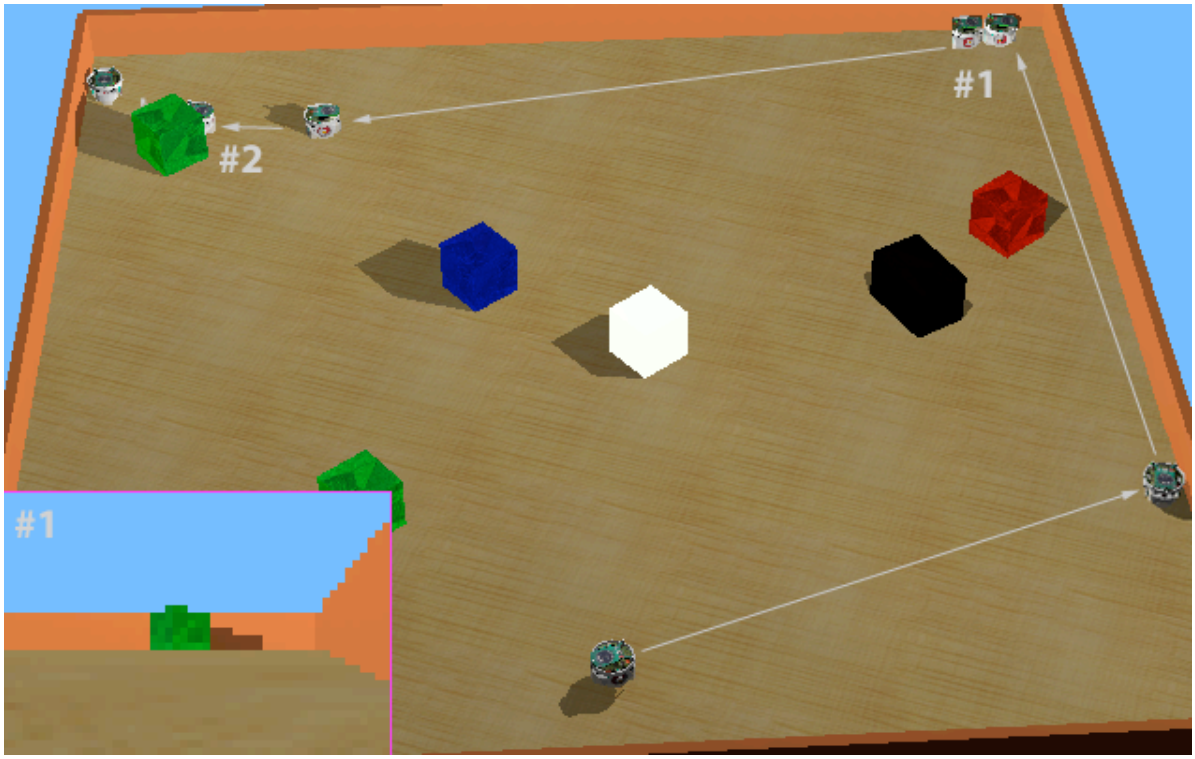


The Artificial neural network that we designed has two input layers Distance and Camera and two hidden layers Blocker and Directions. Each node in the layer Distance represents one IR-proximity sensor on the E-puck robot. The camera layer is represented by five nodes. Each nodes represent column in the snapshot taken by the camera mounted on the E-puck robot. Pre-processing of the snapshot splits the picture into five columns and calculates the amount of green in each column. If the columns in the snapshot contains any green the robot will turn in that direction of the given column(s) that contains green. See figure 3. The hidden layer Direction decides which way the robot should go. Right, left, forward or back. It is dependent on the input from the input layers and Blocker. It has activation function Step with a threshold of 0.6. The hidden layer Blocker is setup to handle obstacles that the proximity sensors in the front of the robot discovers. It has activation function Step with the threshold of 0. Finally the output layer Wheelspeed that sets the correct speed to the wheels, so it turns right, left or goes straight ahead (equal values). It has the activation function tanh.

Figure 3: Columns of the camera view

We uses the same design for the hard wired and the learning ann. The difference lays in the weight of the arcs. In the hard wired ANN the arcs are manually set by us through out testing and common sense. The arcs in the learning ANN is first set to an initial value and then changed throughout **Back Propagation**. Specifications for each layer is given in the ANN script section.

5. Performance of Hard-Wired Robot



As seen on the figure above, the robot will go in fairly straight line, while there's no objects blocking it's way, or any green objects can be seen. When it hits a wall, it will turn to the side where the path is free and go that direction. This happens on the first position annotated in **#1**, and on the position before that (between **#1** and start position). On the second position in **#1**, the green box will appear in the cameras view, and the robot will adjust its direction to head to the object. As one can see the path is not as close to the wall after **#1** as it is before **#1**, implying that the robot has adjusted it's path.

Between annotation **#1** and **#2**, the robot continuously adjusts it self. When the object is on the left side of the camera, it turns left, when it is on the right side, it turns right. When the robot gets to the target it won't stop, but "mark" the target and drive ahead and search for more targets.

No matter where the start position of the robot is, it will go in straight lines and avoid any obstacles it meets (as long as it doesn't get stuck), and look for reactions in the camera sensor. When it eventually sees it it will turn to the object an drive towards it.

The hard wired robot is a bit fond of hogging to the walls. It might have been smart to make the robot sway in a direction to have it gaping from the wall.

6. Training cases and general strategy for back prop

For training the ANN, we use an off-line training method. We have generated a file with a long list of different cases and optimal target for the back propagation to run any number of epochs on.

To generate cases we constructed a script which creates N amount of data. This script is based on random choice with a couple of constraints. So instead of manually writing down numerous cases, we create a set of similar data sets with some variations. With the dynamic scripting we can also favor some of the wished behavior by adding more of that kind of data (I.e for moving straight ahead.)

Example of the data generated will be:

```
[[-1, -1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0], [-0.6, 1]]
```

When there is any blocking the robot on the right front side, and the camera doesn't see anything, rotate to the left.

```
[[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0], [1, 1]]
```

If there are no blocking objects, and the camera doesn't see anything, drive straight ahead.

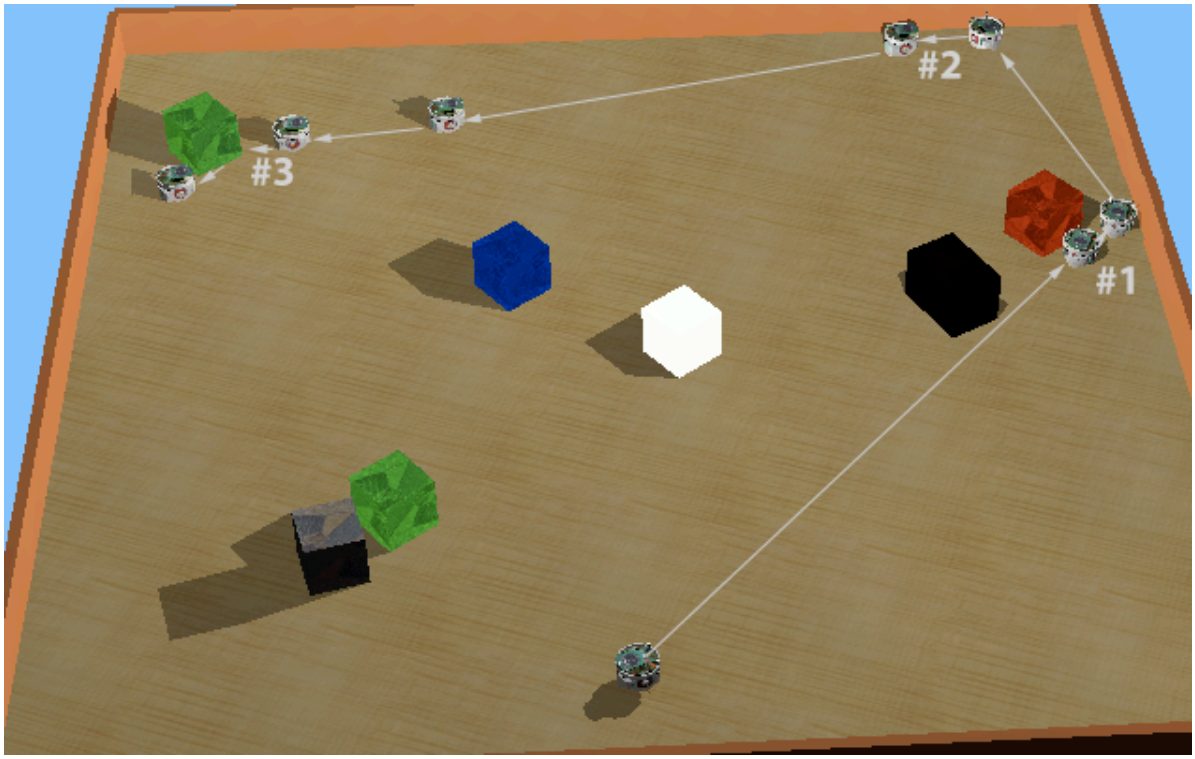
```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0.1, 0.1, 0.1], [-0.5, 0.5]]
```

If there's no blocking object, and the camera sees the desired object on the left side, turn to the left.

```
[[1, 1, 1, 1, 1, 1, 1, 1, 0.1, 0.1, 0.1, 1, 1], [0.5, -0.5]]
```

If there's no blocking object, and the camera sees the desired object on the right side, turn to the left.

7. Performance of Learning Robot



The robot using back propagation, functions really well. When the input says there's no obstacles, and the camera can't see any of the desired objects, the robot follows a forward path, but sways a bit to the left. The swaying may contribute to the robot not hogging to the walls as much as the hard wired one. As one can see on the **#1** annotation on the image above, the robot avoids the red box by turning right, and then turning left again before hitting the wall. As with the hard wired robot, when it gets to position **#2**, it sees a green box, and therefor adjusts his wheel speed to head straight for the object. As one can see from the position between **#2** and **#3** and **#3**, the robot adjusts it self to match up with the box again, after swaying of the path a bit.

As with the hard wired robot, the learning robot won't stop at the object it searches for, unless it get stuck. So when it's found, it crashes in to it, and then moves on to find other green objects.

The robot can be set to different starting position, the same will happen. The robot will wander around avoiding obstacles until it gets a green object in sight, and then it will try to head for the object. The learning robot can some times have difficulty with the white object. The white object, has a high amount of green value (from RGB), and will give some read out value for the camera sensor. The robot might be too sensitive on the camera input.