Introduction

The work in this session is designed to explore **how learning can be applied to an agent's perceptions** of its environment. We will show how learning can be added to the Bot code, as a final wrapup for this example.

Getting Started

Download the file called ***simpleBot4.zip*** from the module Moodle page. Unzip it. The Python file for this week's tasks is **simpleBot4.py** **make sure that you can run it.** You can run this from the command line, or import the file into an IDE. Also install the ***playsound*** and the **pillow** libraries, by using

pip install playsound

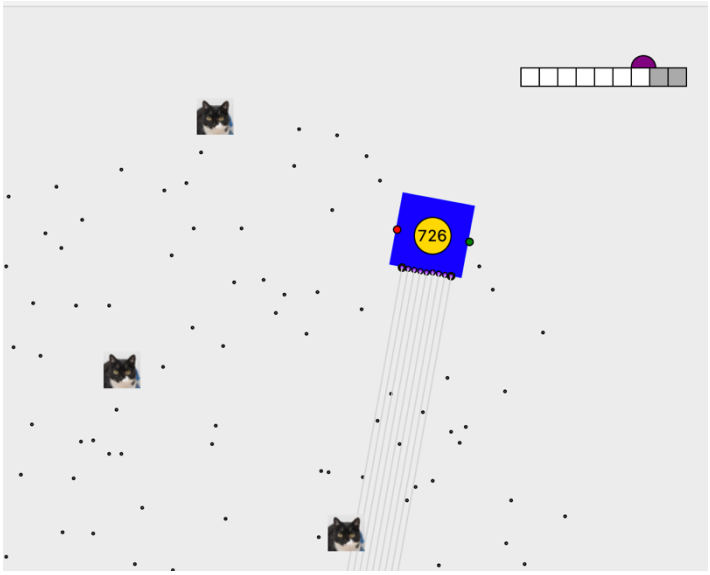
pip install pillow

on MacOS you will also need to do

pip install PyObjC

If you have problems with the sound library, just delete the line *playsound("385892__spacether__262312steffcaffrey-cat-meow1.mp3", block=False)* from *simpleBot4.py*. This line is not an important part of the task.

This is a variant on the Bot code from the previous sessions. Now, the bot has an elementary visual system. It has nine visual sensors, that can see a certain distance into the environment. This is represented by the ***look*** method in the ***Bot*** class, which looks at nine positions in front of the Bot, indicated by the grey lines coming from the bot. The ***look*** method returns a list of nine numbers, which represent what the bot can "see"—if the value is zero, there is nothing in front of that sensor (at least for 400 pixels distance), and if there is something in front, this number gets closer to 1.0 the closer it is to the sensor.



This is shown visually in the top corner of the window, where the nine squares represent the inputs to the nine sensors. So, in the example above, the rightmost two sensors are activated to about 0.5, as illustrated by the mid-grey colour. This is because the last couple of sensors are "seeing" the cat to the bottom of the screenshot.

The other new method in the *Bot* is ***collision***. This detects collisions with *Cat* objects, which wander around the screen at random. Collision returns *True* if the bot bumps into a cat; otherwise *False*.

Tasks

The aim of today's exercises is to get the bot **to learn avoid—or at least warn—the cats**.

Training phase

The first step is to gather a training set; that is, we will let the bot wander at random around the space, and keep a record of the sensor values and the collisions. To do so:

1. make a copy of *main* called *train*, and create an empty list called *trainingSet* (just before *moveIt*), and pass this as a parameter to *moveIt*.
2. Make a copy of *moveIt* called *trainIt,* and change the reference in *train* from *moveIt* to *trainIt*.
3. Then, in *trainIt* just after the call to *collision* add a new line that appends to *trainingSet* a tuple consisting of the list returned from *look* and the *True/False* value returned from *collision*. To do this, you will need to add a variable to collect the information returned from *look* a few lines earlier.
4. As we did in a previous week, replace *sys.exit()* with *window.destroy()* (you will have to pass *window* as a parameter to *trainIt;* or (more advanced) remove all graphics code from *trainIt*), and put a *return* statement after that.
5. Change the code so that the bots move for lots of times (say, 10,000, perhaps even 100,000), and speed up the animation so that there is minimal delay.

```
([0.5930951245032592, 0.5930951245032592, 0.5930951245032592, 0.5930951245032592, 0.5930951245032592, 0, 0, 0, 0], False)
([0.6162806125677086, 0.6162806125677086, 0.6162806125677086, 0.6162806125677086, 0.6162806125677086, 0, 0, 0], False)
([0.6393796591458534, 0.6393796591458534, 0.6393796591458534, 0.6393796591458534, 0.6393796591458534, 0, 0, 0], False)
([0.6518663595315405, 0.6518663595315405, 0.6518663595315405, 0.6518663595315405, 0.6518663595315405, 0, 0, 0], False)
([0.6643520698501413, 0.6643520698501413, 0.6643520698501413, 0.6643520698501413, 0.6643520698501413, 0, 0, 0], False)
([0.6768366753451963, 0.6768366753451963, 0.6768366753451963, 0.6768366753451963, 0.6768366753451963, 0, 0, 0], False)
([0.6893200428254057, 0.6893200428254057, 0.6893200428254057, 0.6893200428254057, 0.6893200428254057, 0, 0, 0], False)
([0.7018020168108621, 0.7018020168108621, 0.7018020168108621, 0.7018020168108621, 0.7018020168108621, 0, 0, 0], False)
([0.7142824146711807, 0.7142824146711807, 0.7142824146711807, 0.7142824146711807, 0.7142824146711807, 0, 0, 0], False)
([0.7267610204341391, 0.7267610204341391, 0.7267610204341391, 0.7267610204341391, 0.7267610204341391, 0, 0, 0], False)
([0.7392375768208644, 0.7392375768208644, 0.7392375768208644, 0.7392375768208644, 0.7392375768208644, 0, 0, 0], False)
([0.7517117748858874, 0.7517117748858874, 0.7517117748858874, 0.7517117748858874, 0.7517117748858874, 0, 0, 0], False)
([0.7641832403785134, 0.7641832403785134, 0.7641832403785134, 0.7641832403785134, 0.7641832403785134, 0, 0, 0], False)
([0.7766515155492661, 0.7766515155492661, 0.7766515155492661, 0.7766515155492661, 0.7766515155492661, 0, 0, 0], False)
([0.7891160345249041, 0.7891160345249041, 0.7891160345249041, 0.7891160345249041, 0.7891160345249041, 0, 0, 0], False)
([0.8015760894383867, 0.8015760894383867, 0.8015760894383867, 0.8015760894383867, 0.8015760894383867, 0, 0, 0], False)
([0.8140307830026032, 0.8140307830026032, 0.8140307830026032, 0.8140307830026032, 0.8140307830026032, 0, 0, 0], False)
([0.8264789607607003, 0.8264789607607003, 0.8264789607607003, 0.8264789607607003, 0.8264789607607003, 0, 0, 0], False)
([0.8389191120997023, 0.8389191120997023, 0.8389191120997023, 0.8389191120997023, 0.8389191120997023, 0, 0, 0], False)
([0.8513492218829313, 0.8513492218829313, 0.8513492218829313, 0.8513492218829313, 0.8513492218829313, 0, 0, 0], False)
([0.8637665414706368, 0.8637665414706368, 0.8637665414706368, 0.8637665414706368, 0.8637665414706368, 0, 0, 0], True)
([0, 0, 0, 0, 0, 0, 0, 0, 0], False)
([0, 0, 0, 0, 0, 0, 0, 0, 0], False)
([0, 0, 0, 0, 0, 0, 0, 0, 0], True)
([0, 0, 0, 0, 0, 0, 0, 0, 0], False)
```

6. Run train: you should end up with a list that looks something like this:

7. Now, we aren't interested in the values of the sensors when the collision has happened—that is too late. So, instead, write a loop that goes through *trainingSet*, and makes a list of the sensor values (say) 5 steps before the collision happened. Call this *warningValues*. Basically, these will form the training set for learning.

Learning phase

We will do a very simple kind of learning from that list. Let's work out how close, on average, we are with any one of the sensors, before a collision is imminent.

1. Loop through the *warningValues* list, find the maximum value in each set of sensor values in that list (rejecting it if it is zero), then find the average of all these maximum values. Call this *dangerThreshold*, and return it from the *train* function.
2. Create a new method in *Bot* called *checkForDanger*, which takes *dangerThreshold* and *registryActives* as parameters. Call this just before the call to *collision* in *moveIt*.

3. Now, in *moveIt*, place the *checkForDanger* method before the line that calls *collision*. *checkForDanger* executes the *look* method to find the current sensor values. If the max of these sensor values exceeds the *dangerThreshold*, then the system has detected that a collision is imminent! The simplest approach to this is to sound a warning.
4. If, for entertainment, you want to make a real sound, then there is a soundfile that you can play using:

   playsound("436589__julien-matthey__jm-transport-ext-horn-01a-car-short-mini-countryman.wav", block=False)

5. The effect of this should be to make the cats move out of the way. Go through the *registryActives*, and if the object is of class *Cat*, execute its *jump* method (or, perhaps a new jump method with a longer jump than before).

An alternative (slightly more complex) response would be to turn the robot away.

Do some experiments. Count the number of collisions that happen, with and without the warning signal/turning away (as with last week, you probably want to put this in a framework where you repeat the experiments automatically). If you want to be more rigorous, calculate whether a collision would have happened anyway, and then you can calculate the number of false positives (the number of times a warning was sounded but there was not going to be a collision anyway).

Extensions

Here are some extensions to the tasks above. We are not anticipating that you will attempt these in the class, but they provide ideas that might form the beginnings of your coursework.

- Experiment with different settings. The number of steps back (5) was somewhat guesswork. You could be more conservative with the *dangerThreshold*. Does doing more training improve it? Do a rigorous comparison of the system without danger detection, with warnings, and with steering away.
- Measure how much difference (if any) does this avoidance makes to the cleaning task.
- Improve the learning. In this class, we have constructed the beginnings of a training set that could be used in conjunction with a machine learning framework such as *scikit-learn*. You would need to create a second set—call this *safeValues*—by looking through *trainingSet* for examples where the current state wasn't followed in the next few steps by a collision. Then, you have a traditional training set for two-class supervised classification in machine learning. Use. Method such as *CART* (Classification and Regression Trees) to distinguish between safe and unsafe situations, rather than the rather crude thresholding method used above. You can read about how to use CART here: https://scikit-learn.org/stable/modules/tree.html A good attempt at this, with good evaluation, would be a fine basis for the coursework for this module: the question would be "can a robotic agent learn to avoid moving obstacles in its environment by using decision tree learning". Other learning methods are possible.
- Improve the cat behaviour: for example, they only respond to the warning if they are close to the bot.