Designing Intelligent Agents  
Spring 2023  
Worksheet 3

Colin Johnson

# 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 a simple form of learning can be added to the *Brain* code and how the *Bot* can respond to this.

# Getting Started

**Download the file** called *simpleBot3.zip* from the module Moodle page. Unzip it. The Python file for this week’s tasks is *simpleBot3.py*; **make sure that you can run it.** You can run this from the command line, or import the file into an IDE. If you are using Jupyter notebooks, there is a .ipynb file. Also install the *playsound* library, by using

pip install playsound

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\_\_262312-steffcaffrey-cat-meow1.mp3", block=False)* from *simpleBot3.py*. This line is not an important part of the task.

A picture containing text, flying, flock

Description automatically generatedThis 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 is positive. It gets closer to 1.0 the closer the obstacle is to the sensor. If something is immediately in front of the sensor, it returns a value of 1.0.

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 numbers are 0.5 because the cat is in the mid-distance.

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*.

The *Brain* class has been adapted to match the new capabilities of the robot. When *thinkAndAct* is run, it now receives a list of numbers called *camera*, which are the sensor values described above; it also receives a Boolean value *collision*, which is *True* if the robot is currently bumping into a cat.

# Tasks

The aim of today’s exercises is to get the robot to learn to avoid—or at least alert—the cats. 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.

We are going to get *thinkAndAct* in the *Brain* class to do two different things. For the first 1000 times it is called, it is going to gather information about the camera values, and then after 1000 time-steps it is going to use the information gathered to give warnings to the cats when it is getting close to them. So, add an attribute called *self.time* to the *Brain* method, which is incremented by 1 each time *thinkAndAct* is called. Add an *if* statement at the beginning of the *thinkAndAct* method with the condition *self.time<1000*.

Add an attribute *trainingSet* to *Brain*, and initialise it to an empty list. Now, inside the *if* statement append a tuple consisting of the list of sensor values in *camera* and the *True/False* value in *collision*. You should end up with *trainingSet* looking something like this:

A picture containing text

Description automatically generated

That is, we have a list of the nine sensor values for all robots for every timestep in the run. Each is paired with either *True*, if the robot is currently colliding with a cat, and *False* otherwise.

Now, we will process that information on the 1000th timestep. Create another *if* statement with the condition *self.time==1000*. Within that if statement, 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*.

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. To do this, write a loop that goes through the *warningValues* list, finds the maximum sensor values for each entry in the list (rejecting it if all the sensor values in the list are zero), then finds the average of all these maximum values. Call this *self.dangerThreshold*.. The idea is that a sensor value above *self.dangerThreshold* represents a typical situation in which a collision will be happening soon.

Now, create a third and final *if* statement in the *thinkAndAct* method of *Brain*. This is active once training is over, i.e. once *self.time>1000*. Note that there is a variable called *dangerDetected* in the *thinkAndAct* method, which is returned as one of the return parameters. Within this third if statement, set *dangerDetected* to *True* if any of the current sensor values in *camera* is above *self.dangerThreshold*. That is, you are predicting that a collision is imminent.

Now take a look at the *Bot* class. If *dangerDetected* has been set to *True*, then the *reactToDanger* method in the *Bot* class is activated. At the moment, this just prints a message to the console. Check if it works—does “dangerous situation” get printed to the console when the robot is close to colliding with a cat?

Now, let’s sound out a warning to the cats! 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)

To simulate the cats’ response to this, write a loop in *reactToDanger* that loops through all of the *agents*, and checks whether each agent is a cat (using *isinstance*). 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 discussed in the lecture, 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. I am 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.