Homework Module:

A Neural Network Controller for Webots

Purpose: Gain a basic understanding of the use of neural networks to control robots

1 Assignment

In this module, you will use an artificial neural network (ANN) of your own design to control a simulated e-puck robot that runs on the Webots system. Thus, this is essentially a double module, in that you must dig rather deeply into both ANNs and Webots. Consequently, this is worth twice the points of a standard module.

You are free to choose the task for your robot. The only condition is that the robot is controlled by nothing more than an ANN. That means that ALL sensor data that you decide to use (camera, infrared/distance, sound) must be sent to the ANN (and to no other action-controlling code), and all actions (i.e. all wheel movements or generation of signals, such as audio) must be governed by the ANN's output. Some possible tasks include:

1. Moving as fast as possible, but avoid all obstacles, while *exploring* the environment; i.e. the robot shouldn't just spin around in circles but should move around in the arena/environment.
2. Starting anywhere in the environment, search for a particular block (e.g. the red one) and move to it.
3. Divide the environment into several rooms and get the robot to move around so that it visits each one.
4. Move through a simple maze from a start to a goal location.
5. Upon experiencing some target stimulus (such as any obstacle or a block of a particular color), the robot begins a *dance*, i.e., a fixed pattern of movement composed of at least a few different actions. The trick here is that the execution of this fixed action pattern (FAP) must be controlled by the ANN such that the network decides when to transition from one action of the FAP to the next. This involves the use of neurons with state, time constants and/or networks with recurrent connections.
6. Move around, exploring the world, and build up a classification of the sensory input patterns that it experiences using unsupervised learning. The patterns that each detector neuron prefers should then be graphically visualized for the user of the system. These sensory input may include infrared/distance sensors, the camera, even audio signals.
7. Something that you choose. This **must** be approved by Keith or Lester before you begin working on it. So feel free to be creative here, but check with us before you get too far. Some tasks are MUCH harder than they seem when you try to implement them in a robot, particularly one controlled by an ANN.

As indicated by these tasks, you are free to use any of the sensory options. In our experience, you can only do limited things with the distance sensors, but they are good at helping the robot to avoid obstacles. The camera is quite satisfactory, and relatively easy to use. You will probably want to pre-process the images before feeding them into your neural network, but in languages like Python, that is easily accomplished with the *image* module (a free download). If you choose to use audio signalling, be sure to check out the *Emitter* class in the Webots documentation.

It is fine if you want to compress the image data in some way before feeding it to your network. For example, you may want to compute a column average of color values and only give these averages to the network, thus reducing the networks visual-input-layer size from MN to N neurons, where M and N are the height and width of an image, respectively.

In this module, you will get the robot to perform the chosen task in two ways: hardwired and via learning. In the hardwired version, you will explicitly set the weights for the ANN to values that are conducive to the task, while in the learning version, back-propgation will be used to train the network.

It is normally easier to begin without learning and then add it once you get the sensors and motors properly interfaced with your network. Starting with an untested network and just **hoping** that learning finds the right behaviors is normally an exercise in extreme futility. So make sure you're network is doing basic things correctly before adding adaptivity to the synapses.

2 The Neural Network

The design of ANN's for controlling robots can involve a lot of trial and error. It is therefore important to have good supporting tools for this task. A significant portion of this module involves the production of a *Generic ANN*, as described in the file generic-ann.pdf, available at: http://www.idi.ntnu.no/emner/it3708/lectures/notes/.

To get full credit for this module, you will need to implement the basic Generic ANN framework, and your scripts should include Layers, Links, and an Execution sequence.

If you choose to implement neural networks in a more ad-hoc manner, without doing the Generic ANN framework, then you will only get partial credit for the module.

A fully functioning Generic ANN framework will allow you to specify all layers, links and the execution sequence **in a script**, which your system will then convert into a working ANN. This script-based approach should greatly improve turn-around time in debugging and testing, so we strongly recommend using the Generic ANN approach despite the extra initial effort that it incurs.

2.1 Back-propagation Learning

A significant portion of this assignment involves your ability to teach the robot in a supervised manner, using standard back-propagation. You are free to incorporate backprop into any aspect of your task, but to get full credit for this module, you will need to have it somewhere and document that the robot's ANN has indeed learned something useful (relative to the task).

A typical approach is depicted in Figure 1, where sensor readings are converted to simple binary values encoding the presence or absence of a nearby object in that sensor's direction from the robot. The ANN

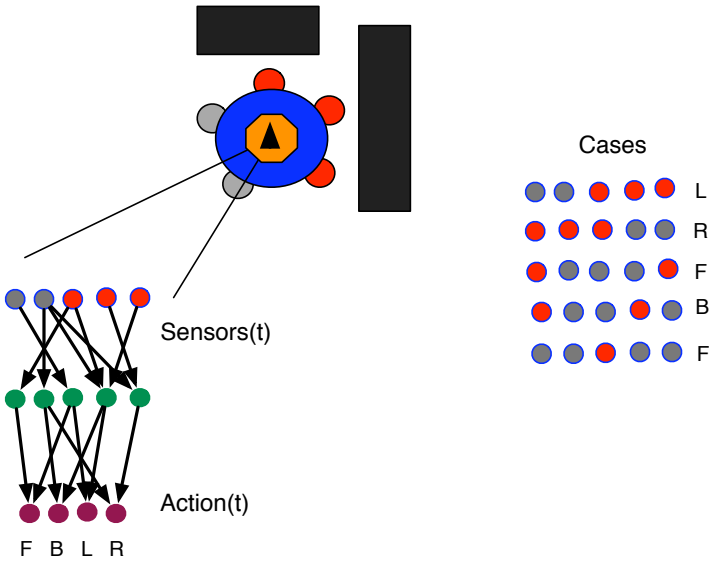

Figure 1: A simple robot (large circle) with 5 distance sensors (small circles) whose colors indicate the presence (red) or absence (gray) of a nearby object. Sensor readings are used as inputs to the robot's ANN (lower left), whose output neurons code for forward (F) and backward (B) movement, and left (L) and right (R) turns. In this scenario, hot sensors on the left and and front of the robot map to the action of turning left. On the right, a sample list of cases (for a task such as obstacle avoidance) are listed, with each containing sensor readings and the recommended action.

then maps sensor readings to robot actions. In the ANN of the figure, the output nodes encode 4 different actions, so auxiliary code might choose the action whose node is most active. Another alternative is to have just two outputs, each of which encodes the speed for the left or right wheel, respectively.

On the left of the figure, several possible cases are depicted. A list of a few dozen such cases would be a typical training set for an ANN, for any of a wide variety of tasks. In the figure, the task is presumably obstacle avoidance, as the movement is opposite to the *hot* sensors.

To train your Webot ANN for a particular task, simply generate (by hand) a set of relevant training cases and perform backpropagation off-line (i.e., when the robot is not actually performing the task in the environment). Then, to test the ANN, place the Webot in its environment and observe the results.

It is easiest to pre-process the sensor readings (which are normally 4-digit integers) into real or binary values before sending them to the ANN. It is these pre-processed values that you would use in your training set as well.

As an alternative to off-line training with a hand-designed case set, you are free to train the robot on-line, though this requires a little more work. As the robot is moving around, you will need to instruct it as to the proper actions to take in different situations. The sensory input cases are then those that the robot actually experiences (although they can still be pre-processed, e.g. converted to binary values). Setting up the code for this may take a little extra time, but once you can interact with the robot by sending it commands that it will convert to target outputs for the ANN, the training can go relatively quickly. It also makes for a very entertaining demo. In addition, it allows you to define the actual task on the fly: by starting with a blank-slate ANN and instructing the robot as to what it should do in various sensory contexts for a few minutes, you can **mold** the robot to do any of a number of simple tasks. So it's a little extra work, but the

payoffs of both generality and increased enjoyment that you can have with the robot can be worth the effort.

3 Webots

The Webots software will soon be available as a floating license from IDI, but if this arrangement is not yet in place when you begin the assignment, simply go to the Webots homepage: http://www.cyberbotics.com/. Then follow the menu path: Products/Webots/download to find the Demo version for your machine. Once the demo version is running, you can download a free PROfessional version for 30 days. You'll need that PRO version, since the Demo version won't allow you to compile new controllers, such as the ANN-based one that you will be building.

Once Webots is installed, fire it up and go to the Help menu for access to the Introductory materials, a comprehensive User's Guide, and a good Reference Manual. In the Reference Manual, pay particular attention to Chapter 10, "Other APIs". There, you will find very useful overviews of the key classes and methods used in Webots. You will need to read a good deal of this documentation; all important details are not described in this homework-module description.

Webots scenarios are implemented as *worlds*, and each world can contain one or more robots, along with various stationary physical objects, such as walls and blocks. We will provide you with one basic world, *e-puck.wbt*, which you are free to use (as is) or modify. Most changes can be done using the "Scene Tree" window on the left side of the Webots window. Worlds are language independent, so regardless of your choice of language for programming the ANN controller, this same world file should work fine.

When you load *e-puck.wbt* into Webots, you will see a table top with an epuck robot and several colored blocks. Each of these components is defined in the Scene Tree. At the bottom of the Scene Tree is an EPUCK definition. Clicking on that definition shows its attributes; the third attribute is the name of a controller, "webann". You will need to change that to the name of your controller, which needs to have been declared in Webots using the top menu item *Wizard* and choosing the "New Robot Controller" option. The wizard allows you to specify the programming language for your controller, with options such as C, C++, Java and Python. Most of the demo code that comes with Webots is written in C, C++ or JAVA. Python is a relatively new addition, so there are few demos. However, the base epuck-controller class that we provide, *epuck_basic* is Python code. The functionality in epuck_basic should translate easily to other languages; the method or function names are similar across the different API's.

You can define and edit your controller in the Webots window, or you can use your favorite editor and load it into Webots via the *revert* button in the controller-editor portion of the Webots window, which normally appears on the right. In our experience, the editor window provided by Webots is not that helpful for Python; its use of indentation is confusing. So you might opt to use your favorite editor instead.

3.1 File Structure

The normal practice in Webots is to define a project directory, such as *ann-epuck*, inside of the Webots *robots* directory. It then contains 2 directories entitled *worlds* and *controllers*. Inside the former directory, simply place your e-puck.wbt file. However, inside the *controller* directory, you need to create another directory with the EXACT SAME NAME as your controller file (minus the file extension). So the file structure that you add into the Webots/projects/ directory should look something like this (where the robots directory should already exist):

```
robots
        ann-epuck
                controllers
                        my-ann-controller
                                epuck_basic.py
                                my-ann-controller.py
                                any other supporting code for your controller
                worlds
                        e-puck.wbt
```

The file extension on your controller file tells Webots the language of your controller, and it will then invoke the appropriate compilers, loaders, etc. to get your controller up and running.

3.1.1 Shell Variable Bindings in Webots

If you start up Webots by clicking on the ladybug icon, you will have the full power of languages such as C++, Matlab, Python, etc., but you will not necessarily have access to (your normal) bindings of shell variables such as PYTHONPATH. Without those bindings, Python in Webots will not know where to look for your files. It will look in the Webots sub-directory where your controller is defined (as shown above), but not, for example, in any of your normal directories (outside of the Webots file tree). This may or may not matter to you, as it is perfectly fine to simply include all of your code in a Webots directory (such as robots/ann-epuck/controllers/my-ann-controller/ above).

But if you'd like Webots Python to include your normal shell bindings, one simple approach is to start Webots from a terminal window. On the Mac, this is done by executing the following file:

```
Webots/webots.app/Contents/MacOS/webots
```

On Linux or Windows, look in similar locations for the proper executable. The bindings created by Webots will then be appended to those declared in your shell and you should be able to call all of your normal code during a Webots run. On my machine, the files *epuck_basic.py*, *webann.py*, and even the world file, *epuck.wbt* reside in directories outside the Webots file tree. However, it is still important that the controller code resides in a directory with the same name as that of the controller file (i.e *my-ann-controller.py* is in directory *my-ann-controller* above).

3.2 Controller Modules and Classes

Figure 2 illustrates relevant portions of the Webots class hierarchy, along with one possible hierarchy of neural-network classes.

Webots provides the module named *controller*, which contains all of the robot-component classes such as *Camera*, *Emitter*, and *DistanceSensor*. The *controller* module also defines the class *Robot*, which provides easy access to these components on simulated (or real physical) robots. Finally, *controller* contains the class *DifferentialWheels*, a subclass of Robot. This represents a special class of robots: those that have 2 wheels and use differential steering (i.e., the robot turns by applying different velocities to its wheels).

We provide you with the class EpuckBasic (in file epuck_basic.py), which is a subclass of *DifferentialWheels*. Should you decide against using *EpuckBasic* in your controller, at least be sure to inherit from *DifferentialWheels*, as depicted by the class *YourClass* in Figure 2.

Note that although the code that we provide is written in Python, these same Webots classes (such as DifferentialWheels, Robot, Camera, etc.) are available in C++, JAVA, and MATLAB versions of the Webots controller-building code.

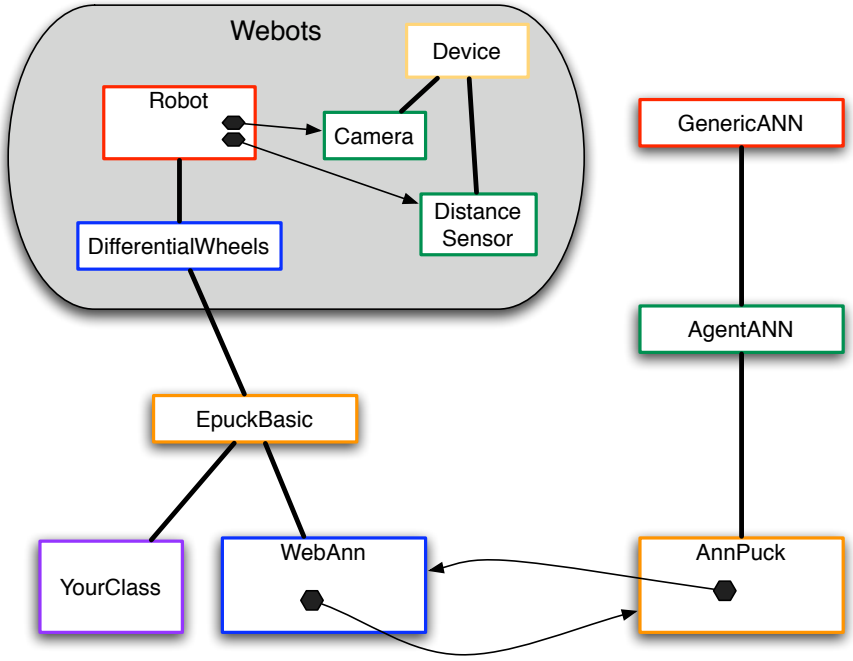

Figure 2: Class hierarchy for Webots and GenericANN descendants. Small black hexagons represent slots/attributes of a class.

3.2.1 The EpuckBasic Class

The file *epuck_basic.py* contains the class EpuckBasic. Even if you do not code your controller in Python, you are wise to read through the code and comments in that file. EpuckBasic is built as an abstraction so that IT3708 students and other users can ignore a lot of the lower-level details of interfacing with Webots robots. If you use a language other than Python, you may want to define something similar to EpuckBasic in your language. Since most of the Webots classes are the same in all (supported) object-oriented languages, your C++ or JAVA versions of EpuckBasic will probably use many of the same calls as in the Python version.

Your ANN should control the robot via EpuckBasic (or similar code that you write). In doing so, it will use a few important methods:

1. **get_proximities** - This returns a vector containing the values of the 8 distance sensors. These should all be integers between 0 and 4096.
2. **snapshot** - This returns the current picture from the robot's onboard camera, and it returns it as an *image* object. Python's *image* class provides a host of primitives for working with these objects. You

are wise to download and install the image module; basic Python installations do not include it.

3. **move, move_wheels**, and **set_wheel_speeds** - Any (or all) of these can be used to set the wheel speeds of the robot. Normally, your code will set wheel speeds based on output values of motor-layer neurons, but the wheels will not be activated to run at those speeds until a call to *do_timed_action* occurs.

4. **run_timestep** and **do_timed_action** - The former calls the latter, which sends the *step(timestep)* command to the robot, causing it to run for *timestep* milliseconds (of simulation time) with the current settings of the wheel speeds.

To force the robot to move, you thus have to do 2 things: a) set the wheel speeds, and b) call **run_timestep** or **do_timed_action**. The timestep is actually stored as a property of the world. You can access and modify it by clicking on the "WorldInfo" attribute at the top of the Scene Tree in the Webots window. The WorldInfo attribute named *basicTimeStep* is an integer denoting the number of milliseconds in a simulated timestep. It has a default value of 32, meaning that anytime you call run_timestep, the simulator will perform actions for 32 milliseconds of simulation time. How long this takes on your computer can vary - but in the world being simulated, it's 32 milliseconds worth of action.

The EpuckBasic class includes a *basic_setup* method which loads the WorldInfo.basicTimeStep into it's own *timestep* slot. This method also instantiates the camera and distance sensors.

3.2.2 The GenericANN Class and its Descendents

The upper right of Figure 2 displays the class GenericANN. This is the class that manages the creation (from scripts) and running of artificial neural networks (ANNs). It's child class, AgentANN, handles ANNs that are used in agents, such as a robot. AgentANN includes code for binding slots of the agent (such as sensors or motors) to layers of the ANN, along with code for transferring data from these slots to the appropriate neurons, and vice versa. AnnPuck, a child of AgentANN, is specialized for dealing with Epuck agents.

We provide this part of the hierarchy only as a suggestion, as you are free to implement the Generic ANN concept in many ways.

3.2.3 The WebAnn Class

The file *webann.py* defines the class WebAnn as a subclass of EpuckBasic. We provide this class as an example, but we do not recommend that your controller class inherits from WebAnn. It should inherit from EpuckBasic, but it might involve similar code to that in webann.py.

Notice that the initialization routine for WebAnn has, in its final line, a call to the epuck2.annpuck2 (a.k.a. AnnPuck) constructor. Hence, the ANN is simply a property of the WebAnn object. Our AnnPuck objects include an *agent* slot, in which the WebAnn is stored. You are free to implement the ANN-agent interaction in many ways; we merely provide a glimpse of our implementation as one possible approach.

The bottom of wabann.py contains a "Main" section. In the file containing your controller, you need to include "Main" code at the bottom for creating and activating your controller. Webots will run this code when you click on the start button of a simulation.

3.3 Epuck Details

Many of the details of the epuck robot, such as it's dimensions, sensor parameters, etc., or NOT easily available via the Webots window. To access these, go to the directory webots/projects/default/protos/ and then check out these text files (in separate subdirectories):

- *sensors/EPuck_DistanceSensor.proto* - the specfications for the epuck sensors. To understand these, you'll need to check the Webots documentation.
- *robots/EPuck.proto* - the complete specification for the epuck robot. This references the DistanceSensor.proto specification, among others.

Many of the physical details of the epuck are already incorporated into the EpuckBasic class. For example, dimensions such as the axle length and wheel radius are used to calculate rotation angles during calls to the various *spin* methods. You should not need to worry too much about epuck specifications unless you choose to implement a very advanced task.

3.4 Image Processing

In the file *imagepro.py*, we provide the Python module *imagepro* as a supplement to Python's *image* module. The functions in this file are used by the EpuckBasic code, and you may find them useful for your project. In the EpuckBasic code, we convert Webots camera data into a Python image object, which is returned by the *snapshot* method. The image class offers many useful image-processing methods for further manipulation of snapshots.

3.5 Webots Examples

Webots comes with many sample robots and worlds. Most can be found in the *robots* directory:

Applications/Webots/projects/robots

Support for Python is a very recent addition to Webots, so only a few Python examples come with Webots. In the following directory:

Applications/Webots/projects/packages/python/controllers

you will find two directories, *driver* and *slave*, each of which contains a Webots controller written in Python.

3.6 Auxiliary Files

The files *prims1.py* and *kd_array.py* are necessary support for EpuckBasic and/or imagepro, so be sure to include them.

4 Deliverables

All deliverables should be part of a short (5-10 page) report.

1. An overview of the basic structure of your code, preferably via one or a few diagrams and 1-2 pages of text. **(5 points)**
2. A clear description of the task that your epuck robot is supposed to perform. **(2 points)**
3. A script of the main ANN used in your Webots runs. **(1 point)**
4. A circuit diagram (layers, links, a few nodes, etc) of your ANN, along with a brief description of how it works. **(2 points)**
5. **Informative** text, pictures, graphs and/or diagrams illustrating the performance of your **hard-wired** robot. **(5 points)**
6. A working demo of the hard-wired robot (in simulation). **(5 points)**
7. A description of the training cases and general strategy for backprop training of your Webot. If you do not hand-design the training set, then document some of the *on-line* cases being used by the robot. **(2 points)**
8. **Informative** text, pictures, graphs and/or diagrams illustrating the performance of your **learning** robot. **(4 points)**
9. A working demo of the learning robot (in simulation). **(4 points)**

5 Important Practical Issues: Please Read!!

- We extend the legal group size to 3 for this module. So you can work alone or in groups of 2 or 3 (but no larger). ONE report and demo per group should be delivered, no more.
- This project will also be DEMONSTRATED for the instructor and course assistants on the same date that the report is delivered. We do not expect perfect performance by your robot: it may not solve the task very well. However, if your system crashes several times during the demo, then you can expect to lose some points. The robot, however, may crash into walls! That's no problem.