Practical Session 6 – Reinforcement Learning

A Basic Multi-Armed Bandit

We're going to start today by implementing a basic multi-armed bandit algorithm (UCB1) and exploring its behaviour in theory. We'll then see if we can tune the algorithm to converge in a good time, before applying it to our emergent software system.

Create a new project with a new file for a main component and (in a resources folder) another one for an interface UCB. In the interface definition, use the following type:

```
interface UCB {
        void setActions(String actions[])
        int getAction()
        void consumeData(dec reward)
        }
```

Next we'll write our main component, which will be a simple test driver for our UCB implementation. We'll read in simulated data for a set of actions, which we can use to explore the behaviour of UCB1.

Open the source file for your main component and add the following code, which uses a JSONEncoder to read in a list of values:

```
data Row {
        dec values[]
        }

component provides App requires io.File, data.json.JSONEncoder parser, UCB ucb {

        int App:main(AppParam params[])
                {
                File fd = new File("test_data.txt", File.FILE_ACCESS_READ)
                Row rows[] = parser.jsonToArray(fd.read(fd.getSize()), typeof(Row[]), null)

                String actions[]
                for (int i = 0; i < rows[0].values.arrayLength; i++)
                        actions = new String[](actions, new String("a" + i))

                ucb.setActions(actions)

                for (int i = 0; i < rows.arrayLength; i++)
                        {
                        int a = ucb.getAction()
                        ucb.consumeData(rows[i].values[a])
                        }

                return 0
                }

        }
```

We now implement the UCB1 algorithm in a component UCB.dn as follows:

```
component provides UCB requires util.Math math {

	String learningActions[]
	dec rewards[]
	dec counts[]
	int totalCount
	int currentAction

	void UCB:setActions(String actions[]) {
		learningActions = actions
		rewards = new dec[actions.arrayLength]
		counts = new dec[actions.arrayLength]
		}

	int selectAction() {
		for (int i = 0; i < counts.arrayLength; i++) {
			if (counts[i] == 0.0) return i
			}

		dec maxVal = 0.0
		int maxInd = 0

		for (int i = 0; i < counts.arrayLength; i++) {
			// UCB1
			dec bonus = math.sqrt((2.0 * math.natlog(totalCount)) / counts[i])
			dec val = rewards[i] + bonus

			if (val > maxVal)
				{
				maxVal = val
				maxInd = i
				}
			}

		return maxInd
		}

	int UCB:getAction() {
		return currentAction
		}

	void UCB:consumeData(dec reward) {
		//update learning state
		counts[currentAction] += 1.0
		dec n = counts[currentAction]
		dec value = rewards[currentAction]
		dec newValue = ((n - 1.0) / n) * value + (1.0 / n) * reward
		rewards[currentAction] = newValue
		totalCount ++

		//choose new action
		currentAction = selectAction()
		}

	}
```

Compile the entire project using a terminal window in the root directory with the command:

```
dnc .
```

Now download the `test_data.txt` file from the GitHub repository and run the main component.

We now need to write some code to let us see what's really happening in the learning system. In the main component, add some code to reveal how much time is spent taking each action and print this out at the end of the main function.

The results at this stage will look very skewed; this is because we're not yet *normalising* the reward data but are instead feeding the raw data directly into UCB.

UCB expects that all reward data is provided as a real number in the range [0…1] so we need to normalise our actual reward data into this range. To do this you'll need to decide on what the smallest and largest rewards are likely to be and use this information to translate the reward data into the needed range. For each reward value read in from the file, your translated value will be fed into the `consumeData` function instead of the raw reward value.

Now run your test again to see how the distribution of choices has been affected.

Experiments and Tuning

Next we explore how the UCB1 algorithm behaves in more detail. First analyse the data file to discover which of the three actions is the best one to take. Now compare this to the overall amount of time that the algorithm spends in this action.

Next, divide the execution time of the algorithm into fixed-size blocks (say, 20 actions) and extract the amount of time spent in each action per block over time.

How might we adjust the UCB1 algorithm so that it converges on the best action more quickly?

Programmatically Driving an Emergent System

In Practical 5 we introduced the Interactive Perception component which provides a command line interface for the Perception module API. Although this is a very convenient way to gain access to the Perception module functions, the Interactive Perception requires a human operator to manually lead the assembly process by calling the functions from the command line. We now use a programmatic way to drive the emergent system.

We have written a simple component that you can use to interact with the Perception and Assembly modules programmatically way is the Autonomous Perception component, located in the *pal* subfolder of the *Practical6* folder. This component interacts with both the Assembly and Perception module through RESTful API defined by the EmergentSys component. The EmergentSys component encapsulates both Perception and Assembly modules whilst providing their functions in a RESTful API. If you study the AutonomousPerception component code, however, you will notice that the HTTP requests are abstracted by the RESys component. The RESys component allows HTTP request for the REST API to be

made as if they were regular local calls, thus facilitating the interaction with the Perception and Assembly modules. The Autonomous Perception module provided with this practical assembles a web server by calling setMain to be executed by the EmergentSys component. Then it adds proxy to the web server, changes the web server composition and gets perception data to print it on the screen.

To execute the Autonomous Perception, you should follow the steps below:

1. Compile components in *pal* folder: "dnc . -sp ../repository"
2. Compile components in repository folder: "dnc . -v"
3. Execute in one terminal the EmergentSys component: "dana -sp ../repository EmergentSys.o"
4. In another terminal, execute the AutonomousPerception component: "dana -sp ../repository AutonomousPerception.o"
5. Execute the client you created in the previous practical assignment. In case you do not execute a client, the collection of perception data by the Autonomous perception will always return metrics and events with invalid values.

Applying Machine Learning to Emergent Software

We'll now try connecting our learning algorithm with a live emergent software system, to see if we can learn the best action to take in a real system.

You will need to update the AutonomousPerception component so that it uses your UCB1 algorithm to choose an action (setConfig()), wait some time, then get the reward level (getPerceptionData()) before getting the next action to choose from UCB1.

Note that for the learning algorithm to work properly you will need to convert *cost* into *reward* before sending any data into UCB. This is because, for the web server, a **lower** value of response time is better. This is the opposite direction to that in which UCB works, where it considers a higher value to be better. How would you convert the cost value from the system to a (normalised) reward value to give to UCB?

Once your learning algorithm is connected to the emergent system, you can test your workloads from Practical 5 to see if the learning algorithm correctly finds the best composition for each workload.

Classifiers

With a working learning algorithm, the next thing to do is write a classifier to analyse and label each environment that is experienced by the system. For each separate environment you will need a separate instance of your learning algorithm and its associated state, so that learned actions can be stored for each environment.

How might you write a basic classifier which might be able to distinguish your different workloads? Think about how you can use the event data coming from the emergent system to create different classes of environment.

Try to implement your idea, then test it by switching between different workloads to see if different environments are detected and appropriate actions are learned for each one.

Efficient search space navigation

Another major problem with real-time machine learning is quickly navigating a large search space of possible actions to quickly converge on the best one. Our basic implementation of UCB1 does not use any approach to solve this problem.

How might you achieve this? What information could you use to make more advanced inferences about how different compositions are likely to perform if chosen? Is it possible not to have to try every possible composition and still find the best one?