# AINT351 MACHINE LEARNING 2016/17
# P2.3: NEAREST SEQUENCE MEMORY LEARNING

## Part of your AINT351 laboratory journal coursework

This set of laboratory practical exercises are part of the AINT351 laboratory journal coursework. After you have completed the exercises you must write a corresponding lab report and include it in the final lab journal according to the instructions provided in the AINT351 Coursework specification document available from the module DLE site.

## IMPORTANT NOTE ON PRACTICAL

This should describe:

- What the lab tasks were
- How you tackled them
- Your results and findings.

This assessment requires you to implement and evaluate a reinforcement learning (RL) algorithm called the nearest sequence memory[1] (NSM) algorithm. NSM is an instance-based algorithm for solving partially observable Markov decision problems (POMDPs). The exercises also ask you to apply the NSM algorithm to a partially observable version of McCallum's grid-world, presented in Figure 1, Figure 2 and Figure 3.

## 1. A POMDP

Implement a function called *transition* that takes as parameters, a state from Figure 1 and an action from Figure 2. The function should return the state that is the result of taking the given action in the given state, e.g., given state 1 and action 1, the function should return state 4 and given state 10 and action 3, the function should return state 10.

| | | | | |
|---|---|---|---|---|
| 7 | 8 | 9 | 10 | 11 |
| 4 | | 5 | | 6 |
| 1 | | 2 G | | 3 |

*Figure 1: The individual states of McCallum's grid-world.*

[1]The 1995 paper by Andrew McCallum introducing this algorithm, entitled 'Instance-Based State Identification for Reinforcement Learning' is available on the module's DLE site under the reinforcement learning section.

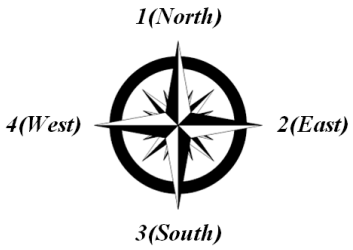


*Figure 2: The actions of McCallum's grid-world.*

Implement a function called *observation* that takes a state as a parameter and returns the observation value for that state according to Figure 3.

| | | | | |
|---|---|---|---|---|
| 9 | 5 | 1 | 5 | 3 |
| 10 | | 10 | | 10 |
| 14 | | 14 G | | 14 |

*Figure 3: The observations for each state in the POMDP version of McCallum's grid-world.*

Implement a function called *rndStartState* that returns any random state value apart from the goal state, 2

## 2. Random episodes

Implement a function called *rndEpisode*. The *rndEpisode* generates a sequence of *steps* (observation, action and discounted reward values) that lead from a random starting state to the goal state. Actions are chosen at random. The *rndEpisode* function should get a start location from the *rndStartState* function and use a loop to take random *steps* until it reaches the goal state. Each step consists of selecting a random action and updating the state and observation by using the *transition* and *observation* functions you implemented above.

For each step, the function should add a row to a matrix called *episode*. Each row should contain three values. The first value should be the current observation. The second value should be the current action taken and the third value should be the discounted reward. We don't know the discounted reward values until we reach the goal state, so set the third value to 1.0 initially.

Add a second loop that calculates the correct discounted reward value for each row after the goal has been reached. The discounted reward should be 10 for the final step leading to the goal state and should be discounted by a factor of $\gamma = 0.9$ for each step back towards the start of the episode.

The *rndEpisode* function should return two values. The first value should be the number of steps that was needed to reach the goal state from the start state. The second value should be a matrix containing the last 20 rows of the *episode* matrix, describing the 20 steps leading up to the goal state. The information about the preceding steps can be ignored.

If less than N steps were needed to reach the goal state, the matrix should be padded out with row of zeros so that the function always returns a 20x3 matrix. The rows of zeros should be added before the rows describing the random steps so that the last row, row 20, always contains the values describing the step leading to the goal state.

### 3. Random Trials

Implement a function named *rndTrial*. This function should create a 3D matrix called *LTM* (long term memory) made up of random episodes produced by calling the *rndEpisode* function you implemented, so that the expression LTM(:,:,3); would return the third 20x3 matrix produced by the third call to the *rndEpisode* function. The *rndTrial* function should take the number of episodes to be added to LTM as an argument.

The *rndTrial* function should return a vector with two values. The first value should be another vector containing the number of steps used by each of the episodes. The second value should be the LTM.

You should be able to plot the individual number of steps taken for 1000 random episodes using the MATLAB command:

```
plot(rndTrial(1000));
```

The result should look similar to Figure 4 but it should not be exactly the same.

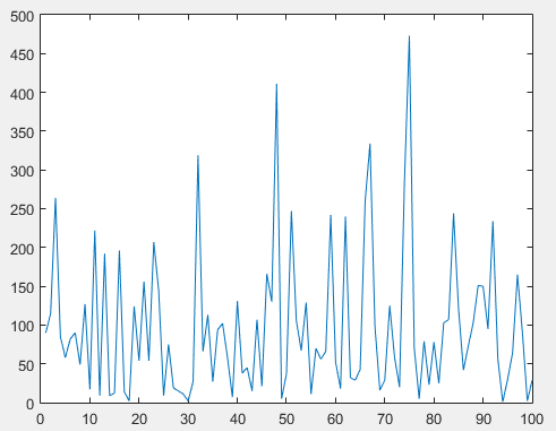


*Figure 4: The number of steps required by 100 random episodes.*

## 4. Proximity

Implement a function called *proximity*. The *proximity* function counts how many steps are similar between the current short-term memory (STM) episode and an arbitrary step in the LTM matrix. The *proximity* function should take five arguments. The first argument should be a 3D matrix of the same format as the LTM matrix above. The second argument should be the index of one of the episodes in the LTM matrix. The third argument should be the index of one of the steps in that episode. The fourth argument should be a 2D matrix of the same format as the *episode* matrix above. This matrix represents the current content of the algorithms STM. Finally, the fifth argument should be an observation value. The function should return a vector with two values. The first value is the proximity value indicating how *near* the step at the given point in the LTM matrix is to the given STM episode and observation. The second value is the vector describing the matching LTM step, observation, action and discounted reward. The rules for calculating the proximity are given below. The *proximity* function should first initialise the proximity to 0.0.

If the observation value recorded in LTM, under the episode and step indices given in the second and third arguments to the proximity, is the same as the current observation value given in the fifth argument, the proximity value should be set to 1. In this case, the function should also match previous steps in LTM with the steps in the current STM episode given in the fourth argument as described in task 6c below.

If the function set the proximity value to 1 above, the proximity function should start comparing the observation and the action values recorded in LTM and STM. First, initialise a separate STM step index value to point to the last step recorded in STM. Write a loop that compares the preceding steps in the current STM episode with the LTM steps preceding the step used in task 6a. above (indicated by the LTM indices). For each match, the proximity value should be increased by 1 and the LTM and STM step indices should be decreased by 1. When there is a mismatch, the function should terminate.

## 5. Nearest Sequence Memory

Implement a *KNearest* function. The *KNearest* function calls the *proximity* function for all the steps recorded in LTM and keeps a list of the 10 (K) steps with the highest proximities, i.e., the 10 nearest sequences. The function should loop through all the episodes and steps in the given LTM matrix and call the *proximity* function for each one. If the proximity value for either one of the steps is greater than 0, a new step-and-proximity vector containing four values (observation, action, discounted reward and proximity) should be added to a vector of nearest steps.

If nearest steps vector already contains 10 step-and-proximity vectors, the function should check whether the proximity of the latest step to be checked is greater than the minimum proximity value recorded in the nearest steps vector. If it is greater, the new step-and-proximity vector should replace the one with the minimum proximity in the nearest step vector.

## 6. NSM Action Selection

Prepare to implement the *nearest sequence memory* (NSM) algorithm by doing the following. Make a copy of the *rndTrial* function and rename it *NSMTrial*. Change the code in the *NSMTrial* function so that instead of calling the *rndEpisode* function, it instead calls a function called *NSMEpisode.* The *NSMEpisode* function takes the LTM matrix as an argument.

Make a copy of the *rndEpisode* function and rename it *NSMEpisode*. Change the code in the new function so that it takes the LTM matrix as an argument and instead of creating random action, it gets actions by calling a function named *NSMSelectAction*. The *NSMSelectAction* function takes the LTM matrix, the current STM *episode* matrix and the current observation as arguments.

Implement the *NSMSelectAction* function. The *selectAction* should calculate the average discounted reward for each action based on the K nearest sequences and return the action with the highest average discounted reward. The *NSMSelectAction* function should return a random action 10% of the time. When not returning a random action, the *NSMSelectAction* function should call the *KNearest* function to get the vector of nearest steps. It should use these to calculate the mean discounted reward for each action (some actions may not be present in the K-nearest sequences. These should have a value of 0.0. The function should return the action with the highest mean discounted reward.

## 7. NSM Trials

Run the *NSMTrial* function and plot the individual number of steps taken for 50 episodes using the MATLAB command:

```
plot(NSMTrial(50));
```

The result should like similar to Figure 5 but it should not be exactly the same.

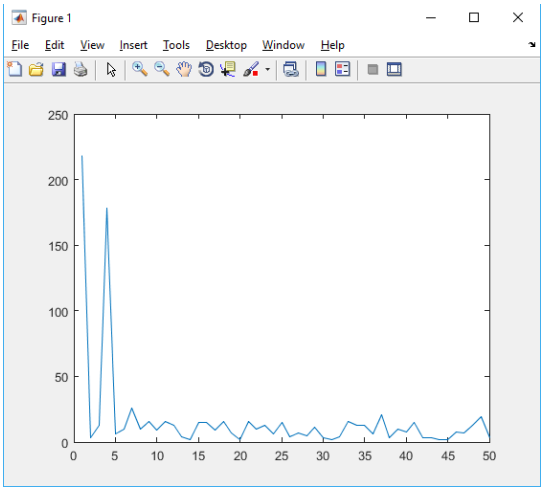


*Figure 5: The number of steps required by 100 episodes using NSM learning.*

## 8. NSM Experiment

Write an experiment function that runs 100 trials and calculates the means and standard deviations in performance for each episode number. Plot the means and standard deviation using the 'shadedErrorBar' function[2] as shown in Figure 6.

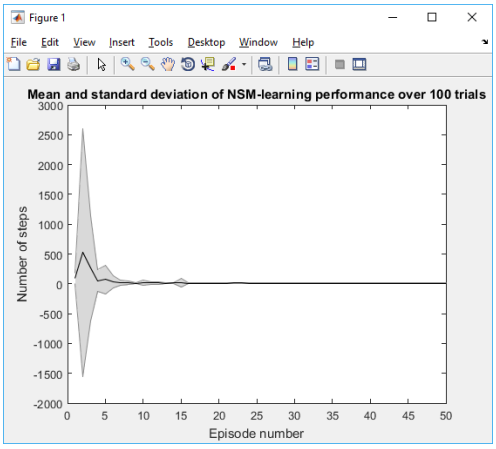


*Figure 6: NSM-learning experiment data plot.*

[2] The shadedErrorBar function is available at https://uk.mathworks.com/matlabcentral/fileexchange/26311-raacampbell-shadederrorbar. You must register for a MathWorks account using your Plymouth Univesity email in order to download it.