Report Module DBD04 Learning Robots

Bastiaan Ekeler

Marnick Menting
S050212
Billy Schonenberg
S051293
Gerrit Willem Vos
S040878

September 2009

1.1 Introduction

Within the field of robotics and product design a new paradigm on intelligent systems and robots has been explored the past few years which is the focus of this module. Learning algorithms provide students and researchers the opportunity to design robots and products that are able to learn from sensory, embodied input and seemingly random database knowledge. Complex patterns of interaction and intelligent decisions don't have to be coded up front but can be learned by the software, creating a robust and adaptive system, capable of adjusting to dynamic situations.

In this report we will show you how we've applied a reinforcement learning algorithm through theory and practice and how we think this can be used within product design. Reinforcement learning is one of the three main types of learning algorithms, the others being supervised and unsupervised learning. Both supervised and unsupervised learning is best used for a type of classification problem where mapping and modeling of input towards output has priority. However, reinforcement learning learns from acting in the world towards a specific goal to achieve. Every action has impact on the environment and the environment provides feedback in the form of rewards which are used to optimize behavior.

Starting off with an approximation of the optimal state, reinforcement learning adjusts this approximation by trial and error. Sensors define the current state and the algorithm defines the optimal action. If this action yields a reward the system optimizes the parameters and through numerous steps, it learns which actions are optimal in acquiring the goal. For example we'll project it to learning the algorithm on how to ride a bike in which the goal is to ride forward and stay upward [1, page 2]. In the first trial, the algorithm begins riding the bicycle and performs a series of actions that result in the bicycle being tilted 45 degrees to the right. At this point there are two actions possible: turn the handle bars left or turn them right. The algorithm turns the handle bars to the left and immediately crashes to the ground, thus receiving a negative reinforcement. In the next trial the algorithm performs a series of actions that again result in the bicycle being tilted 45 degrees to the right. The algorithm knows not to turn the handle bars to the left, so it performs the only other possible action: turn right. It immediately crashes to the ground, again receiving a strong negative reinforcement. At this point the algorithm has not only learned that turning the handle bars right or left when tilted 45 degrees to the right is bad, but that the "state" of being titled 45 degrees to the right is bad. By performing enough of these trial-and-error interactions with the environment, the algorithm will eventually learn how to prevent the bicycle from ever falling over.

Reinforcement learning principles yielded a number of different algorithms of which Actor-Critic, Value Iteration and Q-learning are probably best known. Actor-Critic makes use of an actor algorithm which chooses the best new action while the critic continuously "criticizes" the actor by comparing it to the value estimate. Value-Iteration, on the other hand, requires that we find the action that returns the maximum expected value over each action possible. But rather than finding the correct mapping between states, Q-learning finds a mapping between state/action pairs and Q-values. In each state there's a Q-value for the possible actions in which the Q-value for each action is the sum of the reinforcements received when performing the associated action. By avoiding the calculation of the maximum expected value over every action this algorithm is much faster in both learning and computational power.

1.2 The Platform

During the course we started working with the AdMoVeo robot platform developed within the faculty of Industrial Design, at the University of Technology Eindhoven [4]. This robot is about the size of a cd and the sensors include two line readers at the bottom, three infrared distance sensors at the sides and in the front with sensibility of 0 to 20cm, two light sensors in the front, two sound sensors at the sides and two optional encoders coupled to wheels. The actuators include two motors driving two wheels, a buzzer and a RGB color LED integrated into the acryl chassis. An XBee module is optional for wireless communication. Overall an Arduino board is used to control the robot. The reason we choose for this platform is the possibilities it has with the embedded sensors. It allowed us to freely experiment with the Q-learning algorithm and several applications. Below you'll find two images of the robot to give an impression on the size and functionality of the robot.

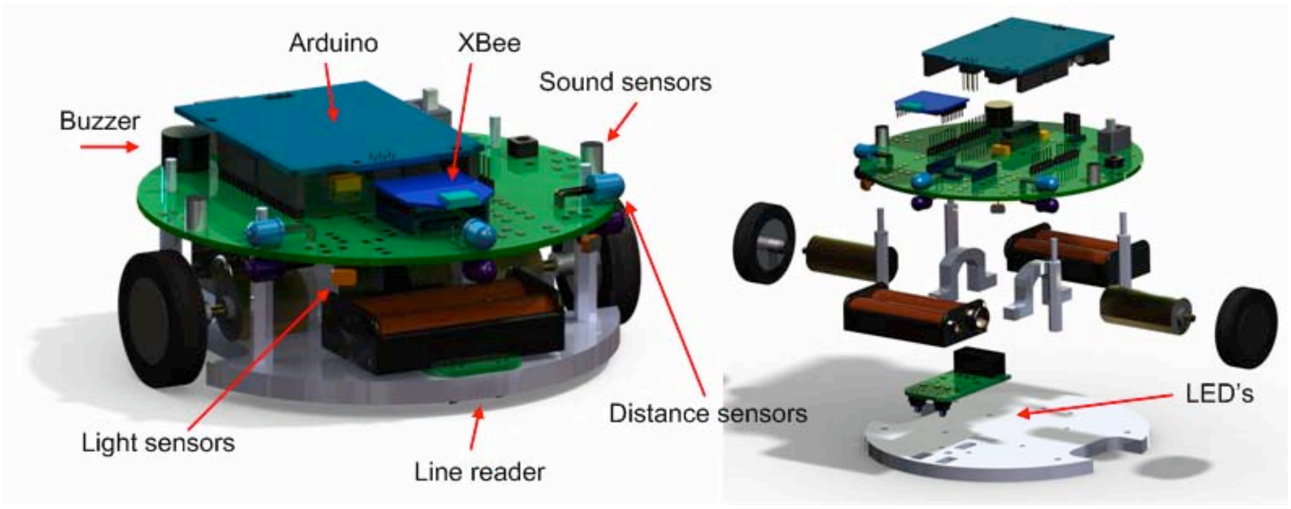

Fig 1: A 3d Rendering of the AdMoVeo robot [4]

1.3 Concept

For the initial conceptualization our aim was to create a game of Hide and Seek between two robots. Thus if one robot has been caught the other one would run off and hide, while the other robot starts searching for him. Linking this to the platform that would mean that the seeker has to detect a light source from the second robot and ride towards it in the most optimal way. When the hiding robot is tagged this would mean a negative reward and the associated state of “hiding” would yield a negative reinforcement. This hiding behavior would merely mean that the robot would run away from the light source from the seeking robot.

To implement a working algorithm the different elements of the game were filtered out of which finding a light source and define the optimal path towards it was the main objective, while wall avoiding and path following in a maze is the secondary action. Due to our inexperience with learning algorithms, creating two different robots with different algorithms and intelligent behavior was abandoned due to the high improbability that we would be able to create this before the deadline.

2.1 Q-learning Algorithm

Q-learning has the advantage over, for example, Actor-Critic, that the lack of policy greatly simplifies the analysis of the algorithm making it easier to see a convergence in the early stages of learning. Thus, proof of learning is easier detected making it more ideal for experimenting with learning algorithms.

Learning is done by determining the maximum Q-value of an action in each state, this maxQ value is given a reward and each time the agent is given a reward new values are calculated for each combination of a state *s* and action *a*. Thus, the robot determines the next best action by finding the maximum Q value in its current state for each possible action, after taking this step the algorithm verifies whether this was a good action by comparing the maxQ values for the next actions with the previous action and the reward received. If the next actions have very low Q-values and the current reward is low, the previous step is updated to have a lower Q-value, because the previous action didn’t lead into a rewarding state. This way a trail of high Q-values is created within the state-action diagrams. The formula below has been used to determine the new Q-values for the current state:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{old\ value} + \underbrace{\alpha_t(s_t,a_t)}_{learning\ rate} \times \Big[\overbrace{\underbrace{r_t}_{reward} + \underbrace{\gamma}_{discount\ factor} \underbrace{\max_a Q(s_{t+1},a)}_{max\ future\ value}}^{expected\ discounted\ reward} - \overbrace{Q(s_t,a_t)}^{old\ value} \Big]$$

This formula determines the Q-value from the previous action by multiplying the expected reward for the next action minus the old Q-value with it’s learning rate. This learning rate determines how fast the algorithm learns and is usually a value around 0.3. A higher learning rate would yield results much faster, with the added problem that when freak actions occur it will learn those actions as well. A lower learning rate slows down the learning but is more robust towards freak actions within the context.
Although the robot is able to determine the maximum Q-value for each state-action pair it still doesn’t work smoothly because the robot isn’t able to explore. By building in a small chance of a random action the robot will keep seeking for higher Q-values through different routes. Thus once it finds the current optimum value it will periodically take a different action, trying to find a higher optimum value within the state-action diagram.

2.2 Implementation

In order to come to such a behavior the different sensors and actuators need to be mapped. Fig 2 shows the concept in diagram form. The basic idea is that the the system maps the predefined actions (5 in our case, see the ‘output’ chapter later) to the different states. In order to assess whether or not the action that our robot choses is the right one, there is a reward system in the algorithm, to write all the data in a table which stores each state with the corresponding actions, and their value. For example, if the robot performs an action, random at first, in a specific state, then the value of this action is assessed through the algorithm. If the change in state was beneficial, then the value of the action for that state is updated. The next time that the robot is in that state, it will prefer the action with the highest value.
After the robot has been running for some time, the whole table is updated with the ‘best’ action per value, and so the robot has learned (mapped) the correct actions with the state, and will have ‘learned’ to perform the task. The task, of course, all depends on the settings of the actions, states, and rewards.

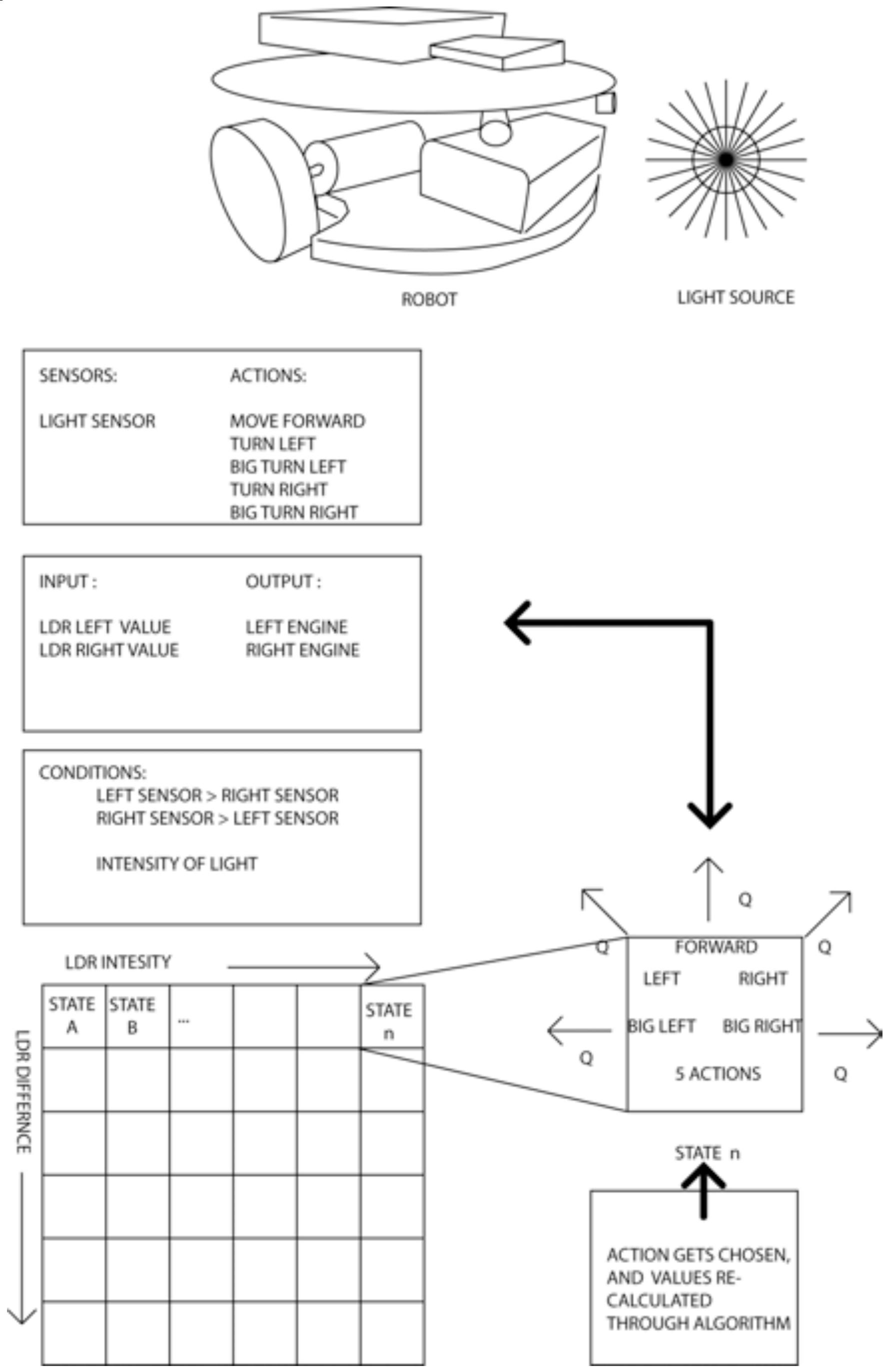

Fig 2: Concept diagrams

2.2.1 Input

To give the robot a mapping of his surroundings in order to determine the current state, we have chosen to use the light intensity sensors. The robot has one of these on his front left side, and one on his front right. Although just one sensor can only measure the intensity of the light, information from the two can be combined to get information about the relative location of the robot.

There are two different aspects to this. The average light sensitivity is one. If this sensitivity is low, this will mean that the robot is probably far away from the light source. However, this does not say anything about the angle.

The difference of the input of the two sensors is the second aspect, and gives an idea about the angle of where the light comes from, and this gives information about where the light source is located. This is useful for the robot to know, so the correct actions (F.I. turn left) can be mapped to the state, by the system. See Fig 3.

Although some experimental testing for using the distance sensors with which the AdMoVeo is also equipped for more advance functions such as collision avoiding, has been done this is not implemented in this prototype.

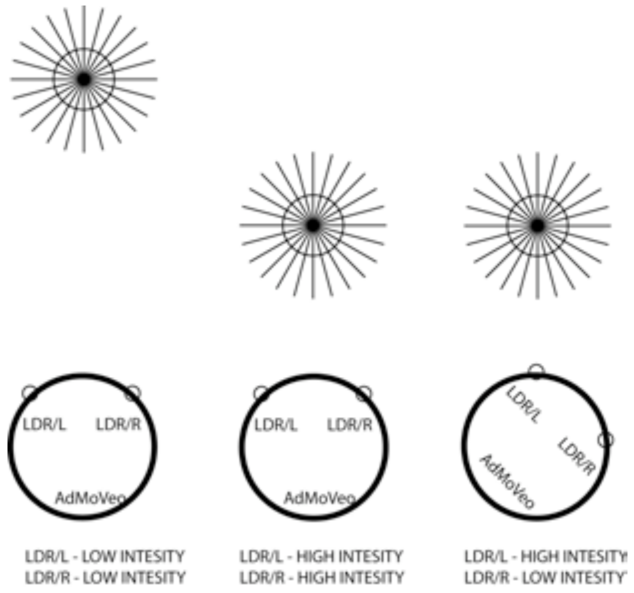

Fig 3. Different positions and Sensor input.

2.2.2 Output

The actuators that are used by our system are the two motors. To make the robot able to search for the lights, and go towards it we need 2 basic actions: ‘turn’ to search, and ‘forward’ to go towards it. However it is more beneficial to have 5 actions: one for forward, and two for each turn – left and right. This is done to achieve the effect that the robot is more sophisticated in his actions. However, the more actions, the more learning it will need. In our experiments, this number of actions proved to be a very good balance between being too crude and too complex. Fig 4 shows an overview of the actions.

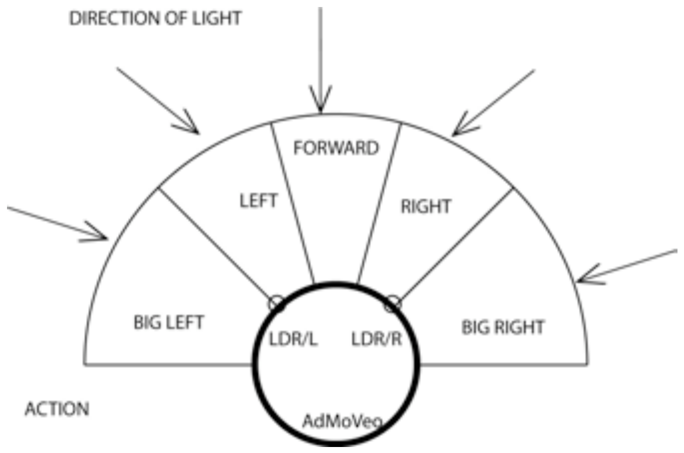

Fig 4. Overview of actions.

2.2.3 Rewards

The rewards in the system are calculated after each action and there is no terminal state. This means the robot does not evaluate its entire list of actions towards the goal (terminal state), but evaluate each step. This means we don't have to put the robot back into the same position after each run, and then let it have another go. It will learn on the fly, and adapt dynamically.

The height of the first reward is the difference between the total intensity in this state minus the total intensity of the previous state. This means if the total intensity of this state is higher that the previous the robot has not been going towards the light, and will get a negative reward. If the robot goes towards the light, than the reward will be positive. Naturally, if changes between the intensity of the state are higher, the reward is also higher.

Another reward that is programmed is a fixed negative reward. If, after 5 actions there is no change in intensity, the last action gets a reward of minus two. The reason for this is we wanted a penalty if the robot is stuck in the same place. The change in intensity is within a margin – when playing with physical objects it is very difficult to define absolutes.

2.3 Expectations

We expect the robots to have a faster learning rate when the states and actions are minimized. Thus when it has little actions to choose from, for example 3, it will be faster in learning which actions have the highest Q-value. The reason would be because the robot would have a higher chance on passing the same State-Action pairs thus they will updated more frequently.
Also, when the robot has learned the optimum path towards a light in a darkened room it should have less difficulty in approaching the same light in a brighter environment. This would be true because learning the robot in a dark environment will mean a high difference in light intensity and a wider range of states and learning tasks, needed to learn the optimum path. When placed in brighter environment with lower differences in light intensity the robot will already have learned the optimal path without it needing to learn optimal Q-values for every State-Action pair.

3.1 Execution

At the time of writing this section we are still updating the algorithm to increase the functionality of the robot. Thus the program discussed in this section is based on the current working version of the algorithm, not the final version.

The program has been designed and programmed within the Processing environment which is able to form a direct link with the AdMoVeo robot through Xbee. Within this paragraph we'll explain the basics of the program, but if your interested in the full code we can direct you to the appendix. The language used is Java and at the start of the program a State-Action grid is initiated, of which the length and content is determined in paragraph 2.2 of this report. This grid will hold the Q-values for each possible State-Action pair.

```
.....
  Q = new float[nStatesLightDiff][nStatesLightIntens][nActions];

  for(int i = 0; i<nStatesLightDiff; i++)
  for(int j = 0; j<nStatesLightIntens; j++)
    for(int k = 0; k<nActions; k++)
       Q[i][j][k] = random(0,100)/1000;
.....
```

The grid is filled with random numbers to create more explorative behavior at the start of the algorithm. We could also fill the grid with 0 but this will take the robot longer to find extreme or optimal values for State-Action pairs. The basic structure of the program is very simple consisting of the following commands:

```
void draw()
{
   getSensorValues();
   calculateAction();
   performAction();
   delay(100); // Pause the robot for a short while to stabilize the sensors before reading them
}
```

By reading the sensor values the algorithm defines the current state, in this case there were 10. These states are used to calculate the optimal action through the Q-learning algorithm and performs the required action. An important part of this program is the reward system, which is defined in the following code:

```
void getReward()
{
  reward = 0;
  averageLight = (rightLightValue + leftLightValue) / 2;
  if(averageLight < oldAverageLight + 5 && averageLight > oldAverageLight - 5 && averageLight <
        (highestLightValue + lowestLightValue) / 2)
  {
    numberRepeats++;

    if(numberRepeats >= MAX_REPEATS_FOR_PENALTY)
    {
      reward -= 2;
      numberRepeats = 0;
    }
  }
  else
    numberRepeats = 0;

  // give reward when light intensity increases compared to the previous state
  reward += (averageLight - oldAverageLight) / 10;

  // give reward for absolute light intensity
  reward += (averageLight / 1000) - 0.1;

  oldAverageLight = averageLight;
}
```

A positive reward is given when the robot's LDR's sense an increased light intensity and for the overall light intensity, a negative reward of 2 is given when the robot fails to increase it's light intensity the number of MAX_REPEATS. This way the robot will move towards the light instead of away from it. This reward is fed into the standard Q-value algorithm which is used to update the State-Action diagram:

```
void calculateAction()
{
  getReward();
  if(oldStateLightDiff >= 0)        // check if the algorithm run at least once
  {
    newAction = getMaxAction(newStateLightDiff,newStateLightIntens);
    Q[oldStateLightDiff][oldStateLightIntens][oldAction] = Q[oldStateLightDiff][oldStateLightIntens][oldAction]
        + LEARNING_RATE * (reward + DISCOUNT_RATE *
        Q[newStateLightDiff][newStateLightIntens][newAction] -
        Q[oldStateLightDiff][oldStateLightIntens][oldAction]);
  }
  // put the new states in the old states
  oldStateLightDiff = newStateLightDiff;
  oldStateLightIntens = newStateLightIntens;

  // Perform the best possible action or do a random action
  if(random(0,100)/100 > EXPLORING_RATE)
    oldAction = getMaxAction(newStateLightDiff, newStateLightIntens);
  else
    oldAction = floor(random(0,nActions));
    println("newStateLightDiff: " + newStateLightDiff + " newStateLightIntens: " + newStateLightIntens +"
        newAction: " + newAction + " reward: " + reward + " q_action: " +
        Q[newStateLightDiff][newStateLightIntens][newAction]);
}
```

The action with the optimal Q-value is filtered out of the State-Action grid using the code stated below. The corresponding values are used to calculate the Q-value of the current state with a learning rate of 0.3 (standard) and a discount rate of 0.95. To enable the robot to deviate from a current optimal path, an exploring rate of 0.1 enables the robot to perform a random action in 10% of the cases.

```
int getMaxAction(int s1, int s2)
```

```
{
  int action = 0;
    for(int x=0; x < nActions; x++)
    {
      if(Q[s1][s2][x] > Q[s1][s2][action]){
        action = x;
      }
    }
    return action;
}
```

3.2 Observations

Using the algorithm described in paragraph 3.1 we observed learning behavior after approximately 1 minute of running the program. This process went faster in a room were all light was filtered out with black cloth, leaving one single light source switched on. Also we observed that when the robot was placed right in front of the light source the learning rate was much higher than when we would put him further away from the light. After the robot had learned how to approach the light we could see a dramatic decrease in the time needed to adjust itself towards a new light source on the other end of the testing room, regardless of the strength and color of the light.

4 Evaluation

Looking at the expected outcome of the first tests and the actual observations we think we succeeded in applying a basic Q-learning algorithm on a working robot. By training the robot in a dark environment, having one light source, the robot was able to learn within about 60 seconds to drive towards the light. And afterwards, even in bright environments, the robot was capable of spotting and driving towards the brightest light source in most circumstances. By creating more sensor states and action possibilities the algorithm took longer to learn but proved to be more precise and better capable in finding different light sources. The next step would be to add the distance sensors to the states to enable the robot to learn how to avoid walls. This would be done in a similar fashion as finding the light; putting a strong negative reinforcement on driving into walls and a positive reinforcement for staying away from them.

Although the application itself was a basic example of Q-learning it does show the potential for projects within Industrial Design. Within human - product interaction many of the actions done by the user are based on reinforcement learning. By learning through trial and error a (range of) product(s) can teach itself the optimal behavior towards the user, customizing the experience. The advantage in this case is that Q-learning is relatively easy to implement for prototyping purposes, capable of learning fast and showing early results when performing a user test. Other reinforcement techniques would be more suitable if the product reaches a final state and has to show more durable and stable behavior.

Overall we are very pleased with the results and we feel we succeeded in grasping the concept of learning algorithms. By dividing ourselves in two groups of two, both working on our own robot and Q-learning, we all understood the algorithm and were able to implement it in a working robot. The presented algorithm was one of the two programs written, and although the other program also showed learning behavior, the presented program proved to be more stable, thus more appropriate for presentation purposes. Because we worked on two robots simultaneously we all had the same learning experience.

5 References

1 M.E.Harmon, S.S.Harmon, *Reinforcement Learning: A tutorial*, Wright State University

2 R.S.Sutton, A.G.Barto, *Reinforcement Learning: An Introduction*, (1998), MIT Press, Cambridge

3 M.Gasser, T.Busey, T.Pegors, *Tutorial on reinforcement learning (Q-Learning)*, Indiana University Bloomington.

4 S.Alers,J.Hu,*AdMoVeo: A robotic platform for teaching creative programming to designers*, Faculty of Industrial Design, University of Technology Eindhoven

6 The Program

```
//************************************//
```

```
// Description: This program will let the AdMoVeo robot (www.admoveo.nl) learn how to ride towards light
using the Q-learning reinforcement learning algorithm.
// Programming language: Processing
// Platform: PC + Arduino + AdMoVeo
//
// This project was done during the Master module Learning Robots, lectured by dr. E. Barakova
// Faculty of Industrial Design, Eindhoven University of Technology
// Performed by students: Bastiaan Ekeler, Marnick Menting, Billy Schonenberg, and Gerrit Willem Vos
// Last Modified: September 18th, 2009
//************************************//

import processing.serial.*;
import nl.tue.id.creapro.admoveo.*;

AdMoVeo admoveo;

// Constants are CAPITALIZED
float LEARNING_RATE = 0.3; // the rate at which the Q values are updated
float EXPLORING_RATE = 0.1; // the rate at which a exploitative or explorative action is taken
float DISCOUNT_RATE = 0.9; // the rate at which future rewards are as important as current rewards
int MAX_REPEATS_FOR_PENALTY = 5; // maximum of times
int COM_PORT = 9; // The COM port used, seen in Control Panel > System > Windows Device Manager >
Ports > USB
int MOTOR_POWER = 155; // The power at which the motors should run

// other variables
float leftLightValue; // The current value of the left light sensor, mapped between 0 and 100
float rightLightValue; // right sensor
float averageLight; // The current average light intensity, mapped between 0 and 100
float oldAverageLight; // the previous average light intensity

float highestLightValue = 0; // the highest light intensity ever found
float lowestLightValue = 101; // the lowest light intensity ever found

float reward;
int newStateLightDiff = 1;
int oldStateLightDiff = -1;
int newStateLightIntens = 1;
int oldStateLightIntens = -1;

int oldAction = -1;
int newAction = 0;

int numberRepeats;

int nStatesLightDiff = 8; // the number of light difference states you have
int nStatesLightIntens = 2; // the number of light intensity states you have
int nActions = 6; // the number of possible actions
float[][][] Q;

void setup()
{
  admoveo = new AdMoVeo(this, "COM" + COM_PORT);
  admoveo.getLeftLightSensor().enable();
  admoveo.getRightLightSensor().enable();

  // initialize the table with a q value for every state-action pair
  Q = new float[nStatesLightDiff][nStatesLightIntens][nActions];
  for(int i = 0; i<nStatesLightDiff; i++)
  for(int j = 0; j<nStatesLightIntens; j++)
    for(int k = 0; k<nActions; k++)
       Q[i][j][k] = random(0,100)/1000;
}

void draw()
{
    getSensorValues();
    calculateAction();
    performAction();
    delay(100); // Pause the robot for a short while to stabilize the sensors before reading them
}

void getSensorValues()
{
  println("leftLightSensor: " + leftLightValue + " rightLightSensor: " + rightLightValue);

  float average = (leftLightValue + rightLightValue) / 2; // The average of current light sensor values

   // remember the highest and the lowest value ever
  if(average > highestLightValue)
    highestLightValue = average;
  if(average < lowestLightValue)
    lowestLightValue = average;

  float averageHighLow = (lowestLightValue + highestLightValue) / 2; // The average of the lowest and
highest light sensor value ever found

  // define the states based on the difference between left and right light sensor
  float diff = (leftLightValue - rightLightValue);
  if(diff < -30)
    newStateLightDiff = 0;
  else if(diff >= -30 && diff < -20)
    newStateLightDiff = 1;
  else if(diff >= -20 && diff < -10)
    newStateLightDiff = 2;
  else if(diff >= -10 && diff < 0)
    newStateLightDiff = 3;
  else if(diff >= 0 && diff < 10)
    newStateLightDiff = 4;
  else if(diff >= 10 && diff < 20)
    newStateLightDiff = 5;
  else if(diff >= 20 && diff < 30)
    newStateLightDiff = 6;
  else
    newStateLightDiff = 7;

  // compare the average with the average of highest-lowest, define the states based on absolute light
intensity
  if(average < averageHighLow)
    newStateLightIntens = 0;
  else
    newStateLightIntens = 1;
}

void calculateAction()
{
    getReward();
    if(oldStateLightDiff >= 0) // check if the algorithm run as least once
    {
      newAction = getMaxAction(newStateLightDiff,newStateLightIntens); // get the action with the highest Q
value
      // the magic formula that does the learning and updates the Q value of the previous state
      Q[oldStateLightDiff][oldStateLightIntens][oldAction] = Q[oldStateLightDiff][oldStateLightIntens][oldAction]
+ LEARNING_RATE * (reward + DISCOUNT_RATE * Q[newStateLightDiff][newStateLightIntens][newAction]
- Q[oldStateLightDiff][oldStateLightIntens][oldAction]);
    }
    // put the new states in the old states
    oldStateLightDiff = newStateLightDiff;
    oldStateLightIntens = newStateLightIntens;

    // Perform the best possible action or do a random action
    if(random(0,100)/100 > EXPLORING_RATE)
      oldAction = getMaxAction(newStateLightDiff, newStateLightIntens);
    else
      oldAction = floor(random(0,nActions));
      println("newStateLightDiff: " + newStateLightDiff + " newStateLightIntens: " + newStateLightIntens +"
newAction: " + newAction + " reward: " + reward + " q_action: " +
Q[newStateLightDiff][newStateLightIntens][newAction]);
}

// return the action with the highest Q value given the LightDiff state and LightIntens state
int getMaxAction(int s1, int s2)
{
    int action = 0;
      for(int x=0; x < nActions; x++)
      {
        if(Q[s1][s2][x] > Q[s1][s2][action]){
          action = x;
        }
      }
      return action;
}

void getReward()
{
    reward = 0;
    averageLight = (rightLightValue + leftLightValue) / 2;
    if(averageLight < oldAverageLight + 5 && averageLight > oldAverageLight - 5 && averageLight <
(highestLightValue + lowestLightValue) / 2)
    {
      numberRepeats++;

      if(numberRepeats >= MAX_REPEATS_FOR_PENALTY) // if the light intensity stays the same for a
number of times and it is lower than the average, penaltize
      {
        reward -= 2;
        numberRepeats = 0;
      }
    }
    else
      numberRepeats = 0;

    // give reward when light intensity increases compared to the previous state
    reward += (averageLight - oldAverageLight) / 10;
    // give reward for absolute light intensity
    reward += (averageLight / 1000) - 0.1;

    // give penalty for making the light intensities different
    //reward -= abs(rightLightValue - leftLightValue) / 100;

    oldAverageLight = averageLight;
}

// Send the actual commands to the wheels of the robot to drive
void performAction() {
  switch(newAction)
  {
    case 0:
    admoveo.getLeftMotor().backward();
    admoveo.getRightMotor().forward();
    admoveo.getLeftMotor().setPower(MOTOR_POWER);
    admoveo.getRightMotor().setPower(MOTOR_POWER);
    delay(100);
    break;
    case 1:
    admoveo.getLeftMotor().forward();
    admoveo.getRightMotor().backward();
    admoveo.getLeftMotor().setPower(MOTOR_POWER);
    admoveo.getRightMotor().setPower(MOTOR_POWER);
    delay(100);
    break;
    case 2:
    admoveo.getLeftMotor().backward();
    admoveo.getRightMotor().forward();
    admoveo.getLeftMotor().setPower(MOTOR_POWER);
    admoveo.getRightMotor().setPower(MOTOR_POWER);
    delay(200);
    break;
    case 3:
    admoveo.getLeftMotor().forward();
    admoveo.getRightMotor().backward();
    admoveo.getLeftMotor().setPower(MOTOR_POWER);
    admoveo.getRightMotor().setPower(MOTOR_POWER);
    delay(200);
    break;
    case 4:
    admoveo.getLeftMotor().forward();
    admoveo.getRightMotor().forward();
    admoveo.getLeftMotor().setPower(MOTOR_POWER);
    admoveo.getRightMotor().setPower(MOTOR_POWER);
    delay(150);
    break;
    case 5:
    admoveo.getLeftMotor().forward();
    admoveo.getRightMotor().forward();
    admoveo.getLeftMotor().setPower(MOTOR_POWER);
    admoveo.getRightMotor().setPower(MOTOR_POWER);
    delay(150);
    break;
  }
  admoveo.getLeftMotor().setPower(0);
  admoveo.getRightMotor().setPower(0);
}

// Just read out the sensors and put them in the global variables, leftLightValue and rightLightValue
void inputAvailable(Sensor sensor, int oldValue, int newValue){
    if(sensor == admoveo.getLeftLightSensor()){
        admoveo.execute("updateLeftLight", AdMoVeo.NOW);
    }
    else if(sensor == admoveo.getRightLightSensor()){
        admoveo.execute("updateRightLight", AdMoVeo.NOW);
    }
}

void updateLeftLight(SensorStatus s){
  leftLightValue = s.get(admoveo.getLeftLightSensor());
  leftLightValue = map(leftLightValue, 0, 1023, 0, 100) - 5;
}

void updateRightLight(SensorStatus s){
  rightLightValue = s.get(admoveo.getRightLightSensor());
  rightLightValue = map(rightLightValue, 0, 1023, 0, 100);
}
```