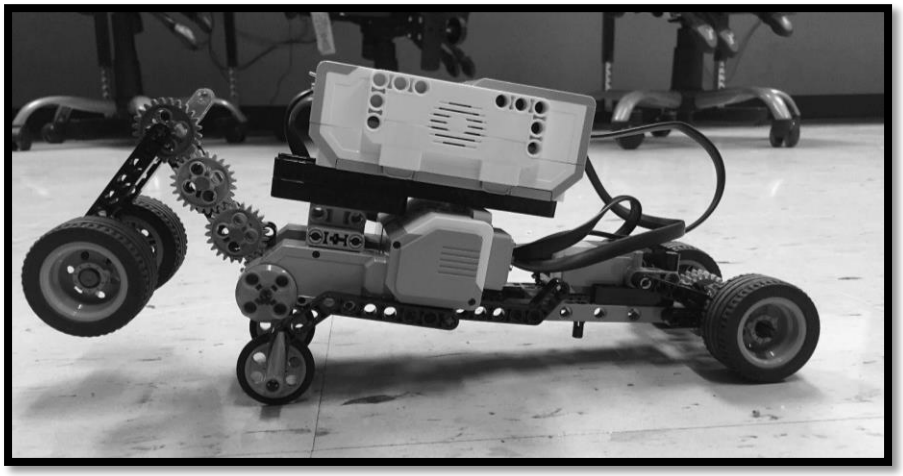
Reinforcement Learning with Lego Mindstorms

Author: Tyler Timm

Spring 2016 Undergraduate Research



Introduction

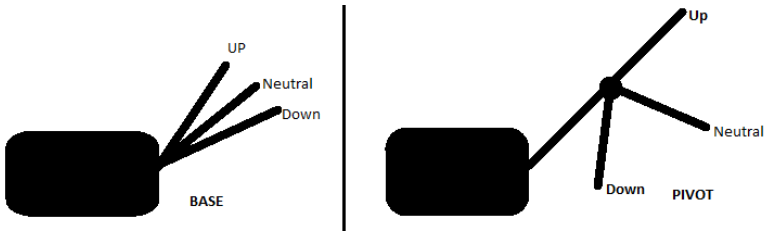
Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. It allows machines and software agents to automatically determine the ideal behavior within a specific situation by giving it reward feedback. There are many different techniques that are used for reinforcement learning, but the one I chose to implement is known as Q-Learning. The point of this research was to build a simple robot that would use reinforcement learning to learn how to walk. I chose to build the robot using Lego Mindstorms for a couple reasons. First, the parts are easy to work with and the University of Wisconsin La Crosse had many replacement parts available in case of sensor failure. Secondly, programming these robots is very simple yet powerful when you use the Lejos language. Lejos is very similar to Java and allows many of the same data structures which makes it easy to pick up for anyone with previous programming experience. I made a simple video demonstrating the results that can be found at https://www.youtube.com/watch?v=bVbT9zkPIvs . If you would like the source code feel free to email me at tyler1397@hotmail.com.

Q-Learning

Q-learning is a reinforcement technique that is used to find an optimal action policy for a given MDP. The formula for q-learning is Q(S, A) = Q(S, A) + a (r + g (Q(S' , A')) – Q(S, A)).

Q(S, A) = current state action pair
S = current state
A = current state's next action
a = alpha (the learning rate)
r = reward the current State Action pair receives for taking the action
g = gamma (the discount factor)
S' = the new State it will end up in after taking the action
A' = the new State's action
Q(S', A') = next state action pair

The first thing I had to do in order for this experiment to work was to figure out my state space. I built the robot with two motors, one controlling the base arm, and one controlling the pivot arm. I decided that each arm would have three positions: *UP*, *NEUTRAL*, and *DOWN*.



Each position is represented by a number, *UP* = 0, *NEUTRAL* = 1, and *DOWN* = 2. A state in the state space consists of a pair of numbers that represent each arm's current position. The first number represents the position of the base arm, and the second represents the position of the pivot arm. Each state maps to an action that will transition it to the next state. There are a total of six possible actions labeled 0-5. Each number represents a position of the robots arm. For example, action 0 would simply mean that the robot should rotate its base arm to the *UP* position. How this was handled will be discussed later under the robot section. Below I have included some useful information to help describe the state space as well as the actions associated with it.

Action	Description
0	Rotate base arm to the *UP* position
1	Rotate base arm to the *NEUTRAL* position
2	Rotate base arm to the *DOWN* position
3	Rotate pivot arm to the *UP* Position
4	Rotate pivot arm to the *NEUTRAL* Position
5	Rotate pivot arm the *DOWN* Position

State (56 total)	Actions
{0, 0} Base arm up, pivot arm up	{0,1,2,3,4,5}
{0, 1} Base arm up, pivot arm neutral	{0,1,2,3,4,5}
{0, 2} Base arm up, pivot arm down	{0,1,2,3,4,5}
{1, 0} Base arm neutral, pivot arm up	{0,1,2,3,4,5}
{1, 1} Base arm neutral, pivot arm neutral	{0,1,2,3,4,5}
{1, 2} Base arm neutral, pivot arm down	{0,1,2,3,4,5}
{2, 0} Base arm down, pivot arm up	{0,1,2,3,4,5}
{2, 1} Base arm down, pivot arm neutral	{0,1,2,3,4,5}
{2, 2} Base arm down, pivot arm down	{0,1,2,3,4,5}

Gamma, or the discount factor, was set to 0.9 and never changed. I used a fairly simple reward function, giving -1 if the robot did not move after its action, and +5 if it did. Below is the algorithm I used to implement the q-learning formula:

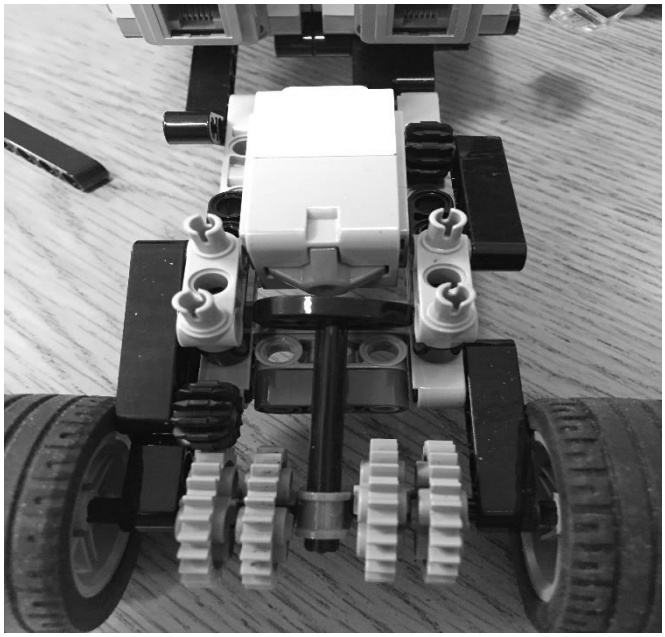
```
epsilon = 0.9
alpha = 0.9
gamma = 0.9
episodes = 20
for(int i = 0; i < episodes; i ++){
        -set current state s to the best possible start state, choose greedily based on current policy
        -decrement Alpha and Epsilon by 0.1 every 2 episodes

        for(int s = 0; s < 55; s ++){
                if(Math.random() < epsilon){
                        -choose the action randomly
                        -observe the next state s', get reward( -1 if it doesn't move and +5 if it does)
                        -update U(s) = U(s) + alpha( reward + gamma(U(s')) - U(s) )
                        -set current state s to s'
                }else{
                        -choose the action greedily based on the current policy
                        -observe the next state s', get reward( -1 if it doesn't move and +5 if it does)
                        -update U(s) = U(s) + alpha( reward + gamma(U(s')) - U(s) )
                        -set current state s to s'
        }
}
```

After many different test runs I found that decreasing epsilon and alpha by 0.1 every 2 episodes gave the best results. By the time alpha is zero, the robot is no longer learning and is simply following the policy that it learned (see final page). I saved the value function, or the value that is being modified by the Q-learning algorithm, within the state class itself which allowed for easier updates. Ideally, the state class should contain two integer values representing the positions of the base and pivot arm, the state's action, and the value associated with the state. The entire state space was stored in a linked list data structure.

The Robot

The robot went through a lot of different prototypes before a final design was used. I was able to build the robot's arm by setting two motors closely together and having one control the base, and another control the gears that turned the pivot. The wheels closest to the robot's arm are stationary to allow for the robot to learn how to walk instead of simply pushing itself on the wheels. Tracking the robot's movement was one of the hardest parts due to the simple sensors that are included with the robot. I initially tried the distance sensor: however, it only worked up to 30 cm away from a wall and I wanted the robot to have more flexibility. My final design involved a pair of wheels, a touch sensor, and a type of piston setup which is located at the front of the robot. Every time the robot goes to take an action it first checks to see if the touch sensor is pressed or not. After the action is completed it then checks it the state of the touch senor has changed (pressed, or not pressed). If it changes, it means the robot has moved. Now with this system the sensor wouldn't always pick up if the robot had moved so I adjusted my reward function to compensate for it. A simple example would be if the touch sensor is initially pushed, but the action caused enough force to press and then release the touch sensor. This would result in the touch sensor state not being changed, and even though the robot moved, the sensor says otherwise. Another big issue with this reward system was there was no way to tell if the robot was moving forward or backward. In order to fix this I installed an additional part that only allows the wheels to spin one way. Below is a picture of the reward sensor.



Due to the robot's primitive sensors and motors I was unable to simply state Motor.A.rotate(140). I found that when I did this, if there was any resistance at all, the motor would not accurately rotate 140

degrees. I was able to come up with a new idea which allowed the robot to reset and recalibrate every episode. The initial positions of the two arms at the beginning of every episode were *UP*, *UP*. When I called reset I would tell both arms to extend until they stall, which resulted in both arms being in the *UP* position. I created two separate arrays called basePositions and pivotPositions which held arm positions. At this point I called recalibrate which would set the "0" entry to the current positions of the arms in the *UP* position. When that is finished it loops over the remaining entries subtracting from the previous entry. To better help visualize let's say that after reset both arms are at 180 degrees. This is what the arrays look like after recalibrate. NOTE: I decrement Pivot positions more because I want the pivot arm to move farther then the base arm does.

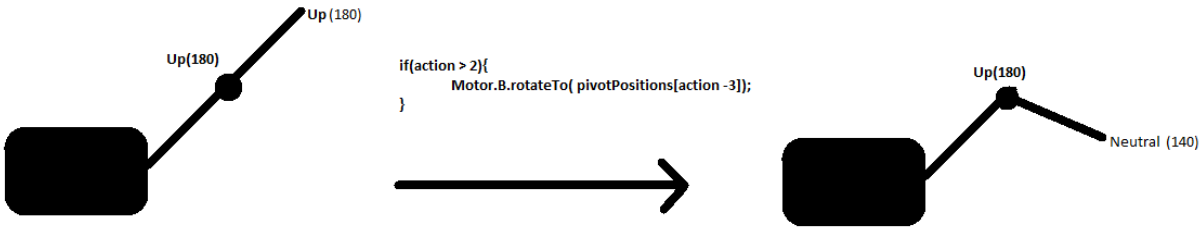
Base Positions

0 (UP)	180
1 (NEUTRAL)	160
2 (DOWN)	140

Pivot Positions

0 (UP)	180
1 (NEUTRAL)	140
2 (DOWN)	100

Since the states action can be any number between 0-5 all I need to do is check to see if the robot is moving its base or its pivot arm. For example, state s = {*UP*, *UP*} action: 4. I see that the action is 3 or greater which means I will be moving the pivot arm. I can now say, Motor.B.rotateTo(pivotPositions[action–3]). This breaks down to saying rotate the pivot arm to the neutral position.



I have included a sample final policy below, if you have any more questions email me and let me know.

This is a draft so data may be added or deleted without notice.

```
0 0 0 -3.6742989541244592
0 0 1 -3.6917116915097017
0 0 2 -3.8066483858818008
0 0 3 -3.942147879707496
0 0 4 -3.6616445081829707 <<Highest start value
0 0 5 -3.7493622058040854
0 1 0 -3.600410598020205
0 1 1 -3.700728322264118
0 1 2 -3.53389540417173
0 1 3 -3.5541242338575145
0 1 4 -3.639349662937985
0 1 5 -3.6353073640507847
0 2 0 -3.8006133097459114
0 2 1 -3.6191476048992404
0 2 2 -3.6776483135419133
0 2 3 -3.5924369011418826
0 2 4 -3.5091926385826233
0 2 5 -3.588205098739641
1 0 0 -3.631320922876912
1 0 1 -3.8193107294859168
1 0 2 -3.631916736506294
1 0 3 -3.913000868757647
1 0 4 -3.7092344101519727
1 0 5 -3.6514348436464727
1 1 0 -3.6564189036981394
1 1 1 -3.8587640028284516
1 1 2 -3.5974547675329998
1 1 3 -3.5348635270098496
1 1 4 -3.734069149098075
1 1 5 -3.73168434596434
1 2 0 -3.672785191356695
1 2 1 -3.8617635119650577
1 2 2 -3.7170227379223886
1 2 3 -3.594544586191715
1 2 4 -3.594769759659062
1 2 5 -3.662785050450614
2 0 0 -3.726890929515837
2 0 1 -3.7587941876441224
2 0 2 -3.8509630234669148
2 0 3 -3.810758963929236
2 0 4 -3.6120188948413734
2 0 5 -3.6992031259872364
2 1 0 -3.6331648748955057
2 1 1 -3.501657576115208
2 1 2 -3.517169409080573
2 1 3 -3.613041820017452
2 1 4 -3.6215730515279585
2 1 5 -1.3304612180322863
2 2 0 -3.4903288163308552
2 2 1 -3.63169063081534
2 2 2 -3.7894501770543396
2 2 3 -3.735822770990959
2 2 4 -1.1393795872961774
2 2 5 -3.507913554921458
```