Embedded Agent for Arm Control

PROGRAMMING ASSIGNMENT

CS394R Fall 2016

1 Motivation

Embedded implementations of reinforcement learning agents will be key to enabling expressive interactions with everyday objects. These agents, which may not always have access to off-board computing resources, must learn from real-world interactions using limited memory and clock cycles. To explore the issues facing embedded agents, I built a robot and solved a simple control problem using reinforcement learning.

2 Introduction

2.1 Learning Platform

I based the platform on the Atmel ATmega328p, a cheap and widely available 8-bit microcontroller. It has 32kb of program memory, 2kb of SRAM and operates at 16MHz. It is used by popular hobbyist electronics boards like the Arduino Uno, and for this project, I used an Arduino Pro Mini. It does not have a floating point unit, though the Arduino runtime provides a software implementation that mimics 32-bit IEEE float.

In order to provide a reasonably complex task for the agent, I decided to build a two degrees of freedom arm with an LED actuator. The arm's joints are SG90 micro-servos, each capable of 155° of rotation. The base joint is fixed to a surface, and the elbow joint is mounted to the base with a pair of three-inch dowel rods. The elbow joint controls another rod which is tipped with an LED. The motors are connected such that the middle of their rotation ranges align.

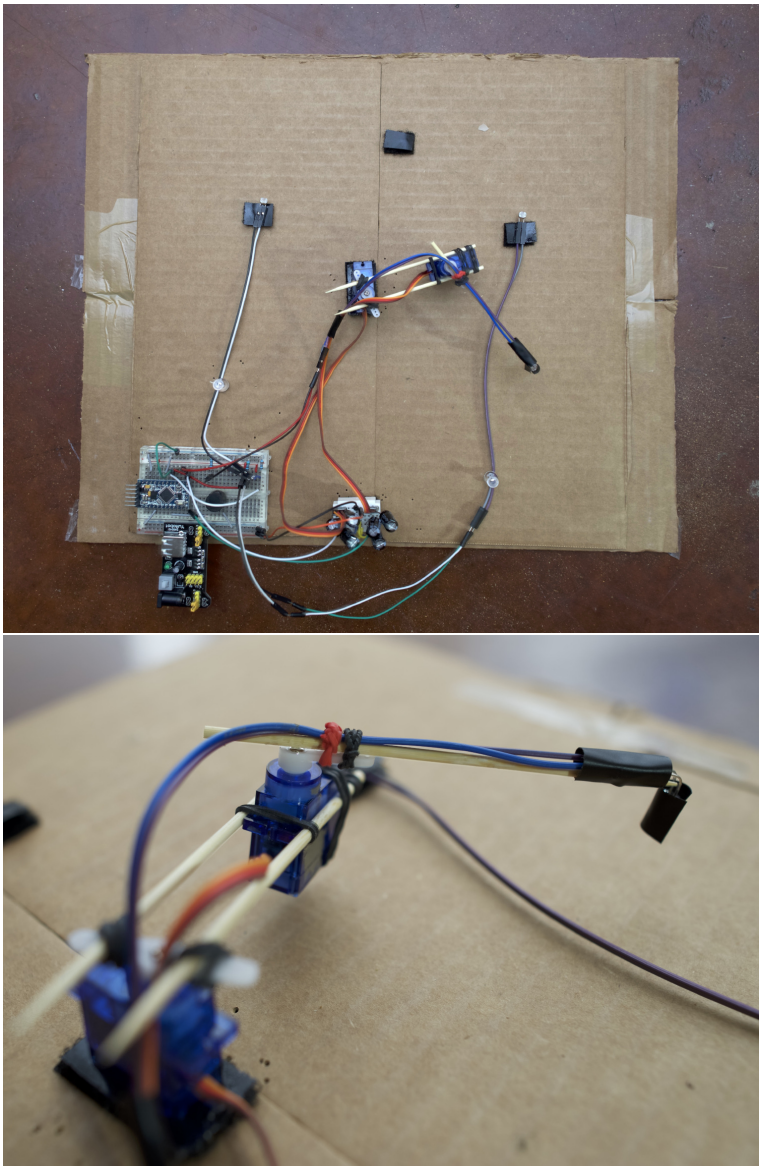

Figure 1: Top: The learning platform. The middle of the joints rotation range is the configuration that has both joints pointing north. Two photocells are pictured, but only the left one was used. Black tape marks the unused position for a third. Bottom: The arm in detail. Note that the LED has its sides covered to reduce the width of its light cone.

2.2 Problem

The agent must point the LED at a photocell fixed to the surface in as few movements as possible, activating the LED as little as possible. It begins from a random initial configuration. The episode ends when the photocell reads above a defined threshold.

$$r(s, a, s') = \begin{cases} 50 & \text{if photocell activated} \\ -2 & \text{if LED activated and photocell not activated} \\ -1 & \text{otherwise} \end{cases}$$

2.3 Learning Approaches

2.3.1 Tabular?

The servo control library used for this project allows motor targets to be set with single-degree precision, so a single joint can have integer positions in $[0°, 155°]$. For each configuration, the LED can be either on or off, so there are a total of 48050 states. At a given time step, the agent may choose to keep a joint fixed, move it left, or move it right and it can choose to activate or deactivate the LED. Assuming we restrict the agent to movements of unit magnitude, this means there are eighteen actions.

A state-action value table based on this representation, assuming 4 byte floats, would occupy more than 4 megabytes of memory. Due to the spread of the light of the LED, it may be feasible to reduce the fidelity of the joint state representation and still achieve good performance, and because the optimal policy will likely always elect to move a joint, we may be able to remove actions which do not move a joint with little adverse effect. Even then, the microcontroller could only theoretically fit 10% of all state action pairs (less without careful optimization, as the stack needs to live in memory as well). A tabular approach is not feasible due to the system's memory constraints.

2.3.2 Function Approximation

Approximation allows tight control of the amount of memory being used to represent the value function. Because of the complex update step however, only at most half of the memory can be used for storing weights. Even then, the update step must be implemented carefully to control the stack size. Consider the episodic semi-gradient one-step Sarsa update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \Big[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}\, \boldsymbol{\theta}_t) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t) \Big] \Delta \hat{q}(S_t, A_t, \boldsymbol{\theta}_t) \quad (1)$$

And in the linear case:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \Big[R_{t+1} + \gamma \, \boldsymbol{\theta}_t^\top \boldsymbol{\phi}_{t+1} - \boldsymbol{\theta}_t^\top \boldsymbol{\phi}_t \Big] \boldsymbol{\phi}_t \tag{2}$$

It is possible to implement the update using only n additional space, where n is the number of weights, but this is easy to do incorrectly. If the action selection step is placed after the memory allocation, the stack will consume $2n$ memory; maximizing the value function over possible next states requires an additional n stack space.

Algorithm 1 Memory-conscious Episodic Semi-gradient One-step Sarsa

procedure UPDATE(S_t, A_t, S_{t+1}, $\boldsymbol{\theta}$)
    $A_{t+1} \leftarrow$ choose action from S_{t+1} according to policy
    Allocate $\boldsymbol{x}$ to be a vector the size of $\boldsymbol{\theta}$, and floats r and a
    $r \leftarrow r(S_t, A_t, S_{t+1})$
    $\boldsymbol{x} \leftarrow \phi(S_{t+1}, A_{t+1})$ ▷ store ϕ_{t+1}
    $a \leftarrow \boldsymbol{\theta}^\top \boldsymbol{x}$ ▷ calculate $v(S_{t+1}, A_{t+1})$ so we can discard ϕ_{t+1}
    $\boldsymbol{x} \leftarrow \phi(S_t, A_t)$ ▷ store ϕ_t
    $a \leftarrow r + \gamma v - \boldsymbol{\theta}^\top \boldsymbol{x}$ ▷ a is now the bracketed term in eq. 1
    $\boldsymbol{x} \leftarrow (\alpha a)\boldsymbol{x}$ ▷ $\boldsymbol{x}$ is now the weight update
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + x$
end procedure

An approximation approach implemented on the microcontroller can use at most 1kb of RAM, or 250 features (less the incidental stack space required during the update step). It was not clear that this would be sufficient for good performance, but it was the only feasible approach. I implemented a semi-gradient one-step Sarsa agent using a linear function approximator.

I have not dwelt on time efficiency since, even with software floating point operations, 16MHz permits a fair amount of computation. For this project, I was satisfied as long as actions could be selected more quickly than the servos could execute them. Meeting this deadline, which was about 100ms, or 1.6 million cycles, was not an issue, even while the device was also streaming logging information over serial. If time performance requirements were tighter, special attention would need to be paid to the action selection process, which involves $|A|$ value function queries, each costing n multiplications

3 Experimental Setup

The agent learns for 50 episodes using $\alpha = 0.2$, $\gamma = 0.99$, following an ϵ-greedy policy with $\epsilon = 0.1$. Then, the agent operates in an evaluation mode for 50 episodes with $\alpha = \epsilon = 0.0$. During this period, episodes are limited to 200

steps in case the agent executes a policy that never reaches the goal. A small delay is used between action execution and sensing to allow the arm to settle. The photocell threshold is calibrated before every session to prevent spurious activations.

3.1 Features

I used a simple discretization of the state space. The range of each joint was divided into 8 20° sections. The two sets of section features along with a binary feature for the LED state were then gridded, resulting in 128 mutually exclusive binary features.

Because of the large number of actions, it was not feasible to maintain separate approximators per action, so I used three action features: two characterizing the direction of each joint's movement, taking values in $\{0, 1, -1\}$, and one binary feature describing whether or not the LED is activated by the action. Clearly, three features is insufficient to encode the differences in value for all actions across the state space, however there was not enough memory to grid these features with the state features. The total number of weights in the representation is still below the theoretical maximum number of weights for the microcontroller, so additional features could have been added with more time.

3.2 Actions

At each time step, the agent selects one entry from each column to form its action. If the LED is already on and the agent chooses an action that includes turning the LED on, the light remains activated for the next timestep. The behavior is symmetric in the LED off case.

Joint 1	Joint 2	LED
Move left	Move left	Turn on
Move right	Move right	Turn off
No movement	No movement	

To better support the state representation, I set the agent's movement increment to 20°. This ensures that the state resulting from any movement has a different feature vector than the previous state. This change does limit the granularity of the agent's movements, but without it, the agent must randomly escape the 20° sections by repeatedly making smaller movements, which makes learning take much longer.

4 Results

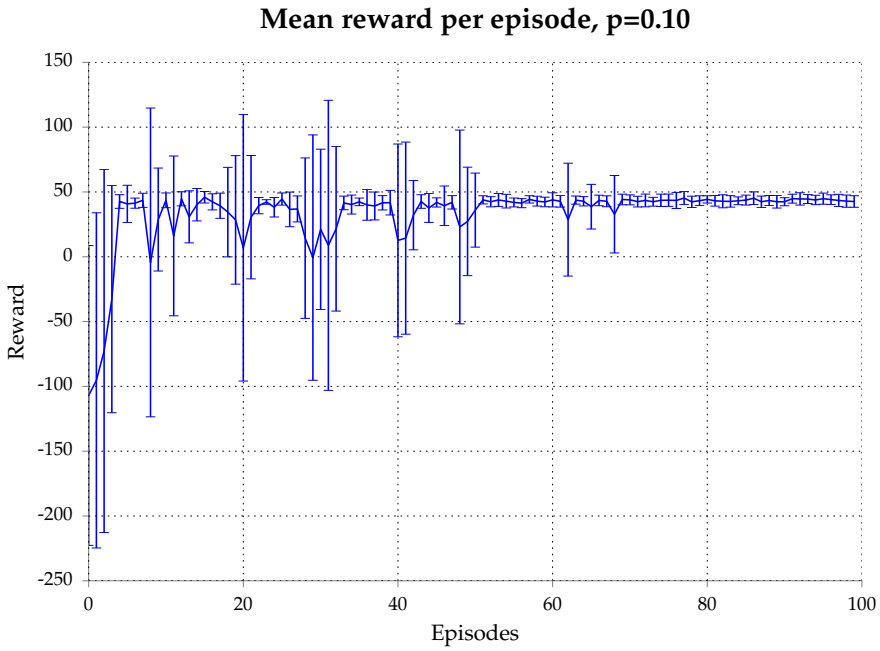

Figure 2: The agent's performance. The first 50 episodes are learning time, the rest are evaluations. Due to the time expense of collecting data from the robot, I could only run ten trials. Still, within reasonable confidences, the agent's evaluation averages a reward of 44, demonstrating that it was able to quickly point the LED towards the photocell from arbitrary start positions.

4.1 Discussion

The agent learns and generalizes a fairly good policy within its first fifty steps. The evaluation period demonstrates that the learned policy performs well from arbitrary start positions.

Even though it points the LED at the photocell consistently, the agent does not learn the LED reward dynamics, opting to simply leave the light on at all times. This is not unexpected, since it lacks features that describe the interaction of the joint position with the value of actions that turn the LED on. The weight associated with the LED activation action-feature is forced to represent the value of LED actions from any state. Averaged across the entire state space, turning the light on has a higher value than turning it off, so it always leaves it on. Efficient

LED activation is not as important as joint movement, so it seems reasonable to prioritize detailed state-space representation over features that would better capture the use of the light.

5 Conclusions

I have demonstrated that it is possible to implement an embedded agent that achieves good performance on an arm control task, even with extraordinary memory constraints. Further work could investigate more sophisticated features, or explore the performance and memory characteristics of policy gradient methods in the same domain.

6 Appendix A: Images

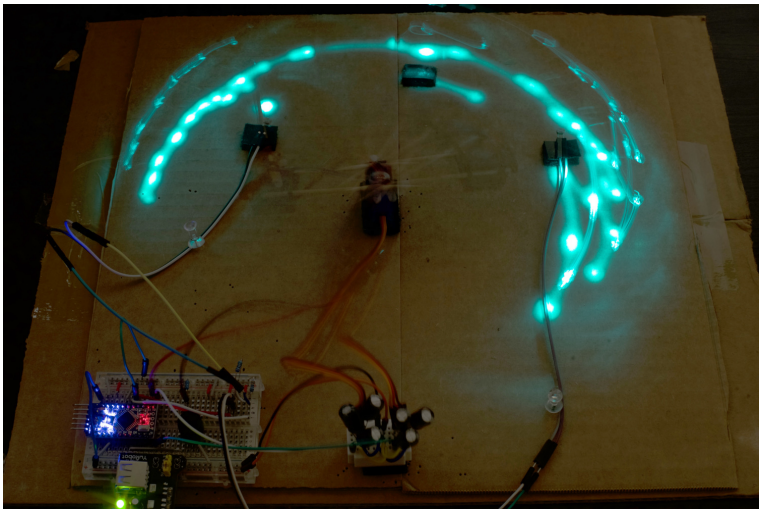

Figure 3: A 30 second exposure taken early in the agent's learning.

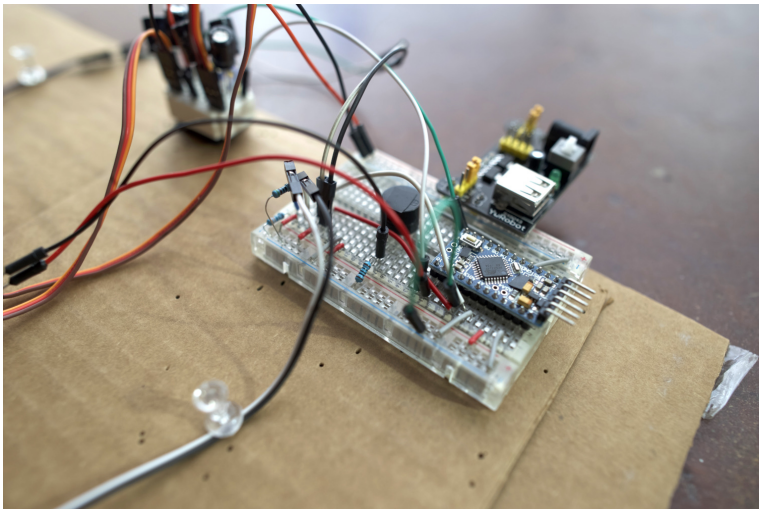

Figure 4: Detail of the breadboard. A piezo buzzer beeps whenever an episode ends. Voltage dividers for the photocells are visible. The board hanging off the side provides voltage regulation. At the top left, two capacitor banks smooth the high draw that occurs when the servos start.

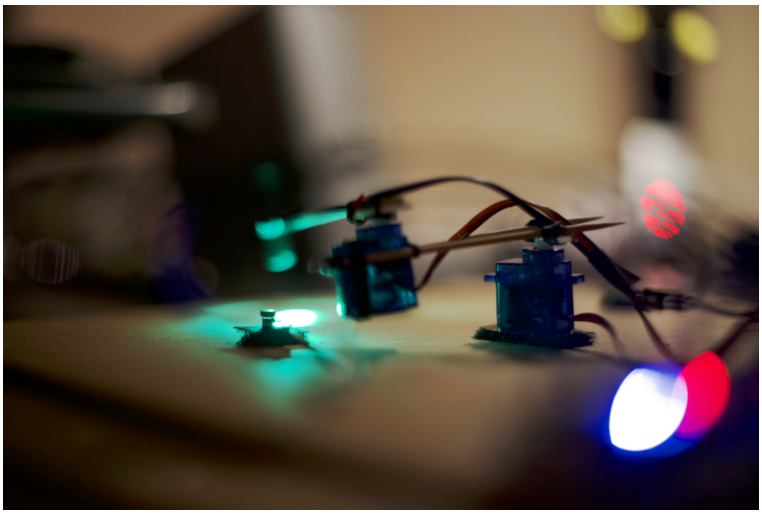

Figure 5: The agent illuminates the photocell.

7 Appendix B: Bill of Materials

Component	Quantity	Unit Price ($)	Note
Arduino Pro Mini, 5v	1	3.00	
Breadboard	1	4.00	
SG90 micro-servo	2	3.00	Any servo with sufficient range of motion.
Dowel rods	3	2.00	
1500uF capacitor	2	1.00	
5v 2.5A power supply	1	8.00	If variable supply unavailable.
330Ω resistor	4	0.01	
LED	1	0.10	The brighter the better.
Photocell	3	0.10	
Rubber bands	8	0.10	
Assorted jumpers		3.00	
Adhesives, project surface		4.00	