ENCODING ROBOTIC SENSOR STATES FOR Q-LEARNING USING THE SELF-ORGANIZING MAP

Gabriel J. Ferrer
Department of Mathematics and Computer Science
Hendrix College
Conway, AR 72032
(501) 450-3879
ferrer@hendrix.edu

ABSTRACT

The self-organizing map (SOM) [1] reduces a large input space to a fixed-size output space. It is trained by means of an unsupervised learning algorithm. One application of the SOM is to transform a set of robot sensor readings into a state space suitable as the input for the reinforcement learning algorithm Q-learning [5].

We have implemented two different formulations of this concept [2][3] using the Lego Mindstorms NXT robot [8], a robot commonly used in undergraduate computer science courses. We compared the performance of our implementations against a traditional Q-learning implementation. We found that the number and type of sensors encoded by the SOM has a significant impact on the quality of the behavior the robot learns.

INTRODUCTION

The Q-learning algorithm [5] is a very popular algorithm for reinforcement learning. It is straightforward to implement and tends to learn effective behaviors. Mahadevan and Connell [6] pioneered the use of Q-learning for a mobile robot to learn behaviors in the physical world.

The main drawback of Q-learning from the point of view of robotic implementation is that it requires all possible combinations of sensor values to be reduced to a finite number of states. Furthermore, the rate at which the algorithm converges to a behavior depends directly on the number of states selected for encoding the sensor values. This becomes particularly troublesome when high-resolution sensors such as sonars are employed.

In order to address this, Touzet [3] and later Smith [2] combined the Q-learning algorithm with Kohonen's Self-Organizing Map (SOM) [1]. The SOM is a type of neural network that learns how to classify its inputs into a fixed number of output nodes. By feeding sensor input into a SOM, the strongest SOM output can be used to define the current state for the Q-learning algorithm.

Touzet [3] implemented and tested his formulation using the Khepera robot [7]. The Khepera is a cylindrical robot of radius 6 cm. It has eight infrared proximity sensors that report distances ranging from 2 to 5 cm. The robots from Smith's paper [2] exist entirely in simulation.

The Lego Mindstorms NXT robot [8] has proven to be popular among undergraduate computer science educators. However, its sensor inputs and performance profile differ significantly from the robot platforms used in the previously mentioned papers. Consequently, we decided to implement both approaches using the Lego

Mindstorms NXT to assess their utility with this popular platform. We have found that we can reproduce their successful results as long as the state of the motors is not part of the state encoding given to the SOM.

The first two sections describe the Q-learning algorithm and the unsupervised learning algorithms used for training a self-organizing map. The third section describes our experimental setup. We then describe and analyze our results, followed by our conclusions.

Q-LEARNING

In reinforcement learning, the agent learns a behavior based on its expected reward. In the Q-learning implementation of reinforcement learning, expected rewards are stored in a two-dimensional array. One index of the array is the current state; the other index denotes an action to be performed in that state. The value of a particular state/action pair is the expected reward for performing the given action from the given state. The elements of this array are called "Q-values".

At each time step, the learning agent performs the following tasks:

1. Calculate the state index based on current sensor values.
2. Calculate the reward earned by the most recent action.
3. Update the Q-value for the previous state/action pair.
4. Select and perform a new action based on the Q-values.

The first two steps (calculating the state and reward) are implemented based on domain constraints. The Q-value is updated according to the following formula:

$$Q(s,a) = (1 - \alpha)\, Q(s,a) + \alpha\, (r + \gamma \max(Q(s',a)))$$

In this formula, the term α is called the "learning rate" and γ is called the "discount". The learning rate controls the speed with which the Q-values change; the discount represents a trade-off between current and future rewards. The state s' is the state that results from applying action a from state s, and r is the reward received after applying a in s.

Actions are selected based on the Q-values. From the current state, the action that is predicted to yield the largest reward (i.e. has the largest Q-value) is selected and performed. Formulated in this way, assuming that every state/action pair is visited an infinite number of times, Q-learning will converge to an optimal controller.

Since infinite state exploration is impossible in practice, action selection becomes a trade-off between *exploration* of the state/action reward space and *exploitation* of the predicted rewards stored in the Q-table. A common solution to this problem is to use an additional constant ε. At each time step, if a randomly generated value is greater than ε, the agent will use the best available action. If not, an action will be selected randomly.

SELF-ORGANIZING MAPS

The self-organizing map (SOM) [1] is a type of artificial neural network. Each output node is associated with a vector representing the "ideal input" for that node. The output of a SOM is determined competitively when an input vector is presented to it. Let the *distance* between an input vector and the ideal input for an output node be defined as

the square root of the sum-of-squared-differences between the two vectors. We then say that the activated output node is the output node whose ideal input has the smallest distance to the input vector.

The output nodes themselves are arranged in a Cartesian grid. On each iteration, the ideal input for the winning output node as well as some of its neighbors in the grid is modified according to the following formula for each affected output node *i* and ideal input element *j*:

$$weight_{ij} = weight_{ij} + (learningRate * (input_{ij} - weight_{ij}))$$

One approach for determining affected nodes is to set a radius parameter. All output nodes within the radius are updated according to this rule. Another common approach is to use a Gaussian neighborhood function $e^{-d2 / (2c2)}$, where *c* is the radius parameter and *d* is the Cartesian distance in the grid between the winning output node and the output node being modified. The result of the Gaussian is multiplied by the learning rate in the above computation for every output node in the SOM. This approach is used in Smith's implementation [2].

In Touzet's implementation [3], the winning output node uses a constant learning rate of 0.9, and each of its Manhattan neighbors in the Cartesian output grid is updated with a learning rate of 0.4. No other output nodes are updated on a given iteration.

The SOM is biologically inspired by the cerebral cortex. The use of neighborhoods enables the SOM to imitate the cortical phenomenon of regions that specialize in recognizing certain sensory data.

EXPERIMENTAL SETUP

We implemented the combination of Q-learning with a SOM in the Java language using the LeJOS implementation version 0.85 [9] for the Lego Mindstorms NXT robot. In our implementations, the winning SOM output node is the state index for the Q-table. Each experiment ran for 240 seconds, which in all cases represented 800 iterations of the Q-learning algorithm. All experiments had 36 state indices for the Q-table and three actions: Both motors forward, left motor backward with right motor stopped, and right motor backward with left motor stopped. Our robot was configured with a single forward-facing ultrasonic sensor and two forward-facing bump sensors, one on the front left, the other on the front right. The building instructions can be found in the author's robotics textbook [10].

We used two different Q-learning implementations without a SOM as controls. In the first implementation (designated **Qa**), the state was derived from the combination of each bumper state, the current selected action (out of three), and one of three discretized sonar states. The first discretized sonar state represented distances from 0-19 cm; the second state was 20-39 cm; and the third state was 40 cm or more.

In the second implementation (designated **Qb**), we did not represent the current action at all; instead, we discretized nine sonar states. The first state represented 0-11 cm; the second state 12-23 cm; and so forth up to the eighth state at 84-95 cm. The ninth state was 96 cm or more.

In all cases, the reward was calculated as follows. If either bump sensor is pressed, the reward is 0.0. If neither bump sensor is pressed, if both motors are going

forward, the base reward is 1.0; otherwise, it is 0.5. If the sonar distance is greater than 20 cm, the base reward is used; otherwise, it is scaled down depending on how close the sonar value is to zero cm. This reward function is designed to teach the robot to drive forward while avoiding obstacles in its path.

We used four different Q-learning implementations with a SOM. We followed each of Smith [2] (designated **QSOM**) and Touzet [3] (designated **QT**), and for each of these implementations, we implemented both with (**a**) and without (**b**) encoding the motor speeds as part of the state. In all four implementations, both the sonar and each bump sensor were part of the input to the SOM. Consequently, **QSOMa** and **QTa** had length-5 input vectors, while **QSOMb** and **QTb** had length-3 input vectors for each SOM. As with our control implementations, the Q-learning tables each had 36 input states; hence, each SOM had 36 output nodes.

When creating the input vectors from the sensor values, we normalized their ranges as follows. Let the scale factor be 100 / (maxSensorValue – minSensorValue). We multiply the scale factor by the sensor reading to produce a scaled input to the SOM. As the sonar range is 0 to 255, while the motor speeds range from -900 to 900, the intention is to normalize the inputs so that the raw sensor range does not distort their impact on the SOM.

For all experiments, the discount (γ) term for updating the Q-table values was 0.5. For **Qa**, **Qb**, **QSOMa**, and **QSOMb**, the learning rate is calculated using the formula $1/(1 + (t/100))$, where t is the current iteration of the algorithm. The neighborhood radius is the learning rate multiplied by 3. For **QTa** and **QTb**, the learning rate is fixed at 0.9. In both cases, the learning rate for Q-learning and the SOM is identical.

The exploration/exploitation trade-off is handled by setting an epsilon value of the learning rate divided by four. If a randomly generated value between 0 and 1 is below epsilon, the next action is selected randomly, with a distribution proportionate to the expected rewards. Otherwise, the action with the highest expected reward is selected.

At startup, the ideal input vectors for each SOM are all set to zero, and all of the Q-values are initialized to 0.5.

RESULTS

We ran at least three experiments for each of the six implementations. We ran seven experiments for **Qa** and four experiments for **QSOMa**. We employ the total reward earned over 800 iterations as our primary quantitative performance metric. Our results are in the table below.

	Qa	**Qb**	**QSOMa**	**QSOMb**	**QTa**	**QTb**
Mean	607.97	578.91	468.86	534.49	456.19	545.61
Std. Dev.	81.92	76.95	39.39	160.41	85.07	57.98
Median	608.75	667.5	485.11	587.64	442.62	560.77
Minimum	506.47	528.67	410.2	354.25	378.72	481.55
Maximum	723	540.55	495	661.59	547.22	594.5
Mean/Iter.	0.76	0.72	0.59	0.67	0.57	0.68

StdDv/Iter.	0.1	0.1	0.05	0.2	0.11	0.07

In terms of the quantitative data, the SOM-based solutions perform somewhat worse than the traditional Q-learning implementations. The creation of the "b" implementations was inspired by the particularly poor performance of **QSOMa** and **QTa** relative to **Qa**. When we examined the ideal inputs of a SOM created by a run of **QSOMa**, we observed that the encoded motor speeds ranged from 2% to 50%, while the encoded sonar value was stuck between 90% and 94%. In other words, the SOM learned a state encoding for the sonar that paid no heed whatsoever to any significant variance in sonar input. In the case of **QTa**, a much larger range of sonar values (10% to 100%) is present in the ideal inputs; nevertheless, performance was about equally bad.

We realized that for this task it was not necessary to encode the motor speeds as part of the state. Since the expected reward is conditioned both on the current state and the selected action, it seemed that including the action in the state was redundant. While this change made no significant difference to the performance of our control implementation, it did have a very positive effect on the performance of our SOM-based implementations. In fact, two of the three experiments with **QSOMb** produced obstacle-avoiding behaviors that were arguably superior to those of our control implementations.

For these two experiments, an inspection of the ideal inputs for the SOM is helpful in understanding their behavior. In both cases, the ideal inputs assumed that the bumpers were not pressed at all; for those state elements, there was no variation. However, the sonar value varied from approximately 40% to 95% in both cases. When the sonar readings were at the low end of this range, it was clear from observing the robot's behavior that the high-reward action was to turn. Otherwise, it drove forward. Once the learning algorithm had stabilized, these experiments produced extremely reliable obstacle-avoiding robots that never hit anything.

For **QTb**, successful runs learned to rely more on the bump sensors once the learning algorithm converged. The rewards tended to remain high because in spite of relying upon the bump sensors, the robot still spent very little time within 20 cm of an obstacle. An inspection of the ideal inputs for the SOM for **QTb** provides a clear rationale for this behavior. There were several output nodes whose ideal inputs included high values for the bump sensors, and it was for those states that Q-learning predicted a high reward for a turn. We believe that the different patterns of SOM learning between the **QT** and **QSOM** runs is explained by the high, non-decreasing learning rate employed by **QT**.

For **Qa** and **Qb**, reliance on the sonar vs. the bump sensors varied significantly between runs. The higher-scoring runs tended to rely more on the sonar; the lower-scoring runs relied more on the bump sensors.

In all cases, poorly-performing controllers exhibited one of two behaviors: They either learned to avoid obstacles by perpetually turning, or they alternated between each of the three actions in turn. In both cases, the robots avoided hitting things largely by going nowhere. Since the reward for non-forward driving when not near an obstacle is 0.5, it is to this reward value per iteration that the poor performers tended to converge. So after a fashion, even the unsuccessful runs learned to avoid obstacles; they failed to learn to actually go anywhere while doing so, however.

CONCLUSION

We have described an empirical comparison between traditional Q-learning and two different formulations of Q-learning with the Self-Organizing Map. Our implementation employs the Lego Mindstorms NXT robot, a common platform used in undergraduate education. We have shown that the successful results from the research literature for combining these approaches can be replicated using the NXT platform as long as the motor speeds are not included as part of the input vector for the SOM.

Our results leave open a number of questions. It remains puzzling as to why the inclusion of the motor speeds in the input vector pollutes the SOM so badly in the case of **QSOMa**, as well as why **QTa** performs so badly in spite of avoiding this problem. This also raises the question of the scalability of the approach with increasing numbers of sensor inputs. We plan to experiment with multiple sonar inputs to investigate the possibility of scaling in further depth.

Another potential direction for future work would be to replicate Touzet's attempt to improve robot learning by disregarding reinforcement learning entirely [4]. In this new approach, instead of a reward function, a goal state is specified, and the robot uses the ideal inputs of the SOM to guide its actions. A second SOM encodes a relationship between actions and transitions. In light of our problems with motor speed encoding, it will be an interesting challenge to try to replicate this result on the NXT as well.

We will mention in closing that we have used QSOM as an assignment in an upper-level undergraduate Artificial Intelligence course. Our students (who, in previous assignments, had already implemented both the SOM and Q-learning separately from each other) were very successful in implementing this combination for several tasks, including obstacle avoidance, pursuit, and light-finding.

REFERENCES

[1] Kohonen, T. *Self-Organizing Maps*, 3rd Edition. Springer, 2001.

[2] A.J. Smith. "Applications of the Self-Organizing Map to Reinforcement Learning". *Neural Networks* 15:1107-1124, 2002.

[3] C. Touzet. "Neural Reinforcement Learning for Behavior Synthesis". *Robotics and Autonomous Systems* 22(3-4):251-281, December 1997.

[4] C. Touzet. "Modeling and Simulation of Elementary Robot Behaviors using Associative Memories". *International Journal of Advanced Robotic Systems* 3(2):165-170, June 2006.

[5] C.J. Watkins and P. Dayan. "Q-Learning". *Machine Learning* 8(3-4):279-292, 1992.

[6] S. Mahadevan and J. Connell. "Automatic Programming of Behaviour-Based Robots Using Reinforcement Learning". In *Proceedings of the National Conference on Artificial Intelligence*, 768-773, 1991.

[7] F. Mondada, E. Franzi & P. Ienne, "Mobile Robot Miniaturisation: A Tool for Investigation in Control Algorithms," *Third International Symposium on Experimental Robotics*, Kyoto, Japan, October 1993.

[8] Lego Education. "Lego Mindstorms Education Base Set". http://www.legoeducation.us/store/detail.aspx?ID=1263&bhcp=1, retrieved December 11, 2009.

[9] LeJOS, Java for Lego Mindstorms. http://lejos.sourceforge.net/, retrieved December 11, 2009.

[10] Ferrer, G. *Introduction to Robotics using Lego Mindstorms NXT.* Lulu.com, 2009.