Implementing an Adaptive Robot with Multiple Competing Objectives in a Service Industry Environment

Fletcher Lu[1], Lorena Harper[2]
[1]University of Ontario Institute of Technology,
[2]University of Maryland Eastern Shore
fletcher.lu@uoit.ca, laharper@umes.edu

Abstract

This research paper presents the implementation of an adaptive learning algorithm from artificial intelligence known as Reinforcement Learning in a robot that must deal with more than one reward where the rewards may come into conflict with each other. A case that practically illustrates this problem is in a service industry environment where a robot is implemented to pick up objects for cleaning and/or attending to clients. An example of conflicting rewards for this application would be achieving a high reward for quickly picking up objects which typically conflicts with minimizing any damage inflicted on the object during the picking up process.

The innovative component of our research is that we will use multiple competing rewards for some states in contrast to the single reward per state method traditionally used in Reinforcement Learning. We compare these two approaches through the implementation of a Lego Mindstorm robot that has been programmed with both learning methods. The objective of our robot is to pick up objects quickly without damaging the object. We illustrate the conditions under which it is advantageous to use a single state for competing rewards over a multi-state approach through practical comparisons on efficiency for our Lego robot.

The objective of this research is to broaden the adaptability of learning robots. The impact on service industries, such as hotel and restaurant service, of this research would be to increase the acceptability of adaptable robots into fields of manual labor that have traditionally been limited due to the inflexibility of robots in dealing with dynamic situations.

Introduction

Although robots have been used to perform tasks since the beginning of the industrial age, it is only in the past 20 years, since the advent of artificial intelligence research, that has allowed for the implementation of adaptive robots that can handle tasks that are not completely repetitive [1]. A learning robot fundamentally differs from a robot that performs a repetitive task in that the learning robot can modify its behavior with sensory feedback. One of the most popular learning methods for implementing such robots is the

Reinforcement Learning system [1]. This approach has been used to implement robots that can act as tour guides [2], robotic nurses [3], and automobile drivers [4]. The learning component of these robots makes them more flexible so that they can modify their behavior.

Reinforcement Learning operates by modeling the environment as a network of states, S, where actions, A, can be taken to move from one state to another. Rewards, R, are associated with each state and the general goal is to maximize the long-term rewards one may obtain as one navigates through the environment. In order to achieve this goal one needs to develop a function known as a 'policy' (represented by the symbol π) which maps states to action: $\pi(S) \rightarrow A$. The main objective is to find what is known as an optimal policy, π^* that produces the best long-term rewards navigating through the environment using function π^* [5].

The Reinforcement Learning approach to modeling the world and developing an optimal policy works very well for a broad range of applications. However it is generally limited in the sense that it assumes a single reward for any given state [6] [7]. There are a variety of situations where this is not necessarily the case. For instance, in a service industry robot, we may wish to implement a robot that can pick up objects for cleaning purposes, such as cleaning up hotel rooms. One natural single state is to have successfully grabbed an object for pick up. For an efficient cleaning robot to achieve this state of successfully grabbing an object for pick up, two problems however must be overcome:

1. Enough pressure must be applied to the object with the robot's grabbing apparatus so that the object does not slip through.
2. Excessive amounts of pressure must not be applied or the robot would damage the object.

These two challenges can be viewed as competing rewards to achieve the same state. This state is the grabbing of the object, which is essentially applying just the right pressure to pick up the object without damaging it. The competing computed rewards would be, the more pressure applied, the greater the chance of successfully picking up the object, with an opposing reward of the less pressure applied, the greater the chance of successfully not damaging the object for pick up.

An alternative approach, which is often used, is to split the two competing rewards into separate states, one state for each reward. However this is not a natural solution for many real situations such as the cleaning robot problem. For the cleaning robot problem, for instance, the state of the amount of pressure applied to pick up an object is more naturally a single state.

The primary goal of this research is to discern the advantages of combining competing rewards in a single state versus separating the rewards into separate states. We also will determine the conditions under which it is more efficient to use separate states for each competing reward versus our combined reward/single state alternative.

The major benefit to adding in a multiple reward structure for each state is that we provide more options in implementing robots. This greater flexibility in implementation allows for greater adaptability in learning robots.

Background on Reinforcement Learning

A Reinforcement Learning system models the environment as a network of states, where transitions from one state to the next follow a Markov reward process on a finite set of N states. Recall that we transition from one state to another by choosing an action that moves us into another state, given the current state we are in and the action chosen. This produces a set of transition probabilities. We choose actions based on a policy function π, which maps states to actions. The goal is to find an optimal policy that produces the best long-term rewards [1]. To do so, we start by arbitrarily choosing a policy and evaluating it for long-term rewards. This evaluation is possible due to the Markov process assumption [8].

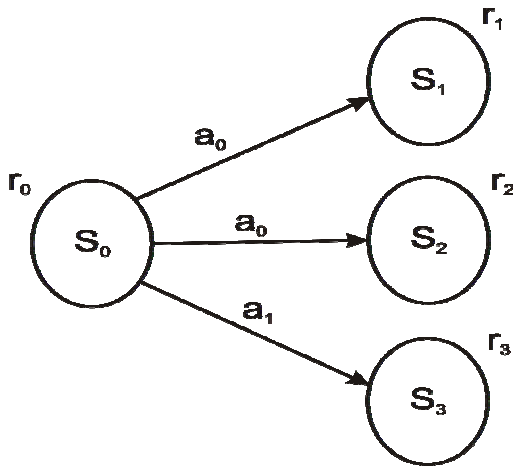

Figure 1: Program Flow Chart

A Markov process implies that in a given state, n, one transitions to a next state, m, by the conditional probability model $P(S_{i+1}=m|S_i=n)$, where we assume the transition probabilities do not change over process time i (stationarity assumption) [9]. Such a transition model can be represented by an N x N matrix P, where P(n,m) denotes $P(S_{i+1}=m|S_i=n)$ for all process times i. The reward R_i observed at time i is independent of all other rewards and states given the state S_i visited at time i. We also assume the reward model is stationary and therefore let **r**(n) denote $E[R_i|S_i=n]$ and σ(n) denote $var(R_i|S_i=n)$ for all process times i. Thus, **r** and σ represent the vectors (of size N x 1) of expected rewards and reward variances respectively over the different states n=1,...,N.

The value function **v**(n) is defined to be the expected sum of rewards obtained by starting in a start state S_0=n. That is, **v** is a vector given by:

$$v = r + \gamma \Pr + \gamma^2 \Pr + \ldots \quad (1)$$

where $0<\gamma<1$ is a discounting term used on the rewards.

We can solve this equation by substituting the infinite sequence with vector **v** to produce the equation:

$$v = r + \gamma P v . \quad (2)$$

Therefore, if P and **r** are known then **v** can be calculated explicitly by solving the matrix equation:

$$(I - \gamma P)v = r . \quad (3)$$

We compute an estimate of the transition probabilities to produce our matrix P during our learning phase of the learning algorithm. With our estimate of P and the rewards we collect during the learning phase to produce the vector r, we can create an estimate of the long-term rewards through the value function v [5]. All these values are dependent on the fixed policy function π. Once we have evaluated this policy, we can then iteratively improve it by modifying the policy for each state to choose a better action giving improved long-term rewards. It has been proven that one can non-trivially improve a policy at every iteration of this process until an optimal policy has been reached [11].

A variety of approaches may be implemented to choose actions for states during the learning phase of the Reinforcement Learning algorithm. For our implementation we will use a random uniform probability method to choose among all possible actions.

When implementing a real-world robot using reinforcement learning, most approaches use the single reward per state method [2][12]. However, in a real-world environment, such restrictions are very limiting when attempting to simulate a more dynamic robot that can adapt to gradiated conditions such as different degrees of pressure and temperature. The situation is especially challenging when attempting to deal with ultimately developing humanoid type robots such as those considered by Peters et al. [13].

Later robotic implementations dealing with more complex environments have created more complex functional type rewards without explicitly considering benefits and drawbacks of actually splitting the rewards into multiple states. Zhumatiy et. al. [14], for instance created a robot with a functional reward dependent on both an obstacle and target value. Our contribution is thus to consider such benefits and drawbacks to single function rewards versus splitting the reward into multiple states.

Lego Robots

For our robotic implementation, we used the Lego Mindstorm NXT kit to build our robot. The NXT kit includes several essential components in order for it to be useful as a learning robot. The most important components are the sensors that allow for feedback to our robot to sense the state of its environment. Without such sensors, it would be impossible to learn whether actions taken by the robot would be yielding positive or negative results. There are three sensors that were used in our robot implementation:

1. a light sensor that could send out a laser beam of light and measure the amount of reflected light when the beam hits an object,

2. an ultrasonic sensor that is used to measure distance by an approach similar to sonar detection, whereby distance is measured by measuring the time needed for an ultrasonic wave to be reflected back to the sensor, and
3. a touch sensor, which can measure the degree to which the sensor button is pressed.

In addition to these sensors, the robot includes servo motors, which are essential in our robotic implementation, so that we can apply continuous pressure through the motors on an object without destroying the motors themselves.

In addition to the hardware of the NXT kit that made the robot learning possible, is the software component. The standard Graphical User Interface system for programming NXT robots is very limited. However, Lego allows for others to develop programming compilers for its system by making its byte code available for translation. In our case, we used the NXC programming language, which stands for Not eXactly C [15]. This language allows for the creation of arrays, which are essential for the matrix vector computations in the Reinforcement Learning algorithm. In addition, it includes random number generators and probability computations needed for our uniform random choices during our learning phase as well as the computations of the probability matrix P. Also the language allows for concurrent programming mechanisms such as semaphores and mutexes to avoid multiple simultaneous access to controlled systems such as motors.

Our Picking-Up Robot

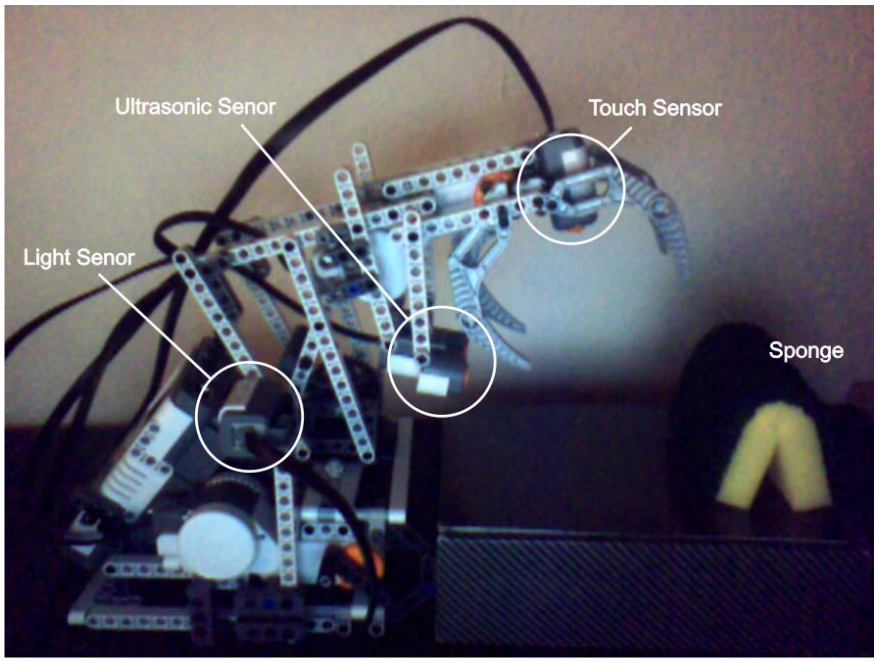

Figure 2: Picking-Up Robot

For our robot implementation, we wished to create a robot that could pick up objects. The variables included:

1. the amount of pressure the robot could apply to the object in order to pick it up and

2. the speed at which it would try to pick up the object.

The robot needed to be able to sense if the object was being damaged during its picking up process, as well as whether it had dropped the object because the pressure it was applying was insufficient to hold the object.

Figure 2 shows our completed robot. The arm of the robot has both the touch sensor and ultrasonic sensor mounted on the arm. The claws are used to grab the object. In order to avoid destroying objects during our tests, we used a sponge to simulate a delicate object. The touch sensor mounted above the claw would act as a sensor to decide when the object had been grabbed as well as if it too much pressure was being applied to the object or if the object was dropped. The idea is that the touch sensor needs to be somewhat pressed in order for the robot to be aware that it has successfully gotten a hold of the object. However, if too much pressure is applied then the sponge would be squeezed excessively and the touch sensor would be pressed beyond a certain threshold indicating the object had been damaged.

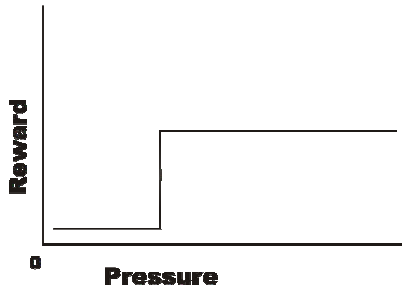

Figure 3: Pick-up Reward

Recall from our introduction that we have two competing rewards for a single state. This state represents the amount of pressure applied to an object to pick it up. One of the rewards would be a zero-one function, where if enough pressure is applied to the object, then it would be enough to pick the object up producing a reward of one. Before this threshold pressure is reached, this reward returns zero. Figure 3 illustrates this reward, which we will call the Pick-up Reward.

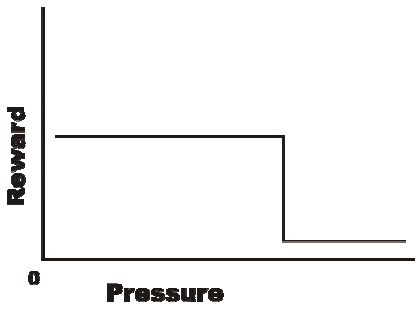

Figure 4: Not Crushed Reward

The competing reward also is dependent on the pressure state. It returns a value of one as long as pressure is below a certain breaking threshold. Once the pressure applied to the object exceeds that breaking threshold, the object is considered to have been damaged and a value of zero is returned. Figure 4 illustrates this reward, which we will call the Not Crushed Reward.

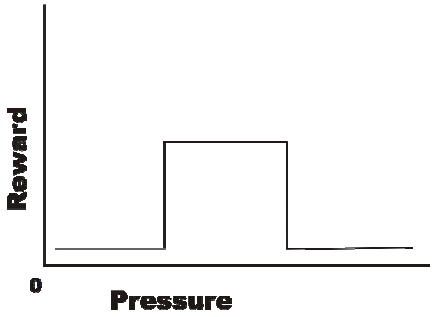

Figure 5: Combined Reward

The two competing rewards combined can be viewed as the function illustrated in figure 5. As pressure increases, zero reward is obtained until enough pressure is applied to pick up the object, then a reward of one is obtained. This reward of one is only obtained if concurrently not too much pressure is applied or the breaking threshold is reached and the reward returned is zero.

The actual sponge object we used can be dropped as well. So if an insufficient amount of pressure is applied, then the sponge can be dropped, which we would consider a failed attempt.

Our objective during our experiments is to pick up the object as quickly as possible and return the arm to an initial start position. The goal of picking up an object as quickly as possible is directly affected by how careful the robotic arm must be at picking up this delicate object. The faster the object is picked up, the more likely it is damaged because greater pressure is generally needed to avoid the object from being dropped when it is being moved at faster speeds.

Theoretical Analysis

The key issue we are addressing is whether there is an advantage to using a combined reward structure in a single state (when it is natural to do so) or is it best to artificially separate the rewards into two separate states in order to fit the natural Reinforcement Learning model. In our service industry application, the single state of the pressure applied to the object we are attempting to pick up results in two rewards that somewhat oppose each other. Each reward is a natural direct result of the pressure state and thus should both be assigned during that state.

A combined reward structure for a single state intuitively requires a greater degree of computation and depending on the complexity of how the rewards should interact, this could slow down computation time significantly. However discretizing the rewards into separate states could also slow computation time. Recall from equation 3 of the background section on Reinforcement Learning that we need to solve an N x N matrix equation in order to compute value estimates during a learning phase. By splitting a single state into two states we actually increase the dimension size of our matrix to (N+1) x (N+1), also slowing computation time. The theoretical time to solve equation 3 is in the worst case $O(N^3)$. Thus the theoretical increase in time complexity with the added state would be:

$$(N+1)^3 - N^3 = 3N^2 + 3N + 1.$$

Therefore, theoretically, if the state is only visited once per sampling estimate, then as long as the runtime for computing the more complex single state reward, R, is:

$$R(combined) < 3N^2 + 3N + 1$$

or $O(N^2)$, (i.e. the reward must run in quadratic time to the number of states or less) then the single combined state should be faster. However, if the state is visited N times per sampling estimate, the bound decreases to:

$$R(combined) < 3N + 3 + 1/N$$

or O(N), (i.e. the reward must run in linear time to the number of states or less). Note that these runtimes are based on a worst-case scenario of a fully connected network of states. The more sparsely connected the network of states, the lower this threshold would be.

In addition to the running time of the reward computation is the added possible advantage of using a function to compute our combined reward. Consider that the combined reward that takes into account multiple factors to produce its reward for its state could be the result of a complex interaction between the two or more rewards. If one were to discretize the rewards into separate states, the only way the rewards interact directly is through a discounting summation of equation 1. This means that they, at best, proportionately weight the two rewards and add them together as the most complex interaction when split into separate states. In contrast, a single state reward function could create a much more complex combined reward.

Experiments

In our experiments, we compared our combined reward in a single state to two other approaches. One alternative approach was to choose the state that assigned the Pick-up Reward first and then immediately follow that with a deterministic transition to the Not-Crushed reward state. However, if the Pick-up reward failed, then the robot immediately reset to try again during learning trials. The second alternative approach was to choose the Not-Crushed reward state first then immediately followed with the Pick-up Reward state but only if the Not-Crushed reward had not failed. We followed this approach since it seems intuitively obvious that if one fails to pick up the object, there is no point in testing if the object is crushed. Similarly, if the object was crushed, there is no need to check if the object was picked up.

We ran 12 trials where each trial allowed for 3 minutes of learning to find an optimal choice of speed and pressure to pick up our sponge object without crushing it. We recorded how many practice attempts were tried during each three minute learning phase. We also recorded the times each practice attempt took and whether they failed to pick up the sponge or crushed it. Finally for each trial we recorded an exploitation phase where the robot demonstrates the best learned attempt to pick up the sponge.

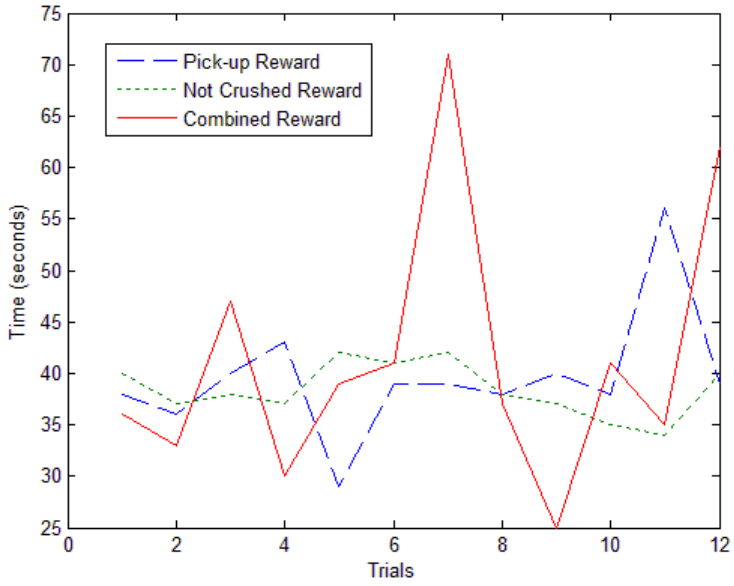

Figure 6: Time of Learned Pick-up

Figure 6 illustrates graphically the time that it took the robot to pick up the object using the best learned approach in each trial. As can be seen graphically, the Combined Reward in a single state for our pressure value, has the greatest variability in terms of learned result. It not only produced the best learned result (shortest time to successfully pick up the sponge without crushing it), but also the worst learned result (longest successful time) over our 12 trials. We tested a null hypothesis that the means of the learned trials for each approach were different from each other ($\mu_i \neq \mu_j$, where i and j represent our three different reward approaches, $i \neq j$). With 99% confidence interval, none of the means could be accepted as different. Thus we would accept that all three approaches on average produce comparable learned best pick-ups. But the variance of the Combined Reward method did significantly differ from the other approaches. Using an F distribution test on the variances, the Combined Reward approach has a 98% confidence that the variance differs from the variance of the other two approaches ($f_{ij} = 4.6$, $f_{ik} = 25.3$, where i is the Combine Reward approach, j is the Not Crushed Reward approach, and k is the Pick-up Reward approach).

Looking at figure 6, the difference in variance implies that the Combined Reward approach tends to have the most variability in how well it learns. The reason for this wider variability

is due to two factors at work during the Combined Reward approach. The first factor is that the Combined reward requires a two threshold test, which takes slightly longer to compute, which can result in fewer practice trials in our 3 minute learning time. With fewer practice trials to learn from, this could lead to worse learned results. The second factor is that the more nuanced reward can give a better measure of value function, leading to better learned results.

What can be drawn from both our experimental results and our theoretical analysis, is that using a function that combines multiple factors to produce a combined reward can yield better learned results because the reward can take into account more complex relationships between the rewards than the simple weighting that would result in splitting rewards into separate states. But this benefit can be balanced out by the added computation time needed to produce this more complex reward. Thus, sufficient learning time is needed for the benefits of the more nuanced combined reward in a single state to be of benefit. The amount of actual time to compute the combined reward should not exceed $O(N^2)$, assuming a constant number of visits to the state per learning trial.

Conclusion

In this paper the researchers compared the traditional single state per reward approach with using a combined reward that incorporates at least two or more reward factors into a state. This combined reward allows for more subtle computations especially for reward factors that may oppose each other. By allowing for more complex computed reward structures, we may produce better learned results. We experimented on a practical implementation of this problem with a Lego robot that was charged with the task of learning to pick up a delicate object. The robot needed to apply sufficient pressure to pick up the object but not too much pressure or the object would be crushed.

In terms of theoretical results, we demonstrated that in the worst case, the combined reward should not exceed $O(N^2)$ computation time or the benefit in terms of efficiency would be outweighed by cost to compute the reward. However, this is a worst case scenario, and it is quite possible that the combined reward function must be significantly faster than $O(N^2)$ depending on the graph connectivity of the Reinforcement Learning model and the number of visits to the combined state.

From our robot experiments we demonstrated that the benefits of the more subtle combined reward can be outweighed by the extra time taken to compute the reward. In our experiments, it resulted in greater variability in terms of learned ability. Because we fixed the amount of learning time for our robot, the extra time needed to learn resulted in fewer practices to learn the task. These fewer practices competed with the benefit of the more subtle reward to cause greater variability in terms of performance for our learned task. Thus, for those considering using a complex function to compute rewards, they must take into account the amount of learning time allotted. If a sufficient amount of learning time is allowed, then the more subtle combined reward structure can produce better learned results. How much time should be allotted most likely is application dependent.

In terms of future work, we plan to expand on this investigation to explore a variety of possible reward functions with the goal of implementing them in a practical service robot.

References

[1] Sutton, R., & Barto, A. G., "Reinforcement Learning: An Introduction", MIT Press, Cambridge, Massachusetts, 1998.

[2] Burgard, W., Tranhanias, P., Hahnel, D., Moors, M., Schulz, D., Baltzakis, H., Argyros, A. "Tourbot and WebFair: Web-Operated Mobile Robots for TelePresence in Populated Exhibitions", IEEE/RSJ Conf. IROS, 2002, pp. 77-89.

[3] Pollack, M. E., Engberg, S., Matthews, J.T., Thrun, S., Brown, L., Colbry, D., Orosz, C., Peintner, B., Ramakrishnan, S., Dunbar-Jacob, J., McCarthy, C., Montemerlo, M., Pineau, J., & Roy, N., "Pearl: A Mobile Robotic, Assistant for the Elderly", AAAI Workshop on Automation as Eldercare, 2002.

[4] Krodel, M. & Kuhnert, K., "Reinforcement Learning to Drive a Car by Pattern Matching", Pattern Recognition: 24th DAGM Symposium. Springer Berlin/Heidelberg, 2002, pp 322-329.

[5] Lu, F, Patrascu, R., Schuurmans, D., "Investigating the Maximum Likelihood Alternative to TD(lambda)", 19th International Conference on Machine Learning, 2002, pp 403-410.

[6] Michie, D., and Chambers, R. A., "BOXES: An experiment in adaptive control." Machine Intelligence, Vol 2, 1968, pp 137-152.

[7] Barto, A. G., Sutton, R. S., and Anderson, C. W., "Neuronlike elements that can solve difficult learning control problems." IEEE Transaction on Systems, Man, and Cybernetics. Vol 13, 1983, pp 535-549.

[8] Witten, I. H., "An adaptive optimal controller for discrete-time Markov environments", Information and Control, 1977, Vol 34, pp 286-295.

[9] Watkins, C. J. C. H., "Learning from Delayed Rewards." PhD. Thesis, Cambridge University, 1989.

[10] Lu, F., "Exploring Model-Based Methods for Reinforcement Learning", PhD. Thesis, University of Waterloo, 2003.

[11] Sutton, R., "Learning to predict by the method of temporal differencing", Machine Learning, Vol. 3, 1988, pp 9-44.

[12] Montemerlo, M., Pineau, J., Roy, N., Thrun, S., Verma, V., "Experiences with a Mobile Robotic Guide for the Elderly", Proceedings of the AAAI National Conference on A.I., 2002, pp 587-592.

[13] Peters, J., Vijayakumar, S. and Schaal, S., "Reinforcement Learning for Humanoid Robotics." Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots, 2003, pp 1-20.

[14] Zhumatiy, V., Gomez, F., Hutter, M., Schmidhuber, J., "Metric State Space Reinforcement Learning for a Vision-Capable Mobile Robot." Intelligent Autonomous Systems, 2006, pp 272 – 281.
[15] Hansen, J., "Not eXactly C (NXC): Programmer's Guide", Version 1.0.1. b33, bricxcc.sourceforge.net, 2007.

Biography

FLETCHER LU is an Assistant Professor in the Faculty of Health Sciences at the University of Ontario Institute of Technology. He is peer-review published in areas of machine learning and artificial intelligence. He received a Bachelors of Mathematics with distinction as well as a PhD. in Computer Science from the University of Waterloo.

LORENA HARPER is a senior in the Department of Math and Computer Science at the University of Maryland Eastern Shore pursuing a Bachelors degree in Computer Science. She is a member of the Upsilon Pi Epsilon Honor Society and Vice President of the university's Math and Computer Science student club.