Reinforcement Learning in a Noisy Environment: Light-seeking Robot

KARY FRÄMLING
Laboratory of Information Processing Science
Helsinki University of Technology
PL 5400, 02015 TKK
FINLAND

Abstract: Despite many promising results from the use of reinforcement learning in simulated robot worlds, its use in real robot worlds is relatively rare. This paper addresses challenges related to real robot worlds and shows how reinforcement learning combined with linear function approximation can solve many of them. Experiments are performed using a light-seeking robot built with the Lego Mindstorms Robotics Invention System.

Key-Words: - reinforcement learning, exploration strategies, light-seeking robot, linear approximation, gradient descent

1 Introduction

Trial and error seems to be one of the main ways that animals learn to solve problems. The success or failure of a trial should help to modify behaviour in the "right" direction. The scientific research area called reinforcement learning (RL) is one of those studying such behaviour. RL methods have been successfully applied to many problems where the agent has to explore its environment and learn interactively. Depending on the current state of the environment and the agent, the agent has to take actions without a priori knowledge about how good or bad the action is, which may be known only much later when a goal is reached or when the task failed.

RL has been used in many robotic tasks, but most of them have been performed in simulated environments. Only few results have been reported on the use of RL on real robots. Real-world applications are challenging because they involve noise coming from sensors, non-deterministic actions and changes in the environment. Real-world experiments are also longer than simulated ones, so learning must be relatively rapid and possible to perform without causing damage to the robot.

Due to these requirements, most real-world experiments reported simplify the problem by pre-processing sensor values. Sensor values from sonar and light sensors are typically continuous, but they are often discretized even into binary values using value intervals. Such discretization requires a priori knowledge about the task to learn and leads to a loss of information.

In order to avoid discretization, we here use a linear function approximator together with gradient descent for making a robot learn how to go towards a light. The task is to learn the correct association between three sensor values, i.e. light readings in three directions, and five possible actions. Even though this task seems simple, sensor noise and other real-world considerations make it more complicated than it would be otherwise. The experimental results also show the influence of training parameters and the exploration policy used.

After this introduction, Section 2 explains the learning methods used in this paper. Section 3 presents an overview of previous work on the use of RL in robotics, followed by test results in Section 4 and conclusions.

2 Reinforcement learning principles

One of the main domains treated by RL is Markov Decision Processes (MDPs). A (finite) MDP is a tuple $M=(S,A,T,R)$, where: S is a finite set of states; $A = \{a_1, \ldots, a_k\}$ is a set of $k \geq 2$ actions; $T = [P_{sa}(\cdot) \mid s \in S, a \in A\}$ are the next-state transition probabilities, with $P_{sa}(s')$ giving the probability of transitioning to state s' upon taking action a in state s; and R specifies the reward values given in different states $s \in S$. RL methods are based on the notion of *value functions*. Value functions are either *state-values* (i.e. value of a state) or *action-values* (i.e. value of taking an action in a given state). The value of a state $s \in S$ can be defined formally as

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right\} \qquad (1)$$

where $V^{\pi}(s)$ is the state value that corresponds to the expected return when starting in s and following policy π thereafter. The factor r_{t+k+1} is the reward

obtained when arriving into states s_{t+1}, s_{t+2} etc. γ is a discounting factor that determines to what degree future rewards affect the value of state s. The discount factor also indicates the shortest path through the state space because the shorter the path, the less reward is discounted.

Action value functions are usually denoted $Q(s,a)$, where $a \in A$. In control applications, the goal of RL is to learn an action-value function that allows the agent to use a policy that is as close to optimal as possible. However, since the action-values are initially unknown, the agent first has to explore the environment in order to learn it.

2.1 Exploration/exploitation trade-off

Balancing the exploration/exploitation trade-off is one of the most difficult problems in RL for control [12]. The policy π used determines the balance between exploring the environment and exploiting already found, but possibly sub-optimal solutions. Random search achieves maximal exploration, while a greedy policy gives maximal exploitation by always taking the action that has the highest action value.

A commonly used method for balancing exploration and exploitation is to use *ε-greedy exploration* ([12] calls this *semi-uniform distributed exploration*), where the greedy action is selected with probability (1-ε) and an arbitrary action is selected with probability ε using a uniform probability distribution. This method is an *undirected exploration method* in the sense that it does not use any task-specific information.

Directed exploration methods use task-specific knowledge for guiding exploration. Many of these try to guide the exploration in such a way that the entire state space would be explored in order to learn the value function as well as possible. In real-world tasks this is often problematic because exhaustive exploration is impossible and dangerous. However, a technique called "optimism in the face of uncertainty" or "optimistic initial values" offers a possibility of encouraging exploration of previously un-encountered states mainly in the beginning of exploration. It can be implemented by using initial value function estimates that are bigger than the expected ones. This gives the effect that unused actions have bigger value estimates than used ones, so unused actions tend to be selected rather than already used actions. When all actions have been used a sufficient number of times, the true value function overrides the initial value function estimates.

2.2 Q-learning

When reward is not immediate for every state transition, rewards somehow need to be propagated "backwards" through the state history. Temporal Difference (TD) methods [11] do this by supposing that temporally consecutive states should have close value. The main advantage of TD methods over many other RL methods is that they update the value function on every state transition, not only after transitions that result in direct reward. TD methods are currently the most used RL methods.

Q-learning is a TD algorithm that uses state-action values rather than state values. In its simplest form, one-step Q-learning, action values are updated according to

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \quad (2)$$

where $Q(s_t, a_t)$ is the value of action a in state s at time t, α is a learning rate and r_{t+1} is the immediate reward. The max-operator signifies the greatest action value in state s_{t+1} and therefore represents the expected future reward when following a greedy policy. It is important to notice that (2) is only valid for discrete state spaces, usually handled by storing Q-values in a two-dimensional lookup table. The general case of continuous-valued state spaces is treated in the next section.

Q-learning is an *off-policy* method. This signifies that the value function converges no matter what policy is used, but under the condition that the learning rate is sufficiently small and that all state-action pairs continue to be updated [1].

2.3 Generalisation

Generalisation in RL is based on the idea that an action that is good in some state is probably also good in similar states. Various classification techniques have been used for identifying "similar" states. Some kind of artificial neural net (ANN) is typically used for the generalisation. ANNs can handle any state descriptions, not only discrete ones. Therefore they are well adapted for problems involving continuous-valued state variables.

The simplest ANN is the linear Adaline [13], where neurons calculate their output value as a weighted sum of their input values

$$o_j = \sum_{i=1}^{N} w_{i,j} s_i \quad (3)$$

where $w_{i,j}$ is the weight of neuron j associated with the neuron's input i, o_j is the output value of neuron j, s_i is the value of input i and N is the number of

inputs. They are trained using the Widrow-Hoff training rule [13]

$$\Delta w_{i,j} = \alpha(t_j - o_j)s_i \quad (4)$$

where t_j is the corresponding "correct" or "target" value. α is a learning rate parameter that determines the step size of weight modifications. The Widrow-Hoff update rule is a gradient descent method that minimizes the root mean square error (RMSE) between output and target values:

$$RMSE = \sqrt{\frac{1}{M}\sum_{k=1}^{M}\left(t_j^k - o_j^k\right)^2} \quad (5)$$

where M is the number of training examples. By inserting (3) into the RMSE expression and taking the partial derivative, it can be shown that it has only one optimal solution. Therefore gradient descent is guaranteed to converge if the learning rate is selected sufficiently small. The Widrow-Hoff learning rule is a special case of TD learning [1], which signifies that the Q-learning error term in (2)

$$r_{t+1} + \gamma \max_a Q(s_{t+1}, a) \quad (6)$$

can be used as the "target" value.

When the back propagation rule for gradient descent in multi-layer ANNs was developed [8], it became possible to learn non-linear function approximations and classifications. Unfortunately, learning non-linear functions by gradient descent tends to be slow and to converge to locally optimal solutions. This is particularly problematic in RL applications, where convergence of gradient descent cannot always be guaranteed even for Adalines [2].

3 Reinforcement learning in robotics

Due to the challenges related to real-world robotics applications, most experimental RL work has been done in completely simulated environments, e.g. [9] and [10], or in simulators written for simulating real robots, e.g. [3] and [7]. Simulators have also been used for initial learning before transferring the agent to the real robot. Real robots have been used in relatively few cases. One of the main reasons is the presence of *hidden state*, i.e. that it is not possible for the agent to completely determine its own state and/or that of the environment. Complete observability is one of the main assumptions of MDPs and hidden state is therefore a major challenge for RL methods.

Lin [4] used three different learning tasks, 1) wall following, 2) going through a door and 3) docking into a charger. Mainly sonar sensors were used; light sensor readings were only used in the docking task, where a lamp indicated the docking position. Most state variables were converted into binary values. There were 16 possible actions, consisting in turning and going forward in different directions. Relatively large non-linear ANNs (one per action) were used for learning the reward function. Actions were selected according to Boltzmann-distributed probabilities

$$\text{Prob}(a_j) = \exp(Q(s_i, a_i)/T) / \sum_k \exp(Q(s_i, a_k)/T) \quad (7)$$

where T (called *temperature*) adjusts the randomness of action selection.

Mahadevan [5] used a task that consisted in finding boxes and pushing them into corners. The task was decomposed into three sub-tasks (*behaviours*); 1) box finding, 2) box pushing, 3) unwedge from stalled states. The OBELIX robot used eight sonar sensors for detecting boxes, one infrared "bump" sensor for indicating contact with a box and a "stuck" indicator based on motor current. Sensor values were converted into 18 binary state values. Five actions were possible, one for going straight forward and four for turning right/left by 22 or 45 degrees. Actions were selected using ε-greedy exploration with $\varepsilon = 0.1$. State generalization was performed using a Hamming distance-based technique and a clustering technique.

Instead of learning the behaviours, Mataric [6] learned selecting the appropriate pre-programmed behaviour as a function of state. The task consisted in having multiple robots learn to "home" pucks, i.e. find pucks and take them back into a "home area". Four binary state variables were used, which makes the state space small enough to be treated by a lookup-table. Five actions (behaviours) were possible. Untested actions in a given state were used if possible; otherwise the greedy action was used. Plain Q-learning was not able to learn the task at all, so the reward function was modified to use *progress estimators* to produce more immediate reward and guide the learning.

What is common to these approaches is that state indicators are greatly pre-processed, preferably into binary values. Task-specific knowledge is also used for tuning reward functions by hand. Immediate reward is often necessary in order to guide the learning sufficiently for it to be successful. However, immediate reward may also lead to locally optimal solutions instead of solving the main task.

In the following section, results are shown for a task where an agent should learn to associate three continuous-valued state indicators and five actions so that it reaches a light source as quickly as possible. This task resembles many of those in this section, but using a linear function approximator makes it possible to avoid discretization of state variables.

4 Experimental results

Experiments were performed using a "robot" built with the Lego Mindstorms Robotics Invention System (RIS). The RIS offers a cheap, standard and simple platform for performing real-world tests. In addition to Lego building blocks, it includes two electrical motors; two touch sensors and one light sensor. The main block contains a small computer (RCX) with connectors for motors and sensors. Among others, the Java programming language can be used for programming the RCX.

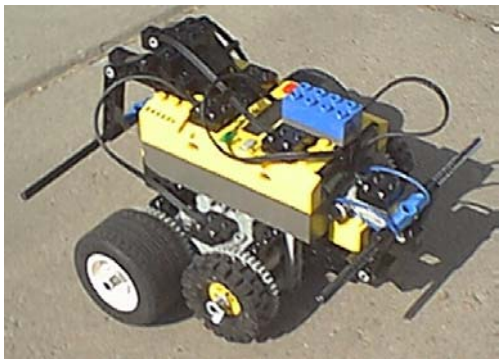

Figure 1. Lego Mindstorms robot. Light sensor is at the top in the front, directed forwards. One touch sensor is installed at the front and another at the rear.

The robot had one motor on each side; touch sensors in the front and in the back and a light sensor directed straight forward mounted in the front (Figure 1). Robots usually have more than one light sensor, which were simulated by turning the robot around and getting light readings from three different directions. One light reading was from the direction straight forward and the two others about 15 degrees left/right, obtained by letting one motor go forward and the other motor backward for 250 milliseconds and then inversing the operation. The light sensor reading from the forward direction after performing an action is directly used as the reward value, thus avoiding hand tuning of the reward function.

Five actions are used, which consist in doing one of the following motor commands for 450 milliseconds: 1) both motors forward, 2/3) one forward, other stopped, 4/5) one forward, other backward. Going straight forward means advancing about 3 cm, actions 2/3 going forward about 1 cm and turning about 15 degrees and actions 4/5 turning about 40 degrees without advancing.

The robot starts about 110 centimetres from the lamp, initially directed straight towards it. Reaching a light value of 80 out of 100 signifies that the goal is reached, which means one to fifteen centimetres from the lamp depending on the approach direction and sensor noise. The lamp is on the floor level and gives a directed light in a half-sphere in front of it. If the robot hits an obstacle or drives behind the lamp, then it is manually put back to the start position and direction. The test room is an office room with noise due to floor reflections, walls and shelves with different colours etc. The robot itself is also a source of noise due to imprecise motor movements, battery charge etc. However, the light sensor is clearly the biggest source of noise as shown in Figure 2, where light sensor samples are indicated for two different levels of luminosity.

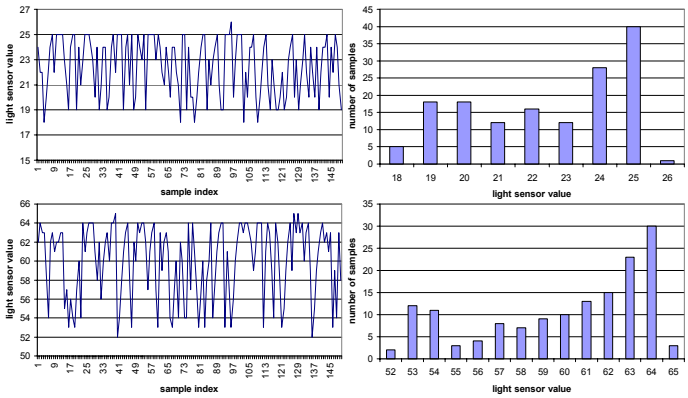

Figure 2. 150 light samples for two different light conditions, taken with 500 millisecond intervals. Average values are 22.5 and 60.0. Raw values are shown to the left, value distribution to the right.

When using an ANN there is one output per action, where the output value corresponds to the action-value estimate of the corresponding action. With five actions and three state variables, a 5x3 weight matrix can represent the weights (no bias input used here). A "hand-coded agent" with pre-defined weights (Table 1) was used in order to prove that a linear approximator can solve the control task and as a reference for judging how good the performance is for learning agents. These weights were determined based on the principle that if the light value is greatest in the middle, then make the forward-going action have the biggest output value. In the same way, if the light value is greater to the left, then favour some left-turning action and vice versa for the right side.

The hand-coded agent reached the goal after the 10-episode average of 19.3 steps. In order to compare with agents using ε-greedy exploration, the hand-coded agent was also tested for the two values of ε used here, i.e. 0.1 and 0.2 (selecting a random action every 10 and 5 times). The 10-episode average for $\varepsilon = 0.1$ is 20.8 steps and for $\varepsilon = 0.2$ it is 31.1. One of the biggest reasons that the average episode length grows so much when increasing the value of ε is that if random actions are used close to the lamp, then the

agent might not be able to reach the light at all anymore. The hand-coded agent with $\varepsilon = 0.2$ was moved back to the start position twice.

Table 1. Hand-coded weights. One row per action, one column per state variable (light sensor reading).

Action	Left	Middle	Right
Forward	0.1	0.8	0.1
Left/forward	0.6	0.3	0.1
Right/forward	0.1	0.3	0.6
Left	0.6	0.2	0.2
Right	0.2	0.2	0.6

Learning agents used the same linear function approximator architecture as the hand-coded agent. Weights are modified by the Widrow-Hoff training rule (4). Experiments were performed both with weights initialised to small random values in the range [0, 0.1) and with weights having optimistic initial values in the range [1, 2). Such weights are optimistic because weight values after training should converge to values whose sum is close to one. This is because state variable values and reward values are all light sensor readings, so the estimated reward value should be close to at least one of the state variable values. If the RL is successful, then the estimated reward should even be a little bit bigger since the goal by definition of RL is to make the agent move towards states giving higher reward.

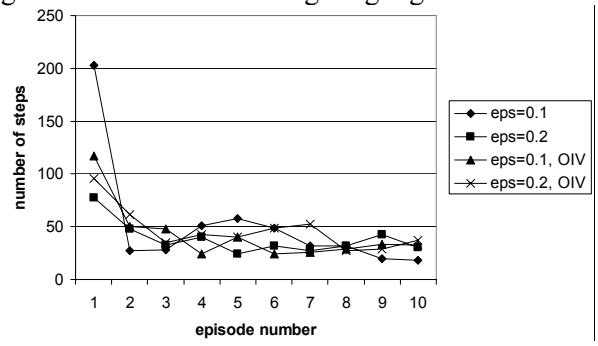

Figure 3. Number of steps per episode, average values from 5 runs. OIV indicates "optimistic initial values".

Actions were selected using ε-greedy exploration with $\varepsilon = 0.1$ and $\varepsilon = 0.2$. The learning rate α in formula (4) was 0.0001 in all experiments, which was determined through experimentation. Bigger values (such as 0.001) lead to weight oscillations, which make the learning fail. Since light sensor values range from zero to 100, it is also easy to see from formula (4) that weight modifications become too large with α values greater than 0.0001.

Figure 3 shows the number of steps per episode as an average from five runs per agent. The number of runs was limited to five due to the long experiment times that are typical for real-world robot experiments. Despite the limited number of runs, some interpretations remain possible. For instance, using $\varepsilon = 0.1$ with small random initial weights gives very long initial episodes. Since learning increases the weights of the first action used, that action is selected all the time until other actions are tested and can update their weights, i.e. every ten steps as an average. This tends to give a policy where only one or two actions continue to be used for over 100 steps, which usually does not make the robot move closer to the light. However, with $\varepsilon = 0.1$ and small random initial weights, there seems to be an advantage in finding an optimal policy as shown by the last two episode averages. Using optimistic initial values gives an advantage with $\varepsilon = 0.1$, but with $\varepsilon = 0.2$ it doesn't provide any advantage compared to the agent using small random initial values.

Table 2. Statistics from five runs for different ε values and weight initialisation methods. "Average 3 best" values are the average value of the three best episodes in Figure 3. OIV stands for "optimistic initial values".

Agent	Total steps	Average 3 best	Manual resets
ε=0.1, rand.	516	21.6	15
ε=0.2, rand.	388	27.4	13
ε=0.1, OIV	426	24.9	16
ε=0.2, OIV	468	30.1	16

Table 2 resumes some statistics for the four agents. The number of total steps shows that the agent using $\varepsilon = 0.2$ and random initial weights performs the best. This advantage is further illustrated by the results obtained when taking the average of the three best episodes. With $\varepsilon = 0.2$, the hand-coded agent had an average episode length of 31.1 steps, so with this ε-value both learning agents perform better. The total number of manual resets to the start position for all episodes and runs with the agent also indicates that the agent with $\varepsilon = 0.2$ learns a "harmless" policy the quickest. With $\varepsilon = 0.1$, the hand-coded agent had an average episode length of 20.8 steps, which is not achieved by neither learning agent using the same ε value. It should, however, be pointed out that the agent using random initial weights and ε=0.1 has a very good average performance for the last two episodes, 19.4 and 18.4.

Even though only immediate reward is used, using reward discounting might still somehow be useful. Taking into account already known next-state action value estimates might make learning faster by

using already learned next-state action values. Q-learning was used with three different discount rates, 0.0 (no discounting), 0.2 and 0.5. ε was set to 0.2 and weights were initialised to small random values. As shown by the results in Figure 4 and Table 3, using discounting makes the agent perform worse. In fact, discounting seems to make the agent get "blocked" into repeatedly using the same actions in the beginning of exploration.

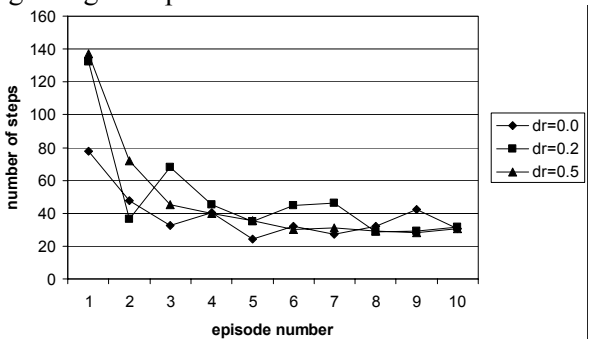

Figure 4. Effect of discount rate (*dr*) on learning. Weights initialized to small random values, ε=0.2.

Table 3. Numerical statistics for different discount rate (γ) values. Weights initialized to small random values, ε=0.2.

Agent	Total steps	Aver. 3 best	Man. resets
γ=0.0	388	27.4	13
γ=0.2	498	29.9	29
γ=0.5	479	29.1	17

5 Conclusions

Even though the light-seeking robot task seems simple, it is still in many aspects comparable with others performed earlier using real robots. Both the robot itself and the environment are sources of noise, which makes the learning task highly stochastic. Especially the use of a linear function approximator makes it possible to overcome or at least reduce these challenges that are typical in real-world tasks. Using a linear function approximator also makes it possible to use continuous-valued state variables directly without value discretization or scaling.

The best learning agents outperformed a hand-coded one, which gives an indication of the advantages that can be obtained by using a learning and adaptable control model instead of using a pre-programmed and static one.

Future work will attempt to solve more difficult tasks involving delayed reward and contradictory goals, e.g. reaching a light source while avoiding obstacles. In such tasks the importance of selecting a good exploration policy can be expected to further increase. Developing exploration policies is therefore a main subject of future research.

References:

[1] Barto, A.G., Sutton, R.S., Watkins C.J.C.H. (1990). Learning and Sequential Decision Making. In M. Gabriel and J. Moore (eds.), *Learning and computational neuroscience : foundations of adaptive networks*. M.I.T. Press.

[2] Boyan, J. A., Moore, A. W. (1995). Generalization in Reinforcement Learning: Safely Approximating the Value Function. In Tesauro, G., Touretzky, D., Leen, T. (eds), *NIPS'1994 proc.*, Vol. 7. MIT Press, pp. 369-376.

[3] Gullapalli, V. (1992). *Reinforcement Learning and its Application to Control*. (Ph.D. Thesis) COINS Tech. Rep. 92-10, Univ. of Mass..

[4] Lin, L.-J. (1991). Programming robots using reinforcement learning and teaching. In *Proc. of the Ninth National Conference on Artificial Intelligence (AAAI)*, pp. 781-786.

[5] Mahadevan, S., Connell, J. (1992). Automatic Programming of Behavior-based Robots using Reinforcement Learning. *Artificial Intelligence*, Vol. 55, Nos. 2-3, 311-365.

[6] Mataric, M.J. (1994). Reward Functions for Accelerated Learning. In Cohen, W. W., Hirsch, H. (eds.), *Machine Learning: Proceedings of the Eleventh International Conference*. Morgan-Kaufmann, CA.

[7] Millán, J. R., Posenato, D., Dedieu, E. (2002). Continuous-Action Q-Learning. *Machine Learning*, Vol. 49, 247-265.

[8] Rumelhart, D. E., McClelland, J. L. et al. (1988). *Parallel Distributed Processing Vol. 1*. MIT Press, Massachusetts.

[9] Rummery, G. A., Niranjan, M. (1994). *On-Line Q-Learning Using Connectionist Systems*. Tech. Rep. CUED/F-INFENG/TR 166, Cambridge University Engineering Department.

[10] Sun, R., Peterson, T. (1998). Autonomous Learning of Sequential Tasks: Experiments and Analyses. *IEEE Trans. on Neural Networks*, Vol. 9, No. 6, 1217-1234.

[11] Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, Vol. 3, 9-44.

[12] Thrun, S.B. (1992). The role of exploration in learning control. In DA White & DA Sofge, (eds.), *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, New York.

[13] Widrow, B., Hoff, M.E. (1960). Adaptive switching circuits. *1960 WESCON Convention record Part IV*, Institute of Radio Engineers, New York, pp. 96-104.