Reinforcement learning benchmarks and bake-offs II

Reinforcement Learning Benchmarks and Bake-offs II

A workshop at the 2005 NIPS conference

Organizers (alphabetically):

Alain Dutech, Tim Edmunds, Jelle Kok, Michail Lagoudakis, Michael Littman, Martin Riedmiller, Brian Russell, Bruno Scherrer, Rich Sutton, Stephan Timmer, Nikos Vlassis, Adam White, Shimon Whiteson.

1 Introduction

It is widely agreed that the field of reinforcement learning would benefit from the establishment of standard benchmark problems and perhaps regular competitive events (bake-offs). Competitions can greatly increase the interest and focus in an area by clarifying its objectives and challenges, publicly acknowledging the best algorithms, and generally making the area more exciting and enjoyable. Standard benchmarks can make it much easier to apply new algorithms to existing problems and thus provide clear first steps toward their evaluation. This is a follow-up workshop from last year's NIPS workshop organized by Rich Sutton and Michael Littman.

Morning session

The morning session involves the 1st RL benchmarking event. The workshop organizers have provided 6 different domains (3 continuous and 3 discrete), and participants apply their RL algorithms on these domains. The organizers have supplied a standardized infrastructure to allow for a set of performance measures to be collected on the cross product of domains and algorithms.

The morning program is as follows:

- 7:30 Michael Littman, introduction
- 7:45 Tim Edmunds, Adam White, software introduction
- 8:00 Michael Littman, results summary
- 8:15 Nikos Vlassis, Shimon Whiteson, discrete domains, results
- 8:40 Roland Hafner, continuous domains, results
- 9:00 break
- 9:15 Spiros Kapetanakis, results
- 9:30 Doina Precup, results
- 9:45 Marcello Restelli, results
- 10:00 Bethany Leffler, results
- 10:15 15-minute wrap up, future events

Afternoon session

The afternoon session involves a discussion on proposals for RL benchmarking, e.g., concrete proposals how to organize a benchmarking competition (potentially web-based), proposals for benchmarks, proposals for performance measures, proposals for software frameworks, etc.

The afternoon program is as follows:

- 3:30 Matthew Molineaux: TIELT (invited talk)
- 4:00 Andrea Bonarini, Allessandro Lazaric, Marcello Restelli: Yahtzee: A large stochastic environment for RL Benchmarks
- 4:15 Risto Miikkulainen: Scaling up to Real World Tasks
- 4:30 Bert Kappen: Time dependent control and the role of noise
- 4:45 Pascal Poupart, Nikos Vlassis: Exploiting Domain Knowledge in Reinforcement Learning
- 5:00 break
- 5:15 Roland Hafner, Martin Riedmiller: Case study: Control of a Real World System in CLSquare
- 5:30 Francesco De Comite, Samuel Delepoulle: PIQLE: A Platform for Implementation of Q-Learning Experiments
- 5:45 Adam White, Rich Sutton: A Standard Interface and Software Package for Benchmarking in RL
- 6:00 Discussion
- 6:30 end

2 The benchmarking event

2.1 Participants

The following people participated in the benchmarking event, from nine different countries (US, FR, NL, PT, DE, UK, IN, IT, CA):

- Pablo Samuel Castro (McGill, CA)
- Francesco De Comite (LIFL, FR)
- Tim Edmunds (Rutgers, US)
- Aakanksha Gagrani (IIT Madras, IN)
- Roland Hafner and Martin Reidmiller (University of Osnabruck, DE)
- Dinakar Jayarajan (IIT Madras, IN)
- Spiros Kapetanakis (QinetiQ, UK)
- Karthik and Sreejith (IIT Madras, IN)
- Jason Keller (Rutgers, US)
- Jelle Kok and Nikos Vlassis (U. Amsterdam, NL)
- Sriram Krishna (IIT Madras, IN)
- Thibault Langlois (Universidade de Lisboa, PT)
- Lihong Li and Jin Zhu (Rutgers, US)

- Lihong Li (Rutgers, US)
- Nikita Lytkin (Rutgers, US)
- Risto Miikkulainen, Nate Kohl, Igor Karpov, Joseph Reisinger (UT Austin, US)
- Muralikrishnan and Shrinivas (IIT Madras, IN)
- Ali Nouri (Rutgers, US)
- Bohdana Ratitch, Doina Precip, Joelle Pineau, Marc Bellemare (McGill, CA)
- Marcello Restelli and Andrea Bonarini and Marcello Restelli (Politicnico di Milano, IT)
- Brian Russell (Rutgers, US)
- Oncel Tuzel and Raghov Subbara (Rutgers, US)
- Munu Sairamesh (IIT Madras, IN)
- Shravan (IIT Madras, IN)
- Himanshu Shrimali and Yogishchandra S Kamath (IIT Madras, IN)
- Nikos Vlassis and Pascal Poupart (U. Amsterdam, NL)

2.2 Software and problems

The benchmarking event was based on the RL-Framework developed at the University of Alberta (`http://rlai.cs.ualberta.ca/RLBB/`). The RL-Framework is intended to provide a foundation for building benchmarks for RL agents. A secondary goal is to support RL competitions in which agents are compared in their performance on new problems that are only revealed at the time of a competition.

There are eight benchmarking categories for which we will be collecting data. They are:

- MountainCar (continuous)
- CartPole (continuous)
- PuddleWorld (continuous)
- ContinuousTriathlon (same system run on all continuous domains)
- Blackjack (discrete)
- SensorNetwork (discrete)
- Taxi (discrete)
- DiscreteTriathlon (same system run on all discrete domains)

2.3 General setting

Each domain is run for a certain maximum number of episodes, Nmax. We set Nmax = 10,000 for all problems.

For all of the problems, a set of 50 starting states is created randomly and held fixed for all participants. A total of Nmax episodes will be run, cycling through the 50 starting states, sequentially.

The principle metric reported is the average summed reward over blocks of 50 episodes. Additional metrics recorded are elapsed wallclock time and number of steps per episode. We will compare the solvers on multiple metrics at the workshop.

Use of prior knowledge about a domain and/or offline computation are allowed, provided they are reported in the results. We also emphasize that we are also interested in solvers regardless of their internal notions of performance metrics and encourage participants to describe the internal metrics used.

We have chosen not to explicitly separate training and testing phases. Each solver should strive to maximize the rewards it accumulates over the Nmax episodes.

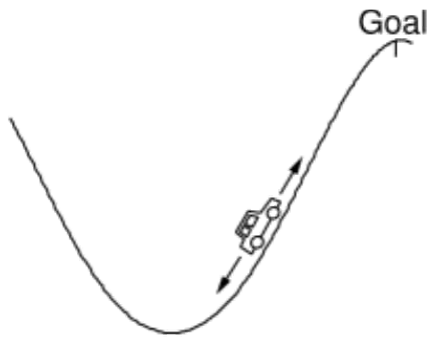
An episode ends if either a terminal state is reached or a certain maximum number of cycles, Cmax is reached. We set Cmax = 300 for all problems. Agents cannot terminate episodes prematurely.

2.4 Problem descriptions

Mountain car

This is the problem described in R. Sutton and A. Barto, Reinforcement Learning: an Inctroduction, MIT Press, 1998.

The objective in this problem is to drive an underpowered car up a steep mountain road, as shown in the figure. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The solution is to first move away from the goal and up the opposite slope on the left. Then, by applying full throttle the car can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way.

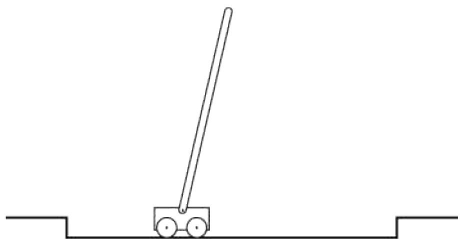


There are two continuous state variables: x1 (the position of the car) and x2 (the velocity of the car). There are three discrete actions: $+1$ (forward throttle), 0 (no throttle), -1 (backward throttle). The starting state of the problem is in the region: $-1.1 \leq x1 \leq 0.49$, $x2 = 0$. Goal states are those with $x1 \geq 0.5$. Finally, the reward is -1 at each step and 0 if a goal state is reached.

Cart pole

The objective in this problem is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over. A failure is said to occur if the pole falls past a given angle from vertical or if the cart runs off the track.

There are four continuous state variables: pole angle (radians from vertical), pole angular velocity (rad/s), cart position (meters from the center), cart velocity (m/s). There are 21 discrete actions, $-10, -9, \ldots, -1, 0, 1, \ldots, 9, 10$, corresponding to various degrees of negative, zero, or positive forces. The starting state is selected from the following region: $-\pi/18 \leq$ pole angle $\leq +\pi/18$, pole angular velocity = 0, $-0.5 \leq$ cart position ≤ 0.5, cart velocity = 0. A failure occurs if |pole angle| $\geq \pi/6$ (pole failed) or |pos| ≥ 2.4 (cart out of limits). Finally, the reward scheme is the following:



0 , if $|\text{pole angle}| \leq \pi/60$ and $|\text{pos}| \leq 0.05$ (balancing),

-1000 , if $|\text{pole angle}| \geq \pi/6$ or $|\text{pos}| \geq 2.4$ (failure),

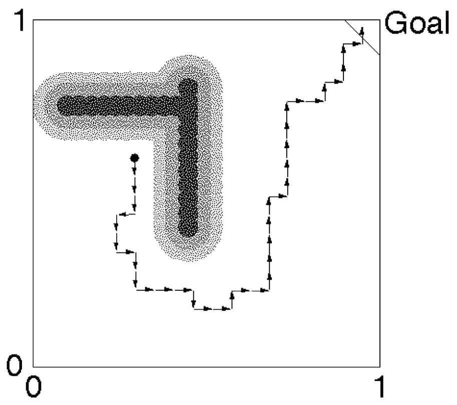
-1 , otherwise

The settings used in the simulator are: cart mass $= 1.0$, pole mass $= 0.1$, pole length $= 0.5$, no friction.

Puddle world

This problem is based on Sutton, R.S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding, NIPS 1996.

The objective in this task is to drive a point robot in a continuous rectangular 2-dimensional terrain to a goal area (unknown to the robot) located at the upper right corner, while avoiding the two puddle areas (see figure). The 2-dimensional area is defined as $[0, 1]^2$. The puddles are oval in shape with a radius of 0.1 and are located at center points $(.1, .75)$ to $(.45, .75)$ and $(.45, .4)$ to $(.45, .8)$.



There are two continuous state variables: the position (x,y) of the robot. There are four discrete actions: 0 - up, 1 - down, 2 - right, and 3 - left. Each action moves the robot by 0.05 in the corresponding direction, up to the limits of the area. A random gaussian noise with mean=0 and std = 0.01 is also added to the motion along each dimension. Starting states are chosen uniformly over $[0, 1]^2$ and the goal states are those with $x + y \geq 0.95 + 0.95 = 1.9$. Finally, there is negative reward at each time step plus additional penalties if either or both of the two oval "puddles" are entered. These penalties are 400 times the distance inside the puddle.

Blackjack

The problem is based on the Blackjack problem described in section 5.1 of the book of Sutton-Barto (1998). Standard rules of blackjack hold, but modified to allow players to (mistakenly) hit on 21. Episodes are prevented from starting in terminal states.

- State space: 1-dimensional integer array, 3 elements,
  element[0] - current value of player's hand (4-21)
  element[1] - value of dealer's face-up card (2-11)
  element[2] - 0-player does not have usable ace (0/1)
- Region for starting states: player has any 2 cards (uniformly distributed), dealer has any 1 card (uniformly distributed)
- Terminal states: Terminates when player "sticks"
- Actions: discrete integers 0-HIT, 1-STICK
- Reward: -1 for a loss, 0 for a draw and 1 for a win.

Sensor network

The distributed sensor network (DSN) problem is a sequential decision making variant of the distributed constraint optimization problem described in: S. M. Ali, S. Koenig, M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In AAMAS'05.

The network consists of two parallel chains of an arbitrary, but equal, number of sensors. The area between the sensors is divided into cells. Each cell is surrounded by exactly four sensors and can be occupied by a target. With equal probability a target moves to the cell to its left, to the cell to its right or remains on its current position. Actions that move a target to an already occupied cell are not executed. It is the goal of the sensors to capture all targets. See figure for a configuration with eight sensors and two targets.

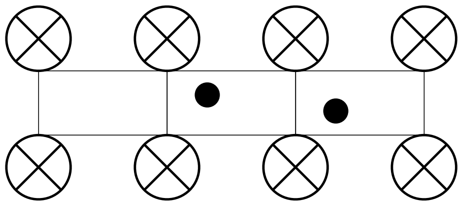


Figure 1: Example sensor network configuration with eight sensors ($\otimes$) and two targets ($\bullet$). The depicted configuration has 37 distinct states. Each sensor can perform three actions resulting in a joint action space of $3^8 = 6561$ actions.

Each sensor is able to perform three actions: track a target in the cell to its immediate left, to its immediate right, or don't track at all. Every track action has a small cost (reward of -1). When in one time step at least three of the four surrounding sensors track a target, it is 'hit'. Each target starts with a default energy level of three. Each time a target is hit its energy level is decreased by one. When it reaches zero the target is captured and removed. The three sensors involved in the capture are each provided with a reward of $+10$. An episode finishes when all targets are captured. In accordance with the other discrete NIPS benchmark problems, each joint action is modeled a single action and the received reward is the sum of the individual agent rewards.

- State space (37): 9 states for each of the 3 configurations with 2 agents, 9 for those with one agent and 1 for those without any agents

- Starting states (3): $[3, 3, 0]$,$[3, 0, 3]$,$[0, 3, 3]$
- Terminal states (1): when both targets have zero hit points $[0, 0, 0]$
- Actions (6561): the actions of each sensor (0=don't track, 1=track left cell, 2=track right cell) are packed into a single integer; the ith sensor's action is: $(\frac{a}{3^{7-i}})$ mod 3
- Reward ($[-8, 54]$): -1 for each sensor focus, $+30$ for eliminating a target

This problem has a relatively small state space compared to the action space. Furthermore, the problem involves multiple targets forcing the sensors to coordinate their actions.

Taxi

This problem is adapted from Dietterich's MAXQ work. The agent controls a taxi on a 5x5 grid with walls (the wall layout is fixed). There are 4 locations on the grid that are possible passenger pick-up/drop-off locations (these locations are also fixed). In each episode, the passenger must be dropped off at a specific drop-off location.

- State variables: taxiLocation [0,24], passengerLocation [0,4] (waiting at pick-up/drop-off [0,3] or in the taxi), drop-offLocation [0,3]
- Region for starting states: taxi is uniformly randomly in any of the grid squares, passengerLocation is uniformly randomly in one of the passenger states, drop-offLocation is uniformly randomly one of the drop-off locations
- Terminal states: 1 - passenger is successfully dropped-off
- Actions: 0-go north, 1-go south, 2-go west, 3-go east, 4-pick up passenger, 5-drop off passenger
- Reward: -1 for an attempted movement (whether it is successful or blocked by a wall), -1 for a successful pickup, 0 for a successful drop-off, -10 for an attempted drop-off with no passenger or at the wrong location, -10 for an attempted pick-up at the wrong location (or if the passenger is already in the taxi).

3 Learning algorithms and benchmarks discussion

In the following pages we provide short summaries of the algorithms that have beed tested in this event. Contributions are grouped by problem. Next we provide short summaries of the afternoon session talks.

LEAP in the Mountain Car domain

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
{bonarini, lazaric, restelli}@elet.polimi.it
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

1 Learning Parameters

Since the MountainCar domain has two continuous state variables, in order to use LEAP we need to discretize them. We have partitioned each variable into 8 equal sized intervals, thus producing 16 initial macrostates. The other learning parameters are reported in the following table.

Parameter	Value
Initial number of macrostates	16
Discount factor	0.95
Learning rate	0.5
Exploration Factor	0.5
Decreasing Rate	0.001
MinExplorationC	2
MinExplorationP	10000

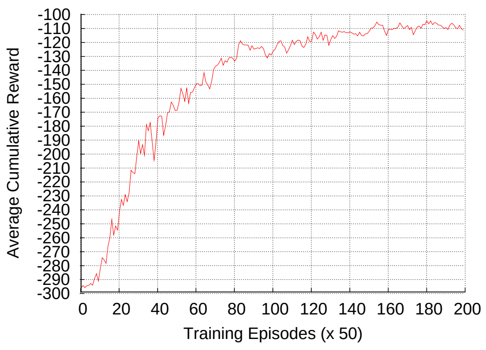


Figure 1: Reward collected by LEAP (10^4 episodes) averaged every 50 episodes.

2 Results

Figure 1 displays the learning curve of LEAP obtained by averaging over 10 runs. The curve starts from -298 and stabilizes around a performance of -110. The cumulative reward averaged over the 10,000 episodes is -149.04.

Figure 2 shows how the number of macrostates changes during the learning process. The growth of macrostates is fast in the first episodes and reaches the number of 60 macrostates. With the same state space discretization, a tabular approach would require 64 states. In this problem, LEAP is not able to rely only on few macrostates, since it is possible to identify the optimal action only by jointly considering the value of the two state variables.

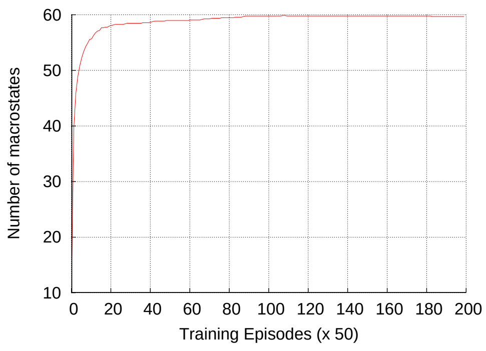


Figure 2: Number of macrostates generated by LEAP.

LEAP in the Cart Pole domain

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
{bonarini, lazaric, restelli}@elet.polimi.it
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

1 Learning Parameters

Since the CartPole domain has four continuous state variables, in order to use LEAP we need to discretize them. We have partitioned each variable into 5 equal sized intervals, thus producing 20 initial macrostates. The other learning parameters are reported in the following table.

Parameter	Value
Initial number of macrostates	20
Discount factor	0.99
Learning rate	0.9
Exploration Factor	0.001
Decreasing Rate	0.001
MinExplorationC	5
MinExplorationP	100

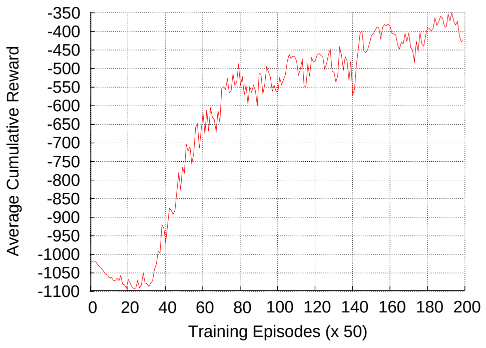


Figure 1: Reward collected by LEAP (10^4 episodes) averaged every 50 episodes.

2 Results

Figure 1 displays the learning curve of LEAP. The curve starts from -1020 and after an initial phase characterized by a low performance, LEAP starts to improve its policy and finally it approaches -350. The cumulative reward averaged over the 10,000 episodes is -627.65.

Figure 2 shows how the number of macrostates changes during the learning process. The number of macrostates grows quite gradually and at the end of the learning process LEAP is using about 430 macrostates. With the same state space discretization, a tabular approach would require 625 states.

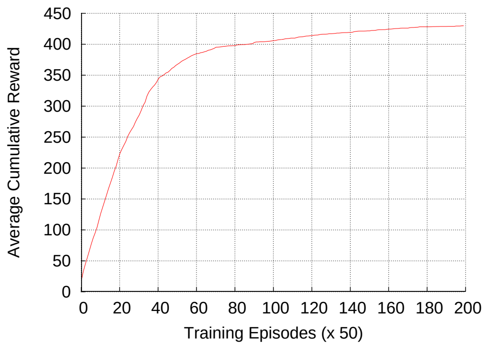


Figure 2: Number of macrostates generated by LEAP.

LEAP in the Puddle World domain

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
{bonarini, lazaric, restelli}@elet.polimi.it
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

1 Learning Parameters

Since the PuddleWorld domain has two continuous state variables, in order to use LEAP we need to discretize them. We have partitioned each variable into 10 equal sized intervals, thus producing 20 initial macrostates. The other learning parameters are reported in the following table.

Parameter	Value
Initial number of macrostates	20
Discount factor	0.95
Learning rate	0.3
Exploration Factor	0.3
Decreasing Rate	0.001
MinExplorationC	2
MinExplorationP	1000

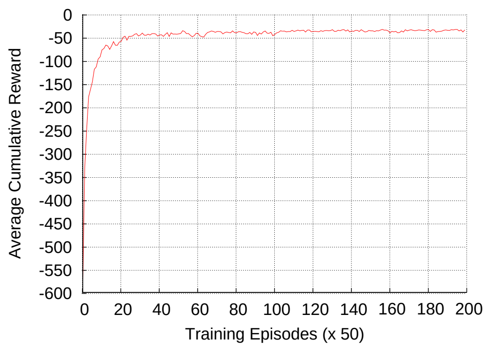


Figure 1: Reward collected by LEAP (10^4 episodes) averaged every 50 episodes.

2 Results

Figure 1 displays the learning curve of LEAP. The curve starts from -457 and stabilizes around a performance of -30 after 1000 episodes. The cumulative reward averaged over the 10,000 episodes is -48.647.

Figure 2 shows how the number of macrostates changes during the learning process. The growth of macrostates is fast in the first episodes and, after 1000 episodes, LEAP has generated an average of 74 macrostates, that are maintained until the end of the learning process. With the same state space discretization, a tabular approach would require 100 states.

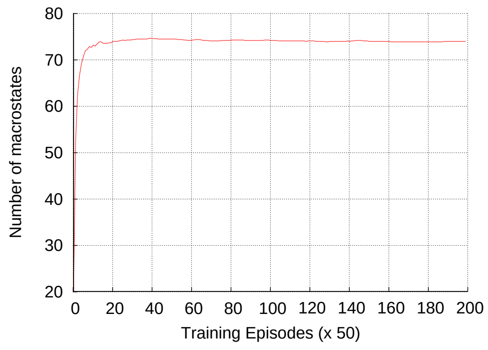


Figure 2: Number of macrostates generated by LEAP.

Efficiently Learning Control in Continuous Domains by Neural Fitted Q Iteration

Roland Hafner and Martin Riedmiller
Neuroinformatics Group
University of Onsabrück

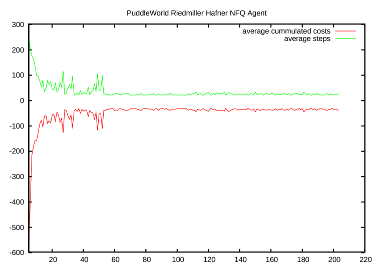
Neural Fitted Q Iteration (NFQ)

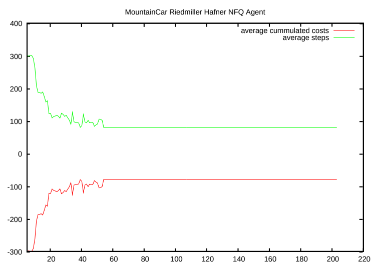
Neural Fitted Q Iteration (NFQ) is a novel RL algorithm for efficient and effective training of a Q-value function represented by a multi-layer perceptron [3], [4]. The method is model-free; it's main point is the principle of storing and reusing transition experiences.

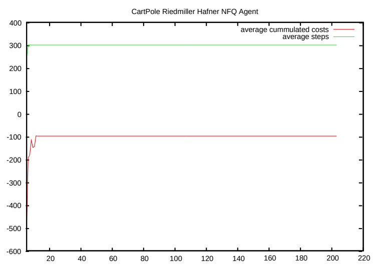
NFQ belongs to the family of fitted value iteration algorithms [2]. In particular, our method is a special realisation of the 'Fitted Q Iteration' framework, recently proposed by Ernst et.al [1]. Whereas Ernst et.al examined tree based regression methods, we propose the use of multilayer-perceptrons (MLPs) with an enhanced weight update method, Rprop. Our method is therfore called 'Neural Fitted Q Iteration' (NFQ).

We argue that the global approximation scheme that underlies MLPs has some favourite properties with respect to generalisation ability. The drawback of such a global approximation scheme - the (negative) mutual influence of adaptation at different inputs - is controlled by doing Q iteration off-line over all transitions in a batch supervised learning scheme (in contrast to on-line updating 'on the fly').

In particular, we want to stress the following important properties of NFQ: (a) the method is model-free. The only information required from the plant are transition triples of the form (state, action, successor state), (b) learning of successful policies is possible with relatively few training examples (data efficiency). This enables the learning algorithm to directly learn from real world interactions, which is currently done in several control and robotic settings [4] and (c) the method is very robust with respect to the choice of the parameters. For all experiments, the same neural network structure was used - only the input layer was adapted to fit the dimension of the state space.







References

[1] D. Ernst and and L. Wehenkel P. Geurts. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[2] G. J. Gordon. Stable function approximation in dynamic programming. *ICML 1995*

[3] M. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In *ECML 2005*, Porto, Portugal, October 2005.

[4] M. Riedmiller. Neural reinforcement learning to swing-up and balance a real pole. *SMC 2005*

NIPS '05 Workshop: Continuous Q and SARSA

Jason Keller
jakeller@cs.rutgers.edu

Agents were trained to solve the MountainCar, PuddleWorld, and CartPole problems using standard Q-Learning, Q-Learning with eligibility traces, and SARSA on a coarsely discretized state space with an ϵ-greedy choice of actions. λ was set to 0.3, ϵ to 0.01, and γ to 0.9 in the implementation of these algorithms. All Q-values were set optimistically to zero. A 20-by-20 state space was used to take advantage of the PuddleWorld problem, and was also used for MountainCar. For CartPole, a 5-by-5-by-5-by-5 state space was used. The table below summarizes the average return results.

Algorithm	MountainCar	PuddleWorld	CartPole
Standard Q	-116.07	-308.00	-125448.47
Q(0.3)	-128.93	-308.21	-78321.17
SARSA(0.3)	-128.73	-307.12	-80136.62

All three agents were fairly well-behaved in completing MountainCar, though the reward for the standard Q algorithm diverged slightly at the end. The agents found a policy for PuddleWorld which reduced the number of steps, but did not greatly change the average return. The agents generally failed in their attempts at the CartPole task, though the agent trained by SARSA happened upon policies that were able to last for 300 steps for between 50 and 250 episodes at a time, which neither Q-learning algorithm was able to do.

XAI[*]: eXplore and Allocate, Incrementally

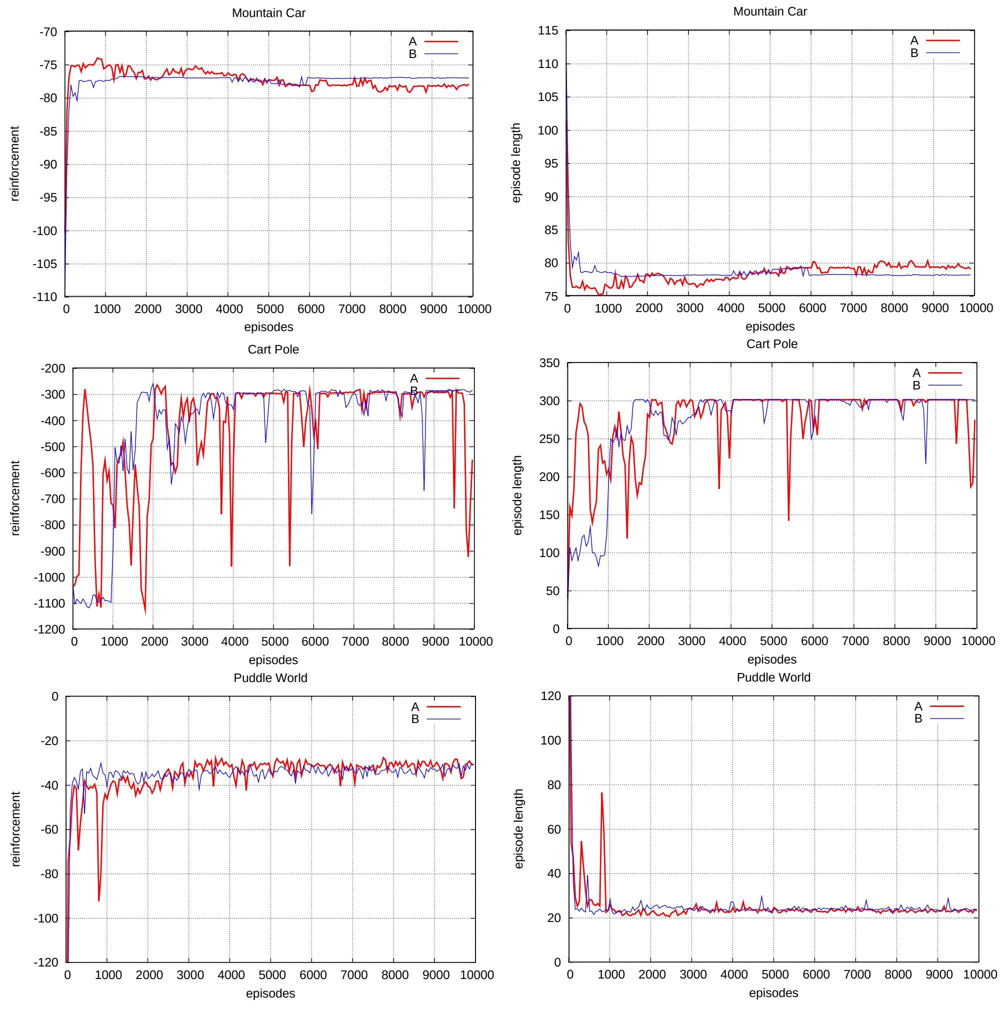
Thibault Langlois
Universidade de Lisboa, Departamento de Informática
Campo Grande – 1749-016 Lisboa, Portugal
`tl@di.fc.ul.pt`

December 5, 2005

The method used for modeling the Value Function allocates memory resources while the agent explores its environment. The model of the Value Function is based on a Radial Basis Functions Network, with Gaussian units. The model is built incrementally by creating new units on-line, when the system enters unknown regions of the space. The model's parameters are adapted using gradient descent and the Sarsa(λ) algorithm. The method does not require a model of the environment neither does it involve the estimation of such a model.

The algorithm consists in starting the experiment with a Value Function that has only one unit centered on the initial state of the environment. As the agent interacts with its environment, new units are created when the system enters previously unexplored areas of the state space.

Once a new unit is created, its center is initialized with the current state vector and remains fixed during training. The width of new units, σ_i is chosen according to the problem at hand. This parameter represents the desired precision of state variables. After a unit is created, the values of σ_i remain fixed.



[*]pronounce "*shy*"

Kernel-Based Reinforcement Learning using KD-trees

Pablo Samuel Castro, Doina Precup, Joelle Pineau, and Bohdana Ratitch
McGill University: School of Computer Science

5th December 2005

Kernel-Based Reinforcement learning was presented by Dirk Ormoneit and Śaunak Sen in their paper *Kernel-Based Reinforcement Learning*, and is the algorithm on which these results are based. This function approximator places Gaussian kernels on sampled points in our state space. New points obtain their value estimates by taking a weighted sum of the value of existing points in its neighborhood. The values are weighed according to a basis function (in our case we used a Gaussian).

The points are stored in a KD-tree, with a strict maximum capacity. For this, the ANN (Approximate Nearest Neighbor) package was used. The difficulty here was that ANN was meant for batch data, but in our case, we are dealing with online data, so a method of adding to the tree was necessary, while still maintaining it well-balanced. ANN was modified to allow for insertion of points, and whenever its size would increase by two-fold, it would force a rebalance of the tree. Also, when the size of our tree would reach its maximum allowed capacity, 10% of the existing points would be removed at random, and the tree would be rebalanced.

Since the functionality of ANN was somewhat altered, memory leaks were discovered as the algorithm was learning. The amount of memory used was increasing enough on each episode that only about 100 episodes could be run safely. In order to get around this problem, every 100 episodes, our tree, along with their value estimates, would be stored on disk. Then the algorithm would start over, but this time reading in the values from the previous run. One benefit of this change is that we can analyze the progress of the algorithm with great detail at regular intervals of the learning process.

This approximator was combined with Sarsa(0) which was available from Bohdana Ratitch's PhD work, thus producing a reinforcement learning algorithm for continuous tasks.

Unfortunately due to the time constraints, most of the time was spent integrating these different algorithms, data structures and interfaces, and fixing memory leaks and integration issues. Because of this, not enough time was spent optimizing parameters. Thus, the results submitted are by no means competitive (especially in the MountainCar environment). However, they are not presented in a 'competitive' manner, but rather, as a means of introducing a new RL framework to the community on which much work can be done.

The only environment which was able to give us satisfactory results given the time constraints was PuddleWorld. For it we used $k = 10$ (for k-nearest neighbors), $\alpha = 0.55$ (learning rate), $\epsilon = 0.1$ (exploration rate), and the maximum allowable size of our kd-tree was 20,000. We used a fixed schedule for α, in other words, it did not decrease throughout training. Finally, we used an activation radius of 0.02 for both dimensions.

For MountainCar, we used $k = 10$ (for k-nearest neighbors), $\alpha = 0.45$ (learning rate), $\epsilon = 0.1$ (exploration rate), and the maximum allowable size of our kd-tree was again 20,000. Again, we used a fixed schedule for α. The activation radii used for both dimensions, respectively, were 0.017 and 0.0014. This α value was giving better results than previous values that were tested earlier. However, because of the deadline, only 5000 episodes were run. The attached data file holds the values obtained in those runs.

Oncel Tuzel
Raghav Subbarao

NIPS RL Benchmarking Workshop
SARSA

Algorithm Description

We implemented epsilon greedy q-learning algorithm (SARSA). The algorithm is as follows:

```
Initialize $Q(s,a)$ to zero
Repeat for each episode
    Initialize $s$
    Choose $a$ from $s$ using $\varepsilon$-greedy policy derived from $Q$
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using $\varepsilon$-greedy policy derived from $Q$
    Update $Q$ as:
        $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    Set new state and action:
        $s = s'$
        $a = a'$
```

Parameter Selection

We used a discount factor of $\gamma=0.99$.

The parameters which affected performance are selected using trial-error by choosing values that perform well for all of the different environments. The parameters are tested with different constant values and different decay rates.

Since we would like to explore the environment more during the initial episodes, ε is initialized to a high value of 0.5. This value is decayed by 10 percent every 50 episodes.

The learning rate parameter is held constant $\alpha=0.3$

These values of α and ε are chosen by trial and error. Notice that for different environments different values might perform better but these values are optimized for the triathlon.

Continuous Environments

We discretize the continuous state values into a discrete space. Each state is discretized into 30 discrete bins. Again 30 is selected by trial and error.

NIPS'05 RL Benchmarks Workshop Algorithm Description

Lihong Li and Jin Zhu
{lihong,zhu}@cs.rutgers.edu
Department of Computer Science, Rutgers University, USA

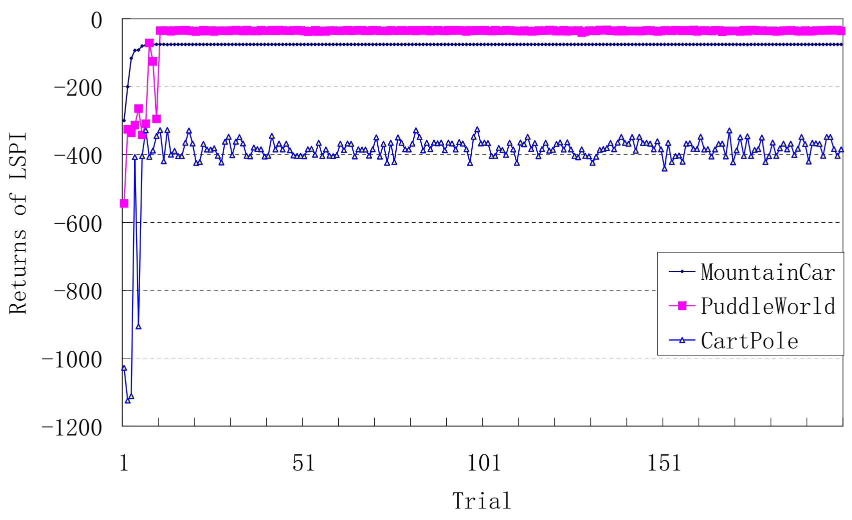
Algorithm and Implementation: We used least-squares policy iteration (LSPI)[1] for our RL agent implementation. A brief summary is as follows:

- The value function was compactly represented by a linear function approximator. The features we have tried include: (i) polynomials, (ii) radial basis functions (RBF), and (iii) CMAC;
- We adopted ϵ-greedy exploration, and learning terminates when ϵ is too small.
- We updated the matrix A and vector b in LSPI every 50 episodes.
- We used the matrix manipulation package, NEWMAT11 (developed by Robert Davies, and available at `http://www.robertnz.net/nm11.htm`).
- We chose to work on continuous domains and use C++.

Results: For MOUNTAINCAR, the features were a grid of 4×4 RBFs evenly located in the state space for each action. For PUDDLEWORLD, the features were 4 tilings of 3×3 grids over the state space for each action. For CARTPOLE, the features were a grid of $2 \times 2 \times 2 \times 2$ RBFs evenly located in the state space for each action; the agent used $\gamma = 0.95$ *internally* for accelerating learning; furthermore, the agent only considered a subset of actions: $\{-10, -6, -2, +2, +6, +10\}$, instead of the original, larger action set. In all three problems, the exploration factor, ϵ, started from 0.5 and was halved when a trial was completed.

Because of space limitation, only the cumulative returns are presented in the figure below. Other details are instead summarized in the table: all quantities are with respect to one episode.

	MOUNTAINCAR	PUDDLEWORLD	CARTPOLE
average return (over all 200 trials)	−78.04	−48.38	−384.7
convergent return (over the last 150 trials)	−75.87	−35.54	−381.0
average time (over all 200 trials)	0.1482	13.17	1.397
average steps to goal (over all 200 trials)	79.04	31.45	292.0



Acknowledgement: We thank Ali Nouri, Michail Lagoudakis, and the workshop organizers for discussions and helps on understanding the problems, algorithms, and the framework.

[1]M. G. Lagoudakis and R. Parr: Least-squares policy iteration. JMLR, 4:1107–1149, 2003.

Model-Based Algorithms in Continuous Domains

Nikita Lytkin nikita.lytkin@rutgers.edu
Michael Littman mlittman@cs.rutgers.edu
Department of Computer Science, Rutgers University, Piscataway, NJ 08854-8019, USA

Algorithms Overview

Two model-based reinforcement learning algorithms are described below. In both cases, the environment was modeled as a Markov Decision Process. Domain state space was discretized, and the corresponding model solved by the Value Iteration method. Grid density was set uniformly in each state space dimension to 100 intervals.

To encourage exploration of a domain, the system was initialized optimistically. Values of all states were set to zero. A high-valued "virtual" state was created. All initial transitions were assumed to lead to the virtual state. All rewards were set to R_{max}.

Actions were chosen greedily based on the current model.

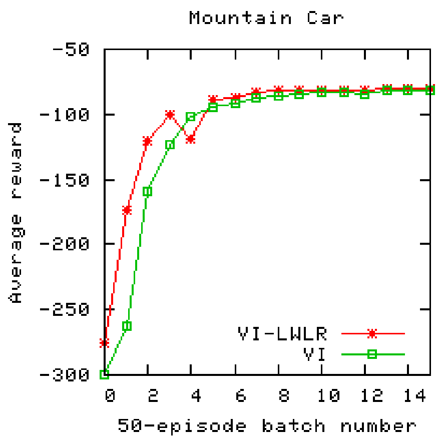
The algorithm described thus far, call it VI, was used as a basis and a baseline for implementation and evaluation of the second algorithm – VI-LWLR.

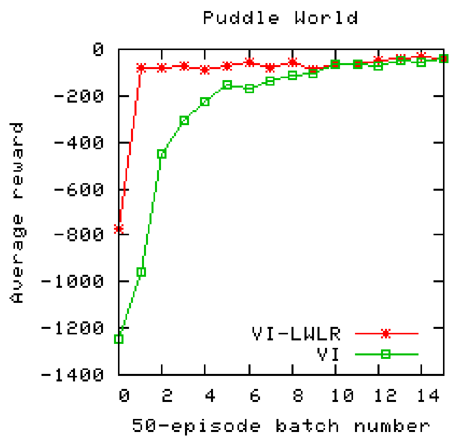
VI-LWLR maintained an additional memory base containing all transitions experienced since the algorithm started in the environment. Each entry in the memory base was of the form $(s, a) \rightarrow s'$, which signified a transition to state s' as a result of performing action a while in state s.

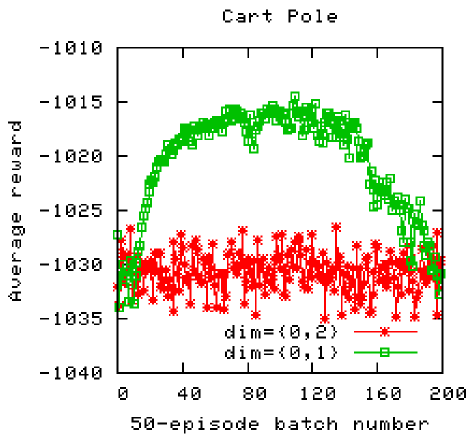
Locally weighted linear regression of the first order was used to estimate transitions from states and actions that the algorithm has not experienced. Kernel used for regression was a Gaussian $K(s, s') = exp\left(-\frac{(s-s')^2}{\sigma^2}\right)$. Estimates obtained by LWLR were used by the VI algorithm when solving the model.

Learning Curves

Mountain Car and Puddle World plots are purposely cut off and zoomed in on the segment where curves for the two algorithms differ.







Model storage and computation requirements grow exponentially in the dimensionality of the state space. Cart Pole proved to be a challenging task making direct application of both VI and VI-LWLR intractable.

The Cart Pole plot shows results achieved by VI utilizing two out of four state space dimensions. Red (lower) line is the plot of the algorithm that used pole angle and cart position. Green (upper) line is the plot of the algorithm that used pole angle and pole angular velocity.

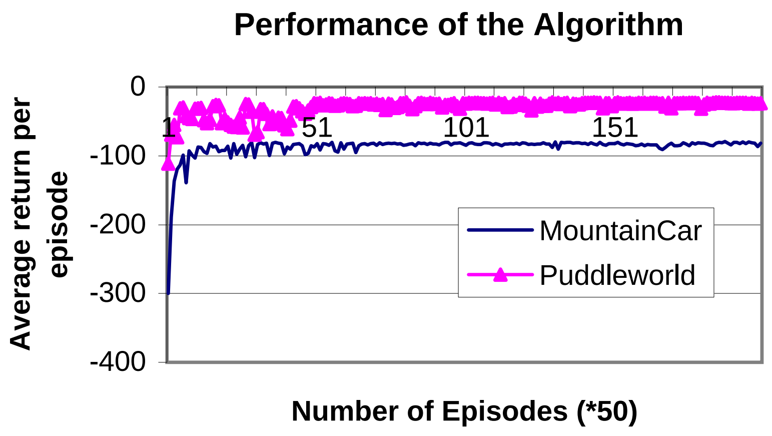
Algorithm Description for NIPS'05 RL Workshop

Ali Nouri [*]
nouri@cs.rutgers.edu

Description: The algorithm implemented here is a model-based learner that uses function approximation for estimation of the parameters of the model (i.e. transition functions and reward functions). Here, we've used neural network to generate transition functions and a rule-based system for the reward functions. Each time a transition is made in the environment, we use that as a training data to our function approximators. Periodically, we build a MDP model of the environment using our experience in the world and our function approximators. In order to build the transition functions, we first discretize the state space, then we use uniform sampling to select some points inside each state and feed them into our neural network. The outputs of the network will be used as sample data to compute the maximum likelihood model of the distribution over the next states. Having built the model, we solve it using Value Iteration technique. The ability to extrapolate beyond the training data allows us to finely discretize the state space, yet maintain a good convergence rate, so the number of intervals for each state variable can be generally a large number. Exploration here is done using epsilon greedy technique and no prior knowledge of the environment was given to the learner except for the discritization factor (which was set by hand for each domain). Due to the short space, other details of the algorithm will not be presented here.

Results: The result for the best set of parameters is shown in the table below. The algorithm is tested in MountainCar and PuddleWorld.

	MountainCar	PuddleWorld
average return (over all trials)	−86.97	−29.57
convergent return (over the last 150 trials)	−83.00	23.74
average time (over all trials)	3.90	0.20



*I would like to thank Lihong Li for his extensive help on function approximation.

Algorithm Description for NIPS'05 RL Workshop

Ali Nouri
nouri@cs.rutgers.edu

Description: The algorithm implemented here is based on RMAX [1] for continuous domains. We discretize the state space and apply RMAX as if we were working in a discrete state space.

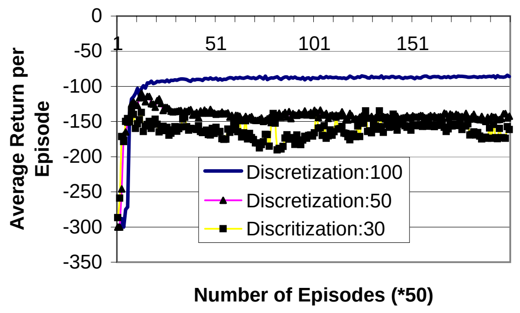
There are a few parameters to tune for this algorithm:

- Known threshold. Number of times to try a state/action before it becomes known. We chose it to be 1 for both MountainCar and PuddleWorld domain. Different values of this parameter didn't work for CartPole.
- Re-evaluation period. The period in which we solve the MDP model using value iteration technique. We chose it to be 3000 steps to have a balance between computation time and performance.
- Discretization size. Determines the coarsness of our discretization. We chose different coarsness and the best value was 100 for MountainCar and 50 for PuddleWorld.
- The program was written in C++ under Linux.

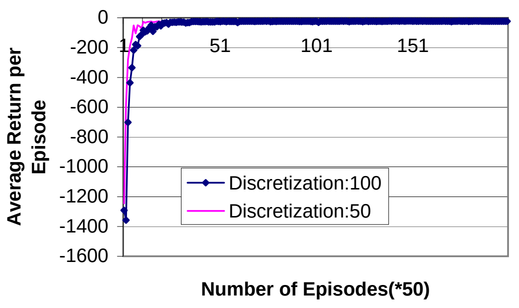
No prior knowledge of the environment was given to the learner except for the discritization size (which was set by hand for each domain).

Results: The result for the best set of parameters is shown in table 1. The algorithm was unable to learn the optimal policy in the CartPole environment, so the results shown here are for MountainCar and PuddleWorld.

	MountainCar	PuddleWorld
average return (over all trials)	−95.20	−36.52
convergent return (over the last 150 trials)	−85.59	−22.85
average time (over all trials)	0.25	0.03



(A) MountainCar Domain



(B) PuddleWorld Domain

References

[1] Ronen I. Brafman, Moshe Tennenholtz, "R-max: A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning", Journal of Machine Learning Research 3 (2002) 213-231.

MDP Solvers for Three Continuous Problem Domains   Brian Russell, Rutgers

There were three solvers for the continuous problems of MountainCar, CartPole and PuddleWorld. The three solvers were essentially identical, and differ only in the hardcoded names of the storage files used and a small correction to a defect in one of the problem environments (i.e. CartPole).

The learning algorithm (discussed below) used by the solvers required Q functions implemented as a two-dimensional array with elements identified with state and action values. States were calculated from observation values passed to the agent_start() and agent_step() functions. Observations were multiple floating point numbers. Each observation value was discretized evenly over its range into integer values. The discretization granularity was set by a preprocessor macro and set to 100 for PuddleWorld, 600 for MountainCar and 120 for CartPole. The discretized observation values were linearized into a single integer state value through an algorithm similar to array element indexing calculation. The only concessions to a problem-specific domain were for CartPole. The concessions were 1) only two of the four observation values were used, namely pole angle and pole angular velocity; and 2) the high-range parameter passed to the discretization function compensated for pole angular velocity values that exceed the range specified in the TaskSpec struct. The first concession made the arrays allocated of manageable size for the Intel hardware and Linux operating system.

The basic learning algorithm was an online temporal learning update for Q functions using the back propagation formula

$$Q(s,a) := Q(s,a) + alpha(s)*(r + gamma*\max_{a'} Q(s',a') - Q(s,a))$$

after each state transition s -> s'. After the update to Q(s,a), policy(s) was immediately updated

$$policy(s) := \operatorname{argmax}_{a} Q(s,a)$$

The agent_start() and agent_step() functions transparently returned an environment-specific action by mapping the internal action determined by policy(s') to the external action set.

The alpha value in the back propagation formula was implemented as a time-dependent function specific to each state value. The implementation used an array of doubles called rate, whose elements were initially set to 1.0 for each state. At each call, rate[state] was incremented by 1.0 after calculation of the alpha value. The actual alpha function was the slowly decreasing 100/(99+rate[state]) which provided faster convergence than the more traditional alpha function 1/rate[state].

Triathlete Implementation for Continuous Problems

By hardcoding generic file names and forcing the state linearization algorithm to use exactly two observation values, a generic solver was created that showed performance identical to the problem-specific implementations. For CartPole, the discretization algorithm complained about pole angular velocity values that exceeded the allowed range, but compensated for this by returning the maximum allowable value for that observation when the observation value exceeded the range limitations.

PIQLE & RL-Benchmarks : the Blackjack task

Francesco De Comité[1]

Tested algorithms and parameter settings

None of our algorithms succeeded in learning significantly for this task. We only report two experiments, one with standard Q-Learning ([SB98] p149), the other using a neural network to memorize/learn the $Q(s, a)$ values. Settings of each algorithm are given in table 1.

Algorithm	ϵ	decay ϵ factor	γ	α	decay α factor	λ	particular
QL3	0	–	1	0.01	constant	–	–
NN2	0	–	1	0.01	constant	–	–

Table 1: Parameters settings

Plot

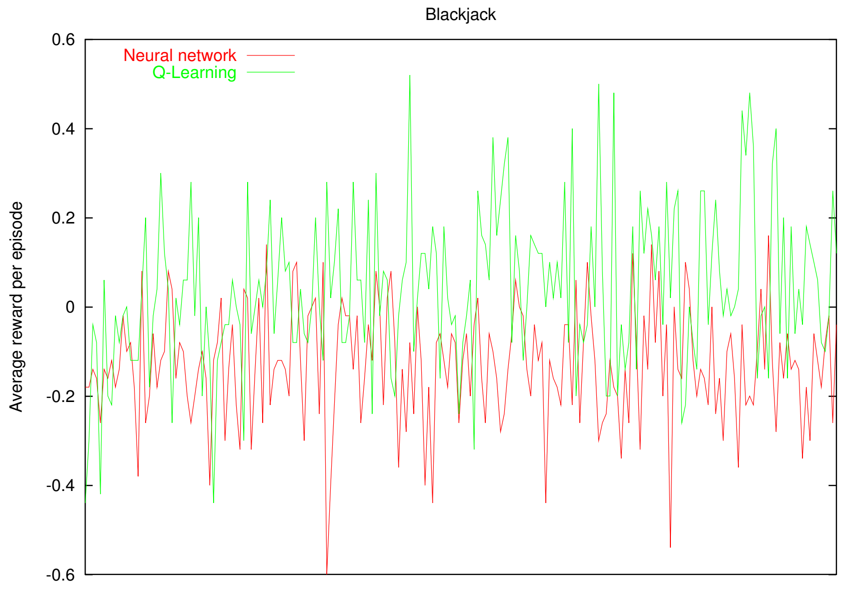


Figure 1: Comparing QL and Neural Networks

References

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998. A Bradford Book.

[1]Grappa,LIFL,UPRESA 8022 CNRS,Université de Lille 1 and Université Lille 3, France e-mail:decomite@lifl.fr

Bayesian RL for Blackjack

Nikos Vlassis (U. Amsterdam) and Pascal Poupart (U. Waterloo)

It is well known that a RL problem can be cast into a Bayesian/POMDP framework by reasoning over the space of beliefs of the unknown transition model [1]. However, solving the derived POMDP is an intractable problem. We present here a simple approach to solve (a general version of) Blackjack by casting the problem as a POMDP and use an approximate solution technique.

We address a version of Blackjack (more general than the benchmark) in which the card drawing distribution is biased in an unknown manner. In reality this could be the case when an unknown set of cards are lost/removed from the deck. Our task is to learn to play for *any* such card distribution. (We know that the benchmark simulator draws cards uniformly in $\{1, \ldots, 10, \mathrm{J}, \mathrm{Q}, \mathrm{K}\}$, but we never make use of this knowledge).

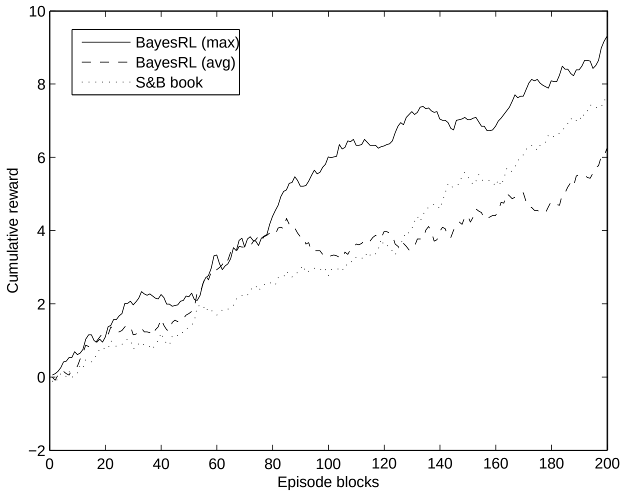
Let $\theta = p(c)$ be a card distribution, with $c \in \{1, \ldots, 10\}$ (where 10 corresponds to $\{10, \mathrm{J}, \mathrm{Q}, \mathrm{K}\}$), and let $b(\theta)$ denote a belief over θ which we model as a Dirichlet distribution (with 10 parameters). Our approach involves an offline step (before starting interacting with the simulator), and an online step (in which the agent is evaluated).

Offline: We sample 500 belief points b_i by drawing uniformly at random Dirichlet counts from $\{1, \ldots, 5\}$. For each belief point b_i we compute its average model $\bar{\theta}_i = \int_\theta \theta b_i(\theta)$ (relative counts) and solve the corresponding MDP over states. (A state s is a triple [playerTotal, dealerCard, playerUsableAce].) We store the resulting Q-function $Q_i(s, a)$ together with the belief point b_i.

Online: We maintain a Dirichlet belief $b(\theta)$, which we initialize to be uniform (all counts 1), and update it everytime we observe a drawn card. We compute the average model $\bar{\theta}$ of $b(\theta)$, and find its nearest neighbor $\bar{\theta}_*$ from all $\bar{\theta}_i$ using Kullback-Leibler divergence. We act greedily wrt $Q_*(s, a)$ for the current state s. After every 100 episodes we update the database: if the KL distance of $\bar{\theta}$ to its nearest $\bar{\theta}_*$ is above a threshold (0.005), we add $b(\theta)$ to the database.

Note that in this Bayesian RL problem there is no exploration/exploitation trade-off, since the unknown card distribution gets sampled at each step (and the model belief is updated) regardless of the current state. Since there is no need for exploration, our strategy strives to exploit by executing the optimal policy of (the closest neighbor to) the average model at each time step.

The figure summarizes the results. We show cumulative rewards (sums over all episode blocks) for the above algorithm (average and maximum; the maximum was obtained in a run in which we didn't update the database online), and for the policy described in the book of Sutton & Barto [2].



[1] M. Duff. *Optimal Learning: Computational procedures for Bayes-adaptive Markov decision processes.* PhD thesis, University of Massachusetts Amherst, 2002.

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

Algorithm used:

The algorithm used is the Monte Carlo algorithm with an epsilon greedy policy.

All the states visited during the episode get their values updated at the end of the episode on the basis of the reward obtained and the policy is also modified accordingly.

However , if the player has less than or equal to 11 points , a hit is forced even if the policy yields a stick.

Also , once every 1000 episodes , the value function is multiplied by 2 in order to reinforce the difference between (s,0) and (s,1).This was experimentally found to give better results than the case where this was not done.

We tried to maximize both the total winnings (i.e wins losses)
as also the percentage of matches won .

No other performance metrics were considered.

Submitted by:

Himanshu Shrimali
Yogishchandra Kamath

Department of Computer Science and Engineering,
IIT Madras .

LEAP in the Blackjack domain

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
{bonarini, lazaric, restelli}@elet.polimi.it
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

1 Learning Parameters

The Blackjack domain has three discrete state variables: current value of player's hand (18 admissible values), value of dealer's face-up card (10 admissible values), and a boolean variable that is true player has a usable ace (2 admissible values). Therefore, LEAP can be initialized using one LE for each state variable, for a total of 30 initial macrostates. The other learning parameters are reported in the following table.

Parameter	Value
Initial number of macrostates	30
Discount factor	0.9
Learning rate	0.5
Exploration Factor	0.1
Decreasing Rate	0.001
MinExplorationC	3
MinExplorationP	20

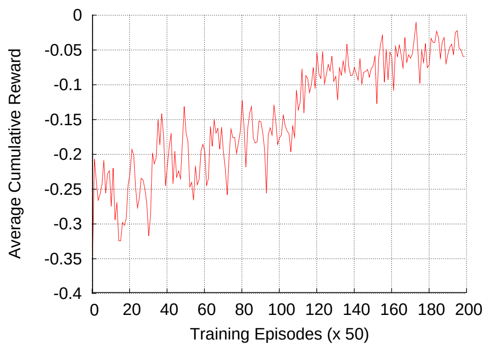


Figure 1: Reward collected by LEAP (10^4 episodes) averaged every 50 episodes.

2 Results

Figure 1 displays the learning curve of LEAP obtained by averaging 50 runs. The curve starts from -0.25 and stabilizes around a performance of -0.05 after 10,000 episodes. The cumulative reward averaged over the 10,000 episodes is -0.14633. The noise affecting the graph is due to the fact that Blackjack is a highly stochastic game.

Figure 2 shows how the number of macrostates changes during the learning process. The growth of macrostates is fast in the first episodes as usual and, after about 4000 episodes, LEAP has generated an average of 38 macrostates, that are maintained until the end of the learning process. The tabular approach would need 360 states.

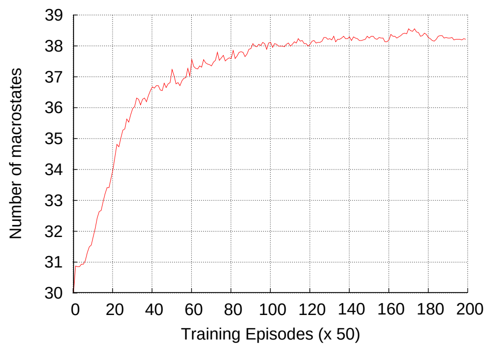


Figure 2: Number of macrostates generated by LEAP.

PIQLE & RL-Benchmarks : the DSN task

Francesco De Comité[1]

Tested algorithms and parameter settings

The huge number of possible actions (6561) leads us to make some modifications to the standard definitions and methods of states, actions, and environments.

- We do not generate all the possible actions for a given state : instead, we have decided to consider only n actions randomly chosen (n becomes a tunable parameter).
- For use with neural networks, we can no longer code an action with a 1 among k coding scheme. Instead, we transformed the action number p into a double $\frac{p}{6561}$. We also tried others coding of actions. We can see no significant differences between those two coding schemes, as far as the behaviour of algorithms are concerned.

Tested algorithms

Even when restricting the number of actions, storing and retrieving $Q(s, a)$ can take a long time, so we have mainly tested algorithms using neural networks to store/learn $Q(s, a)$. Standard Q-Learning algorithms were also ran, in order to be able to compare them with neural networks Q-learners.

Algorithm	ϵ	γ	α	decay α factor	actions	hidden neurons
QL4	0	1	0.01	constant	4	–
QL5	0	1	0.01	constant	7	–
NN3	0	1	0.01	constant	7	5
NN4	0	1	0.01	constant	33	5
NN5	0	1	0.3	$1 - 10^{-6}$	33	5
NN6	0	1	0.3	$1 - 10^{-6}$	263	10

Table 1: Parameters settings

Plot

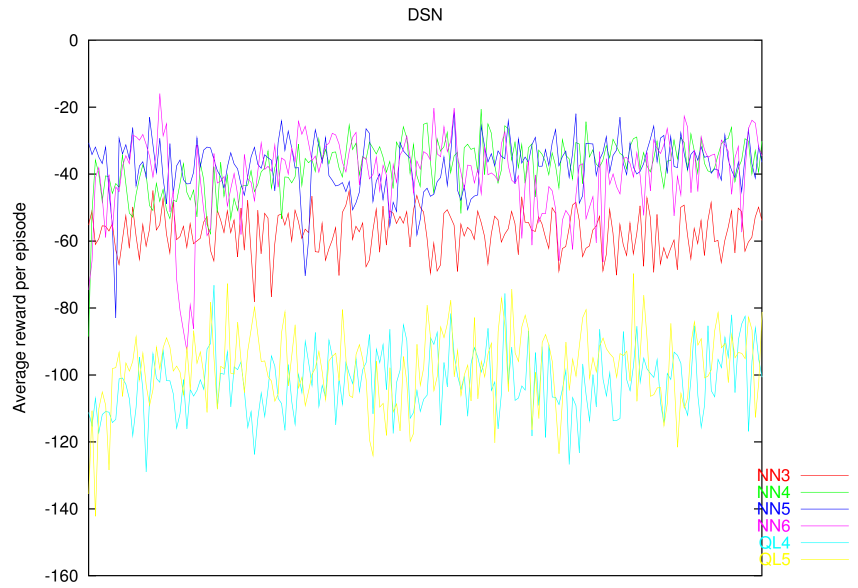


Figure 1: Comparing algorithms

[1]Grappa,LIFL,UPRESA 8022 CNRS,Université de Lille 1 and Université Lille 3, France e-mail:decomite@lifl.fr

Experiments on the DSN benchmark problem

Jelle R. Kok and Nikos Vlassis, University of Amsterdam

We show results for the following tabular Q-learning techniques applied to the Distributed Sensor Network (DSN) problem with 8 agents and 2 targets:

- **MDP learners**. Standard Q-learning [2, 3] using the complete state-action space.

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

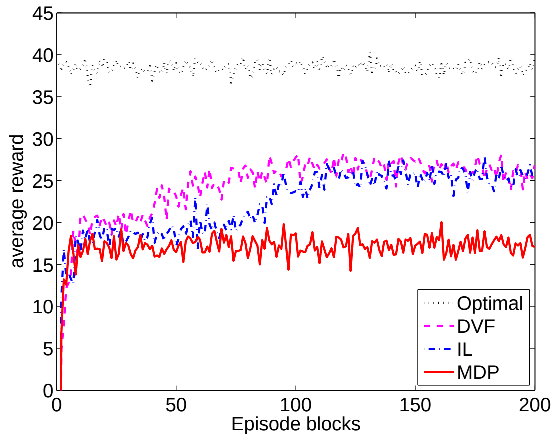
- **Independent Learners (IL)**. Standard Q-learning in which each agent learns Q-values based on the full state information and its own individual action, a_i. In this method, it is assumed that the number of agents is known. Furthermore, the agent explore jointly, i.e. with probability ϵ all agents perform a random action.

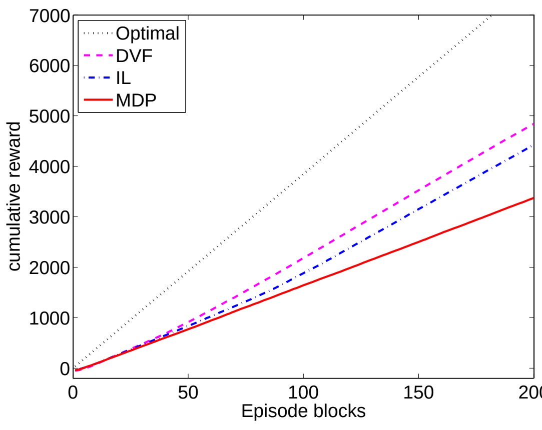
- **Distributed Value Functions (DVF)**. As with IL, each agent i learns individual Q-values. However, each Q-value is updated by averaging over its own Q-value and those of its neighbors, $\Gamma(i)$ [1]:

$$Q_i(s_i,a_i) := (1-\alpha)Q_i(s_i,a_i) + \alpha[R_i(s,a) + \gamma \sum_{j\in\{i\cup\Gamma(i)\}} f(i,j) \max_{a'_j} Q_j(s',a'_j)],$$

  where $f(i,j) = \frac{1}{|i\cup\Gamma(j)|}$. This method additionally assumes that each agent knows its neighbors, possible agents it can coordinate with, in the network. The neighbors of agent i are defined as those agents that are able to track at least one of the cells agent i can track.

Fig. 1(a) and Fig. 1(b) show respectively the average and cumulative reward for the above solution techniques. In all cases, the following parameters were used: $\alpha = 0.2$, $\epsilon = 0.2$, and $\gamma = 0.9$. Both plots also show the results for the, manually implemented, optimal policy (including random exploration actions with probability ϵ).





References

[1] J. Schneider, W.-K. Wong, A. Moore, and M. Riedmiller. Distributed value functions. In *Proc. Int. Conf. on Machine Learning*, Bled, Slovenia, 1999.

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[3] C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

PIQLE & RL-Benchmarks : the Taxi task

Francesco De Comité[1]

Tested algorithms and parameter settings

We used standard Q-Learning algorithm ([SB98] p149), Watkins's $Q(\lambda)$ ([SB98] page 184) ,Peng's $Q(\lambda)$ ([SB98] p182 and [PW94]), a variant of Q-Learning where we train a feed-forward neural network to learn the values $Q(s,a)$, and experimental Q-Learning approximation algorithm, using only integer calculations ([AS04]). Settings of each algorithm are given in table 1.

Algorithm	ϵ	decay ϵ factor	γ	α	decay α factor	λ	particular
QL1	0.5	0.999	0.9	0.5	$1-10^{-10}$	–	–
QL2	0.5	0.999	0.9	0.5	$1-10^{-6}$	–	–
NN	0.5	$1-10^{-5}$	0.9	0.3	constant	–	
QL INT	–	–	–	–	–	–	–
Peng1	0.5	$1-10^{-5}$	0.9	0.3	constant	0.9	traces not replaced
Watkins1	0.5	$1-10^{-5}$	0.9	0.3	constant	0.9	traces not replaced

Table 1: Parameters settings

Plot

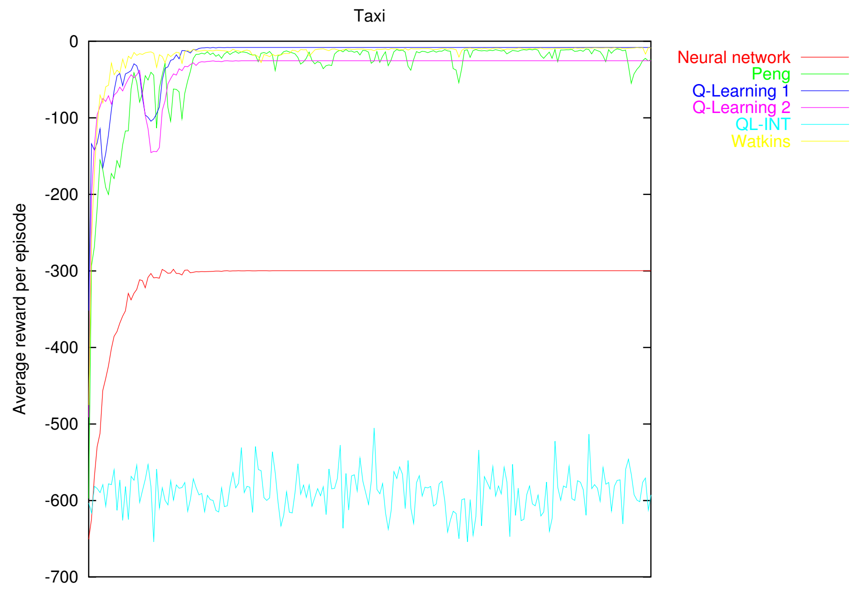


Figure 1: Comparing algorithms

References

[AS04] M. Asadpour and Roland Siegwart. Compact q-learning optimized for micro-robots with processing and memory constraints. *Robotics and Autonomous Systems*, 48(1):49–61, August 2004.

[PW94] Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. In *International Conference on Machine Learning*, pages 226–232, 1994.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. A Bradford Book.

[1]Grappa,LIFL,UPRESA 8022 CNRS,Université de Lille 1 and Université Lille 3, France e-mail:decomite@lifl.fr

Taxi problem

Muralikrishnan V P
Shrinivas M
Dept. of Computer Science and Engineering
Indian Institute of Technology, Madras, India

Implementation

We have implemented Options Learning on the Taxi Problem. The problem was decomposed as per the hierarchies given in [1]. In our implementation, we have incorporated implicit information regarding which option to execute.

```
Root Level:        if( passenger inside the taxi )
                       put( passenger at destination )
                   else
                       get( passenger )

Get Level:         if( taxi at passenger location )
                       pickup( passenger )
                   else
                       navigate( passenger location )

Put Level:         if( taxi at destination )
                       dropoff( passenger )
                   else
                       navigate( destination )
```

Navigate: Agent learns to navigate to the specified destination (option) using Q(0) Learning algorithm[2] with the following parameters:
alpha - 0.1
gamma - 0.9
epsilon - 0.01 (for epsilon-greedy action selection)
while choosing greedy action, ties are resolved randomly

References

1. Thomas G. Dietterich, 2000, Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition, Journal of Artificial Intelligence Research

2. Sutton, R., Barto, A. G., Introduction to Reinforcement Learning, MIT Press

LEAP in the Taxi domain

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
{bonarini, lazaric, restelli}@elet.polimi.it
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

1 Learning Parameters

The Taxi domain has three discrete state variables: taxi location (25 admissible values), passenger location (5 admissible values), and drop-off location (4 admissible values). LEAP uses a LE for each state variable, for a total of 34 initial macrostates. The other learning parameters are reported in the following table.

Parameter	Value
Initial number of macrostates	34
Discount factor	0.95
Learning rate	0.5
Exploration Factor	0.5
Decreasing Rate	0.0005
MinExplorationC	2
MinExplorationP	500

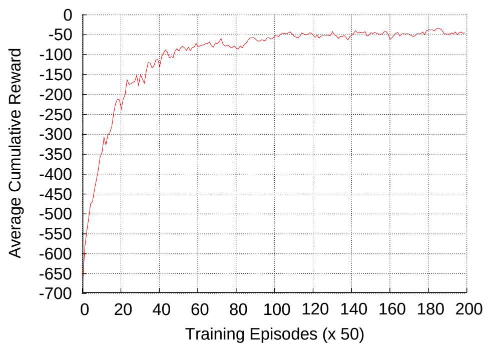


Figure 1: Reward collected by LEAP (10^4 episodes) averaged every 50 episodes.

2 Results

Figure 1 displays the learning curve of LEAP obtained by averaging over 10 runs. The curve starts from -650 and stabilizes around a performance of -50 after 5000 episodes. The cumulative reward averaged over the 10,000 episodes is -103.32.

Figure 2 shows how the number of macrostates changes during the learning process. The growth of macrostates is fast in the first episodes and, after 3000 episodes, LEAP has generated an average of 250 macrostates, and reaches 270 macrostates at the end of the 10,000 episodes. With the same state space discretization, a tabular approach would require 450 states.

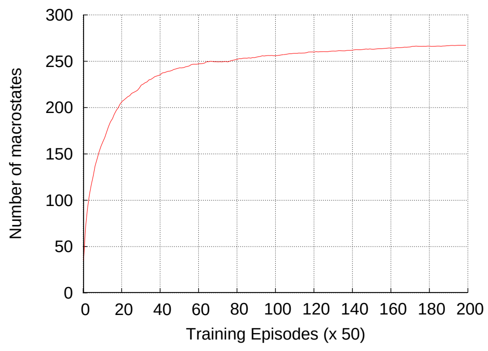


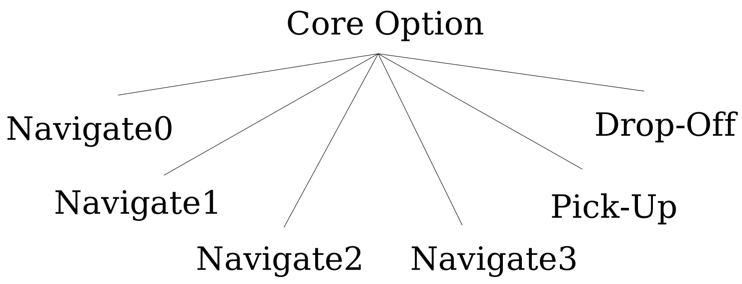
Figure 2: Number of macrostates generated by LEAP.

Hierarchical Taxi Agent

Implemented by Shravan M. N.
Department of Computer Science
IIT Madras, India

The solution uses a one level hierarchy as shown below to solve the Taxi problem.

The Core Option has 4 sub options as its actions & 2 primitive actions.



The Navigate nodes are sub options & have the primitive actions N,S,W & E. The core option's state space is the same as the problem's state space. The state space of the sub options is reduced to only the taxi location. The termination criteria for the Navigate option is when they reach their respective locations(say 0 reaches location 0,0, 1 reaches location 0,4, 2 reaches 3,0 & 3 reaches 4,4).The navigate options get a reward of -1 on every non terminal step & pseudo reward of +5 for reaching its goal, the termination location. The core option terminates when the environment terminates it. The core option gets a -1 on every wrong navigation, -10 on wrong pickup or dropoff & pseudo reward of +1 for a successful navigation, +5 for a succesful pickup & a +10 for a successful dropoff.

All options are implemented as a single class which is instantiated into different options, each with its own appropriate state space & action space. The termination & reward assignment is with the controller which acts as an interface between the environment and the hierarchical agent. The learning method used can be changed by changing the agent_step method of the option. Right now Q-learning & sarsa have been implemented. One important point to be noted is that the actions of the core option are not considered temporally extended. I simply neglect the amount of time needed to execute an option. This will not affect the solution because here there is only one sequence of optimal actions possible & we have to take it no matter what time it takes. So all that the agent executing the core option needs to learn is the right order of actions. Parameters for all the options are same and are alpha=0.1,gamma=0.9,epsilon=0.1. The implementation of the learning algorithm is very similar to that given for mineAgent example with the RL framework. Only the Option framework and the option controller have been implemented newly.

An Optimum Agent for the Discrete Triathlon

Timothy Edmunds
Rutgers University

In order to provide a baseline for comparison, we implemented an agent that executes an optimal policy for each of the environments in the discrete triathlon. The agent makes use of full domain understanding to choose an optimum action for any given observation.

For the discrete sensor network, the three possible target configurations correspond to a choice of three actions, where the appropriate choice will always reduce the health of both targets by one. Thus every episode ends after 3 steps, with a total reward of 42 (-6, -6, 54).

For the taxi, a hand-built lookup table is used to map from state to action such that a shortest path is always taken from the current location to the pickup or drop-off location (depending on whether the passenger is on board). Although the rewards are different for different episodes (since the shortest path length varies with the initial placements), since the same ensemble of 50 starting states, the average reward per ensemble is the same (-8.36) for all 200 ensembles.

Since the discrete sensor network and taxi environments are deterministic, an optimal policy obtains the same result for each ensemble of episodes; the blackjack environment, however, yields different rewards for the same state-action pairs in different ensembles. For this environment, we used dynamic programming to pre-compute a lookup table for actions that would maximize the expected reward in this implementation of the blackjack problem. Note that the policy we found (see Figure 1) differs from the one given in [1]; for this implementation, our policy yields a higher reward. The result of our fixed policy is shown in Figure 2.

Although we have referred to our blackjack policy as optimal, this is only the case under the assumption that the environment truly is a stochastic MDP. Another way to view the environment is as a deterministic POMDP, where the system's random number seed is a hidden variable; if an agent can learn the random number generator's evolution, it can predict what the result of a hit action will be. For completeness, we also implemented an agent that uses perfect knowledge of the system's random seed to maximize its reward; this agent is able to obtain an average reward of 0.595 per episode. The final caveat is that because of the way the environment and interaction is implemented, the agent can actually *affect* the environment's hidden state (by making its own calls to rand()); by doing so, an agent would actually be able to win every hand.

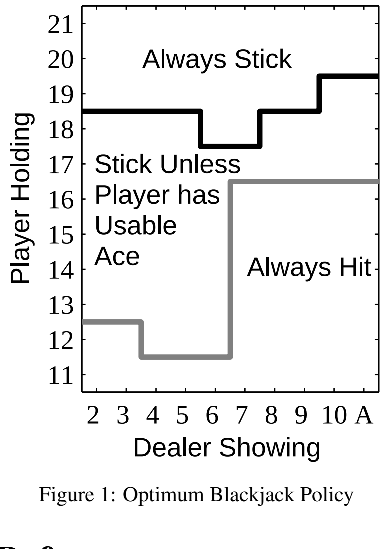


Figure 1: Optimum Blackjack Policy

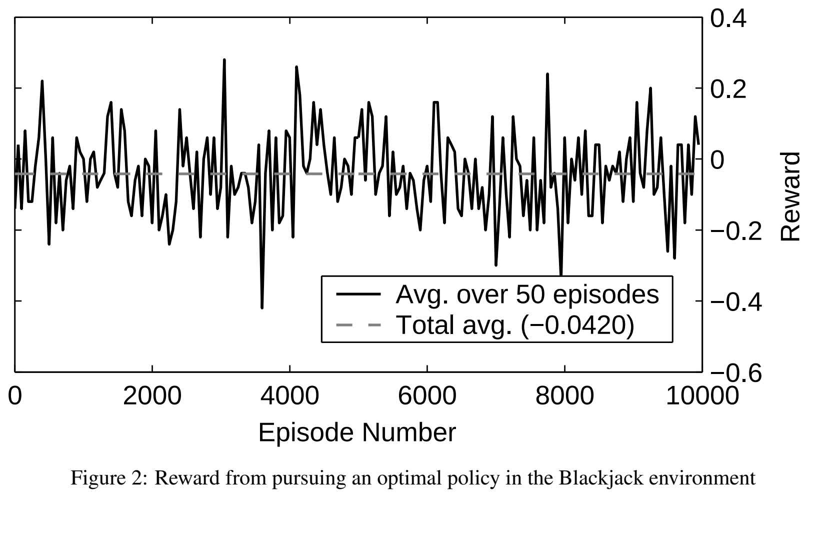


Figure 2: Reward from pursuing an optimal policy in the Blackjack environment

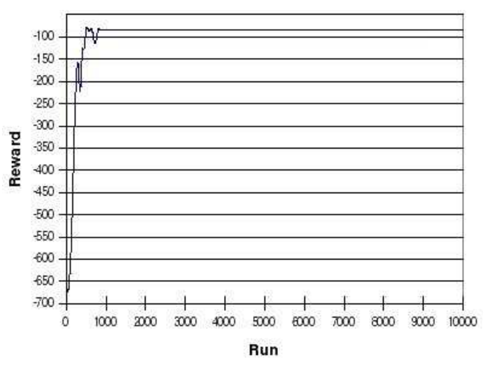
References

[1] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

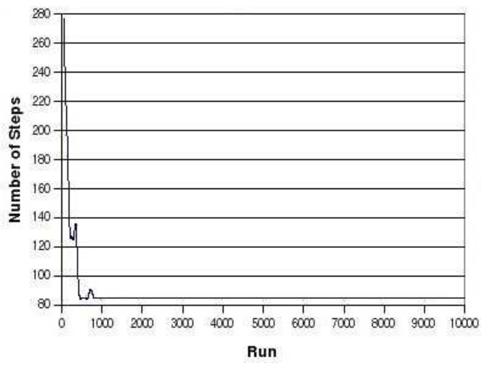
An Rmax Agent for Taxi and Blackjack Environments

Bethany R. Leffler
Rutgers University

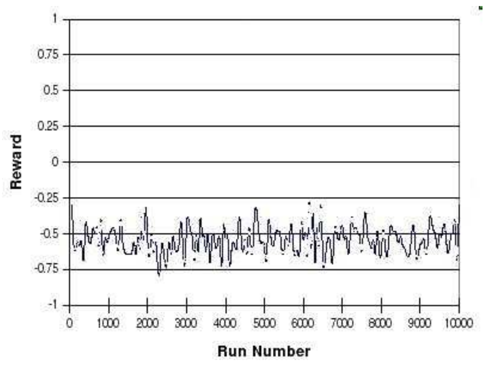
In the discrete environment, I ran the Rmax algorithm as described in Brafman and Tennenholtz [1]. This algorithm uses an optimistic reward structure. All states return the maximum reward until the actual reward is truly known.



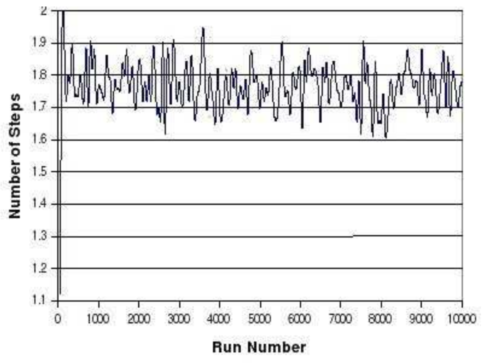
(a) Average reward received in the taxi environment



(b) Average number of steps taken per episode in the taxi environment



(c) Average reward received in the blackjack environment



(d) Average number of steps taken per episode in the blackjack environment

References

[1] R. I. Brafman and M. Tennenholtz. R-MAX — a general polynomial time algorithm for near-optimal reinforcement learning. In *IJCAI*, pages 953–958, 2001.

Neuroevolution Reinforcement Learning

Risto Miikkulainen, Nate Kohl, Igor Karpov, and Joseph Reisinger
Department of Computer Sciences
The University of Texas at Austin
risto,nate,ikarpov,joeraii@cs.utexas.edu

Abstract

Evolution of neural networks, through genetic algorithms or otherwise, has recently emerged as a possible way to solve challenging reinforcement learning problems; NEAT (or NeuroEvolution of Augmenting Topologies), is a particularly powerful such method. Because it is based on evolving a population of solutions, where individuals are evaluated on a number of test episodes, the current setup of the RL Benchmarking Event tasks makes it impossible to compare with it directly. In particular, neuroevolution methods are geared towards finding the best solution (instead of optimizing average reward over the course of learning), and towards reinforcement over the entire episode (instead of during the episode). NEAT in particular is best suited for challenging tasks where topology evolution has time to have an effect, especially those that are non-Markovian and therefore require recurrency. Nevertheless, it is instructive to compare how NEAT solves these problems, and where the particular strengths and weaknesses are.

1 The NEAT Algorithm

The NeuroEvolution of Augmenting Topologies (NEAT) method [6] is a policy-search reinforcement learning algorithm that uses a genetic algorithm to search for optimal neural network policies. NEAT automatically evolves network topology to fit the complexity of the problem by combining the usual search for the appropriate network weights with *complexification* of the network structure. By starting with simple networks and expanding the search space only when beneficial, NEAT is able to find significantly more complex controllers than other fixed-topology learning algorithms. This approach is highly effective: NEAT outperforms other neuroevolution (NE) methods on complex control tasks like the double pole balancing task [5, 6] and the robotic strategy-learning domain [7]. These properties make NEAT an attractive method for evolving neural networks in complex tasks. In this section, the NEAT method is briefly reviewed; see [5, 6, 7] for more detailed descriptions.

NEAT is based on three key ideas. First, evolving network structure requires a flexible genetic encoding. Each genome in NEAT includes a list of *connection genes*, each of which refers to two *node genes* being connected. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows finding corresponding genes during crossover. Mutation can change both connection weights and network structures. Connection weights are mutated in a manner similar to any NE system. Structural mutations, which allow complexity to increase, either add a new connection or a new node to the network. Through mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions.

Each unique gene in the population is assigned a unique innovation number, and the numbers are inherited during crossover. Innovation numbers allow NEAT to perform crossover without the need for expensive topological analysis. Genomes of different organizations and sizes stay compatible throughout evolution, and the problem of matching different topologies [4] is essentially avoided.

Second, NEAT speciates the population so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before they have to compete with other niches in the population. The reproduction mechanism for NEAT is *explicit fitness sharing* [1], where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

Third, unlike other systems that evolve network topologies and weights [2, 8], NEAT begins with a uniform population of simple networks with no hidden nodes. New structure is introduced incrementally as structural mutations occur, and the only structures that survive are those that are found to be useful through fitness evaluations. In this manner, NEAT searches through a minimal number of weight dimensions and finds the appropriate complexity level for the problem.

2 Applying NEAT to the RL Benchmarking Event Tasks

Unfortunately, the tasks seem to have been designed with temporal-difference learning in mind; policy-iteration, and especially population-based methods like NEAT assume a different kind of setting. There are four issues:

1. The population-based methods are designed for off-line learning, i.e. improving the best solution found so far as much as possible during learning. It does not make much sense to plot "average rewards"; instead, progress can be visualized by plotting the average performance of the best individual found so far. Such graphs answer the question "If learning was stopped at a given point in learning, how well would the best solution found so far perform?". In the results that follow, such "expected performance" graphs are used to illustrate progress of learning, as is usual for population-based methods.

   It is of course possible to plot the average reward as well, and below it is done for several of the domains. The main point is that such reward does not work: if we were to optimize it using population-based methods, we would end up with suboptimal solutions. The reason is that a population always contains many individuals that perform poorly: they are there to provide the diversity that allows the algorithm to discover good solutions later. However, if performance is measured with a average reward, it is better to stop the search soon as a competent solution has been found, instead of letting evolution to run its course.

In principle, it might be possible to extend population-based methods to optimize average reward as well. For example, population culture [3] could be used to focus the evaluation episodes only on most promising individuals, i.e. those that have been deemed at least competent by comparing to the best existing individuals. So far, evolution has been an off-line method only; such extensions might allow applying it to online tasks as well.

2. Because each individual is evaluated based on how well it performs over several episodes, reinforcement *during* the episode is not useful. Unfortunately, the reward structure in some of the RL Benchmarking Event tasks makes it difficult, and in some cases impossible, to learn the task using policy-search methods like NEAT.

   For example, in the cart-pole tasks, the reward is the same whether the pole falls early or late in the episode. Such rewards do not allow distinguishing good individuals from the bad. Instead, off-line methods need to use a fitness that penalizes for failing early, and rewards for keeping the pole balanced longer. This is the function used in the experiments below.

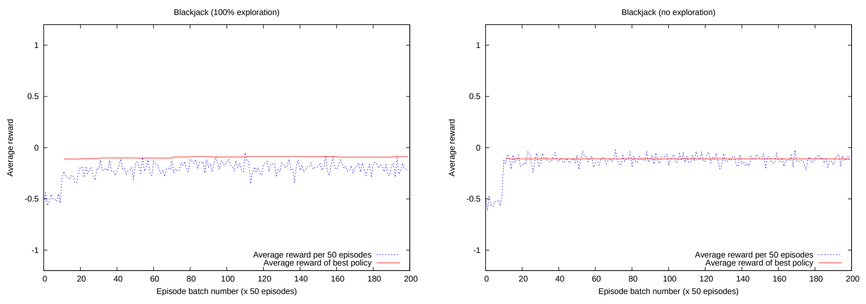
3. Methods such as NEAT assume that the tasks are at least hard enough so that dozens or hundreds of generations are needed to solve them. Only then there is enough time to discover the useful structures and make use of them in constructing good solutions. The tasks in the current competition do not allow that. For example in blackjack and mountain car, solutions are sometimes found in the initial random population, without any evolution at all.

   In future benchmarking events, it would be useful to include more challenging tasks, such as balancing two poles at once, or playing Go or Othello, so that the power of these methods could really be put to test.

4. A very interesting and important class of problems includes POMDPs, or those where some kind of memory is required in addition to the current inputs. Methods such as NEAT, which evolve recurrent network topologies, were specifically designed with such problems in mind. The idea is to discover the proper recurrency, i.e. the proper memory for the task. Such properties of neuroevolution are not tested at all in the current set of tasks.

   Unfortunately no such tasks were included in the RL Benchmarking Event. It turns out that recurrency still allows NEAT to perform better than when recurrency is not allowed. Apparently the networks utilize recurrent connections as an internal representation that makes their performance more flexible, which is an interesting result.

Despite these issues, it is still interesting to see how NEAT performs in the tasks of the RL Benchmarking Event, and what the tradeoffs are. In order to embed NEAT into the provided reinforcement learning setting, it was necessary to add several features to



(*a*) Best-solution fitness ($\epsilon = 1.0$) (*b*) Average-reward fitness ($\epsilon = 0$)

Figure 1: **NEAT performance in the Blackjack domain.** In all graphs, each network is evaluated in 10 episodes during evolution; in order to draw the graph, each new best network is evaluated separately over 1000 episodes. Both the average reward for the entire population and the score of the best network found so far are shown. Even when exploration is enabled, no significant improvement is found after the first generation, therefore making evolution unnecessary.

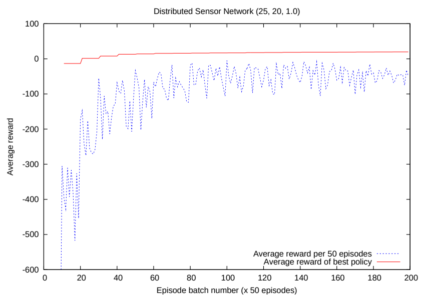
the base system: (1) In order to deal with "noisy" fitness functions in games of chance and domains with random starting conditions, each single individual was evaluated in multiple episodes. (2) In order to compare the best-solution fitness and average-reward fitness, an exploration parameter ϵ was introduced. This parameter allows NEAT to evaluate new policies ϵ fraction of the time, and to evaluate the best policy found so far $1 - \epsilon$ fraction of the time. (3) A modular input and output encoding mechanism is added to translate environment observations into network inputs and to translate network outputs into actions in the environment.

Unless otherwise specified, each network is evaluated 10 times during evolution. In addition, the performance of the best individual in each generation is tested over 1,000 random episodes to get an accurate measure of progress. These evaluations do not affect evolution; they are used simply to draw a graph that accurately measures the expected performance of the current best solution at each point in evolution. The performance plots are averages over 40 randomized runs.

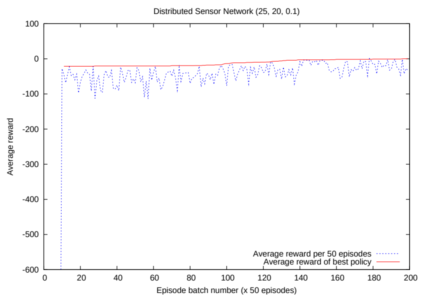
3 Results

3.1 Blackjack

The Blackjack domain is the simplest of the tasks, with optimal performance at about 0, or break-even. This task does not test evolution at all: in an average run, the first generation of 50 random neural networks produces an acceptable solution. In other words, $\epsilon = 0$, i.e. no search at all, generates the best average-reward fitness. Figure 1 shows the average performance of the algorithm.



(*a*) Best-solution fitness ($\epsilon = 1.0$)



(*b*) Average-reward fitness ($\epsilon = 0.1$)

Figure 2: **NEAT performance in the Distributed Sensor Network domain.** Maximizing the average reward leads to suboptimal solutions; the best average reward is achieved when evaluating new networks only 10% of the time.

3.2 Distributed Sensor Network

The Distributed Sensor Network domain is a more complex discrete challenge. Since the high-dimensional actions are represented as a single integer range, each NEAT output neuron serves as a single dimension of the action (or a bit in the base-3 number representing that action). The mid-range output of the neurons is mapped to the 0 action, which is "safe" relative to the other actions. Thus networks are initially more likely to do nothing than to randomly focus and therefore incurring penalties. The population size is 25 and each individual is evaluated 20 times during evolution.

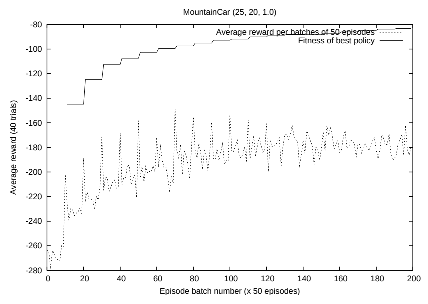
Using the best-solution fitness (Figure *a*), NEAT is able to learn a better policy than when it is set to optimize average reward (Figure *b*). The best average reward performance is achieved with $\epsilon = 0.1$, i.e. when new networks are evaluated only 10% of the time.

3.3 Taxi

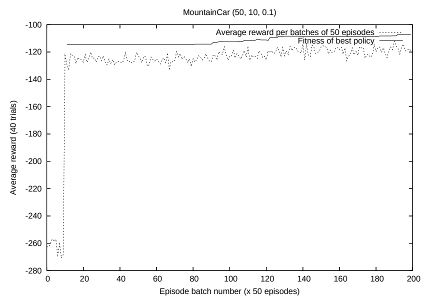
We did not have enough time to work on this domain.

3.4 Mountain Car

By maximizing the best fitness, NEAT is able to find solutions that solve the Mountain Car in an average of 83 time steps (Figure 3*a*). The average fitness on such runs yields an average performance of approximately 192 time steps. With $\epsilon = 0.1$, NEAT's average performance according to the average-reward measure is 120 time steps, however the performance of the best solution found drops to about 100 time steps (Figure 3*b*).

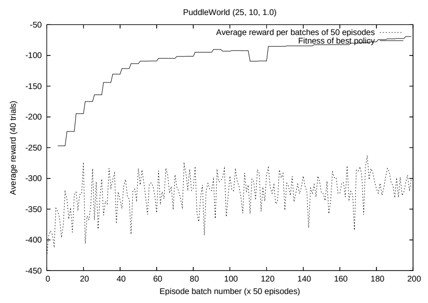


(a) Best-solution fitness ($\epsilon = 1.0$)

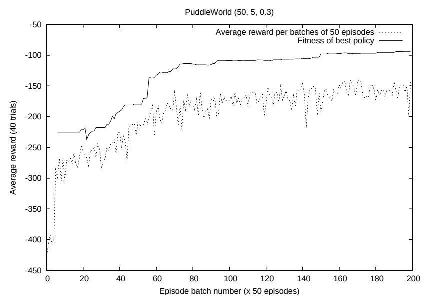


(b) Average-reward fitness ($\epsilon = 0.1$)

Figure 3: **NEAT performance in the Mountain Car domain.** Maximizing the average reward leads to suboptimal solutions; the best average reward is achieved when evaluating new network only 10% of the time.



(a) Best-solution fitness ($\epsilon = 1.0$)



(b) Average-reward fitness ($\epsilon = 0.3$)

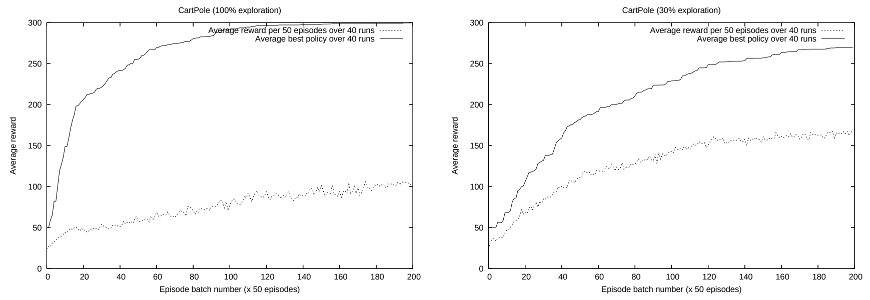
Figure 4: **NEAT performance in the Puddle World domain.** Maximizing the average reward leads to suboptimal solutions; the best average reward is achieved when evaluating new network only 30% of the time.

3.5 Puddle World

Puddle World is difficult for NEAT because the optimal action given two similar input vectors can be wildly different. Nevertheless, maximizing best fitness yields solutions that can find the goal in an average of 69 time steps (Figure 4*a*). The average-reward measure on those runs yields an average time to the goal is 322 steps. With $\epsilon = 0.3$, NEAT's average performance according to the average-reward measure is 197 time steps, but the performance of the best solution found falls to 132 time steps (Figure 4*b*).

3.6 Pole Balancing

The reward structure for this task is not suited to policy-search algorithms like NEAT. NEAT accumulates the rewards received during each step of an episode and only pays attention to the resulting sum. It is essential that this final sum indicates how good the



(*a*) Best-solution fitness ($\epsilon = 1.0$) (*b*) Average-reward fitness ($\epsilon = 0.3$)

Figure 5: **Performance of NEAT in the pole balancing domain with modified reward structure.** As in the other domains, the task is solved quickly when maximizing the best solutions, and maximizing the average reward leads to suboptimal solutions.

policy is; otherwise it is not possible to decide which policies should be discarded and which should be explored further.

Unfortunately, the reward structure for the pole-balancing task does not result in such evaluations: A network that balances the pole for a while can receive the same final score as a network that allows the pole to fall immediately. In fact, it is possible for a network that balances the pole for a short while to actually receive a lower score than a network that drops the pole immediately. For methods like NEAT, such misleading information makes the task impossible to learn.

One way to address this issue is to change the reward structure. For example, when a reward of +1 is given for every non-terminal timestep that elapses, NEAT optimizes the time the pole is balanced. Each episode is limited to a maximum of 300 timesteps, which means that the best score is 300.

When such sensible gradient information is provided, this task is easy for methods like NEAT. By maximizing the best fitness, NEAT is able to find solutions that keep the pole balanced the entire episode, whereas the average-reward fitness is about 100 (Figure 5*a*). With $\epsilon = 0.3$, NEAT's average-reward fitness rises to 170, however the average performance of the best solution found drops to 270 (Figure 5*a*).

References

[1] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154. San Francisco: Kaufmann, 1987.

[2] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Cambridge, MA, 1996. MIT Press.

[3] P. McQuesten. *Cultural Enhancement of Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2002. Technical Report AI-02-295.

[4] N. J. Radcliffe. Genetic set recombination and its application to neural network topology optimization. *Neural computing and applications*, 1(1):67–90, 1993.

[5] K. O. Stanley and R. Miikkulainen. Efficient evolution of neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco: Kaufmann, 2002.

[6] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

[7] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[8] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

LEAP: Learning Entities Adaptive Partitioning

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
{bonarini, lazaric, restelli}@elet.polimi.it
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

Algorithm and Parameters

In the following, we give a brief description of LEAP (Learning Entities Adaptive Partitioning), a new model-free adaptive reinforcement learning algorithm, that we have benchmarked.

In LEAP, the state space is initially decomposed into different partitions, each entrusted to a different Learning Entity (LE) that runs a Q-learning-like learning algorithm. Each partition aggregates states of the original state space into *macrostates* and for each $\langle macrostate, action \rangle$ pair LE computes the Q-value and a measurement of the reliability of such a value. The learning process of LEAP is on-line and does not need any information about the transition model of the environment. The action selection phase is performed by a Learning Mediator (LM) that merges all the action-values learned by LEs and computes the best action in the current state. In all the experiments in the benchmark we adopted a simple mean of Q-values weighted by their variability, in order to strengthen only reliable estimations. During the update phase, each LE compares the expected reward with the target actually received and through a heuristic criterion, the *consistency test*, detects whether the resolution of its own partition in the current state should be increased, in that case the LE is said to be *inconsistent*. When more than one LE is inconsistent, the LM creates a new Joint Learning Entity (JLE), that will operate on a new single macrostate obtained as the intersection of the macrostates of the inconsistent LEs. As a consequence, the basic LE will be deactivated on all the states covered by the more specialized entity. At the same time, an opposite mechanism, the *pruning mechanism*, detects, during the action selection, when a JLE can be removed from the list of LEs. This mechanism simply compares the action proposed by a JLE with the action that would be proposed by the deactivated LEs, and, when those actions are the same, the JLE can be removed.

Through these mechanisms (consistency test and pruning), LEAP builds a multi-resolution state representation that is specialized only where it is needed to learn a near optimal policy.

LEAP needs some parameters to be set depending on the specific problem to be solved. Most of them are typical of on-line algorithms:
Learning Rate: the learning rate used by all the LEs and JLEs during their update phase.
Exploration Factor: the value of ϵ in a simple ϵ-greedy exploration strategy.
Decreasing Rate: the rate used to decrease both learning rate and exploration factor.[1]
Furthermore, LEAP introduces two additional parameters used to put a lower threshold on the minimum exploration of macrostates needed before consistency test and pruning mechanism actually taking place in a macrostate:
MinExplorationC: number of times the least taken action must be executed before the consistency test takes place.
MinExplorationP: number of times the least taken action must be executed before the pruning mechanism takes place.
Both these parameters must be chosen in order to avoid, because of initialization effects, incorrect refining of the state space (i.e. the consistency test) and early pruning of macrostates (i.e. pruning mechanism).

Results

Since LEAP is an adaptive algorithm whose goal is to give compact solutions to complex problems, in all the following results we pursue a tradeoff between optimality and number of macrostates used to compute the solution.

[1]Since aggregation induces additional stochasticity in the environment, even in deterministic problems the learning rate cannot be constant.

TIELT: A Testbed for Learning in Large-Scale Environments

Matthew Molineaux[1,2] and David W. Aha[1]

[1]Intelligent Decision Aids Group; Navy Center for Applied Research in Artificial Intelligence;
Naval Research Laboratory (Code 5515); Washington, DC 20375
[2]ITT Industries; AES Division; Alexandria, VA 22303
{molineaux@aic.nrl.navy.mil, david.aha@nrl.navy.mil}

Motivation

The goal of the Testbed for the Integration and Evaluation of Learning Techniques (TIELT) is to provide machine learning researchers access to large-scale interactive environments (e.g., games). One key avenue of research supported by the system is reinforcement learning. To this end, TIELT provides a model of environments consistent with the reinforcement learning model of an environment and agent which communicate by means of percepts and actions. The following design characteristics have guided the development of TIELT:

1. *Operation*: TIELT should input a model of an environment's state and transitions, a task to be performed, and a description of the inputs required by a learning agent. During a learning episode, TIELT should interpret and translate communications between an agents and an environment, and maintain and record state information. By scheduling a series of episodes as specified by a researcher, it should empirically test the agent using chosen performance metrics.
2. *Agent flexibility*: Agents with many different capabilities and informational needs (e.g., planning, advice-taking, and reinforcement learning agents) should be able to leverage TIELT's capabilities.
3. *Environment flexibility*: TIELT should accommodate a wide variety of game types, including real-time and turn-based, team-based and one-on-one, and fully and partially observable games. Communications should be flexible enough to interoperate with existing environments such as commercial games.
4. *Reuse*: TIELT should permit researchers to study an agent's ability to perform tasks from several games. To this end, TIELT-integrated environments should be useable unchanged with different agents.
5. *Availability*: TIELT should be available for use on all major computing platforms, at no cost.

System Description

TIELT is currently in a late Alpha stage of development. It is freely available for download from http://nrlsat.ittid.com, where we also provide its documentation (e.g., User's Manual and Tutorial), solicit requests for additional functionality, and maintain a bug tracking facility. Written in pure Java, it can be used on all platforms supporting the Java Virtual Machine.

Experiments in TIELT involve a set of user-defined data models that control how episodes are executed:

Environment Model: This encodes a declarative description of an environment, including: a set of *state variables*; a set of *operators* that define the action space of the environment, as well as action preconditions and effects; and a set of *percept models* that describe the space of percepts and in what states they may occur.

Agent Program Interface Description / Simulator Interface Description: These define how TIELT communicates with an agent program and an environment simulator, respectively. That is, they describe communication specifications that TIELT uses to translate messages sent and received by these external systems.

Experiment Structure: This describes an experiment that tests one or more agents in one or more environments. It specifies independent and dependent variables, and when and how to record environment and agent state information for subsequent analysis.

TIELT provides a GUI for creating and editing these data models. By integrating an agent or environment with TIELT, a user gains access to TIELT's libraries of integrated agents and environments. A researcher can, for example, integrate an agent and compare its performance against selected competitor agents.

TIELT has been integrated with a dozen environment simulators, including Full Spectrum Command/Research, (which is used for Army training purposes at Ft. Benning), the MadRTS engine (which is used in the popular *Empire Earth II*™), and Wargus, a *Warcraft II*™ clone implemented using the freeware game engine Stratagus. These three simulators are real-time strategy simulators. It has also been integrated with a few decision systems, including Soar and CaT, a case-based reasoner that learns to win Wargus, as reported in TIELT's first published research investigation (Aha *et al*., 2005). We have improved the functionality of TIELT from our experience with these integrations, and believe it to be a useful tool for experimenting on and comparing learning agents.

References

Aha, D.W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. *Proceedings of the Sixth International Conference on Case-Based Reasoning* (pp. 5-20). Chicago, IL: Springer.

Yahtzee: a Large Stochastic Environment for RL Benchmarks

Andrea Bonarini, Alessandro Lazaric, Marcello Restelli
Department of Electronics and Informatics, Politecnico di Milano
piazza Leonardo da Vinci 32, I-20133 Milan, Italy

Introduction

Yahtzee is a game that is regularly played by more than 100 million people in the world. We propose a simplified version of Yahtzee as a benchmark for RL algorithms. We have already used it for this purpose, and an implementation is available[1].

Description of the game

The player scores points by rolling specific combinations of five 6-sided dice. The game consists of 13 rounds. In each round, the player rolls the 5 dice (up to 3 times) and then scores the roll in one of 13 categories. In the first 6 categories, the goal is to get as many 1s, 2s, 3s, 4s, 5s, 6s as possible, while the other categories are related to special combinations such as: three of a kind, four of a kind, full house, small straight, large straight, chance, and Yahtzee. A Yahtzee is rolled when all five dice show the same face. Each category can be scored only once. After 3rd roll, the player must choose his score; if he cannot enter a score, he must select a zero. If the sum of points scored in the first 6 categories is greater than 62, the player earns a bonus of 35 points. The object of the game is to maximize the total score. The game ends once all 13 categories have been scored[2].

In the simplified version of Yahtzee we propose as a benchmark, the player cannot choose the dice to re-roll, while, in the original version, the player must throw all the dice on its first turn of each round, but he can select the dice to re-roll on its second and third turns. Therefore, the action space is reduced to 14 possible actions: the learner can choose either to fill one of the 13 categories with the obtained result, or reroll the dice. Furthermore, we have removed the possibility to obtain the mentioned bonus.

We propose a representation consisting of 14 variables. One is related to the number of rolls performed in the current round; the other 13 are associated to the 13 categories, and each one contains the score that the player would get by filling it with the current dice values. If the category has been already used, the corresponding position is set to -1. The size of this state space is about $0,51 \cdot 10^{11}$.

Given that in Yahtzee the goal is to maximize the sum of the points scored in the thirteen categories, we propose the following reward function: the reward is equal to -100 when the player wants to fill a category which has been already filled, or the dice are rerolled for more than three times; each time a category is filled, the learner is rewarded with the related score. On rerolling the reward is 0.

Motivations

Yahtzee is a game with very simple rules, and this has determined its great popularity. Among the reasons that make Yahtzee a good benchmark for RL algortihms let us mention the following. The state space is large. The transition model is highly stochastic, but the reward function is deterministic. The game does not require any opponent, while other games, e.g., Backgammon, need it: the learning performance depends on how the opponent is implemented and this makes benchmarking difficult. The state space size can be reduced or enlarged by removing or adding categories; for instance, it would be possible to define a reduced version of the problem with only the first six categories in order to face it, without using function approximation, with standard tabular approaches, since the number of states reduces to about $0,35 \cdot 10^6$. Finally, it is easy to produce simple hand-coded policies that achieve good performances, while it is very challenging to find the optimal policy, even if it is known: recently, Woodward [1] has solved the game computing all of the more than 1 trillion possible outcomes and working out optimal playing strategies.

[1] http://www.elet.polimi.it/upload/restelli/yahtzee.html

[2] For more details on the game of Yahtzee you can refer to the official Yahtzee rules available at: http://www.hasbro.com/common/instruct/Yahtzee.pdf

References

[1] Phil Woodward. Yahtzee: The solution. *Chance*, (1):18–22, 2003.

Scaling Up Towards Real-World Tasks

Risto Miikkulainen
Department of Computer Sciences
The University of Texas at Austin

I propose that the Reinforcement Learning benchmarking event be expanded to include domains that are (1) challenging for even the strongest methods in the foreseeable future, (2) resemble real-world tasks.

The simple tasks included so far do not need to be replaced: such domains can be useful in understanding how and why certain algorithms work, thereby gaining insight that allows improving them in the future. However, they are not useful in measuring how strong the current algorithms are, and results in them may even be misleading in that respect. The algorithms that are designed to work on challenging real-world problems may not have a chance to show their true performance when the task can be solved so easily. For example, in population-based methods such as evolution of neural networks, a solution to the pole-balancing task is often found in the initial population—before the algorith has had a chance to do any search at all! Similarly, in methods that construct a model to guide the search, the tasks can be solved before the model has had a chance to have an affect.

Several domains already exist that fit the challenge. Some of these domains are discrete, others continuous; some are Markovian, others not; some change, others are static. Common to all of them are that they are difficult, scalable, and resemble interesting tasks in the real world. For example:

Balancing two poles simultaneously on one cart. This tasks is significantly harder than balancing a single pole because of the nonlinear interactions between the poles. The version with velocities is not particularly hard, but if the velocities are not given to the learner, the task becomes quite challenging. Moreover, the task can be made arbitrarily difficult by making the tasks closer in length, which allows for scaling the task for the foreseeable future, but also allows learning to utilize easier tasks at first and gradually scaling to harder versions.

Robot auto racing (RARS). RARS is an internet community built around a auto racing simulator that holds races regularly over the internet. The simulator is very good and allows realistic races to be run. Most of the races involve hand-coded drivers, but the simulator can be customized to provide an environment for learning. The cars and tracks can be designed to fit the competition and can be made gradually harder. Initially, the competition can consist of simply fast lap times without other cars, but can eventually include driving strategy as well.

Keepaway soccer. Robotic soccer keepaway task, where e.g. a team of three players keep the ball away from two takers) is challenging because it requires different behaviors at different times. In the beginning, all agents can be constrained to be identical; in the future, teams of agents can be trained together, and the game can also be scaled up to full robotic soccer. The task can be implemented with existing robotic soccer (Robocup) simulators.

Packet routing. Although this domain is relatively easy to solve in the static case, changing network topologies, load levels, and traffic patterns can make it challenging: The learner has to adapt to such changes while maintaining effective routing performance. Similar issues exist in complex computing systems in general.

Othello. At CEC-2006, there is a competition for learning the most effective board evaluation function; this domain would be great for reinforcement learning in more general. It is a well defined real-world task where success is measured in a true competition. Good opponents and simulators exist against which learning can proceed.

Go. Unlike in Othello, the best programs do not compete well with humans in go; any success in this domain would therefore be significant result on its own. It is also different from Othello in that learning may have to take place by playing the learner against itself. It is possible to start with small board competitions (e.g. 7x7) and gradually scale up as the methods get better.

Competitions in such domains are be likely to have two effects: (1) they encourage developing methods that scale up to real-world tasks, and (2) they draw attention to the field and generate excitement about reinforcement learning in the broader community. Such domains can grow with the successes, becoming gradually harder and more realistic. In doing so, they may serve as a catalyst similar to Robocup in robotics, contributing to scaling up reinforcement learning from laboratory-scale experiments to the complex applications in the real world.

Time-dependent control and the role of noise

Bert Kappen
SNN, Radboud University, Nijmegen, The Netherlands
b.kappen@science.ru.nl

Reinforcement learning can be viewed as a way to solve a stochastic control problem. Typically, the control problem considered is time-independent and aims to find a control that specifies for each state of the system what action to take. In contrast, a time-dependent control problem aims to find an optimal control that specifies for each time in the future what the best action is. Examples of time dependent control problems are for instance steering a car or throwing a dart.

In either case, the amount of noise in the problem can affect the optimal policy or control substantially. The case is illustrated the left figure below, where the optimal course of action of a spider depends on how well she is able to control her actions. When sober, the best path is over the bridge, but if she is drunk it is more prudent to walk around the lake.

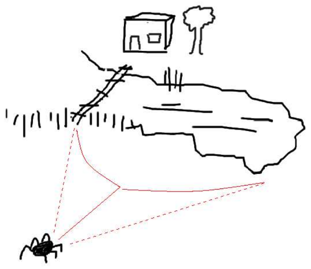


Figure 1: The drunken spider. A spider wants to go home, by either crossing a bridge or by going around the lake. In the absence of noise, the route over the bridge is optimal since it is shorter. However, the spider just came out of the local bar, where she had been drinking heavily with her friends. She is not quite sure about the outcome of her actions: any of its movements may be accompanied by a random sway to the left or right. Since the bridge is rather narrow, and spiders don't like swimming, the optimal trajectory is now to walk around the lake.

In addition to which path to chose, the spider also has the problem *when* to make that decision. Far away from the lake, she is in no position to chose for the bridge or the detour, as she is still uncertain where her random swaying may bring her. In other words, why would she spend control effort now to move left or right when there is a 50 % change that she may wander there by chance? She decides to delay her choice until she is closer to the lake. The question is, when should she make her decision to move left or right?

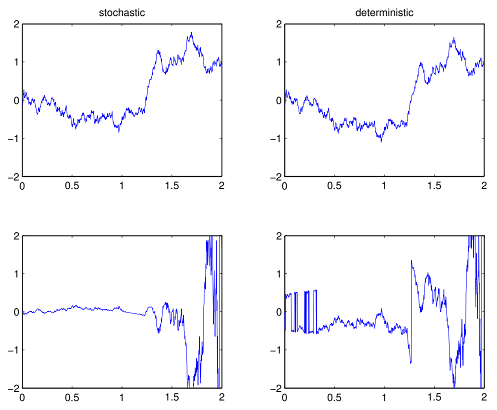


Figure 2: The delayed choice. The task is to move to one of the targets at 1 or -1. When the target is far in the future, the optimal policy is to steer between the targets. Only relatively late, should one aim for one of the targets. The optimal moment depends on the noise. Sample trajectories versus time (top row) and controls versus time (bottom row) under stochastic control (left column) and deterministic control (right column), using identical initial conditions and noise realization. Note, how the optimal stochastic control is practically zero for $0 < t < 1$ whereas deterministic control is too large [1, 2].

It is in these multi-modal examples, that the difference between deterministic and stochastic control becomes most apparent. They are not only of concern to spiders, but occur quite general in obstacle avoidance for autonomous systems and multi-agent systems. In this contribution, I will discuss how noise affects optimal time-dependent control and illustrate the concepts in a number of examples (darts and multi-agents). On the basis of these examples and discussion one or more benchmarks could be defined.

References

[1] H.J. Kappen. A linear theory for control of non-linear stochastic systems. *Physical Review Letters*, 2005. In press.

[2] H.J. Kappen. Path integrals and symmetry breaking for optimal control theory. *Journal of statistical mechanics: theory and Experiment*, 2005. In press.

Exploiting Domain Knowledge in Reinforcement Learning

Pascal Poupart
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
`ppoupart@cs.uwaterloo.ca`

Nikos Vlassis
Informatics Institute
University of Amsterdam
Amsterdam, The Netherlands
`vlassis@science.uva.nl`

Abstract

We argue that exploiting domain knowledge in reinforcement learning provides an easy, accurate and principled way of designing scalable algorithms for online learning in real-world problems. Instead of negatively viewing the exploitation of domain knowledge as "overfitting" or "cheating", bakeoffs and benchmarks should permit and even facilitate the exploitation of domain knowledge.

Reinforcement learning (RL) was originally proposed as a framework to allow agents to learn in an *online* fashion as they interact with their environment. Existing RL algorithms come short of achieving this goal because the amount of exploration is often prohibitive and/or too costly for online learning. As a result, RL is mostly used for *offline* learning in simulated environments.

Common causes of the prohibitively large and costly exploration include a poor balance of the exploration/exploitation tradeoff and a complete ignorance of the transition dynamics. Focusing on the latter cause, complete ignorance of the dynamics implies that a large number of parameters are unknown (e.g., $|A||S|^2$ parameters corresponding to $\Pr(s'|s,a)$ in discrete problems). Whether we take a model-based or model-free approach, a large number of unknown parameters is indicative of the large amount of exploration that may be required to learn a reasonable policy.

In the interest of developing general domain independent algorithms, the RL community traditionally discourages the exploitation of domain knowledge to reduce the number of unknown parameters. Instead, much research has focused on approximating the optimal value function using various function approximators. It is interesting to note that the optimal value function is something rather difficult to characterize, hence approximation heuristics often tend to be poorly motivated and inaccurate. In contrast, it is much easier and accurate to directly characterize what is known and unknown about the transition dynamics and/or the optimal policy. For instance, in mobile robotics, it is easy to realize that a single noise model (independent of the x, y, θ-pose) is sufficient for the robot to know the effects of its controls.

Online reinforcement learning is generally difficult, especially when the transition dynamics are completely unknown. A promising research direction consists of the design of algorithms capable of exploiting domain-specific structural properties. To that effect, it is quite instructive to exploit domain knowledge. Here domain knowledge may be used to constrain the uncertainty about the transition dynamics or even to directly constrain the space of policies. The advantages of using domain knowledge are that it is often readily available and it offers a principled and accurate way of reducing the amount of exploration required. In the long run, the development of general domain independent algorithms remains desirable. To that effect, domain-specific algorithms should be generalized to automatically detect and exploit structural properties arising in some domains.

In the light of the above discussion, RL bakeoffs and benchmarks should encourage the exploitation of domain knowledge as this will help the design of principled and accurate approaches capable of exploiting structural properties when they arise in a problem. Concretely, the exploitation of domain knowledge could be encouraged and facilitated by specifying any known structural properties for each problem. This could be done in words or by providing alternate encodings reflecting those structural properties.

Case study: control of a real world system in CLSquare

Roland Hafner
Neuroinformatics Group
University of Osnabrück
Germany
Roland.Hafner@uos.de

Martin Riedmiller
Neuroinformatics Group
University of Osnabrück
Germany
Martin.Riedmiller@uos.de

1 Abstract

CLSquare (CLS^2 , Closed Loop System Simulator) is a software package for benchmarking in reinforcement learning problems (amongst others) that is under development in our group for several years now.

The base concept of CLS^2 is to provide a modular software package for integrating various plants and controllers that can interact in a standardized framework. This framework provides base modules for graphical visualization, testing between the learning process and statistic analyzing of the performance of the controller. The integration of own modules for plants and controllers is straight forward and the migration from and to CLS^2 with other software packages is supported by several interfaces. To fit your special demands you can plug in your own modules for test procedures, statistics and visualization without leaving the standardized framework. The software framework itself and every module can be parameterized through configuration files. This allows a rapid setup and modification of new experiments.

In contrast to other software packages CLS^2 takes the control or system theory point of view on the problem of modeling the interfaces of the software modules. The controller interacts with the plant in a servo loop manner. A plant is defined as a module that gets an action respectively a control variable as input and provides a new state, as an scalar dependent on the actual state, as output. The input for the controller is the actual observation or state produced by the plant. One benefit of this interface structure is that researchers can exchange controllers or plants to compare their work without taking care for the special method or modeling of the control problem itself.

In our current work we are focusing on reinforcement learning algorithms for online learning of control of multi-variable real systems. As an example, a reinforcement controller learns how to control the velocity of a single dc motor to an arbitrary changing set point by interaction with the real system. Another example of a more complex system is the velocity control of an omnidirectional mobile robot with three dc motors. We use the CLS^2 system for both experiments. For the single dc motor the interface to the real motor is implemented directly in CLS^2 , for the mobile robot an TCP interface to an complex robot control program exist.

We will present the integration of a real standard dc motor as a plant and the integration of a reinforcement controller for online learning in CLS^2 as a case study. In this case study we will concentrate on the special demands the integration of a real world system makes. As a consequence we show how the interfaces and the structures of CLS^2 are build to meet this demands. The work-flow and the possibilities of CLS^2 as a modular and flexible system for benchmarking in reinforcement learning will be discussed on the example of an online reinforcement learning controller for the velocity control of the motor. Furthermore we show that it is possible to compare divers, even classical, learning and control techniques in the same framework. This is done by just plugging in a PID controller as replacement for the reinforcement controller and comparing the performance in the standardized framework.

PIQLE & RL-Benchmarks : the continuous benchmarks

Francesco De Comité

November 24, 2005

1 Introduction

To connect PIQLE with the RL-Framework, we just had to define a new environment, consisting of three JAVA classes :

- ActionRL : allowing to describe the n possible actions.
- EtatRL : to store the p float components of a state.
- ContraintesRL : a vanilla-like environment used to connect the RL-Framework provided environment with the learning algorithms.

Those three classes are problem-independent : for the learning algorithms, a state is just an array of float, an action is just an integer.
We also had to define a new kind of `Agent`, a little bit different from our conception of what must be an `Agent` in the PIQLE framework. This new kind of `Agent` implements the methods required by the RL-Framework.

2 Settings

2.1 The problem settings

An important part of the work in Reinforcement Learning consists in storing 3-uples (state,action,value). In most cases, PIQLE stores those 3-uple in hash maps, where the couple (state,action) plays the role of the key. While in the RL-Framework actions are discrete, the only problem is to discretize the description of states. We have chosen to :

1. Associate an hash code to each state, by rounding its values, and then apply an hash function on those integer values.
2. Define that two states are equal if, in none of their dimensions, the ratio of the difference between their values and the maximum possible difference of two values for this dimension, exceeds 40% .

Definition 1 *Two states are equal if, in none of their dimensions, the ratio of the difference between their values and the maximum possible difference of two values for this dimension, exceeds 40%.*
Otherwise stated :
Two states $\vec{X} = (x_0, \cdots, x_n), \vec{Y} = (y_0, \cdots, y_n)$ are equal for PIQLE iff :

$$\forall i \in \{0, n\}, abs(x_i - y_i) < 0.4 \times (i_{max} - i_{min})$$

Where i_{max} and i_{min} respectively represent the upper and lower bound for the ith dimension value of a state. All those informations are available from the description of the task.

This kind of *distance* between states seems to give good results, while not very difficult to compute. It has some relation with the concept of tiling, which is not still implemented in PIQLE.
The basic setting (0.4) may be tuned for each problem. If such a modification is used, it will be indicated into the associated file.
We used version 1.0.1 of `NIPSBenchmarks`. As we began to be used to our *modus operandi*, we thought it would be time-wasting to rethink the files and directories organization to cast it with the 1.0.2 version.

3 Results

Settings, tested algorithms and results for each continuous problem are described in the corresponding files.
At this time, we do not have results from other algorithms, so we do not know how good (or bad) our results are. But at first glance, we can say that, at least, our algorithms do learn, considering that the reward curve is always far above the random behaviour reward curve.

4 General remarks

- The RL-Framework method `agent_end` should perhaps also tell which was the last reached state. Otherwise, we had to define a fake final state in order for our algorithms to learn.
- The present version of RL-Framework allows RL-Framework environments to be interfaced with PIQLE. We believe that a short work can be done in order to make the interface works in the other way, that is to say interface our PIQLE environments with other RL algorithms.
- Time lakes us to deepen the study : PIQLE's connection with Weka may help to generalize and describe what the algorithms have learned, by applying machine-learning techniques to $(s, a, Q(s, a))$ tables : PIQLE contains methods that can extract different kinds of `arff` files from the stored $(s, a, Q(s, a))$ 3-uples.

RL-Framework: NIPS RLBB and Beyond

Adam M. White and Richard S. Sutton
Department of Computing Science
University of Alberta
Edmonton, AB, Canada T6G 2E8
{awhite,sutton}@cs.ualberta.ca

In reinforcement learning (RL) an agent interacts with a running environment program by selecting actions in various states to maximize the sum of scalar rewards given to it by the environment. This dynamic interaction immediately illustrates the inherent difficulty of establishing a set of problems against which researchers can compare their learning algorithms. RL agents must be tested on environments encoded as interactive programs. RL problems cannot be encoded as traditional numerical benchmarks. Empirical analysis in RL requires passing around dynamic environment software instead of data sets, introducing a number of compatibility and standardization issues.

We have developed a standard protocol, which logically separates the agent and environment into two disjoint entities. If every agent and environment interacts with the interface in the same way, then the format of each can be standardized, and we can establish a platform on which a benchmarking system for collecting and publishing empirical results in RL can be built. Using our model we define the agent, environment, interface and benchmark as distinct entities. Our protocol standardizes communication by allowing only the interface to interact with the agent and environment, allowing user to interact by specifying a learning experiment through the benchmark. The communication standard is realized through a set of agent and environment functions that must be implemented to facilitate interaction with the interface.

We have developed a mature version of the interface called RL-Framework.The current version of the interface software supports direct function call communication between agents and environments written in C. The interface software also provides a standard mechanism for multi-language communication between agents and environments written in different programming languages commonly used in RL. For example, using pipe communication an agent written in C can be tested on an environment written in Java on the same benchmark that would be used with direct language communication. RL-Framework provides a transparent benchmark system, where the mode of communication does not effect the experimental setup, execution sequence or results.

We will present the interface specification and highlight additional features and functionality used to increase the flexibility and future scalability of our communication specification. We will also detail the software framework and its major functionality, while giving intuition into the various design choices and intended usage methodologies.