

Course:CPSC522/Reinforcement Learning with Function approximation

From UBC Wiki

< Course:CPSC522

Contents

- 1 Reinforcement Learning with Function Approximation
 - 1.1 Abstract
 - 1.1.1 Builds on
 - 1.1.2 More general than
 - 1.2 Content
 - 1.2.1 Elements of RL
 - 1.2.1.1 Policy
 - 1.2.1.2 Reward Function
 - 1.2.1.3 Value Function
 - 1.2.1.4 Model(optional)
 - 1.2.2 Example
 - 1.2.2.1 Tic-Tac-Toe
 - 1.2.3 Motivation (Look-up Table Versus Function Approximation)
 - 1.2.4 Function Approximation
 - 1.2.5 Bellman Equation
 - 1.2.5.1 Convergence Rate
 - 1.2.6 Q-Learning
 - 1.2.7 SARSA
 - 1.2.8 Generalization
 - 1.2.9 Generalization with Multi-Layer Perceptrons (Neural Network)
 - 1.2.9.1 Online training
 - 1.2.9.2 Offline training
 - 1.2.10 When RL Fails
 - 1.3 Annotated Bibliography
 - 1.4 To Add

Reinforcement Learning with Function Approximation

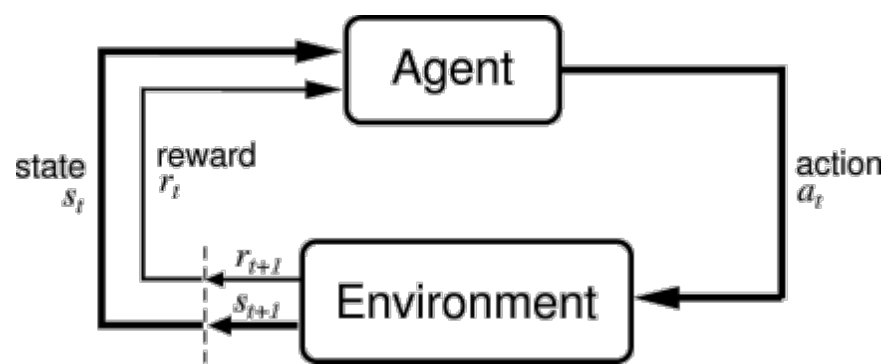
Reinforcement Learning is an area of machine learning (https://en.wikipedia.org/wiki/Machine_learning) inspired by behaviorist psychology (<https://en.wikipedia.org/wiki/Behaviorism>), concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. For smaller problems machine learners use a tabular representation of the data called a Look-up Table (LUT) in order to make the decisions leading to a maximal reward. Here we are introducing a new method called Reinforcement Learning (https://en.wikipedia.org/wiki/Reinforcement_learning) with Function Approximation (https://en.wikipedia.org/wiki/Function_approximation) to overcome issues with LUTs such as limited memory and how they are impractical when dealing with huge data.^[1]

Principal Author: Mehrdad Ghomi

Collaborators: Jiahong Chen, Jordon Johnson

Abstract

If we assume that our estimates of value functions are represented as a table with one entry for each state or for each state-action pair, then we have a particularly clear and instructive case; but it is limited to tasks with small numbers of states and actions. The problem is not just the memory needed for large tables, but the time and data needed to fill them accurately. In other words, the key issue is that of *generalization*. This is a severe problem. In many tasks to which we would like to apply reinforcement learning, most states encountered will never have been experienced exactly before. This will almost always be the case when the state or action spaces include continuous variables or complex sensations, such as a visual image. The only way to learn anything at all on these tasks is to generalize from previously experienced states to ones that have never been seen. Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To a large extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called function approximation (https://en.wikipedia.org/wiki/Function_approximation) because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. ^[1]



Builds on

Function approximation (https://en.wikipedia.org/wiki/Function_approximation) is an instance of supervised learning (https://en.wikipedia.org/wiki/Supervised_learning) (a primary topic studied in machine learning (https://en.wikipedia.org/wiki/Machine_learning)), artificial neural networks, pattern recognition, and statistical curve fitting. In principle, any of the methods studied in these fields can be used in reinforcement learning.

More general than

Function approximation is used to replace inefficient and impractical Look-up Tables. We can also use neural networks (http://wiki.ubc.ca/Course:CPSC522/Neural_Network) to replace the LUTs and approximate the Q-function in Q-Learning (<https://en.wikipedia.org/wiki/Q-learning>), a model-free reinforcement learning technique in which optimal action-selection of a policy for any given finite Markov Decision Process (MDP) (http://wiki.ubc.ca/Course:CPSC522/Markov_Decision_Process) will be learned. Later on we will develop a method called Reinforcement Learning with Back Propagation (<https://en.wikipedia.org/wiki/Backpropagation>), which will build on RL with Function Approximation.^[2] For a very cool applet for playing and working with Q-Learning click here (<http://www.applied-mathematics.net/qlearning/qlearning.html#robot>)^[6]

Content

There are traditionally two different learning paradigms: unsupervised learning and supervised learning. *Reinforcement Learning* (RL) can be introduced as a third learning paradigm. RL is characterized as learning that takes place via interaction of a learning agent with its environment. Like supervised learning, a feedback signal is required; however, for RL this feedback signal represents a reward, not a desired value. RL is therefore not supervised learning. The purpose of RL is to learn which actions yield the greatest reward.^[2]

Elements of RL

Policy

In RL, the policy (denoted by the symbol π) defines a learning agent's way of behaving at any given time. It is a function of the rules of the environment, the current learned state and how the agent wishes to select future states or actions. It provides a mapping from perceived states to actions to be taken. An optimal policy is one which accumulates the greatest rewards over the specified term. The policy is in effect realised by the value function that is being learned using RL and changes as the value function is being learned. You may come across the terms “on-policy” & “off-policy”. The difference is simply that on-policy is both learning and using (to control actions) the same policy that is being learned. “Off-policy” methods learn one policy, while using another to decide what actions to take while learning.^[2]

Reward Function

A reward function defines the goal. Given any state of the environment, the reward function identifies how good or bad this state (or a given action while in this state) is.^[2]

Value Function

The difference between a value function and a reward function is subtle. Whereas the reward function measures the reward associated with the immediate state of the system, a value function represents a prediction of reward at some future time, based on the current state. It takes into account all future rewards that may ensue as a result of reaching the current state.^[2]

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

$$V(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{t_n} r_T$$

Model(optional)

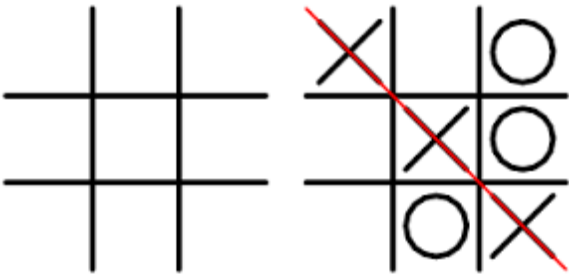
In RL, a model is typically the set of probability functions that determine how an agent would behave in any given state of the system. While it is a requirement for some types of RL (e.g. Dynamic Programming), it is not necessary for the type of learning that will be addressed in this course.^[2]

Example

Tic-Tac-Toe

Tic-Tac-Toe (<https://en.wikipedia.org/wiki/Tic-tac-toe>) or “Noughts-and-Crosses” is a simple game played on a 3x3 grid. The objective is to

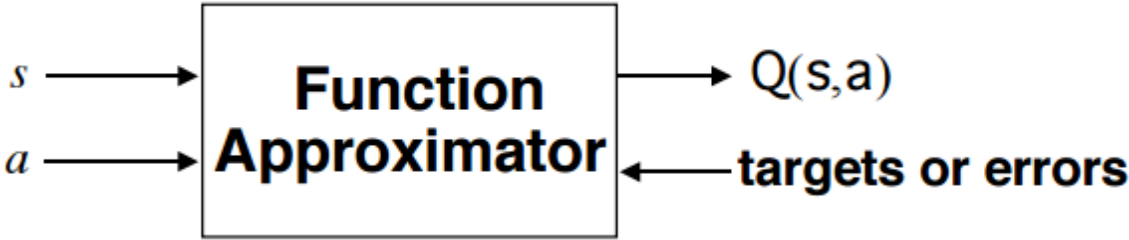
be the first to make a line of crosses or noughts across the grid. The environment in this case, is the board. The reward function is simple, but can only be defined for the final state to indicate a win, a loss or a draw. Given that the environment of all possible states is small and reasonably manageable, the value function can be implemented as a table indexed by all possible states. (One may even wish to take advantage of the symmetrical nature of the board and reduce the number of actual states referenced in this table). The policy takes into account the rules of the game and its choices are dictated by all possible states that it can select given its current state. The effect of RL is to learn a value function that can reliably predict if a state (the next move) will result in a win or not. If a loss and draw are considered to be equal, then the terminal states in the table that represents the value function can be set to 1 or 0. All others would be initialized to 0.5 (i.e. probability of a win is known to be no better than chance).Tic-tac-toe has a small finite number of states and the value function can be implemented as a look-up table. In the above figure, each separate board position has an associated probability. As play proceeds, the computer will perform a search of all possible moves and select that move for which the value function indicates the greatest likelihood of winning. Given that the value function will return 0.5 for all initial states, the earliest games in the learning process will be unlikely to win against an intelligent player. Most moves will be greedy moves, i.e. selection of a move that has the highest value. However it is useful at times to select moves randomly. These moves, although not necessarily the best, are termed exploratory moves as they help to experience states that would otherwise not be encountered. During play, the learning process updates the entries in the table, attempting to improve their accuracy. Over time and after playing many games, the value function will return values that represent the true probabilities of winning from each state.^[2]



Motivation (Look-up Table Versus Function Approximation)

Although current Reinforcement Learning methods have shown some good successes in some application domains, we still need to address several important theoretical issues before having a fully competent and versatile learning method. One important challenge is dealing with problems with large or continuous state/action spaces. In these problems, the look-up table approach is not feasible anymore since there is no way to enumerate all values (there are an infinite number of them). Function approximation (FA) techniques should come at rescue, but our understanding of the interplay between function approximation and RL methods is limited in a number of important ways. In the early stages of RL research, there were many examples and counter-examples that show the feasibility or the infeasibility of using FA in RL. Gradually, the theory started to develop, and the conditions and situations that FA can be used in a RL setting were revealed. However, this problem has not been solved completely and researchers still need to work on it, to create methods that work more efficiently for a larger class of problems. ^[9]

Function Approximation



Optimal and Approximate Value Function When we start learning, the value function is typically randomly initialized:

$$V(s_t)$$

It will be a poor approximation to what we actually want, which is the optimal value function denoted as follows:

$$V^*(s_t)$$

This is the value function that will lead to us making the best decisions. By best, we mean the best or most reward possible. Thus the approximate value at some state s_t is equal to the true value at that state, plus some error in the approximation. We could write this as follows:

$$V(s_t) = e(s_t) + V^*(s_t) \text{ [2]}$$

One thing that needs to be clear and understood is that when using approximation, convergence is no longer guaranteed. The reason is that using the neural Network at each iteration of training we approximate the value-function by changing the weights of the network. In this case we have infinite number of state-action members for which we don't know exactly the Q-values. We just use weights and biases of the network, which are obtained by training the network using a finite number of discrete (quantized) state-action spaces from the LUT, to approximate the Q-values.

Bellman Equation

Richard Bellman (https://en.wikipedia.org/wiki/Richard_E._Bellman) (1957) defined the relationship between a current state and its successor state. It is often applied to optimization problems that fall in a branch of mathematics known as dynamic programming (https://en.wikipedia.org/wiki/Dynamic_programming).

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

The equation is recursive in nature and suggests that the values of successive states are related. In RL, the Bellman equation computes the total expected reward. Typically we want to follow a sequence of events (a policy) that generate the maximum rewards (the optimal policy). Here we introduce two new terms. The immediate reward r and the discount factor γ . ^[2]

Convergence Rate

Let the error in the value function at any given state be represented by:

$$e(s_t)$$

Then:

$$V(s_t) = e(s_t) + V^*(s_t)$$

and:

$$V(s_{t+1}) = e(s_{t+1}) + V^*(s_{t+1})$$

Using the Bellman equation:

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

we can derive that:

$$e(s_t) = \gamma e(s_{t+1})$$

Concluding that the error in successive states is related. But we know that already, since we know that the future rewards $V(s)$ are themselves related. However, the interesting thing is that we can show that these rewards actually converge. I.e. after enough training no longer causes them to change. For a decision process where a reward is available at a terminal state, we know that the reward is known precisely. I.e. there is no error in the reinforcement signal. Given:

$$e(s_t) = \gamma e(s_{t+1})$$

we deduce that with enough sweeps through the state space, I.e. enough opportunities to learn, we will eventually reach the condition where $e(s)$ is zero for all t . When this is the case, then we know we have the optimal value function.

$$V(s_t) = e(s_t) + V^*(s_t)$$

For the optimal value function, we can say that we have state values that satisfy the Bellman equation, for all t : [2]

$$V(s_t) = r + \gamma V(s_{t+1})$$

Q-Learning

Q-learning (<https://en.wikipedia.org/wiki/Q-learning>) is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Additionally, Q-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations. It has been proven that for any finite MDP, Q-learning eventually finds an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable.

```
Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$ 
      (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ 
  until  $s$  is terminal
```

SARSA

The Sarsa algorithm is an On-Policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The name Sarsa actually comes from the fact that the updates are done using the quintuple $Q(s, a, r, s', a')$. Where: s, a are the original state and action, r is the reward observed in the following state and s', a' are the new state-action pair. The procedural form of Sarsa algorithm is comparable to that of Q-Learning:

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Choose  $a$  from  $s$  using policy derived from  $Q$ 
    (e.g.,  $\epsilon$ -greedy)
  Repeat (for each step of episode):
    Take action  $a$ , observe  $r, s'$ 
    Choose  $a'$  from  $s'$  using policy derived from  $Q$ 
      (e.g.,  $\epsilon$ -greedy)
     $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'; a \leftarrow a';$ 
  until  $s$  is terminal

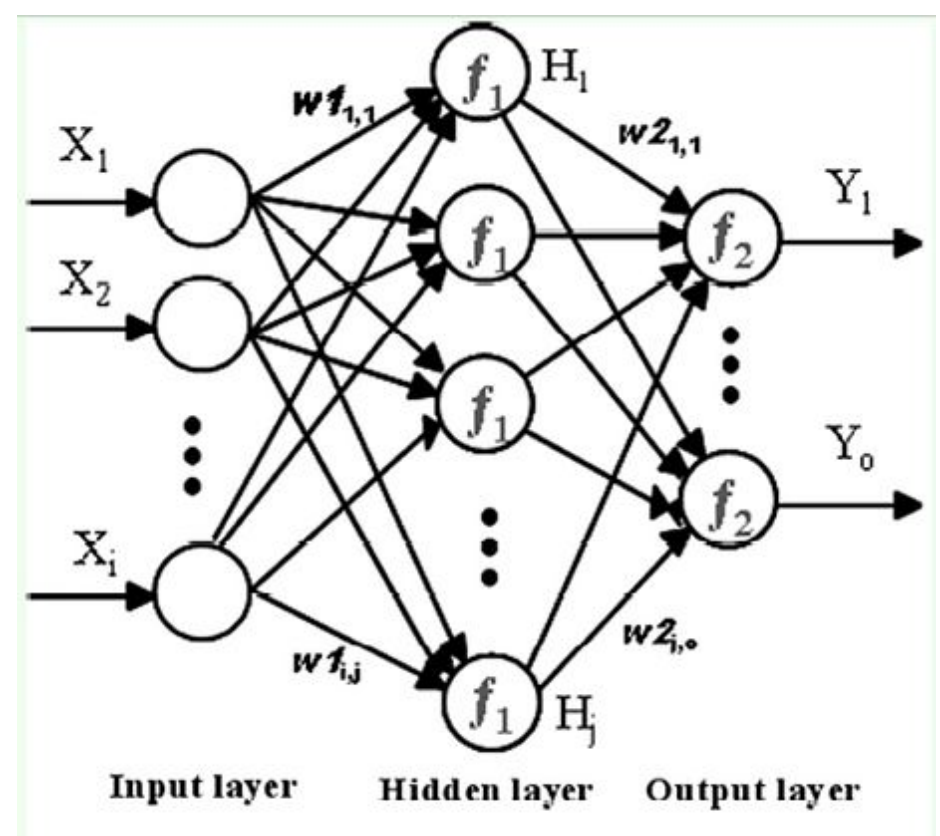
```

Generalization

Reinforcement learning systems must be capable of generalization if they are to be applicable to artificial intelligence or to large engineering applications. To achieve this, any of a broad range of existing methods for supervised-learning function approximation can be used simply by treating each backup as a training example. Gradient-descent methods, in particular, allow a natural extension to function approximation of all the techniques developed in other related areas, including eligibility traces. Linear gradient-descent methods are particularly appealing theoretically and work well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. Linear methods include radial basis functions, tile coding, and Kanerva coding. Backpropagation methods for multilayer neural networks are methods for nonlinear gradient-descent function approximation.^[1]

Generalization with Multi-Layer Perceptrons (Neural Network)

For instructional purposes, the implementation of RL is by far the easiest using look-up tables. Each value or Q-value (<https://en.wikipedia.org/wiki/Q-learning>) is indexed by the states of the problem. Initially randomly initialized, as each state is visited, and gets updated. However, as the number of states increases so does the size of the look up table and the number of computations to adequately populate it. In practice, with a world with many dimensions, the number of states will be large. 1020 states for a Back-Gammon game (<https://en.wikipedia.org/wiki/Backgammon>), not only is memory a concern, but more so generalization. A sparsely populated table is unlikely to be effective. The answer is to generalize the value functions. There are many approaches including the application of non-linear approximators such as mutli-layer perceptrons (<https://en.wikipedia.org/wiki/Perceptron>). However, the area is still one of active research. In fact the application of neural nets (https://en.wikipedia.org/wiki/Artificial_neural_network), although promising and shown to be highly successful in Back-Gammon is regarded as a delicate art! Two approaches to the practical assignment are suggested. Both require the use of a neural network (multi-layer perceptron (https://en.wikipedia.org/wiki/Multilayer_perceptron)) trained using the Backpropagation (<https://en.wikipedia.org/wiki/Backpropagation>) algorithm. Note there are issues with both approaches suggested below: ^[2,3]



Online training

The Q-function is implemented as an MLP. As Temporal Difference(TD) (https://en.wikipedia.org/wiki/Temporal_difference_learning) updates are backed up, the supervised values for the Backpropagation training targets are generated according to TD as applied to the Q-function. Thus the MLP is undergoing training as it is being used.

Offline training

The Q-function is implemented as a look-up table during training. As TD updates are backed up, the table is updated. Upon completion of RL training, the contents of the look-up table are then used to train a neural net. Once trained, the neural net is used to replace the look-up table in the RL agent. We then observe improvements in the agent's behavior now using a generalized Q-function. For playing and working with a very useful tool for Neural Networks click here (<http://aispace.org/neural/>).^[4] For playing and working with a very useful tool for Multi-Layer Perceptron click here (<http://www.sund.de/netze/applets/BPN/bpn2/ochre.html>).^[5] For Java implementation of different types of Neural Nets click here (<http://neuroph.sourceforge.net/documentation.html>)^[7]

When RL Fails

While RL and in particular temporal difference learning, seems generally suitable for learning many varieties of control problems, in order to do so, a learning agent requires interaction with its environment. In certain cases this might not be desirable or possible. For example, learning to fly an airplane. You would not let an untrained agent take control of a real aircraft. Either some software would

be necessary to restrict the agents actions to “safe” actions or possibly, to train it first using a simulator. (An approach which works well with real pilots).
How about applying RL in the health/medical realm? RL might be able to provide personalized care by learning to adjust medications on an individual basis. For example in anesthesiology, where the delivery of anesthetic is carefully governed based upon monitoring a patients vital signs. However again, the application of RL is unacceptable because initially, the agent will require exploration before it can learn to offer optimal dosing. Unlike training a pilot, in this case there is no simulator available for pre-training. With regards to this, one solution from the literature is to adopt an approach which attempts to off-line learn from collected data.^[8]

Annotated Bibliography

[1 (http://wiki.ubc.ca/Course:CPSC522/Reinforcement_Learning_with_Function_approximation#Abstract)] Richard S. Sutton and Andrew G. Barto (1998). *Reinforcement Learning: An Introduction*. The MIT Press. ISBN 0262193981, 9780262193986 (https://books.google.ca/books/about/Reinforcement_Learning.html?id=CAFR6IBF4xYC&redir_esc=y) Reinforcement Learning: An Introduction (HTML format) (<http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html>)

[2 (http://wiki.ubc.ca/Course:CPSC522/Reinforcement_Learning_with_Function_approximation#Content)] UBC ECE 592 Course - Instructed by: Dr. Sarbjit Sakaria ECE 592- Architecture Design for Learning Systems (https://courses.ece.ubc.ca/592/EECE592_WebSite_2009/Welcome.html)

[3 (http://wiki.ubc.ca/Course:CPSC522/Reinforcement_Learning_with_Function_approximation#Generalization_with_Multi-Layer_Perceptrons_.28Neural_Network.29)] Recurrent Neural Network Based Approach for Solving Groundwater Hydrology Problems: Ivan N. da Silva, José Ângelo Cagnon and Nilton José Saggioro (<http://www.intechopen.com/books/artificial-neural-networks-architectures-and-applications/recurrent-neural-network-based-approach-for-solving-groundwater-hydrology-problems>) - ISBN 978-953-51-0935-8, Published: January 16, 2013 under CC BY 3.0 license.

[4 (<http://aispace.org/neural/>)] David Poole (<https://www.cs.ubc.ca/~poole/>), Alan Macworth (<http://www.cs.ubc.ca/~mack/>). AISpace.org (<http://www.aispace.org/index.shtml>) - Copyright © 1999 - 2010

[5 (<http://www.sund.de/netze/applets/BPN/bpn2/ochre.html>)] The OCHRE applet (Optical CHaracter REcognition) (<http://www.sund.de/netze/applets/BPN/bpn2/ochre.html>)

[6 (<http://www.applied-mathematics.net/qlearning/qlearning.html#robot>)] Dr. Ir. Frank Vander Berghen (<http://www.applied-mathematics.net/qlearning/qlearning.html#robot>) - Kranf Site - Q-learning Crawling Robot applet


[7 (<http://neuroph.sourceforge.net/documentation.html>)] Neuroph Java Neural Network Framework - Zoran Sevarak (<http://neuroph.sourceforge.net/documentation.html>).

[8] Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method, Martin Riedmiller, Neuroinformatics Group, University of Osnabrück, 49078 Osnabruck

[9] Reinforcement Learning and Artificial Intelligence (RLAI) - University of Alberta - RLFA Webpage (<http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLFA.html>)

To Add


Fun tool for Q-Learning (<http://www.applied-mathematics.net/qlearning/qlearning.html#robot>)
Reinforcement Learning for Cellular Telephones (<http://web.eecs.umich.edu/~baveja/Demo.html>)
Backpropagation for Image Compression (<http://neuron.eng.wayne.edu/bpImageCompression9PLUS/bp9PLUS.html>)
Cat and Mouse Game Reinforcement Learning (<http://www.cse.unsw.edu.au/~cs9417ml/RL1/applet.html>)



cc creative commons
SOME RIGHTS RESERVED

Permission is granted to copy, distribute and/or modify this document according to the terms in Creative Commons License, Attribution-NonCommercial-ShareAlike 3.0.

The full text of this license may be found here: CC by-nc-sa 3.0
(<http://creativecommons.org/licenses/by-nc-sa/3.0/>)



Retrieved from "http://wiki.ubc.ca/index.php?title=Course:CPSC522/Reinforcement_Learning_with_Function_approximation&oldid=395180"
Category: CPSC522

- This page was last modified on 11 February 2016, at 00:29.
- This page has been accessed 3,719 times.