

REINFORCEMENT LEARNING IN  
GAME PLAYING

By

KEAN GIAP LIM

Bachelor of Science

Oklahoma State University

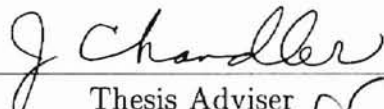
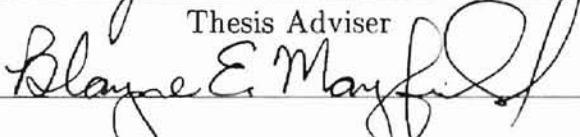
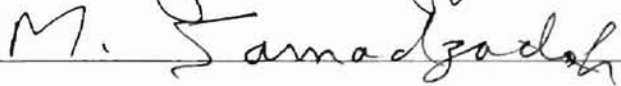
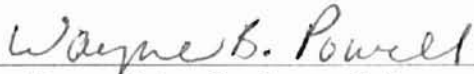
Stillwater, Oklahoma

1995

Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
MASTER OF SCIENCE  
December 1998

REINFORCEMENT LEARNING IN  
GAME PLAYING

Thesis Approved:

  
\_\_\_\_\_  
Thesis Adviser  
  
\_\_\_\_\_  
  
\_\_\_\_\_  
  
\_\_\_\_\_  
Dean of the Graduate College

## PREFACE

This study is conducted to understand the internal workings of reinforcement learning. In the movie called “Terminator II”, in a clip, Arnold Schwarzeneger told the little boy he was protecting from the Terminator that “My CPU is a neural net computer. The more I interact with humans, the more I will learn and understand about humans.” Reinforcement learning (RL) is one mechanism that improves an agent’s intelligence by evaluating the feedback that it receives from the environment with which it interacts. RL rewards well chosen actions and punishes bad decisions. The RL algorithm that was experimented with in this study is  $Q$ -learning. The agent was given the task of learning to play the trivial game of tic-tac-toe. Without any winning strategy encoded into the agent, the agent improved its moves selection by playing against its opponent. The first half of the study examined the parameters of  $Q$ -learning. The second half of this study used a neural network to generalize the agent’s experience. The advantages and disadvantages of both generalized  $Q$ -learning and  $Q$ -learning are discussed.

## ACKNOWLEDGEMENTS

I would like to sincerely thank all my committee members; Dr. J. P. Chandler (Chair), Dr. William Nick Street (ex-Chair), Dr. Blayne Mayfield, and Dr. Mansur Samadzadeh for assisting with my research. Without their guidance, friendship, and all the different kinds of help, I would not have completed this study. I would like to extend my appreciation to the Computer Science Department for giving me the opportunity to be a part of the graduate program.

I would also like to specially thank my mom and dad, who work very hard day and night, to provide the opportunity for me to further my education at Oklahoma State University. Their sacrifices, unlimited love, and supports made my dream of studying at the graduate school come true. I will never forget my sister, Lim Wei Cheng, for putting up with me for six years of my educational life in Stillwater and for the numerous suggestions that she gave me throughout this study.

Finally, I would like to express my appreciation for all the special people who work at the Center for Computer Integrated Manufacturing, especially Dr. Manjunath Kamath who gave me the opportunity to be a part of this research team. Without this opportunity, I would not have enjoyed and appreciated the value of carrying out my research. My thanks also go to Partha Ramachandran who introduced and helped me greatly with  $\LaTeX$ , and also for reviewing this report.



## TABLE OF CONTENTS

I	LITERATURE REVIEW	1
1.1	Introduction . . . . .	1
1.2	Markov Decision Processes . . . . .	2
1.3	Reinforcement Learning . . . . .	3
1.3.1	Policy . . . . .	5
1.3.2	Environment . . . . .	5
1.3.3	Reward Function . . . . .	6
1.3.4	Value Function . . . . .	8
1.4	Exploration vs. Exploitation . . . . .	10
1.5	Temporal Difference Learning . . . . .	12
1.5.1	$Q$ -Learning . . . . .	13
1.5.2	SARSA . . . . .	16
1.6	Generalization in Reinforcement Learning . . . . .	17
1.7	Artificial Neural Networks . . . . .	17
1.7.1	LMS, Delta Rule, ADALINE, or Widrow-Hoff rule . . . . .	24
1.7.2	Multilayer Neural Network . . . . .	28
1.7.3	Backpropagation or Generalized Delta Rule . . . . .	29
1.7.4	Batch and Online Training . . . . .	32
1.8	$TD(\lambda)$ . . . . .	32
1.9	Game Playing . . . . .	35
1.9.1	General . . . . .	35
1.9.2	Reinforcement Learning in Game Playing . . . . .	36
II	RESEARCH OBJECTIVE AND METHODOLOGY	39
2.1	Research Objectives . . . . .	39
2.2	Details of Implementation . . . . .	40
2.3	Details of the Opponent . . . . .	41
2.4	Details of Neural Network Training . . . . .	42
2.5	Method of Analysis . . . . .	44

III EMPIRICAL RESULTS	45
3.1 Bounded Random Walks	45
3.2 Tic-Tac-Toe	47
3.2.1 Experiment 1. Learning Rate	47
3.2.2 Experiment 2. Discount Rate	48
3.2.3 Experiment 3. Exploration Rate	50
3.2.4 Experiment 4. <i>Q</i> -learning Versus SARSA	52
3.3 Function Approximator	53
3.3.1 Experiment 1. Raw Board Representation	54
3.3.2 Experiment 2. Feature Selection	55
IV SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS	59
4.1 Summary	59
4.2 Conclusions	61
4.3 Recommendations	62
4.4 Concluding Comment	63
APPENDICES:	
A. GLOSSARY	68
B. SAMPLE PROGRAM CODE	70

## LIST OF TABLES

I	The process of backing up transition value . . . . .	10
II	Effect of discount on $Q$ -value . . . . .	15
III	Facts about the most publicized chess match between a computer and a human . . . . .	36
IV	Performance of the tic-tac-toe opponent that is used in the experiments. Its performance is tested by playing against a random move generator . . . . .	42
V	$Q$ -values generated by SARSA . . . . .	46
VI	$Q$ -values generated by $Q$ -learning . . . . .	46
VII	$Q$ -learning versus SARSA: Mean Equity of the first and next 25,000 games . . . . .	53

## LIST OF FIGURES

1	Reinforcement learning system . . . . .	5
2	A tic-tac-toe example . . . . .	6
3	Branch and bound heuristic search (left) and value updates of the second iteration of path ACDE(right) . . . . .	9
4	The need for exploration . . . . .	11
5	$Q$ -learning: An off-policy TD control algorithm . . . . .	14
6	An example MDP with rewards . . . . .	15
7	SARSA: An on-policy TD control algorithm . . . . .	16
8	Two-layer perceptron . . . . .	18
9	The exclusive-or, a classification problem that is not linearly separable	23
10	The mean square error surface . . . . .	25
11	Gradient descent orthogonal search direction . . . . .	26
12	A small learning rate converges (top) and a slightly larger learning rate may diverge (bottom) . . . . .	27
13	A multilayer feedforward neural network . . . . .	28
14	The sigmoid threshold function . . . . .	29
15	Garry Kasparov versus Deep Blue: Game 6 final position . . . . .	37
16	Reinforcement learning simulation class model . . . . .	41
17	Evaluation function neural network for a tic-tac-toe example . . . . .	44
18	Bounded random walks . . . . .	45
19	Mean square error: Effects of learning rate on the performance . . . . .	47
20	Equity: Effects of learning rate on the performance . . . . .	48
21	Equity: Discounted versus nondiscounted learning . . . . .	49
22	Mean square error: Discounted versus nondiscounted learning . . . . .	49
23	Mean square error: Exploration versus exploitation . . . . .	51
24	Equity: Exploration versus exploitation . . . . .	51
25	Equity: $Q$ -learning versus SARSA . . . . .	52
26	Mean square error: $Q$ -learning versus SARSA . . . . .	53
27	Equity: An 18-15-9 network trained with backpropagation is used to approximate the value function . . . . .	54
28	Mean square error: An 18-15-9 network trained with backpropagation is used to approximate the value function . . . . .	55
29	Mean square error of a network in which the input representation incorporated hand-selected features and was trained with backpropagation	56

30	Equity: A network in which the input representation incorporated hand selected features ( <i>feature</i> ) and was trained with backpropagation compared to ANN using raw board representation ( <i>regbp</i> ) and lookup table ( <i>table</i> ) . . . . .	57
31	Number of losses: Comparing the results obtained by the lookup table to a network in which the input representation used hand-selected features . . . . .	58

## CHAPTER I

### LITERATURE REVIEW

#### 1.1 Introduction

In recent years, researchers from different fields have shown growing interest in the field of reinforcement learning (RL). RL methods have been applied in many different domains [9] as a practical computational tool for constructing autonomous systems that improve themselves with experience. The report of National Science Foundation (NSF) Winter 1996 Workshop in RL [13] states that researchers are surprised by the failures and successes of RL and there are still many open questions to be answered.

Game playing has been an important topic in the Artificial Intelligence world and much RL research has examined this problem domain. One success story that stands out in the domain of game playing is Gerry Tesauro's TD-Gammon backgammon [32, 33, 34]. This program plays backgammon as well as the best human players. Since then, many researchers have attempted to recreate the success of TD-Gammon in other games such as Go and Chess. However these attempts have been less successful.

This thesis presents research for conducting an experiment with RL applied in the domain of game playing. This study serves mainly as a theoretical tool for studying the principles of agents learning to act by constructing an RL agent that learns to play the game of tic-tac-toe.

## 1.2 Markov Decision Processes

Each day we make many decisions, and today's decisions have impacts on tomorrow's and tomorrow's on the day after tomorrow's. We observe the situation that we are in at a point in time and choose one of the available actions. We, the decision maker, receive an immediate reward or cost and evolve into a new situation at a subsequent point in time. At this subsequent point in time, we repeat the same process over and over again. This process is known as *sequential decision making*.

The key ingredients of this sequential decision making model are the following:

1. A set of decision epochs.
2. A set of system states.
3. A set of available actions.
4. A set of state and action-dependent immediate rewards.
5. A set of state and action-dependent transition probabilities.

At each decision epoch (or time), the system provides the decision maker with all necessary information for choosing an action from the set of available actions in that state. As a result of choosing an action in a state, two things happen: the decision maker receives an immediate reward and the current state evolves into a new state at the next decision epoch. As this process evolves through time, the decision maker receives a sequence of rewards, some positive and some negative.

A *Markov decision process* model (MDP) [19] is one particular sequential decision model whereby the set of available actions, the rewards, and the transition probabilities depend only on the current state and action and not on states occupied and actions chosen in the past. In other words, if the current state summarizes everything

important about the actions that produced it, then the sequential decision model is said to have the *Markov property*.

MDP is studied extensively in operation research and optimal control. If an MDP has a finite number of states and a finite number of actions for each state, then it is a *finite MDP*. The context of this study deals only with finite MDP. MDP is important because it plays a critical role in the theory of reinforcement learning. For more information about MDP, please refer to [19].

Classical optimization methods for sequential decision problems, such as dynamic programming, can compute an optimal solution. *Dynamic programming* is a term that refers to the mathematical formulation of a sequential decision problem. If the problem is formulated, then we can find the optimal solution to the problem telling us *which action to choose in a situation at a point in time*. The formulation of the problem requires the five ingredients mentioned above.

However there are several disadvantages of using this method. A major disadvantage of dynamic programming is that it involves exhaustive sweeps through the state space. This makes it very inappropriate for large problems. Secondly this method requires a complete specification of the transition probabilities of each state. This information is normally not available a priori for the vast majority of practical problems. On the other hand, such information can be estimated from experience through trial-and-error with the system. This is the key idea in the field of reinforcement learning; learning from interaction to achieve long-term goals.

### 1.3 Reinforcement Learning

Reinforcement learning (RL) has attracted a lot of attention from researchers in different fields especially in the past ten years. This happened because there were reports of several breakthroughs in learning different tasks using this method. As



mentioned in the introduction, Gerry Tesauro's TD-Gammon [32, 33, 34] learns the game of backgammon and it is capable of playing at a grandmaster level. Crites and Barto [4] used  $Q$ -learning (explained later in Section 1.5.1) in an elevator scheduling task. The average squared waiting time for passengers was approximately 7% less than the best alternative algorithm and less than half the squared waiting time of the most frequently used elevator scheduling algorithms.

Another attractive property of this learning mechanism is that it enables the *agent* to learn the tasks autonomously. In reinforcement learning, the decision maker or the learner is called the agent. *Autonomous learning* means learning without the assistance from a teacher. Instead an autonomous agent learns by experiencing the task. The agent improves its knowledge by evaluating the feedback that it receives from the environment for all chosen actions.

Reinforcement learning is studied predominantly in the field of animal behavioral sciences. Reinforcement learning is a major part of the human learning process, thus making it easy to understand. Do you remember how you learned to ride a bicycle when you were young? You probably fell dozens of times before you successfully learned the appropriate ways to steer, brake, and pedal the bicycle. The objective is to learn to ride a bicycle without falling.

In reinforcement learning, the computer is given a *goal* to achieve. Reinforcement learning then learns a mapping from situations to actions by trial-and-error interactions with a dynamic environment, as depicted in Figure 1. The agent's goal is to discover which actions yield the most reward by trying them based on its experience with the environment.

There are four fundamental parts in reinforcement learning. These are the policy, environment, reward function, and value function.

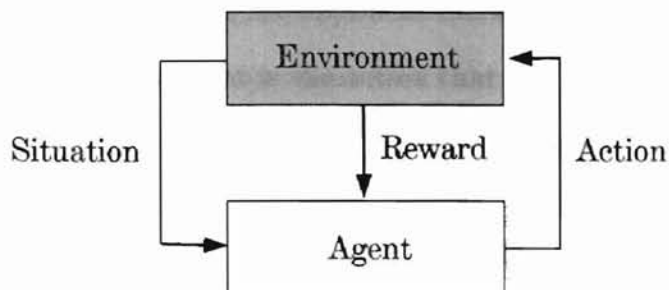


Figure 1. Reinforcement learning system

### 1.3.1 Policy

The policy  $\pi$  is the decision-making function of the agent, telling it what action to perform in each state. A policy is a mapping from states to actions. In reinforcement learning, we are trying to obtain the optimal policy  $\pi^*$  where

$$\pi^* = \{s_1 a_1, s_2 a_2, s_3 a_3, \dots, s_{T-1} a_{T-1}, s_T\}.$$

At each point of time, or epoch  $t$ , each pair  $s_t a_t$  of the optimal policy  $\pi^*$  tells us that taking action  $a_t$  when the agent is in state  $s_t$  yields the highest value in the long run [3, 19]. State  $s_T$  is the terminal state. The terminal state  $s_T$  is our goal.

This is the core of the reinforcement learning agent because the other components serve to improve the policy. Ultimately it is the policy itself that determines how well the agent performs.

### 1.3.2 Environment

The *environment* is a simulation model with which the agent interacts. For example, an RL game-playing agent interacts with its opponent to increase its experience. In this case, the opponent is the environment. A decision-making agent seeks to achieve its goal or goals despite uncertainty about its environment. Consider a tic-tac-toe learning agent that is in state  $s_t$  with three possible actions  $a_1$ ,  $a_2$ , and  $a_3$  to choose.

If it chooses action  $a_1$ , that means the opponent has action  $a_2$ , and  $a_3$  to select. Then the uncertainty of the environment is the action that will be chosen by the opponent. Thus, the agent is uncertain about the next state  $s_{t+1}$  because it does not know which action will be chosen by the opponent as depicted in Figure 2.

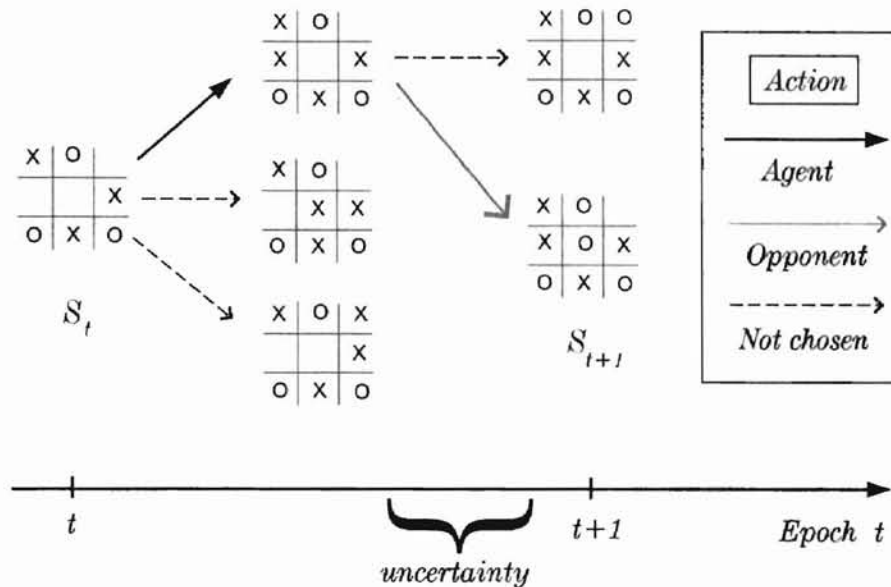


Figure 2. A tic-tac-toe example

### 1.3.3 Reward Function

The *reward function* is also known as the reinforcement function or the utility function. The reward function defines the goal of the reinforcement learning system. It maps the state of the environment to a single number. When the agent achieves the goal, then the action that leads to the goal state is *rewarded* or reinforced with a positive reward. On the other hand, if the agent makes a mistake, then the agent is punished for making that bad decision. This *punishment* is also referred as the negative reward. For instance, if the goal of a tic-tac-toe agent is to learn to win, a positive reward of 1 is assigned for taking the action that leads to the winning state.

Similarly the move that loses a game is punished with a negative reward of -1. So the reward function for a tic-tac-toe agent is

$$R(x) = \begin{cases} +1 & \text{win} \\ 0 & \text{draw} \\ -1 & \text{loss} \\ 0 & \text{otherwise} \end{cases}$$

where *win*, *draw*, and *loss* are terminal states, and *otherwise* defines the reward for all other intermediate moves of a game. As you may have noticed, all the other intermediate actions are neither punished nor rewarded. Let us take the game of chess as an example using the same reward function. The game of chess is won by *good* defenses and traps throughout the game. So what is wrong if a positive reward is assigned for a good chess move such as a knight fork (knight fork is an attacking move where the knight is in a position to attack more than one of its enemy)? In this case, the agent will choose the intermediate move that has the highest immediate reward. As a result, the agent prefers actions that are advantageous in the short term, not actions that leads to a win. Unless one of the agent's goal is to learn knight fork instead of winning the game, it should not be encoded in the reward function for an agent. It is crucial to indicate again the importance of rewarding and punishing *only* actions that meets the goal or goals. This is the reason why all intermediate moves are not rewarded nor punished.

In contrast to reinforcement learning, all of today's successful chess-playing programs that search the position-tree for every move, material winning is the most important component of the objective function. In chess and checkers, the end of the game is usually not visible for being too far down the tree. Thus intermediate positions are evaluated in a tree-searching method.

The reward function is myopic; it determines the immediate reward that an agent

will receive by taking a certain action. It is only one component of reinforcement learning that *defines* the goal or goals of the learning system. The objective of an RL agent is to maximize the total reward it will receive in the long run, meaning, in this case, over many games played. So we need something else that tells the agent which action to choose from the beginning till the end of every trial. To do this, we need a value function.

### 1.3.4 Value Function

The *value function* is the means that specifies what action selection is the best in the long run. Values indicate the *long term* desirability of a state taking into consideration all the states that are likely to follow. The value function is helpful because it can be used to improve the policy. A reinforcement learning system changes its value functions to make them close to the optimal solution.

A heuristic evaluation function is a form of value function that we are mostly familiar with. Heuristic search is an efficient and intelligent search method to generate good solutions without having to search exhaustively. Heuristic search, however, incorporates prior knowledge about state values into the search algorithm, thus making it inflexible [23]. *Q*-learning (explained later in Section 1.5.1) does not require prior knowledge of the domain, but rather, learns a value function through experience.

To have a better understanding about the value function, let us look at how the heuristic search and the value function are used to find a solution. Let us consider the *branch and bound* heuristic search. At each step of the branch and bound process, we select the most promising of the states we have so far. We expand all the branches of the most promising state. We stop if one of them is a solution. The left half of Figure 3 shows an example where state A is the beginning state. First, expand all of state A's branches, producing two successor states, B and C. In this example, we

attempt to minimize the value of the objective function. Next, state C is chosen to be expanded because it has a lower value, 4, than AB which is 5. Now the paths that are expanded are AC(4), ACD(6), ACK(11), and ACL(12). These paths are put into a list that is sorted by the path's value. Now branch AC is expanded to ABF(6) and ABI(8). This time, ABF and ACD have the same value of 6. Thus state D and F are expanded and generate ABFH(10), ABFG(12), and ACDE(15). This makes path ABI the shortest path with a value of 8. Finally when ABI is expanded to ABIJ(11), path ABFH becomes the optimal solution with its path value of 10. This is because there is no other branch that has a smaller value than ABFH.

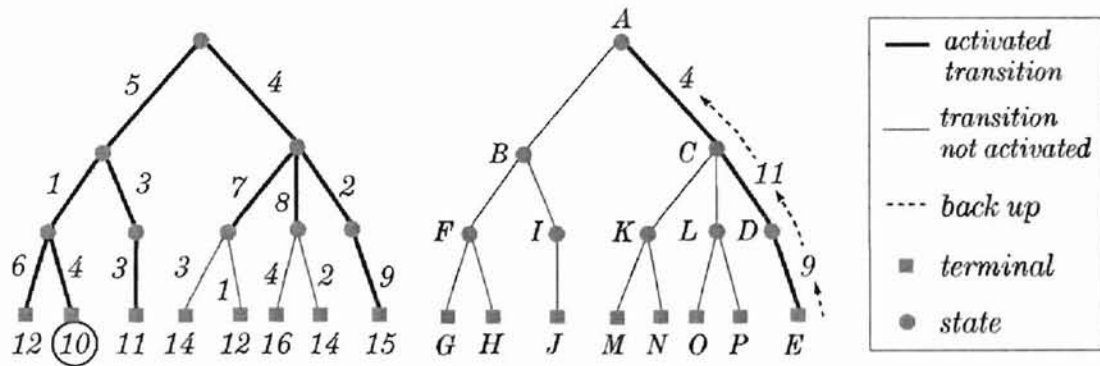


Figure 3. Branch and bound heuristic search (left) and value updates of the second iteration of path ACDE(right)

Next, let us look at the value function. Reinforcement learning updates each value of the transition that the agent has chosen. This process of re-estimating values is called the *back up* process. As mentioned before, RL learns through experimentation. A simple value update function can be written as

$$v_{transition} = r_{transition} + \min v_{next\ transition}$$

where  $v_{transition}$  is the value of a transition,  $r_{transition}$  is the reward of a transition, and  $\min v_{next\ transition}$  is the minimum value of the next transition. Each transition value

is initialized to zero. Suppose that the agent has chosen path ACDE for the first two trials. Table I shows how the value of each transition of path ACDE is backed up for two iterations. In the second iteration,  $\min v$  of C→D is 9 because the value of branch D→E is updated to 9 in the first iteration. Through trial and error, the agent keeps backing up transition values of paths that it chooses. If the agent continues to choose different actions, it will eventually learn that the optimal path is ABFH.

Table I. The process of backing up transition value

Transition	A→C $v_{AC} = r_{AC} + \min v$	C→D $v_{CD} = r_{CD} + \min v$	D→E $v_{DE} = r_{DE} + 0$
Iteration 1	4+0=4	2+0=2	9+0=9
Iteration 2	4+0=4	2+9=11	9+0=9

But how does the agent decide which transition to choose? This is the topic of discussion in the next section.

#### 1.4 Exploration vs. Exploitation

A reinforcement learning agent's objective is to maximize the total expected reward over some time period. If we always choose the action that yields the highest estimated value, as in greedy search, then we are *exploiting* the current knowledge of the value of the actions. Otherwise we are *exploring*. Exploration enables us to improve the estimate of the non-greedy action's value that may generate greater total reward in the long run. Consider the example shown in Figure 4 where the starting state is 1. If the agent always chooses the path that leads to the highest immediate reward, it takes path 1 → 3. It is obvious that the agent needs to explore to realize a better path by trying 1 → 2 → H that yields a total reward of 10, which is higher than path 1 → 3 → H that has a cumulative reward of 6.

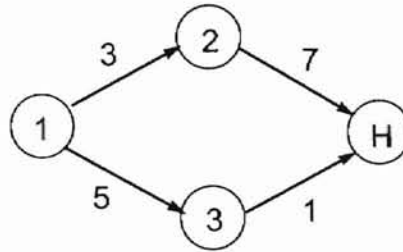


Figure 4. The need for exploration

The example given above can be applied to *learn* the optimal policy of general decision problems. This example is based on the reinforcement learning method that is applied in this study that learns the mapping from each *state-action* pair to a *value*. However most games like chess or checkers that uses heuristic search method to find for moves, the value depends on the *position* (state).

To ensure that the agent will keep exploring all actions, it is necessary for the agent to continue to select them. There are two approaches to ensure this, called the *on-policy* learning and *off-policy* learning [30]. On-policy methods attempt to evaluate and improve the same policy that they use to make decisions. In off-policy methods, the policy used to generate behavior, called the behavior policy, is unrelated to the policy that is evaluated and improved, the estimation policy. This separation has the advantage that the estimation policy can keep getting more greedy while the behavior policy can continue to sample all possible actions. One behavior policy that is commonly used in RL is the  $\epsilon$ -greedy policy [30] where  $0 \leq \epsilon \leq 1$ . If  $\epsilon$  is set to 0.1, this means that the agent chooses one action randomly in every ten decisions. The agent behaves greedily for the other nine moves.

$\epsilon$ -greedy action-selection is not suitable for problems in which it is unacceptable for the agent to experiment with a non-greedy action that is very bad. This is because when the agent explores, it chooses an action with equal likelihood among all actions.



A slightly more sophisticated alternative to  $\epsilon$ -greedy is *softmax* or *Boltzmann exploration* [30]. This is used in *simulated annealing* [10]. In this case, an action is chosen probabilistically according to the distribution

$$P(a) = \frac{e^{Q(a)/\tau}}{\sum_{a' \in A} e^{Q(a')/\tau}}$$

$Q(a)$  is the value of action  $a$ , and  $\tau$  is the temperature parameter that can be reduced over time to reduce exploration. If  $\tau$  is set high, each action is chosen with equal probability. As  $\tau$  approaches 0, the values of the actions are distanced from each other.

### 1.5 Temporal Difference Learning

The key idea to reinforcement learning as agreed by many researchers is temporal difference learning (TD) [28], which is a technique for recursively learning an evaluation function.

The power behind TD learning is that it eliminates the “*rollout*” method [29] used in a traditional method such as dynamic programming. The “*rollout*” method backs up or refines values of the policy only when it reaches the final state. TD learning improves its policy by making estimates from another estimates. This mechanism is actually closer to how humans learn. Suppose you have experience drawing a circle; you do not need to wait till the circle is completely drawn before knowing what adjustments are needed if you want to draw a well-rounded circle.

Temporal-difference learning is separated into two categories: multi-step prediction learning or single-step prediction learning. In single-step prediction problem, you know the *correctness* of a prediction after a step of the prediction. Whereas in multi-step prediction problem, the correctness of a prediction is not determined until more than one step after the prediction. It is beneficial to update the prediction

made several steps in the past using new observations gathered after that. For example, a bad move made in a chess game may not be revealed until several sequence of moves later. It has been shown that treating most prediction problems as multi-step converges faster. However the algorithm of multi-step prediction is slightly more computationally intensive and harder to implement. The algorithm of multi-step prediction, called TD( $\lambda$ ) is discussed after the discussion of artificial neural network.

The algorithm of single-step prediction is a special case of TD( $\lambda$ ) known as TD(0). The single-step algorithm that is very well known is  $Q$ -learning. A variation of  $Q$ -learning called SARSA is also discussed in detail later.

### 1.5.1 $Q$ -Learning

$Q$ -learning is an example of an off-policy TD control algorithm that was developed by Watkins in 1989 [39]. The notation  $Q(s, a)$  is used to represent the estimated value taking action  $a$  in state  $s$ . The goal is to learn the value function,  $Q : S \times A \Rightarrow \mathfrak{R}$ . Traditionally this function is implemented as a table, with a value for each state-action pair. Its simplest form, 1-step  $Q$ -learning is defined by

$$\Delta Q(s, a) = \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (1.1)$$

where  $r$  is the reward,  $\alpha$  is the step-size parameter or learning rate,  $\gamma$  is the discount rate,  $s', a'$  is the state-action pair in the subsequent point in time  $t + 1$ , and  $\max_{a'} Q(s', a')$  is the maximum  $Q$ -value for picking any action  $a'$  in state  $s'$ . The algorithm for  $Q$ -learning using table lookup is shown in Figure 5. To illustrate how the learning function is used, consider the following settings for a tic-tac-toe agent that learns to win where  $\alpha = 0.5$ ,  $\gamma = 0.5$  and all values of  $Q(s, a)$  are set to 0. The opponent selects its moves randomly, and the agent wins the first game. Thus, the agent receives a reward of 1 for winning the game by taking action  $a_w$  in state  $s_w$ .

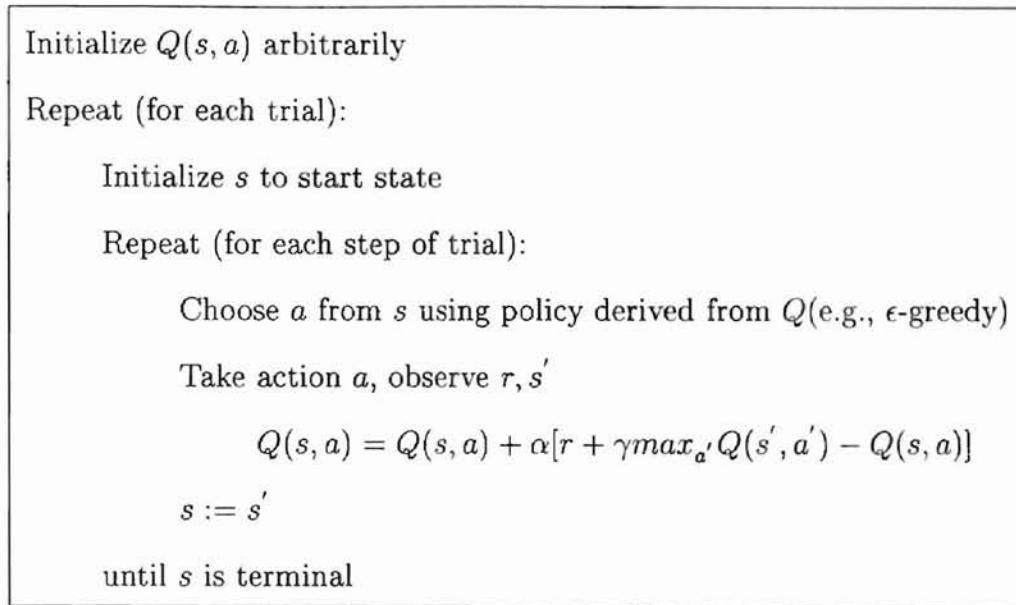


Figure 5.  $Q$ -learning: An off-policy TD control algorithm

According to (1.1),

$$\begin{aligned}
 \Delta Q(s_w, a_w) &= 0.5 [1 + [0.5(0) - 0]] \\
 &= 0.5(1 + 0) \\
 &= 0.5
 \end{aligned}$$

Since the initial value of  $Q(s_w, a_w)$  is 0, the updated value of  $Q(s_w, a_w)$  is 0.5.

The agent continues to play another game, and the agent reaches state  $s_{w-1}$ . The opponent responded by taking an action that ended up in state  $s_w$ . So the  $\max_{a'} Q(s', a')$  is  $Q(s_w, a_w)$  which is 0.5. Thus  $Q(s_{w-1}, a_{w-1})$  is

$$\begin{aligned}
 Q(s_{w-1}, a_{w-1}) &= Q(s_{w-1}, a_{w-1}) + \alpha[r_{t+1} + \gamma Q(s_w, a_w) - Q(s_{w-1}, a_{w-1})] \\
 &= 0 + 0.5[0 + 0.5(0.5) - 0] \\
 &= 0.125
 \end{aligned}$$

A discounted positive  $Q$ -value is assigned to  $Q(s_{w-1}, a_{w-1})$  because the agent may reach state  $s_w$ . If each action is executed in each state an infinite number of times on an infinite run and  $\alpha$  is decayed appropriately, the  $Q$ -values will converge with

probability 1 to  $Q^*$  where  $Q^*$  is the optimal  $Q$ -value [8].

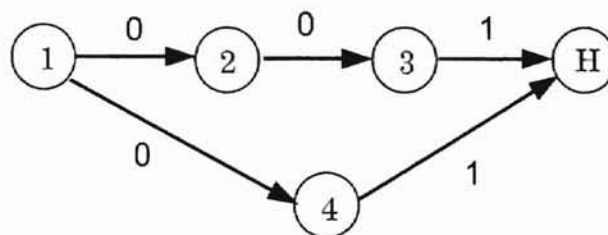


Figure 6. An example MDP with rewards

A discount rate,  $\gamma$ , is introduced to account for the time value of rewards. It measures the value at time  $t$  of a one-unit reward received at time  $t + 1$ . A one-unit reward received  $t$  periods in the future has present value of  $\gamma^t$ . To see the effects of  $\gamma$ , let us look at Figure 6. The figure shows that a positive reward of 1 is only assigned when state H is reached. The effect of  $\gamma$  on each  $Q$ -value is shown in Table II.

Table II. Effect of discount on  $Q$ -value

Path	1→2	2→3	3→H	1→4	4→H
Discounted $Q$ -values ( $\gamma = 0.5$ )	0.25	0.5	1	0.5	1
Undiscounted $Q$ -values ( $\gamma = 1$ )	1	1	1	1	1

Taking the discount rate into account affects the agent's preference for policies. If this example is an undiscounted problem, then both paths to state H have the same  $Q$ -value of 1. In contrast, using the discount rate, the agent prefers path 1→4→H because the goal state H is reached in fewer steps.

The  $Q$ -value of the terminal state is always 0. A  $Q$ -value is assigned to every state-action pair that involves a state transition. The terminal states are the only exception since they are the only states that have no transition to another state.

## 1.5.2 SARSA

SARSA is a variation of  $Q$ -learning that was developed by Sutton [30]. Just like  $Q$ -learning, the SARSA agent begins by taking an action  $a$  in state  $s$ . It then waits for the opponent to respond. After the opponent has moved, state  $s$  evolves into state  $s'$ . Then the SARSA agent needs to choose an action  $a'$  based on its behavior policy. This is the step that distinguishes SARSA from  $Q$ -learning. SARSA's chosen action,  $a'$ , may be an exploratory action. So  $Q(s', a')$  of SARSA will not always be the highest value, while  $Q$ -learning always backs up each  $Q(s, a)$  by discounting the highest  $Q$ -value of the  $s', a'$  pair. The SARSA algorithm is shown in Figure 7.

SARSA is an on-policy learning algorithm because it learns by improving the same

```

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

Figure 7. SARSA: An on-policy TD control algorithm

policy that it uses to select all actions. If the behavior policy of SARSA always acts greedily, then SARSA and  $Q$ -learning are the same.

## 1.6 Generalization in Reinforcement Learning

The artificial neural network functions as an approximation mechanism of the  $Q$ -values instead of storing them in a table. Why do we want to approximate the  $Q$ -values when we can have the actual  $Q$ -values stored in a table? The table  $Q$ -values are updated each time the corresponding state-action pair is encountered during training. This means that at the end of the learning process, there are possibly many  $Q$ -values that have not been updated because these state-action pairs were not experienced. The ability of an artificial neural network to generalize experiences allows the agent to make better decisions even in situations that it has never experienced before. Generalization allows the network to evaluate a state and generates a response based on features that are similar to some states that the network has encountered. Therefore, generalization may speed up learning by discovering commonalities among states.

In addition, because of the large number of states in many environments, using a lookup table to represent the function is not feasible. This shortcoming is known as the ‘curse of dimensionality’. Generalization of these large spaces can be achieved using a function approximator such as artificial neural network (ANN) [1]. This is a very powerful learning tool. It is essential to understand artificial neural networks if we want to scale up reinforcement learning to solve larger problems.

## 1.7 Artificial Neural Networks

Today, there is a large body of literature on the subject of artificial neural networks, ANN for short. An ANN is a simplified model of the brain cells, or *neurons* that are massively interconnected, usually simulated in software by using a computer. In the 1950s and 1960s, when the mass enthusiasm began, this field was known as connectionism. That was when the *perceptron* was introduced by Frank Rosenblatt [21], and this is the first network we will discuss.

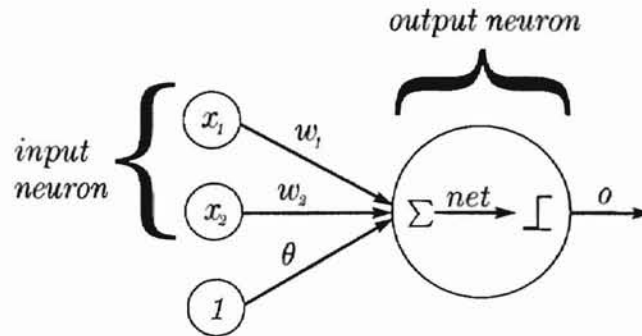


Figure 8. Two-layer perceptron

Figure 8 illustrates a two-layer perceptron. The connections between neurons determine the function of the network. Associated with each connection is a weight  $w_i$ . The strength of the weights is represented by real numbers. The neurons sum up the weights of the activated inputs. If the sum of the inputs is greater than some threshold value,  $\theta$ , the neuron turns on (outputs a 1), otherwise it is off (outputs a 0). The function that represents the threshold value is called the *activation function*.

In supervised learning, an artificial neural network including the perceptron learns a mapping from a set of inputs to the corresponding target outputs. *Learning* takes place by updating the weights and thresholds so that the network generates the desired outputs corresponding to the given inputs. The output of the perceptron is given by

$$o = \text{hardlim} \left( \sum_i (w_i x_i) + \theta \right).$$

where  $x_i$  is the  $i^{\text{th}}$  input neuron,  $w_i$  denotes the weight that connects input neuron  $i$  to the output neuron, and  $\theta$  is the threshold value. Let us denote  $net$  as the weighted sum of inputs. The *hardlim* activation function is defined as

$$o = \text{hardlim} \left( \underbrace{\sum_i w_i x_i}_{net} + \theta \right) = \begin{cases} 1 & \text{if } net \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The perceptron learning rule indicates how the weights and thresholds are adjusted if the network is not producing all the desired outputs. In other words, the perceptron keeps changing both its adjustable parameters. The error of the network,  $e$ , is the difference between the desired output's value,  $y$ , and the estimated output value produced by the network,  $o$ . The perceptron stops adjusting the parameters if it generates all the desired outputs based upon the examples that the perceptron has seen. The perceptron learning rule is

$$\begin{aligned}w &= w + ex, \\ \theta &= \theta + e, \quad \text{where} \\ e &= y - o.\end{aligned}$$

Let us now look at a trivial example to see how a perceptron works. Suppose you would like a vending machine that accepts one and two dollar bills. The way the machine separates them is by using its somewhat primitive scanning device to read in three selected features on the face of the bill. The first feature is the seal of the Department of the Treasury (both bills have this), then the fullness of the president's hair (Jefferson has more hair than Washington does), and finally whether there exist the word "THE UNITED STATES OF AMERICA" on top of the president's picture (the one dollar bill has it whereas the two dollar bill does not). The features of the input can be represented as a vector such that

$$x = \begin{bmatrix} seal \\ hair \\ USA \end{bmatrix}.$$

If any of the feature exists, a 1 can be used to indicate that; and a -1 to indicate otherwise. So the respective input vector for the one dollar bill and the two dollar



bill are

$$x_{\$1} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \quad \text{and} \quad x_{\$2} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$$

First of all, we need to select a network architecture to represent this task. Since there are three features that the vending machine scans, so we need three input neurons. And we want only one answer, whether it is a one dollar or a two dollar bill. Hence one output node is essential. The output is either a 0 or a 1 because the *hardlim* activation function is used. In this example, an output of 1 means a one dollar bill and vice versa. This artificial neural network *architecture* is a two-layer 3-1 network. In this example, it is a two-layer network because it consists of one layer of three input neurons, and another layer that consists of one output neuron.

The weights and thresholds of the perceptron are commonly initialized with small random numbers. Let us pick 0.2, 0.3, 0.1, and -0.4 for  $w_1, w_2, w_3$ , and  $\theta$  respectively. Now we are ready to train the perceptron to recognize both the one and two dollar bill. Let us see if these initial values of the network parameters actually produce the two outputs that we desire. This could happen because this is not a complicated example. We begin the training with a one dollar bill where  $w_1 = 0.2, w_2 = 0.3, w_3 = 0.1$ ,  $x_1 = 1, x_2 = -1, x_3 = 1$  and  $\theta = -0.4$ .

**Iteration 1:**

$$\begin{aligned} o &= \text{hardlim} \left( \sum_i w_i x_i + \theta \right) \\ &= \text{hardlim}(0.2(1) + (0.3)(-1) + (0.1)(1) + (-0.4)) \\ &= \text{hardlim}(-0.5) \\ &= 0. \end{aligned}$$

We expect the perceptron to output a 1; instead it outputs a 0 which is an error. The perceptron learning rule needs to be applied to change the weights and threshold

in order to yield the correct answer. First we need to calculate the error.

$$e = y - o = 1 - 0 = 1.$$

Then the weights update are

$$w = w + ex$$

$$w_1 = 0.2 + (1)1 = 1.2$$

$$w_2 = 0.3 + (1) - 1 = -0.7$$

$$w_3 = 0.1 + (1)1 = 1.1$$

The threshold update is

$$\theta = \theta + e = -0.4 + (1) = 0.6$$

These are the adjustments to the weights and threshold where now  $w_1 = 1.2$ ,  $w_2 = -0.7$ ,  $w_3 = 1.1$  and  $\theta = 0.6$ . But does the perceptron recognizes the one dollar bill right now? Let us find out.

$$\begin{aligned} o &= \text{hardlim}(1.2(1) + (-0.7)(-1) + 1.1(1) + 0.6) \\ &= \text{hardlim}(3.6) \\ &= 1. \end{aligned}$$

Next we look at whether the perceptron recognizes the features of a two dollar bill with its current weights and threshold values where  $x_1 = 1$ ,  $x_2 = 1$ ,  $x_3 = -1$ .

**Iteration 2:**

$$\begin{aligned} o &= \text{hardlim}(1.2(1) + (-0.7)(1) + 1.1(-1) + 0.6) \\ &= \text{hardlim}(0) \\ &= 1. \end{aligned}$$

$$e = y - o = 0 - 1 = -1.$$

$$w_1 = 1.2 + (-1)1 = 0.2$$

$$w_2 = -0.3 + (-1)1 = -1.3$$

$$w_3 = 1.4 + (-1)(-1) = 2.4.$$

$$\theta = 0.6 + (-1) = -0.4$$

Let us prove that the network recognizes the two dollar bill:

$$\begin{aligned} o &= \text{hardlim}(0.2(1) + (-1.3)1 + (2.4)(-1) + (-0.4)) \\ &= \text{hardlim}(-3.9) \\ &= 0. \end{aligned}$$

We know that the perceptron recognizes the two dollar bill now, but does it still recognize the one dollar bill? Since we have only two training samples, the third iteration tests the network with the one dollar bill again.

**Iteration 3:**

$$\begin{aligned} o &= \text{hardlim}(0.2(1) + (-1.3)(-1) + (2.4)1 + (-0.4)) \\ &= \text{hardlim}(3.5) \\ &= 1. \end{aligned}$$

Well since the weights and threshold generate the correct output of 1 indicating that the input is a one dollar bill, no adjustment to the parameters is needed after the third iteration. So the network parameters have converged after two iterations with  $w_1 = 0.2$ ,  $w_2 = -1.3$ ,  $w_3 = 2.4$ , and  $\theta = -0.4$ . The perceptron learning rule is proved to converge to parameters that accomplish the desired classification [7], given that such parameters exist. Remember that this perceptron is trained to recognize only the one and two dollar bill. If the perceptron is presented with a twenty dollar bill in which all three features are present, the perceptron will output a 0. It is not capable of always classifying correctly any inputs other than one and two dollar bills because it is not trained to do so. This simple example presented above is a *pattern recognition* problem, for which artificial neural networks are often used.

The neural network parameters draw a hyperplane in the three-dimensional weight space (since there are three inputs in this example) to linearly separate the one and two dollar bill into two categories. That is why a twenty dollar bill is classified as a two dollar bill, because it falls into the two dollar half of the weight space. The

perceptron keeps adjusting the parameters because it has not found the hyperplane in the weight space that correctly separates the two patterns.

Even though the perceptron is powerful enough to solve many classification problems, it is limited to problems that are linearly separable. In 1969, Minsky and Papert documented the shortcomings and capability of the perceptron [15]. One problem that is not linearly separable by one hyperplane is the classical *exclusive-or*, as depicted in Figure 9.

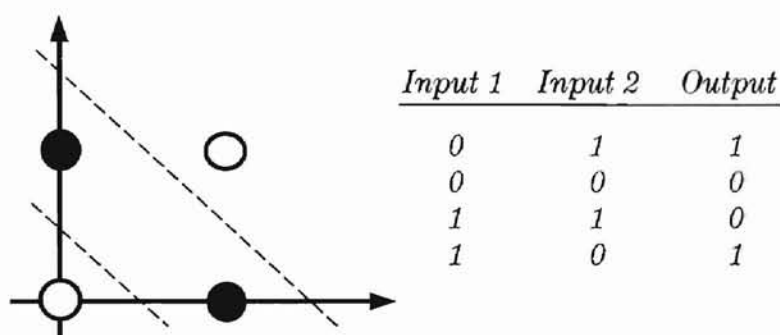


Figure 9. The exclusive-or, a classification problem that is not linearly separable

In order to learn problems that are not linearly separable, we can build a multilayer neural network with one or more “hidden” layers to learn a more sophisticated function. This is also known as a multilayer perceptron or multilayer feedforward network. Multilayer neural networks can be trained to solve problems that are not linearly separable, such as the exclusive-or, by using the backpropagation algorithm. But before looking at backpropagation, first we have to understand the learning system behind it. This learning system is known as the Least-Mean-Square rule.

### 1.7.1 LMS, Delta Rule, ADALINE, or Widrow-Hoff rule

In 1960, Widrow and Hoff introduced an adaptive algorithm known as the Least-Mean-Square (LMS) rule. It is also known as the delta rule, the Adaptive Linear Neuron (ADALINE), and the Widrow-Hoff rule. Like the perceptron rule, the LMS algorithm is also an *adaptive or optimization algorithm* which means the parameters (weights and thresholds) can be properly adjusted to achieve the correct values. The LMS algorithm is applicable to a two-layer network, consisting of a layer of input neurons and another layer of output neurons. However, the LMS algorithm also suffers from not being capable of learning nonlinear functions. Yet it is essential to understand the LMS algorithm because the backpropagation algorithm is a powerful extension of the LMS rule.

The LMS algorithm observes the performance of the network during training. The *performance* of a network is better when the network is closer to classifying all the patterns correctly. As mentioned before, if not all the patterns are classified correctly, then the network has *erroneous* parameters. Consequently, if we have a way to measure the error of the network parameters, we know the performance of the network.

The performance measure of the LMS rule is the *approximate mean square error*. The LMS rule tries to reduce the approximate mean square error in order to obtain the best performance network. This is where the name Least-Mean Square (LMS) comes from. The mean square error (MSE) is defined as

$$MSE = \frac{1}{n} \sum_{i=1}^n [(y_i - o_i)^2] = \frac{1}{n} \sum_{i=1}^n [(\delta_i)^2]$$

where  $n$  is the number of input-output pairs,  $y_i$  is the  $i^{th}$  desired output,  $o_i$  is the  $i^{th}$  estimated output, and  $\delta_i$  is the difference between  $y_i$  and  $o_i$ . At each iteration, the

LMS algorithm *estimates* the mean square error by

$$\widehat{MSE} = (y - o)^2 = (\delta)^2$$

where the mean square error is replaced by the square error. The error of the output is reduced through adjustment of the weights and thresholds. A space has dimension  $n$  when points in it can be specified by  $n$  coordinates. The two-dimensional plane requires two coordinates  $(x, y)$ , three-dimensional space requires three, and so on. Thus the approximate mean square error at each point in time is a coordinate on the error surface that is specified by the weights and thresholds. For a two-layer network, the error surface with respect to the network parameters is a paraboloid as shown in Figure 10.

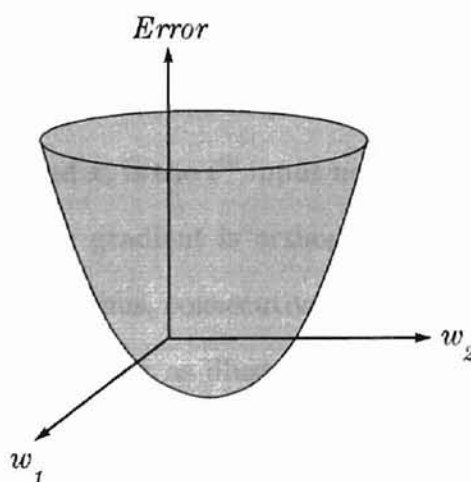


Figure 10. The mean square error surface

The bottom of the paraboloid, the *minimum* or the *optimum* point, is where the error is reduced to zero. To reduce the approximate mean square error is to descend downhill to the minimum point on the error surface. The steepest slope downhill is the negative of the *gradient*,  $\nabla_w \widehat{MSE}$  because the gradient vector is pointing uphill. This method of minimizing the error is known as *steepest descent* or *gradient descent*.

The gradient with respect to the network weights is defined as

$$\nabla_w \widehat{MSE} = \left[ \frac{\partial \widehat{MSE}}{\partial w_i}, \frac{\partial \widehat{MSE}}{\partial w_{i+1}}, \dots \right]$$

such that

$$\begin{aligned} \frac{\partial \widehat{MSE}}{\partial w_i} &= \frac{\partial \delta^2}{\partial w_i} \\ &= 2\delta \frac{\partial \delta}{\partial w_i} \\ \frac{\partial \delta}{\partial w_i} &= \frac{\partial (y - o)}{\partial w_i} \\ &= \frac{\partial}{\partial w_i} \left[ y - \left( \sum_i^n w_i x_i + \theta \right) \right] \\ &= -x_i \\ \frac{\partial \widehat{MSE}}{\partial w_i} &= -2\delta x_i \\ \frac{\partial \widehat{MSE}}{\partial \theta} &= -1 \end{aligned}$$

where  $\frac{\partial \widehat{MSE}}{\partial w_i}$  is the first derivative of the  $\widehat{MSE}$  along the  $w_i$  axis,  $n$  is the number of neurons in the input layer, and  $x_i$  is the  $i^{\text{th}}$  input neuron. The derivative is zero at the minimum point, therefore the gradient is orthogonal (perpendicular or tangent) to the previous search direction. Thus, consecutive search directions of gradient descent are always orthogonal to each other as illustrated in Figure 11.

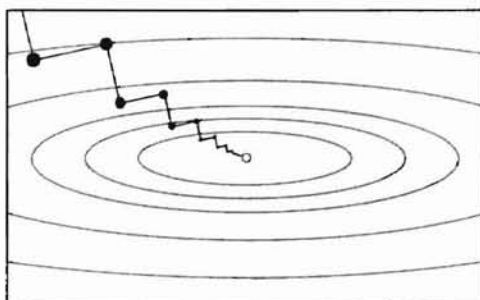


Figure 11. Gradient descent orthogonal search direction

Using the approximation of  $\nabla_w \widehat{MSE}$ , the LMS algorithm is

$$w_i = w_i + \Delta w_i$$

$$\theta = \theta + 2\alpha\delta$$

such that

$$\begin{aligned} \Delta w_i &= \alpha(-\nabla_w \widehat{MSE}_i) \quad (\text{negative gradient}) \\ &= -\alpha(-2\delta_i x_i) \\ &= 2\alpha\delta_i x_i \end{aligned}$$

where  $\alpha$  is a constant learning rate. The learning rate decides the magnitude of the gradient. Hence a small learning rate is normally used to keep the gradient from changing too fast. If the magnitude of the new gradient is greater than the previous, then the parameters may not converge (illustrated in Figure 12) [7]. The selection of the best learning rate is obtained through trial and error.

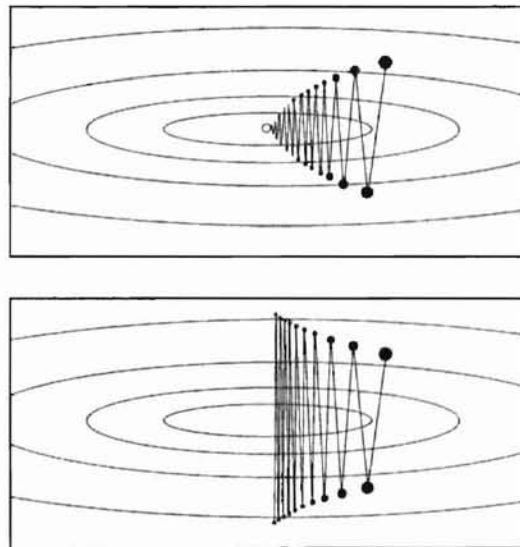


Figure 12. A small learning rate converges (top) and a slightly larger learning rate may diverge (bottom)



## 1.7.2 Multilayer Neural Network

Up to this point, we understand that both the perceptron and the ADALINE network are incapable of solving problems that are not linearly separable. Researchers believed that this barrier could be overcome by building a different network architecture and using a more powerful algorithm to train the network. Studies were conducted in constructing an algorithm to train a multilayer neural network. An additional layer of neurons, called the “hidden layer”, was added in between the input and the output layer. Finally Rumelhart, Hinton, and Williams introduced the backpropagation algorithm that is capable of training a multilayer neural network. By applying the backpropagation algorithm, a multilayer network is capable of classifying nonlinear problems. Figure 13 illustrates a 2-3-1 multilayer feed-forward neural network architecture that has 2 input units, 3 hidden units, and 1 output unit.

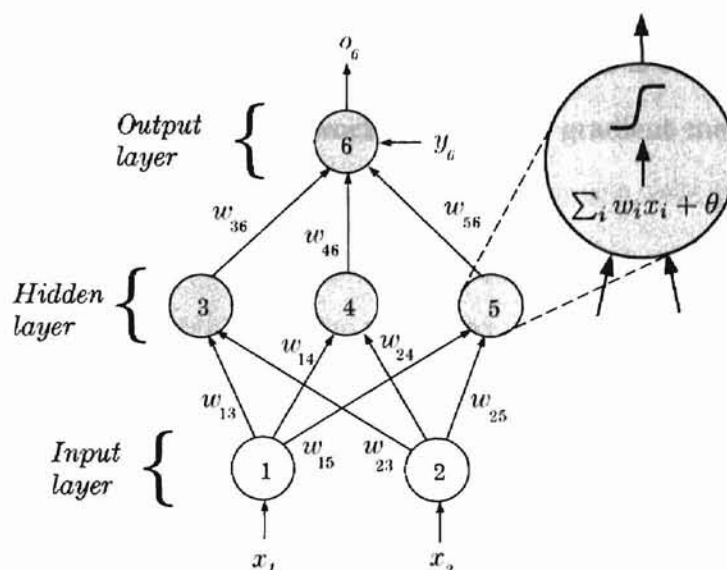


Figure 13. A multilayer feedforward neural network

The notation  $w_{ij}$  denotes the weight connecting neuron unit  $i$  in a previous layer

with neuron unit  $j$ ,  $x_i$  is the input to unit  $i$ ,  $y_j$  is the desired output of the current input, and  $o_j$  is the output value of neuron  $j$ .

There is no clear understanding as to how many hidden layers should be used. Normally, a network of either one or two hidden layers are used in practice. In addition, it has been shown that a one-hidden-layer MLP, if given enough hidden units, is capable of classifying any nonlinear function. Thus a three-layer network is sufficient for handling most classification problems.

### 1.7.3 Backpropagation or Generalized Delta Rule

The backpropagation technique of Rumelhart, Hinton, and Williams [22], has become the most commonly used neural networking algorithm. The term “backpropagation” refers to the way the partial derivatives are efficiently computed in a backward propagating sweep through the network [28]. It is the choice of optimization algorithm used for this experiment because it is very well-studied.

Backpropagation is a simple optimization method, but it is obsolete and probably should never be used for production work. Conjugate gradient methods are always faster and, if properly coded, just as robust.

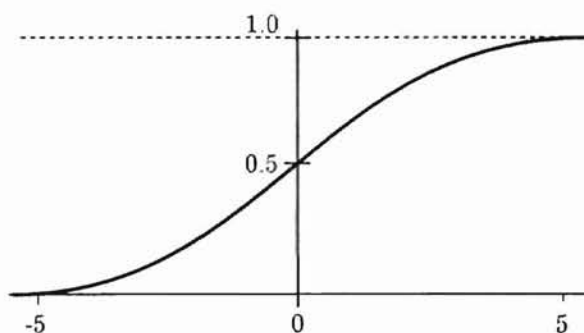


Figure 14. The sigmoid threshold function

The backpropagation algorithm will work with many activation functions. The most commonly used activation function is:

$$o_j = \frac{1}{1 + e^{-net_j}} \quad \forall j \in \text{hidden, output}$$

where  $o_j$  is the output value of the  $j^{\text{th}}$  neuron either in the hidden or in the output layer. The input to neuron  $i$  is denoted by  $x_i$ . The S-shaped activation function depicted in Figure 14 is known as a *sigmoid*. Let  $w_{ij}$  denote the weight connecting neuron  $i$  in a previous layer to neuron  $j$  in the successive layer. Please note that the output of neuron  $i$  is essentially the input to neuron  $j$  such that  $o_i = x_j$ . So the definition of  $net_j$ , the weighted sum of neuron  $j$  is computed by:

$$\begin{aligned} net_j &= \sum_i w_{ij} o_i + \theta_j \quad \forall j \in \text{hidden, output} \\ &= \sum_i w_{ij} x_j + \theta_j \end{aligned}$$

where neuron  $j$  is either a neuron of the output layer or a neuron of a hidden layer, and  $\theta_j$  represents the *bias* value corresponding to the  $j^{\text{th}}$  neuron. Every hidden and output neuron has its own bias value.

In gradient descent, the current value of the weights is moved in the direction in which the expected error falls most rapidly, the direction of steepest descent. The gradient,  $\nabla_w \widehat{MSE}$  can be obtained by using the chain rule. This way, the mean square error can be distributed among all the weights of the network. The first part of the problem is to find how  $\widehat{MSE}$  changes as the weight  $w_{jk}$  changes

$$\begin{aligned} \frac{\partial \widehat{MSE}}{\partial w_{jk}} &= \frac{\partial \widehat{MSE}}{\partial o_k} \frac{\partial o_k}{\partial net_k} \frac{\partial net_k}{\partial w_k} \quad \forall j \in \text{hidden}, \forall k \in \text{output} \\ &= -(y_k - o_k) o_k (1 - o_k) o_j \\ &= -(y_k - o_k) o_k (1 - o_k) x_k \end{aligned}$$

where  $w_{jk}$  denotes the weight connecting the  $j^{\text{th}}$  hidden neuron to the  $k^{\text{th}}$  output neuron, and  $y_k$  denotes the target output for the  $k^{\text{th}}$  neuron in the output layer.

Secondly, we need to relate the change in  $\widehat{MSE}$  to the change in  $w_{ij}$ , the weights that connect neurons from the input layer to the hidden layer. Any change to the weight  $w_{ij}$  changes  $o_j$ , which becomes the input of unit  $k$ . By using the chain rule, the gradient with respect to  $w_{ij}$  is therefore the sum of the changes to the output value of each hidden and output neuron of the network. Let us denote that  $i \in \text{input}$ ,  $j \in \text{hidden}$ , and  $k \in \text{output}$ , then

$$\begin{aligned}
 \frac{\partial \widehat{MSE}}{\partial w_{ij}} &= \frac{\partial \widehat{MSE}}{\partial o_k} \frac{\partial o_k}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial w_k} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} \\
 &= \sum_k -(y_k - o_k) o_k (1 - o_k) w_{jk} o_j (1 - o_j) o_i \\
 &= \sum_k -\delta_k w_{jk} o_j (1 - o_j) o_i \\
 &= -o_i o_j (1 - o_j) \sum_k \delta_k w_{jk} \\
 &= -o_i \delta_j.
 \end{aligned}$$

Therefore, the negative gradient of the sample error,  $\delta_j$ , is computed by a back-propagation process defined by:

$$\delta_j = o_j (1 - o_j) (y_j - o_j), \quad \forall j \in \text{output}$$

$$\delta_j = o_j (1 - o_j) \sum_{k \in \text{output}} w_{j,k} \delta_k, \quad \forall j \in \text{hidden},$$

If  $\delta_j$  is computable, the update equation can be written. The weight update rule is

$$\begin{aligned}
 w_{ij}^{t+1} &= w_{ij}^t + \alpha \delta_j^t o_i^t \\
 &= w_{ij}^t + \alpha \delta_j^t x_j^t
 \end{aligned}$$

where  $\alpha$  is the learning rate, and  $o_i$  is the output value of neuron  $i$  which is equivalent to  $x_j$ . Each neuron in the hidden and output layer has a bias value. The bias of unit  $j$ ,  $\theta_j$ , is adjusted by

$$\theta_j^{t+1} = \theta_j^t + \alpha \delta_j^{t+1}.$$

A momentum coefficient,  $\eta$ , can be used to speed up convergence. This is because it keeps the process moving in a consistent direction. Using momentum, the weight

update formula becomes:

$$\Delta w_{ij}^{t+1} = \eta \Delta w_{ij}^t + \alpha \delta_j^t x_j^t.$$

#### 1.7.4 Batch and Online Training

There are two ways to train a neural network; *batch* and *online* training. Batch or offline training means a set of training examples is obtained and used to approximate the function before it is used in the application. Thus the parameters are updated *only* when the entire training set is presented to the network. Then the gradients of all examples are computed, and the *average* of all the gradients is used to get a better estimate of the gradient [7].

Online training involves continuous weight adjustment of the network by using the data gathered while the system is in operation. This way of training is capable of adapting to a time-varying function. This is essential in temporal difference learning because the target functions change over time. However, there are mixed successes in online applications. This problem exists because the network “forgets” previously learned examples as more new examples are presented for training. In practice, it can be overcome by storing old examples and retraining on them, or by learning slowly and training extensively [31].

#### 1.8 TD( $\lambda$ )

TD( $\lambda$ ) is a multi-step prediction algorithm. This multi-step algorithm makes greater alteration to more recent predictions. This algorithm uses an exponential decay,  $\lambda$ , where predictions  $n$  steps in the past are weighted according to  $\lambda^n$  for  $0 \leq \lambda \leq 1$ . The weight update equation of TD( $\lambda$ ) is given by

$$\Delta w_t = \alpha (P_{t+1} - P_t) \sum_{n=1}^t \lambda^{t-n} \nabla_w P_n$$

where  $\alpha$  is a constant learning rate,  $P_t$  is the prediction at time  $t$ ,  $P_{t+1}$  stands for the prediction made in time  $t + 1$ , and  $\nabla_w P_n$  is the gradient of the  $n^{\text{th}}$  prediction error with respect to the weights.

It is important to note that this TD rule is an offline algorithm. So each  $\nabla_w P_n$  needs to be recorded and the weights are changed at the end of the trial. The case of TD(0) is very like that of conventional backpropagation. TD(0) is essentially a single-step prediction algorithm. The TD error (TDE) is backpropagated to each weight and it is determined only by the most recent observation when  $\lambda$  is 0. The weight update rule is

$$\Delta w_t = \alpha(P_{t+1} - P_t)\nabla_w P_t$$

So each individual weight,  $w_{ij}$ , is updated using

$$\begin{aligned} w_{ij}^{t+1} &= w_{ij}^t - \alpha \sum_{k \in \text{output}} \left( \frac{\partial TDE^t}{\partial o_k^t} \frac{\partial o_k^t}{\partial w_{ij}^t} \right) \\ &= w_{ij}^t - \alpha \frac{\partial TDE^t}{\partial o_j^t} \frac{\partial o_j^t}{\partial w_{ij}^t} \\ &= w_{ij}^t - \alpha \frac{\partial TDE^t}{\partial \text{net}_j^t} \frac{\partial \text{net}_j^t}{\partial w_{ij}^t} \\ &= w_{ij}^t + \alpha \delta_j^t o_i^t. \end{aligned}$$

where  $o_i$  is the estimated output of neuron  $i$ , and  $\delta$  is computed using the backpropagation process discussed in the previous section.

But TD( $\lambda$ ) is treated slightly differently. The process of backpropagation produces an “eligibility” term for each weight. The gradient of each prediction error that is exponentially decayed is known as the *eligibility*. As a temporal difference is determined at each time step, it is *broadcast* to all weights. Then the temporal difference is combined with their eligibility to determine the changes to the weights [28]. In order to reduce the computational resources, the online version of the algorithm is normally preferred. If we have a way to compute the eligibility incrementally, we can

do the computation online. The eligibility,  $e$ , in every time step is computed by

$$\begin{aligned}
 e_{t+1} &= \sum_{n=1}^t \lambda^{t+1-n} \nabla_w P_n \\
 &= \nabla_w P_{t+1} + \sum_{n=1}^t \lambda^{t+1-n} \nabla_w P_n \\
 &= \nabla_w P_{t+1} + \sum_{n=1}^t \lambda \cdot \lambda^{t-n} \nabla_w P_n \\
 &= \nabla_w P_{t+1} + \lambda \sum_{n=1}^t \lambda^{t-n} \nabla_w P_n \\
 &= \nabla_w P_{t+1} + \lambda e_t.
 \end{aligned}$$

If the neural network has  $k$  output neurons, then each weight has  $k$  eligibility traces. Then the  $k^{\text{th}}$  eligibility of the weight from neuron  $i$  to neuron  $j$  is defined as

$$\begin{aligned}
 e_{ijk}^{t+1} &= \frac{\partial P_k^{t+1}}{\partial w_{ij}^{t+1}} \\
 &= \frac{\partial P_k^{t+1}}{\partial \text{net}_j^{t+1}} \frac{\partial \text{net}_j^{t+1}}{\partial w_{ij}^{t+1}} \\
 &= \lambda e_{ijk}^t + \delta_{kj}^{t+1} o_i^{t+1}.
 \end{aligned}$$

where  $o_i$  is the output of neuron unit  $i$ ,  $P_k^{t+1}$  is the new prediction of the  $k^{\text{th}}$  output node, and  $e_{ijk}^t$  is the  $k^{\text{th}}$  eligibility at time  $t$  of the weight connecting unit  $i$  to unit  $j$ .

The  $\delta_{kj}^t = \frac{\partial P_k^t}{\partial \text{net}_j^t}$  is computed using the backpropagation algorithm defined as:

$$\delta_{kj}^t = \begin{cases} o_j^t(1 - o_j^t) & \text{if } k = j \\ 0 & \text{if } k \in \text{output and } k \neq j \\ \sum_{j \in o_i} \frac{\partial P_k^t}{\partial \text{net}_j^t} \frac{\partial \text{net}_j^t}{\partial o_i^t} \frac{\partial o_i^t}{\partial \text{net}_i^t} = \sum_{j \in o_i} \delta_{kj}^t w_{ij}^t o_i^t (1 - y_i^t) & \text{otherwise.} \end{cases}$$

Let us try to understand the equation for updating  $\delta$ . The first and second conditions indicates how to compute  $\delta$  that corresponds to weights that connect to the output layer. So if we are trying to determine the  $k^{\text{th}}$   $\delta$  of a weight that connects to the  $k^{\text{th}}$  output neuron, then  $\delta$  is computed using the first condition. The rest of the  $\delta$ s are set to 0. The  $\delta$  that corresponds to a weight that connects an input neuron to a hidden neuron is calculated by following the third condition. It should be noted

that the equation of TD( $\lambda$ ) for computing  $\delta$  is slightly different when  $\lambda > 0$ . The incremental version of TD( $\lambda$ ) weight update rule is

$$w_{ij}^{t+1} = w_{ij}^t + \alpha \sum_{k \in \text{output}} (P_{t+1}^k - P_t^k) e_{ijk}^t.$$

TD(0) is the focus of this study. It has been shown in Tesauro's TD-Gammon and Thrun's NeuroChess papers that 0 is the optimal value for  $\lambda$ . The TD( $\lambda$ ) algorithm is provided because most of the literature that discusses TD( $\lambda$ ) does not provide the details of the implementation. The details of the algorithm provided above comes from an unpublished paper written by Sutton [27]. This paper by Sutton is a follow-up of his introductory paper to temporal difference learning and TD( $\lambda$ ) [28].

## 1.9 Game Playing

### 1.9.1 General

Many outstanding names in the history of computer science touched upon the domain of game playing. These researchers include Claude Shannon, the father of information theory; Alan Turing, renowned for his contributions to the theory of computation and for his work during World War II in deciphering German war codes; and Herbert Simon, the father of artificial intelligence.

From the very beginning of the development of digital computers, researchers became interested in studying how computers might solve complex problems by examining the game of chess. In 1944, John von Neumann presented the *minimax* algorithm for selecting the move to make in chess [16]. In 1950 Claude Shannon published the paper that detailed the procedures to implement computer chess [26]. Till today, chess programs that are written are based on Shannon's ideas. Simultaneous with Shannon's work, Alan Turing published his approach to the automation of chess strategy [38]. His ideas were very similar to Shannon's. Finally in 1955, Alan Newell,



John Shaw, and Herbert Simon wrote the Newell, Shaw, and Simon (NSS) program that attempted to simulate the human mind's approach to selecting moves in chess [18].

In recent game playing developments, an IBM's supercomputer named Deep Blue defeated the current World Champion and possibly the best ever human chess player, Garry Kasparov, 3.5 to 2.5 in a six-game match [17]. Table III shows the facts about both the contenders. The most shocking news about the match happened in the sixth game, in which Deep Blue defeated the World Champion in just nineteen moves. Figure 15 shows the final position of the historical sixth game.

Table III. Facts about the most publicized chess match between a computer and a human

Facts	Garry Kasparov	Deep Blue
Height	5'10"	6'5"
Weight	176 lbs.	1.4 tons
Age	34 years	4 years
Birthplace	Azerbaijan	Yorktown, NY
Number of processors	50 B Neurons	32 P2SC Processors
Moves per second	2	200 million
Power source	electrical/chemical	electrical
Next career	champion	pharmaceutical design

### 1.9.2 Reinforcement Learning in Game Playing

The one application that is always mentioned for its pioneering success applying reinforcement learning (RL) in game playing is Samuel's checkers playing system [24, 25]. The value function is learned and represented by a linear function approximator, and the training is done similarly to TD updates.

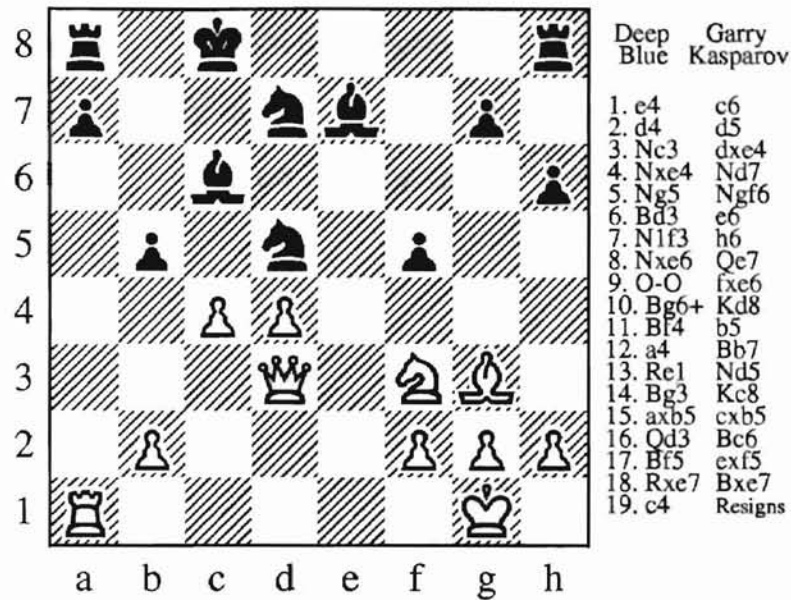


Figure 15. Garry Kasparov versus Deep Blue: Game 6 final position

TD-Gammon is successful in applying TD in learning to play backgammon [32, 33, 34]. Backgammon has approximately  $10^{20}$  states. Tesauro uses a combination of TD and a three-layer ANN with 80 hidden units as the function approximator for generalizing the experience. The successful result was achieved by constant self play. The program always acts greedily in choosing the move with the best chance to win. The success of using this strategy is rather surprising considering all the studies that have been done to discover better exploration methods. The training required several months of computer time for training on 1,500,000 games. This program has competed at the very top level of international human play.

Many researchers have attempted to reproduce the success of TD-Gammon by using the TD learning approach in other games such as chess. Gould [6] implemented Morph by using Adaptive Predictive Search (APS), a learning framework that, given little initial domain knowledge, increases its predictive abilities in complex problem domains such as chess. The result shows that the GnuChess [14] level of play is higher than Morph. Sebastian Thrun [35] combined TD learning and Explanation

Based Neural Network (EBNN) to train NeuroChess. The EBNN learning algorithm [36] enables the computer to learn more accurately from less training data by taking advantage of other previously acquired knowledge, even if it is inexact, to significantly improve accuracy for the new learning task. His research shows that EBNN's generalize better than ANN's using backpropagation. The main drawback of NeuroChess is that it does not develop good chess openings. Jonathan Baxter [2] created KnightCap, a chess program that learns by combining  $TD(\lambda)$  with minimax search. This program learns to play only the middle and end games. KnightCap selects chess openings from a database. These results are not very successful compared to TD-Gammon. Several very successful chess-playing programs have been developed over the years [12, 11, 17]. However, none of these programs used reinforcement learning.

## CHAPTER II

### RESEARCH OBJECTIVE AND METHODOLOGY

#### 2.1 Research Objectives

The main objective of this research is to conduct a study in the area of reinforcement learning. This study attempts to understand the internal workings of an RL agent learning to act. This is accomplished by implementing the reinforcement learning framework and experimenting with it in the domain of game playing. The learning mechanism that is of interest in this study is the temporal difference learning method.

Tic-tac-toe is commonly used to begin the understanding of reinforcement learning method in the domain of game-playing. Tic-tac-toe is a small-size problem with  $3^9$  states in its state space. It is used to answer most of the questions of this study. The first area of concern is to find out how learning is affected by the learning rate, the discount rate, the exploration rate, and by the difference in off-policy method and on-policy learning method. The agent is implemented to play many games of tic-tac-toe by trial and error. Then its performances are recorded to conduct the analysis. In this part of the study, the  $Q$ -value of each state-action pair is stored in a lookup table.

This research then switches its attention to examine the effectiveness of approximating the value function using a multilayer feedforward network trained by the

backpropagation algorithm. It is not the intention of this research to optimize the learning rate of the backpropagation algorithm. The goal of this part of the research is to know how reinforcement learning and a function approximator are combined to solve sequential decision problems. This is important because generalization scales up reinforcement learning to solve practical problems that have a large state space. It is the objective in this part of the study to find out whether an RL agent will learn an optimal policy. Analysis will be carried out to find factors that could lead to the success of generalized reinforcement learning.

## 2.2 Details of Implementation

Simulation is the evaluation tool for this research. The simulation is written in C++ to utilize the power of object-oriented programming. It allows convenient interaction between the various modules of the program, as well as for run-time speed. I am using Microsoft Visual C++ 4.0 as the tool for coding and debugging.

The class diagram of this experiment is shown in Figure 16:

1. **SIMULATOR:** The main function of this module is to provide the user with a user interface to configure the experiment model.
2. **RL FRAMEWORK:** A module that models the framework shown in Figure 1. Learning through trial-and-error is done here. The learning algorithms are  $Q$ -learning and SARSA.
3. **POLICY:** In this module, the agent decides which action to take, given the state that it is in at a given time, by following its behavioral policy. The behavior policy that is implemented is  $\epsilon$ -greedy.
4. **ENVIRONMENT:** This is the opponent that the agent is playing against. After the opponent has chosen its move, it returns two things to the agent:

the subsequent state that the system has evolved into, and the reward of the agent's move selection. The opponent used in this experimentation for learning tic-tac-toe is discussed in further detail in Section 2.3.

5.  **$Q$ -value:** The  $Q$ -value is either stored in a lookup table or it is approximated by using an ANN. *Double hashing* [40] was implemented for storing and looking up the  $Q$ -values. The multilayer neural network using the backpropagation algorithm was implemented using source code from Rogers's book [20]. The ANN and the lookup table are both subclasses of  $Q$ -value. This module returns and modifies the  $Q$ -value of each state-action pair.

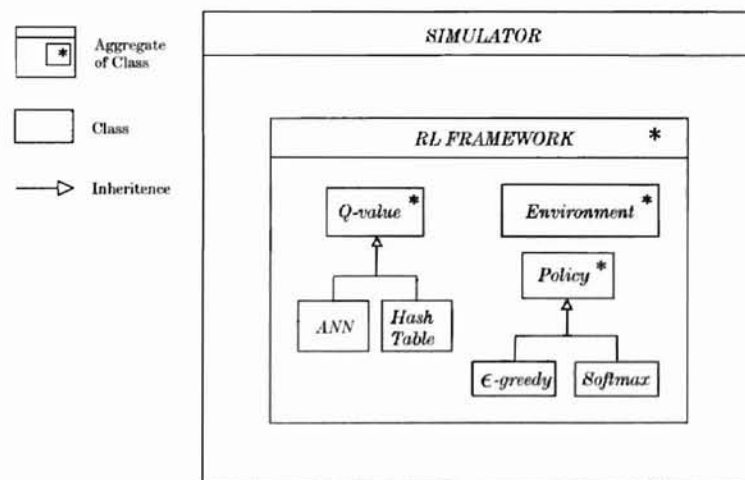


Figure 16. Reinforcement learning simulation class model

### 2.3 Details of the Opponent

The tic-tac-toe opponent is a minimax player that searches two plies for a move. The minimax algorithm has random noise added to its move selection, thus causing it to lose games by missing blocks or missing the winning moves. The stochastic property of this opponent should be ideal for testing reinforcement learning effectiveness in

making decisions *under uncertainty*.

This opponent is put to a test by playing against a random move generator. The performance of the opponent is averaged over ten thousand games. The *equity* is the expected value of the number of losses subtracted from the number of wins. The skill of this opponent is shown in Table IV.

Table IV. Performance of the tic-tac-toe opponent that is used in the experiments. Its performance is tested by playing against a random move generator

Opponent	Wins	Draws	Losses	Equity
Minimax 2-ply with random exploration	70.51	8.63	20.86	49.65

## 2.4 Details of Neural Network Training

The update equation for TD learning used in value function approximation is in accordance with Thrun's [37] training strategy where:

$$Q(s, a) = \begin{cases} +1 & \textit{win} \\ 0 & \textit{draw} \\ -1 & \textit{loss} \\ \gamma \max_{(s', a')} Q(s', a') & \textit{otherwise} \end{cases}$$

It is not the primary goal of this project to fine-tune the network parameters for the fastest learning rate possible. These are the common settings used in the experiments:

- Network weights are initialized randomly between -0.1 and 0.1.
- Weights of the network are adjusted online.
- The original backpropagation algorithm is used. No momentum coefficient is used.

- Each network has one layer of hidden units.
- The sigmoid function is used as the activation function for both the hidden and output layer. The sigmoid function scales its output value in  $[0, 1]$ . In order to scale the reward signal which is in the range of  $[-1, 1]$ :

- Before training the network, the  $Q$ -value in  $[-1, 1]$  is scaled down to the range  $[0, 1]$  before it can be used as the target (denoted by  $o$ ) using

$$o = \frac{Q + 1}{2}.$$

- The output of the network is scaled back into the range of  $[-1, 1]$  before it is used for TD learning:

$$Q = 2o - 1.$$

A multilayer neural network is used to learn the mapping from each state-action pair to its  $Q$ -value. Two input units are used to represent one tic-tac-toe square; 00, 01, and 10 for ' ', 'O', and 'X' respectively. The  $Q$ -value of each action is represented by nine output neurons, where each neuron represents a tic-tac-toe square. Each time, only one of the nine neurons is updated according to the square which is played. Then the training examples are fed to the network to approximate  $Q^*$ , as shown in Figure 17.

Studies have shown that feature selection is important in the effectiveness and success of representing the problem [35]. There are eight different positions in which a player can win a game in tic-tac-toe. The player either wins in one of the three columns, one of the three rows, or one of the two diagonals. Since these are the key features that we look for in a game, this information is used as the input. This neural network uses 00, 01, and 10 to represent ' ', 'O', and 'X'. Then, it uses 48 neurons to represent all eight different winning positions whereby six neurons are used to represent each winning position. These inputs are mapped to nine output



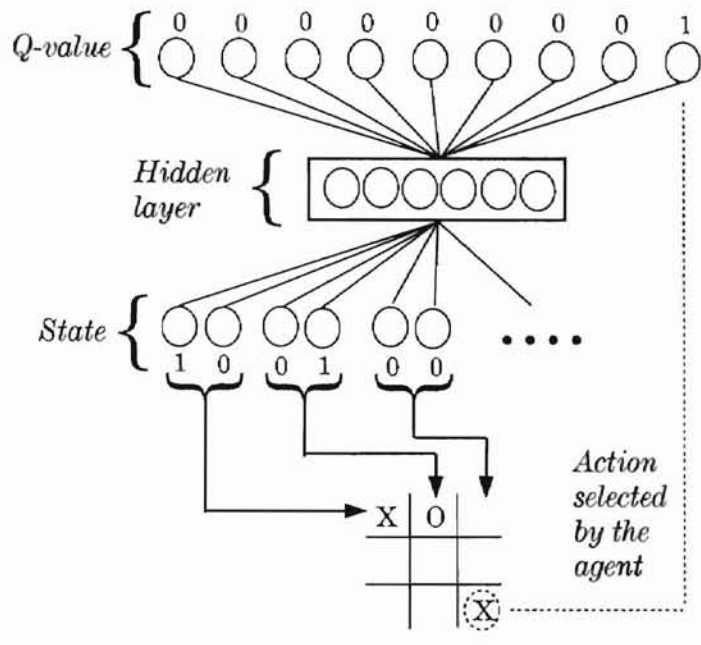


Figure 17. Evaluation function neural network for a tic-tac-toe example

nodes. Each output node represents the  $Q$ -value of one possible action. The results of using this representation are reported in Chapter III.

### 2.5 Method of Analysis

Data were gathered by running the simulation. The data include the wins, losses, draws, mean square error, and equity. The agent's ability to win is measured by its equity. The performance is analyzed by generating charts using these data.

The results were gathered when the agent halted learning temporarily and played one hundred games greedily. It is important to halt learning temporarily because there are explorations involved during the learning process. These data were gathered after every one thousand games of training. The effectiveness of using generalization in reinforcement learning is determined by using results from the lookup table as the benchmark.

## CHAPTER III

### EMPIRICAL RESULTS

#### 3.1 Bounded Random Walks

To test whether both  $Q$ -learning and SARSA actually converge to the optimal policy, a simple test was conducted with a bounded random walk. In this experiment, there are seven states that the agent can be in. The agent starts at state 4 and takes a step to a neighbor randomly. A positive reward of  $+1$  is assigned if the agent reaches state 7, and a negative reward of  $-1$  is assigned for ending the walk at state 1. This problem is shown in Figure 18.

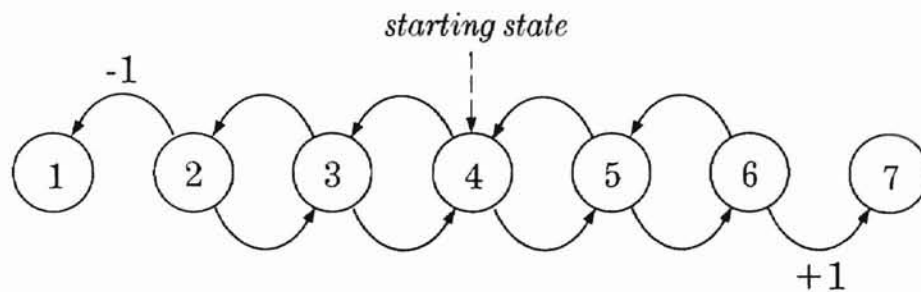


Figure 18. Bounded random walks

The  $Q$ -values of each state-action pair are shown in Table V and Table VI after 1000 trials.

Table V. Q-values generated by SARSA

State/Action	Left	Right
4.	-0.0811	0.0559175
3.	-0.260868	-0.0120768
2.	-0.921006	-0.0876725
5.	-0.00294266	0.264135
6.	0.0762577	0.921555

Table VI. Q-values generated by Q-learning

State/Action	Left	Right
4.	0.466433	0.473384
3.	0.460682	0.463019
2.	-0.927747	0.46221
5.	0.462804	0.463898
6.	0.481388	0.928404

Both SARSA and Q-learning have learned the optimal policy. However, the Q-values produced by Q-learning are somewhat misleading. This is because it only punishes the action for walking to the left from state 2 to state 1. The rest of the Q-values are all positive because of its *max* operator. On the other hand, the Q-values generated by SARSA are more informative. Starting from state 4, a negative  $Q(4, 3)$  of -0.0811 tells us that we will probably ended up getting punished for walking to the left; and  $Q(4, 5)$  of 0.056 tells us that the agent is more likely to be rewarded by choosing to walk to the right.

## 3.2 Tic-Tac-Toe

The  $Q$ -values of experiments 1, 2, 3, and 4 are stored in a lookup table. Using the actual  $Q$ -value allows us to observe the effects of changing the parameters of temporal difference learning. Experiment 5 approximates the  $Q$ -value using an ANN trained with backpropagation. It uses the network architecture that is illustrated in Section 2.4.

### 3.2.1 Experiment 1. Learning Rate

The first experiment tested the learning rate. SARSA was selected to learn the game of tic-tac-toe. The agent played against a minimax opponent that searched 2-ply deep for a move. It followed the  $\epsilon$ -greedy behavior policy to handle exploration. The exploration rate was set to 0.1 and a total of 50,000 games were played by the agent, using two different learning rates of 0.1 and 1.0. Figures 19 and 20 show how the learning curve was affected by the learning rate,  $\alpha$ .

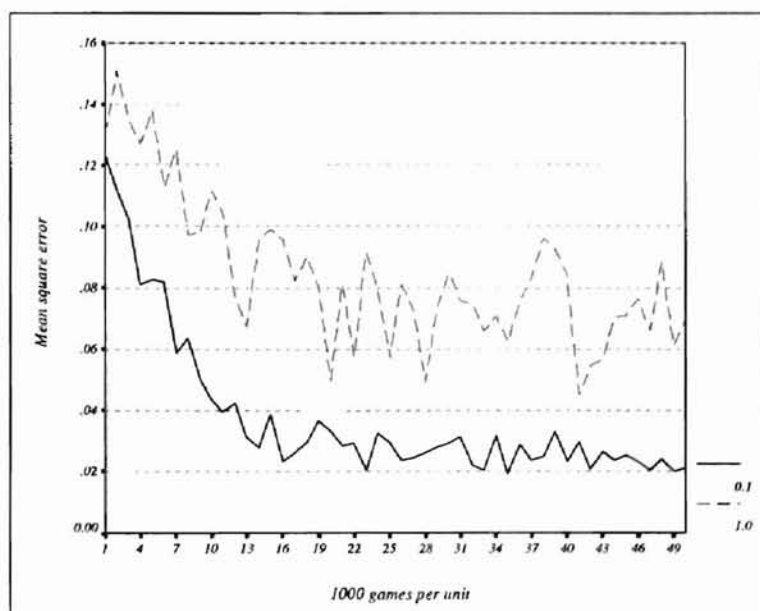


Figure 19. Mean square error: Effects of learning rate on the performance

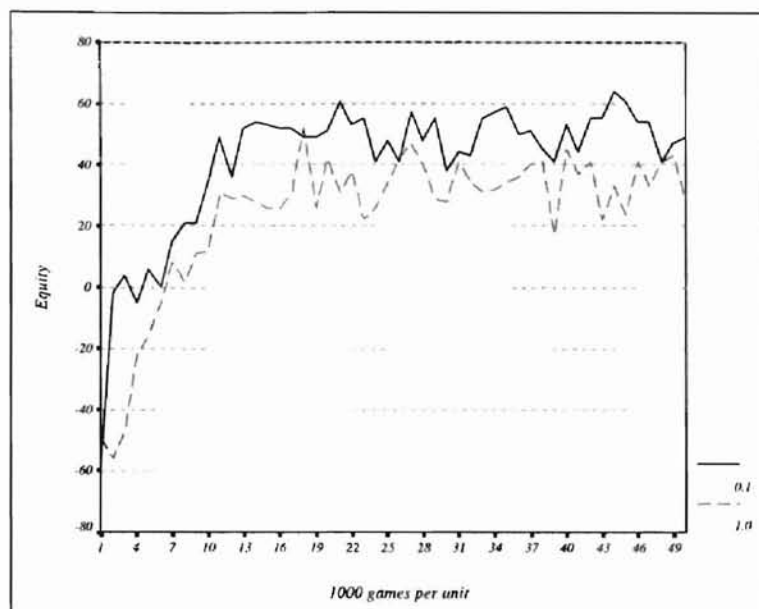


Figure 20. Equity: Effects of learning rate on the performance

When the learning rate was set to 1.0, each  $Q$ -value added the actual temporal difference between  $Q(s_t, a_t)$  and  $Q(s_{t+1}, a_{t+1})$  instead of a fraction of the temporal difference. So,  $Q(s_t, a_t)$  was replaced by  $Q(s_{t+1}, a_{t+1})$  every time  $Q(s_t, a_t)$  was backed up. When the agent explored, a positive  $Q(s_t, a_t)$  could be replaced by a negative  $Q(s_{t+1}, a_{t+1})$ . This is the reason that caused the fluctuations of the mean square error.

### 3.2.2 Experiment 2. Discount Rate

In Section 1.5.1, we say that the usage of a discount rate in the update equation should *only* affect the agent's preference for policies. This also says that it should not affect the agent's performance. An experiment was carried out to verify if this statement holds. The agent played 50,000 games. The learning algorithm is SARSA. The objective in this experiment was to compare undiscounted learning with discounted learning. The learning rate used was 0.1, and the exploration rate was set to 0.1. The

discount rates used were 0.8 and 1.0. Figure 21 and Figure 22 show the equity and the mean square error for both discount rates.

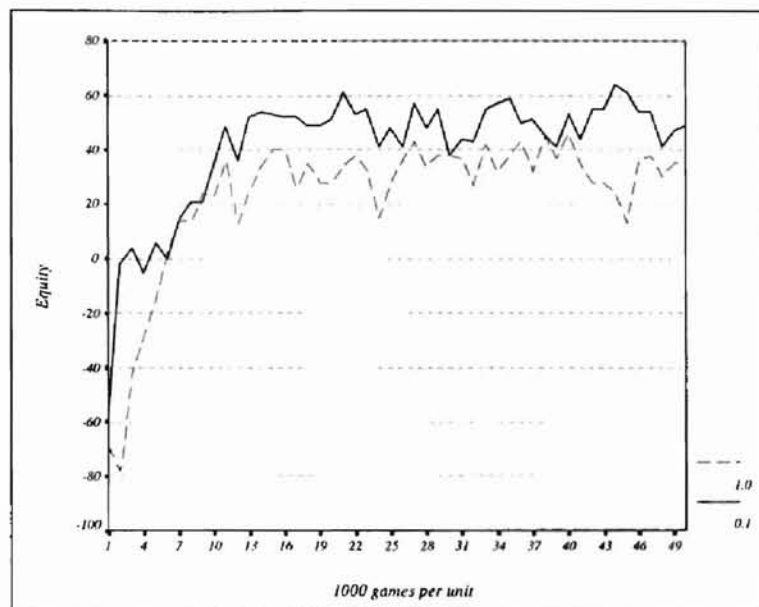


Figure 21. Equity: Discounted versus nondiscounted learning

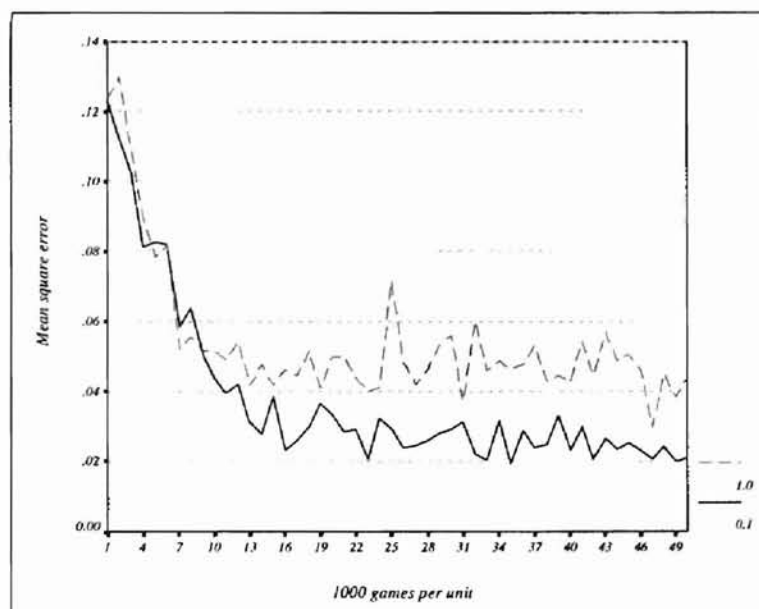


Figure 22. Mean square error: Discounted versus nondiscounted learning

The tic-tac-toe opponent used in this experiment uses a minimax search algorithm

to determine its moves. However, the algorithm has incorporated random moves. This randomness caused the opponent's move selection to be nondeterministic. Using the method of temporal difference learning, the  $Q$ -value of each action will converge to its asymptotic  $Q$ -value if sufficient training time is provided. By not discounting the  $Q$ -value, the agent does not prefer moves that win the game in fewer moves. So when the agent does not prefer to win the game immediately, it indirectly increases the chances for its opponent to settle for more ties. That is the reason why the agent that used a discount rate has better performance. Therefore, the use of a discount rate in a certain class of problems may greatly improve the solution.

### 3.2.3 Experiment 3. Exploration Rate

Exploration enables the agent to discover better decisions given that such decisions exist. When the exploration rate was varied, there was an unexpected result. The experiment used an  $\epsilon$ -greedy strategy as the exploration strategy to test the performance difference between a greedy agent and an agent that explores. Figures 24 and 23 show the differences in the learning curve between an agent that explores ( $\epsilon=0.1$ ) and an agent that does not ( $\epsilon=1.0$ ). The learning rate was 0.1, the discount rate was 0.8, and the learning algorithm was SARSA.

The information shown in the graphs is misleading. If you look at the equity graph, it seems as though the agent that does not explore performed better than the agent that does. The mean equities obtained by averaging the equity of the last 20,000 games are 61.0 and 50.48 for the greedy agent and non-greedy agent respectively. Without exploration, the equity is higher because the greedy agent has only experienced a small subset of the state space, whereas, when the agent explores, it is learning to act in all possible states. Thus, if both agents are allowed to play against a human player after training, the greedy agent will lose more games because it

has many states that are left unexplored. So exploration is a key factor in determining the overall performance of the agent.

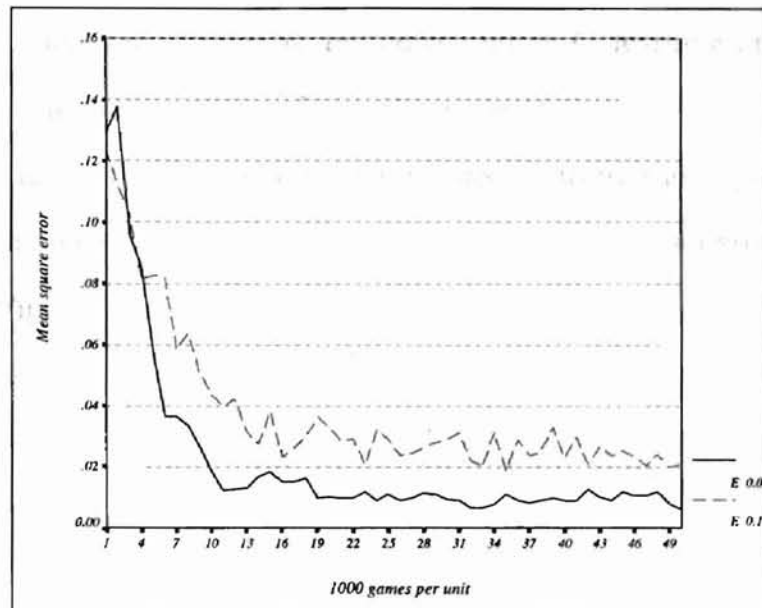


Figure 23. Mean square error: Exploration versus exploitation

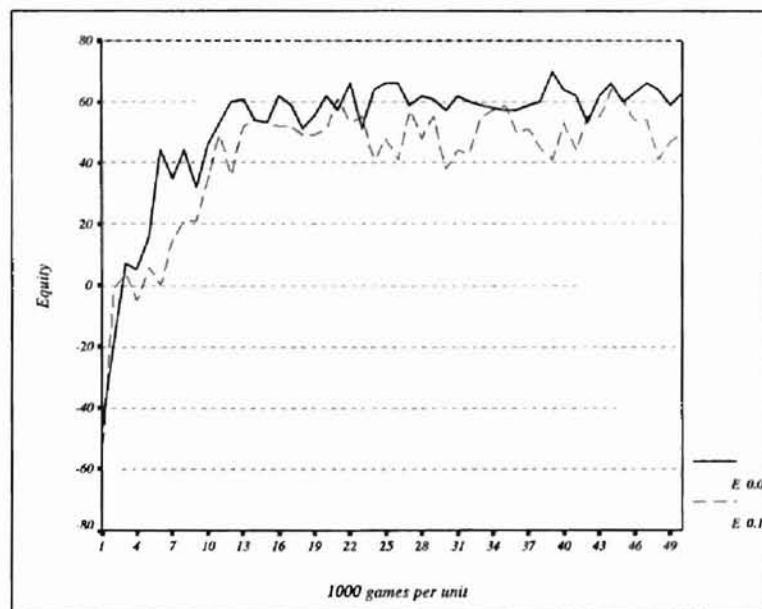


Figure 24. Equity: Exploration versus exploitation



### 3.2.4 Experiment 4. $Q$ -learning Versus SARSA

This section reports the results obtained by comparing  $Q$ -learning with SARSA. This was an effort to find out if there is any advantage in  $Q$ -learning and its variation, SARSA. The agent was trained to learn the game of tic-tac-toe. In both cases, the learning rate was set to 0.1, exploration rate was 0.1 using the  $\epsilon$ -greedy exploration strategy, and discount rate was 0.8. The equity and the mean square error charts are illustrated in Figure 25 and 26.

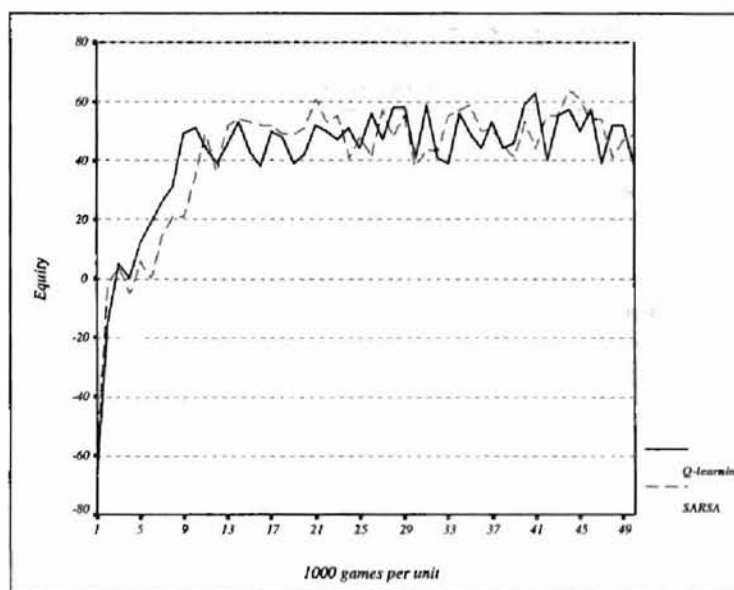


Figure 25. Equity:  $Q$ -learning versus SARSA

$Q$ -learning and SARSA were expected to learn the optimal policy. Indeed, the results show that there are virtually no differences in their performances after they stabilize. To prove this, the equity of both  $Q$ -learning methods were averaged before and after 25,000 games of training. The average equity of the first 25,000 games shows us the agent's speed of learning to win of both algorithms. The remaining 25,000 games were used to determine the mean equity of both  $Q$ -learning methods after the equity stabilized. The results are shown in Table VII. In both cases, the equity increases at almost the same rate. The mean equity of both  $Q$ -learning for the

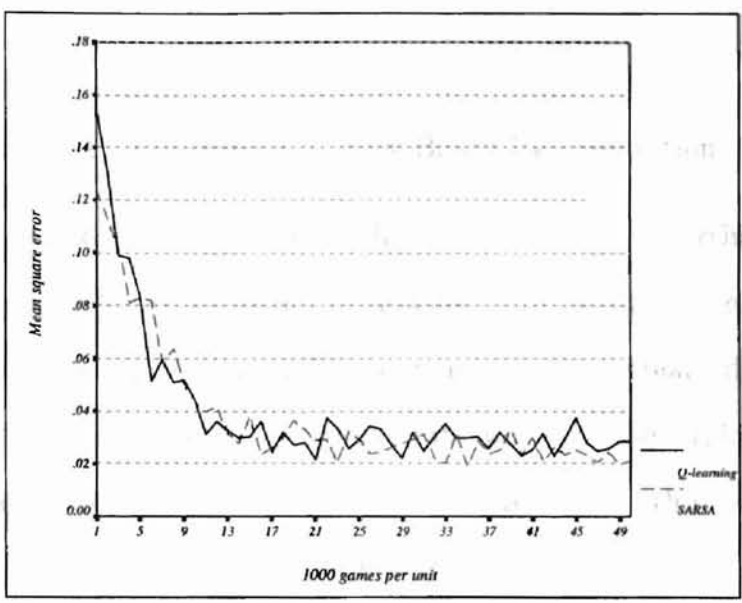


Figure 26. Mean square error: Q-learning versus SARSA

remaining 25,000 games differs only by 0.47. The mean square error of Q-learning and SARSA were properly reduced at a similar rate too. Thus, there is no visible advantage of using either one of them over another.

Table VII. Q-learning versus SARSA: Mean Equity of the first and next 25,000 games

Games	1 - 25,000	26,000 - 50,000
Q-learning	31.37	49.88
SARSA	30.96	50.35

### 3.3 Function Approximator

This section discusses the results obtained by applying a multilayer feedforward neural network using the backpropagation algorithm to approximate the value function. By doing this, we hope to obtain a near-optimal value function. The results obtained by generalizing the value function with different network architectures are provided and

discussed.

### 3.3.1 Experiment 1. Raw Board Representation

The first experiment dealt with using a 3-layer 18-15-9 network architecture to learn the mapping from states to  $Q$ -values of actions. The input to the neural network was the raw board representation of the game in progress (explained in Section 2.4). The learning rate was set to 0.1, discount rate was 0.8, the exploration rate was 0.1, and the agent played 500,000 games. The learning algorithm was SARSA. The objective of this experiment was to find out the efficiency of using this method and the performance of the agent.

As shown in Figure 28, the average mean square error over the last 100,000 games was 0.059. It stabilized after approximately 250,000 games of training.

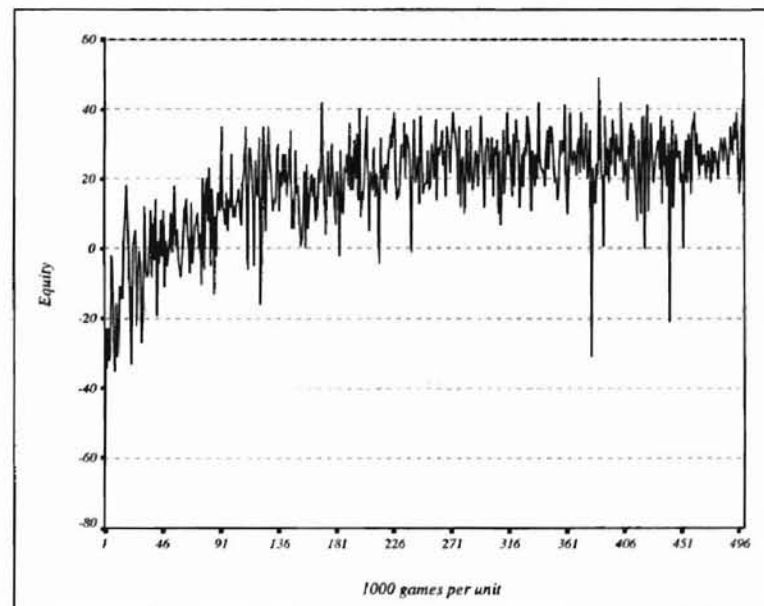


Figure 27. Equity: An 18-15-9 network trained with backpropagation is used to approximate the value function

Comparing the equity obtained by using a lookup table, the equity obtained in this experiment is very much lower than with the lookup table. The equity of this

experiment is shown in Figure 27. Averaging over the last 50,000 games of this experiment, the mean equity was 26.98. This was significantly lower compared to the mean equity of using a lookup table which was approximately 50.

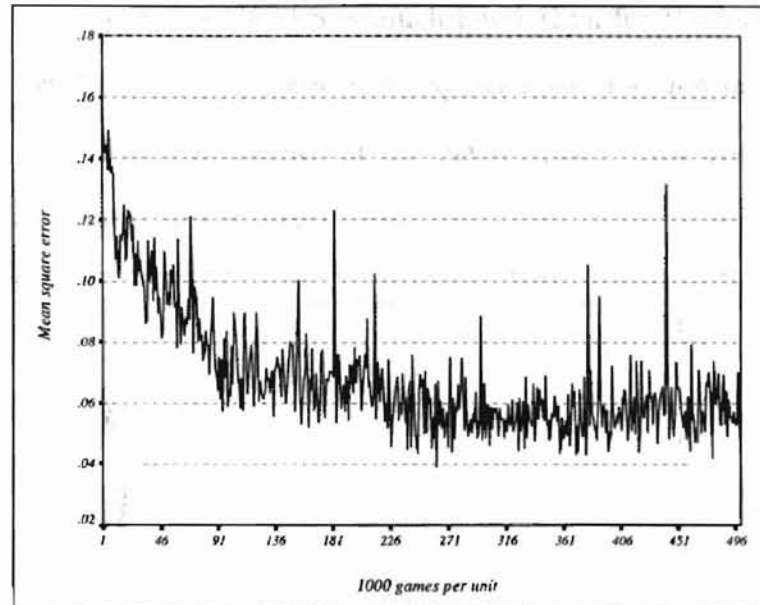


Figure 28. Mean square error: An 18-15-9 network trained with backpropagation is used to approximate the value function

Secondly, the gradient descent method for approximating the value function is slow. It takes close to 350,000 games to reach its maximum equity around 25, whereas when a lookup table was used to store  $Q$ -values, the agent reached the equity of 25 in less than 10,000 games of training. In this case, generalization did not speed up learning and it took a significantly higher number of games to learn a sub-optimal solution using this representation.

### 3.3.2 Experiment 2. Feature Selection

This section reports the findings about using all eight winning positions in tic-tac-toe to represent the input. Let us call this network NETFS and the network that uses raw board representation NETBP. In each case, the agent was trained for 150,000

games. The learning rate was 0.1, the exploration rate was 0.1, and the discount rate was 0.8. The learning algorithm was SARSA. The results of the lookup table, NETFS, and NETBP are compared in this section.

The mean square error of NETFS is much lower than for the lookup table which is shown in Figure 29. However, lower mean square error does not mean that a better policy was learned. The convergence of the mean square error *does not* necessarily imply that an optimal policy is learned [5].

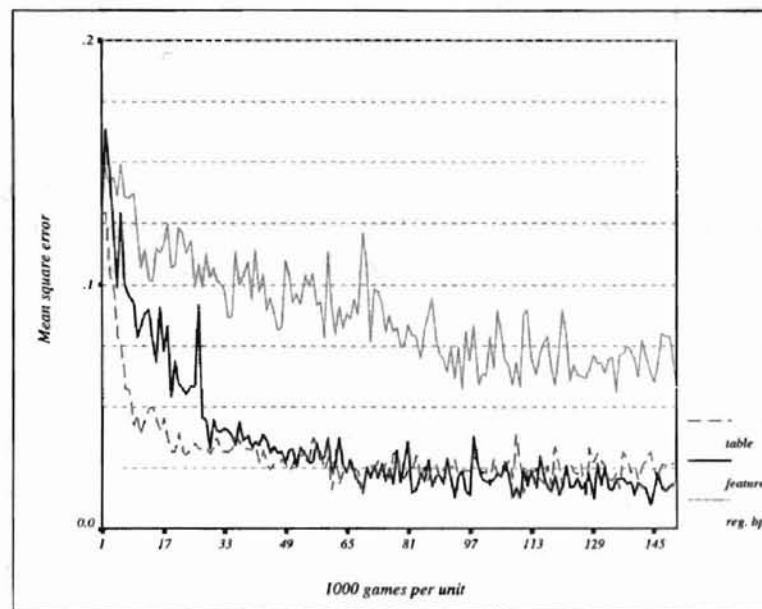


Figure 29. Mean square error of a network in which the input representation incorporated hand-selected features and was trained with backpropagation

Figure 30 compares the equity of the lookup table, NETBP, and NETFS. Averaging the equity of the last 25,000 games, the averaged equity of the lookup table is 17.03 units higher than NETFS. On the other hand, NETFS reached higher equity than NETBP using this representation.

However when we look at the number of losses of the lookup table and NETFS, the numbers are pretty close. This is shown in Figure 31. NETFS resulted in more draws than the lookup table. NETFS has learned a near-optimal policy compared to

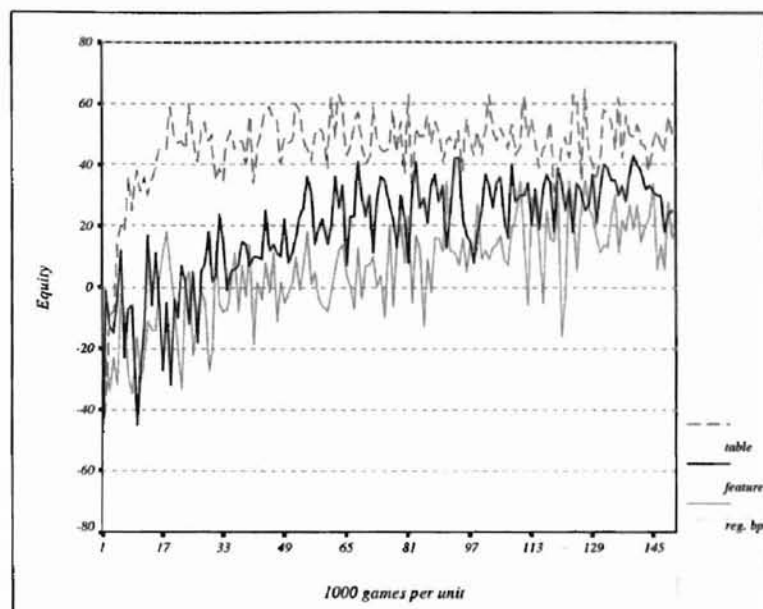


Figure 30. Equity: A network in which the input representation incorporated hand selected features (*feature*) and was trained with backpropagation compared to ANN using raw board representation (*regbp*) and lookup table (*table*)

the policy obtained by the lookup table. In another comparison, NETFS learns significantly faster and it has better performance than NETBP. This experiment shows that the selection of input features to represent problems plays a major role in determining the success of approximating the value function.

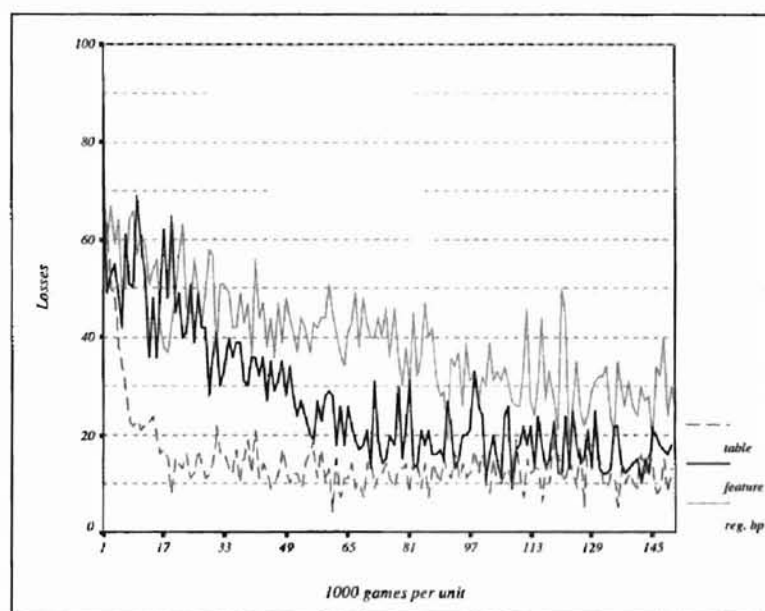


Figure 31. Number of losses: Comparing the results obtained by the lookup table to a network in which the input representation used hand-selected features

## CHAPTER IV

### SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

#### 4.1 Summary

This research studies the internal workings of reinforcement learning. Reinforcement learning solves sequential decision problems through trial and error. This gives it the upper hand for its ability to learn in real time. The learning algorithms of this experimentation were  $Q$ -learning and a variation of it called SARSA. Both these learning mechanisms are one-step learning prediction algorithms that learn a value function, a mapping from a state-action pair to a real number. They update a prediction at time  $t$  to be a fraction closer to the prediction at time  $t + 1$ . These algorithms are considered special cases of a multi-step prediction algorithm known as  $TD(\lambda)$ . At every time step,  $TD(\lambda)$  updates a prediction by exponentially decaying previous predictions based on recency, where  $\lambda$  is the decay factor.  $Q$ -learning and SARSA are essentially  $TD(0)$ , a one-step prediction algorithm in which  $\lambda$  is 0. TD is the abbreviation for temporal difference learning.

In this study, temporal difference learning was used to learn the game of tic-tac-toe. The  $Q$ -values were either stored in a lookup table, or approximated by a multilayer neural network using backpropagation algorithm. The effects of the learning rate, the discount rate, the exploration rate, and both forms of  $Q$ -learning were



experimented with. In these experiments, the  $Q$ -values were stored in a lookup table. If a function approximator was used, then the objective was switched to analyzing the effectiveness and efficiency of generalization in reinforcement learning.

When using a lookup table to store the  $Q$ -values, the results of learning tic-tac-toe were successful. The constant learning rate did not affect the solution except when the learning rate was set close to 1. This was less effective because we want each prediction to be a fraction closer to the next prediction, not taking on the value of the next prediction. Otherwise, a good action could be updated to become a bad action when the next prediction is an exploratory move. The second test dealt with examining discounted versus nondiscounted learning. The effect of a discount rate should affect only the selection of the optimal policies. By using a discount rate, the tic-tac-toe agent prefers actions that leads to a win in the least number of moves. However, this experiment showed that using a discount rate generates a better policy. By not having a preference on the winning strategy when no discount is used, increasing the length of the game inadvertently increases the probability of not winning the game. This is why a discount rate should be used in learning to win in tic-tac-toe.

The third experiment dealt with the exploration rate. A test was conducted to experiment with the outcome of using the greedy policy versus a ten percent exploration rate in its decision makings. When no exploration was used, the result shows a very misleading statistical report. The mean square error was reduced significantly faster compared to the latter. Secondly the results show that the greedy agent actually learned to win at tic-tac-toe better than the agent that explores. However, this does not mean that no exploration is a better strategy. If the agent does not explore, it leaves a lot of the state space unexplored. So when the greedy agent is put into a test after training by playing against a better player, say a human player who will not lose, it will be clear that the agent that explores is a better overall player. Thus

exploration is an important factor in determining the overall performance of an agent.

The second half of the experiment dealt with using a multilayer feedforward network that was trained by the backpropagation algorithm to approximate the value function. A function approximator is commonly used to generalize the value function because it may speed up learning. In addition, the ability to generalize also provides the ability to scale up reinforcement learning for solving problems that have large state spaces. In the first experiment, the input was encoded using the raw board representation. The network then mapped the state of the game to the action's  $Q$ -value. The empirical results showed that it was learning much slower when it is compared to storing the  $Q$ -values in a lookup table. It took literally 3000 percent additional games to reach the performance attained by table  $Q$ -learning. In the second experiment with a generalized value function, the tic-tac-toe board was represented by eight winning positions of tic-tac-toe as the input. By mapping the winning positions of tic-tac-toe to each action's  $Q$ -value, the agent was capable of learning a comparable policy to the results of table  $Q$ -learning. Although this representation of tic-tac-toe reduced the mean square error faster than the table  $Q$ -learning, table  $Q$ -learning had a slightly higher equity. This finding suggests that selection of features for input representation of the problem is critical towards the success of generalized reinforcement learning.

## 4.2 Conclusions

The experiments conducted in this study show that applying  $Q$ -learning in learning the game of tic-tac-toe is indeed successful. The optimal policies of the agent beat the opponent close to seventy percent of the time in just 50,000 games of training. The selection of the discount rate, learning rate, and the exploration rate affects the policy that is learned by the agent.

Experimental findings about combining a multilayer feedforward network using backpropagation and  $Q$ -learning show that it is not very reliable. Different results can be obtained if different representation of the same problem is used. It needs a lot of experience and understandings of the tricks and tips in this field of study in order to make them to work together more successfully. In my experiments, when a function approximator is used, the tic-tac-toe agent learned to win at a pace similar to table  $Q$ -learning. After 10,000 games of training, it reached a point as if it had stopped learning. At this point, the agent lost half of its games. It was discovered later that the agent was actually reducing the mean square error at a very slow pace. Without the analysis of the mean square error, I would not have realized that the agent was improving its value function slowly. After an additional 300,000 games of training, the agent improved its equity from approximately 0 to approximately 25. However it was learning very slowly. It will be beneficial to look at some of the algorithms that are designed to speed up backpropagation. My final conclusion concerning generalized reinforcement learning is that it needs a lot of experimentation to test with different network architectures to represent the same problem, and analyzing the mean square error during the learning process are keys to a better success in the generation of a better policy.

### 4.3 Recommendations

Here are some suggestions for potential future research:

1. Find algorithms that can speed up the backpropagation algorithm and implement it to generalize the value function. Use them to train an RL agent that learns to win tic-tac-toe and several other sequential decision problems. Compare the results to see if every approach produces the optimal solution.
2. First find optimization algorithms for backpropagation. Use them to train an

- RL agent to play tic-tac-toe. If they are successful, then apply this method in the domain of backgammon, chess, or go. Find out how other researchers select features that they think are important in one of those games. Then use those hand-selected features to represent the input and use it to learn that game.

#### 4.4 Concluding Comment

This study provides a close-up understanding of the internal workings of reinforcement learning methods. The empirical results obtained in the experiments are successful when a lookup table is used to store the  $Q$ -value. This method of learning through trial-and-error proved to be very robust *only* when a lookup-table was used, and using a lookup table is prohibitively costly for games with a large state space such as chess.

In my opinion, even though reinforcement learning and supervised learning are considered as two separate fields of study, it seems like the understanding of both fields is compulsory when studying reinforcement learning. I am saying this because most published literature that I have read was concerned with the scalability of reinforcement learning. Researchers hope that the reinforcement learning mechanism can be integrated with different function approximators more successfully in solving large real world problems. Many papers were published that proved the convergence of temporal difference learning algorithms used with different function approximators. But convergence does not mean that an optimal policy is learned. Local function approximators such as CMAC, and radial basis networks are recommended to be applied together with TD learning. However CMAC is not fully scalable even though hashing provides better and more efficient memory usage. Alternative scalable global function approximators such as an artificial neural network trained by the backpropagation algorithm showed that this method of generalizing the value functions may produce only sub-optimal solutions, or in the worst case, may diverge. It is still an on-

going research area to discover better approaches to scaling up reinforcement learning.

## REFERENCES

- [1] D. H. Ackley and M. L. Littman. Generalization and scaling in reinforcement learning. *Advances in Neural Information Processing Systems*, 2:550–557, 1990.
- [2] J. Baxter, A. Tridgell, and L. Weaver. Knightcap: A chess program that learns by combining TD( $\lambda$ ) with minimax search. Technical report, Australian National University, Department of Systems Engineering, November 1997.
- [3] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.
- [4] R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing Systems*, 8:1017–1023, 1996.
- [5] G. J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 261–268. Morgan Kaufmann, San Francisco, CA, 1995.
- [6] J. Gould and R. Levinson. Machine learning: A multistrategy approach: Experience-based adaptive search. *Advances in Neural Information Processing Systems*, 4:579–603, 1994.
- [7] M. T. Hagan, H. B. Demuth, and M. Beale. *Neural Network Design*. PWS, Boston, MA, 1996.
- [8] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, November 1994.
- [9] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
- [10] S. Kirkpatrick, C. D. Gelatt, and Vecchi M. P. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [11] D. N. L. Levy and M. M. Newborn. *All About Chess and Computers : Containing the Complete Works, Chess and Computers*. Computer Science Press, Potomac, MD, 1983.

- [12] D. N. L. Levy and M. M. Newborn. *How Computers Play Chess*. W.H. Freeman and Company, New York, NY, 1991.
- [13] S. Mahadevan and L. P. Kaelbling. The National Science Foundation workshop on reinforcement learning. *AI Magazine*, 17(4):89–97, 1996.
- [14] T. Mann. Gnu chess and XBoard: Frequently asked questions. URL: [http://www.research.digital.com/SRC/personal/Tim\\_Mann/gnuchess/FAQ.html](http://www.research.digital.com/SRC/personal/Tim_Mann/gnuchess/FAQ.html). Newsgroup: *gnu.chess*. (Date accessed: Nov. 1998)
- [15] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.
- [16] J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton Univ. Press, Princeton, NJ, 1944.
- [17] M. M. Newborn. *Kasparov Versus Deep Blue : Computer Chess Comes of Age*. Springer-Verlag, New York, NY, 1996.
- [18] A. Newell, J. Shaw, and H. Simon. Chess-playing programs and the problem of complexity. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pp. 39-70. McGraw-Hill, New York, NY, 1963.
- [19] M. L. Puterman. *Markov Decision Processes*. John Wiley and Sons, New York, NY, 1994.
- [20] J. Rogers. *Object-Oriented Neural Networks in C++*. Academic Press, Huntsville, AL, 1997.
- [21] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.
- [22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6038):533–536, 1986.
- [23] S. J. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.
- [24] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.
- [25] A. L. Samuel. Some studies in machine learning using the game of checkers II. *IBM Journal of Research and Development*, 11(6):601–617, 1967.
- [26] C. E. Shannon. Programming a computer for playing chess. In N. J. A. Sloane and A. D. Wyner, editors, *Claude Elwood Shannon Collected Papers*, pp. 637-656. IEEE Press, New York, NY, 1993.
- [27] R. S. Sutton. Implementation details of the TD( $\lambda$ ) procedure for the case of vector predictions and backpropagation. Technical Report TN87-509.1, GTE Laboratories Incorporated, May 1987.



- [28] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.
- [29] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8:1038–1044, 1996.
- [30] R. S. Sutton and A. G. Barto. *An Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.
- [31] R. S. Sutton and S. D. Whitehead. Online learning with random representation. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 314–321. Morgan Kaufmann, San Mateo, CA, 1993.
- [32] G. Tesauro. Practical issues in temporal difference learning. *Advances in Neural Information Processing Systems*, 4:259–266, 1992.
- [33] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.
- [34] G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–67, March 1995.
- [35] S. Thrun. Learning to play the game of chess. *Advances in Neural Information Processing System*, 7:1069–1076, 1995.
- [36] S. Thrun. Lifelong learning: A case study. Technical report CMU-CS-95-208, Carnegie Mellon University, Pittsburgh, PA, 1995.
- [37] S. Thrun. *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publishers, Norwell, MA, 1996.
- [38] A. Turing. *Faster than Thought: A Symposium on Digital Computing Machines*. B. V. Bowden, Pittman, London, UK, 1953.
- [39] C. J. Watkins. *Learning from delayed rewards*. Ph.D. thesis, King’s College, Cambridge, England, 1989.
- [40] M. A. Weiss. *Data Structures and Algorithm Analysis*. Benjamin/Cummings, Redwood City, CA, 1992.





**On-Policy:** This method evaluates and improves the same policy that it uses to make decisions.

**Policy:** The decision-making function of the agent, telling it what action to choose in each state.

**Q-Learning:** An off-policy temporal difference learning algorithm developed by Watkins.

**Reinforcement Learning:** A learning mechanism that rewards good decisions and punishes bad choices made by the agent; and the learning is autonomous.

**Reward Function:** The definition of the agent's goal that maps the state of the environment to a single number.

**SARSA:** An on-policy temporal difference learning algorithm.

**TD( $\lambda$ ):** A multi-step temporal difference learning algorithm whereby at each time step, TD( $\lambda$ ) updates a prediction by exponentially decaying previous predictions based on recency, where  $\lambda$  is the decay factor.

**Temporal Difference Learning:** A reinforcement learning method that updates a prediction at time  $t$  to be a fraction closer to the prediction after  $t$ .

**Value Function:** The function that specifies what action selection is the best in the long run.

```
cout << endl << "Do you want to train?" << endl << "> ";
```

```
    // Adding exploration rate if wanted
    // Do you want to use? << endl
```

## Appendix B

### SAMPLE PROGRAM CODE

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// main.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>
#include "str.h"
#include "clock.h"
#include "simulator.h"

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Main program of reinforcement learning system
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int main()
{
    ofstream fout;
    RL_Simulator_Class simulator;
    if (simulator.setup(cout, cin) == 1)    // if simulator setup is completed
    {
        cout << endl << "Constructing RL system ..." << endl;
        simulator.construct();
        cout << "Done." << endl;
        clockClass clock;
        cout << endl << "Agent is learning ..." << endl;
        clock.start();
        simulator.learn(cout, fout);    // RL agent is learning
        clock.stop();
        cout << "Learning is done in " << clock << endl;
    }

    // allow user to train more games if data gathered is inadequate
    cout << endl << "Do you want to train more games (y, n)?" << endl << "> ";
    char response;
    long numOfGames;
    double explorationRate;
    cin >> response;
    while (response == 'y')
    { // network weight, move selection, score stat files
        cout << "Please save weights, play, and score file" << endl << endl;
    }
}

```

```
cout << "How many additional games do you want to train?" << endl << ">";  
cin >> numOfGames;  
simulator.setNumOfGames(numOfGames);  
  
    // experiment the effects of changing exploration rate if wanted  
cout << "What exploration rate do you want to use?" << endl;  
cin >> explorationRate;  
simulator.setExplorationRate(explorationRate);  
clockClass clock;  
cout << endl << "Agent is learning ..." << endl;  
clock.start();  
simulator.learn(cout, fout);  
clock.stop();  
cout << "Learning is done in " << clock << endl;  
cout << endl << "Do you want to train more games (y, n)?" << endl << "> ";  
cin >> response;  
}  
return 0;  
}
```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// simulator.h - interface file
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#ifndef SIMULATOR
#define SIMULATOR

#include <iostream.h>
#include <fstream.h>
#include "str.h"
#include "rl.h"
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Provides a user interface to setup and construct the simulation framework
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class RL_Simulator_Class
{
private:
    RL_Framework_Class *agent;
    int tableOrNN;    // hash table of neural network
    int learning;    // RL learning algorithm selection
    int restore;    // restoring network weight or not
    double stepSize;    // step size of Q-learning
    double discount;    // discount rate of Q-learning
    double learningRate;    // learning rate of neural network
    double momentum;    // neural net's momentum
    double exploration;    // agent's exploration rate
    int policy;    // types of behavior policy
    int difficulty;    // opponent's playing level
    int numOfHidden;    // number of neural net hidden nodes
    double rewardWin;    // reward for a win
    double rewardTie;    // reward for a tie
    double rewardLoss;    // reward for a loss
    double rewardNone;    // reward for intermediate moves
    long numOfGames;    // number of games for training
    stringClass scoreFile;
    int display;    // whether to display

public:
    RL_Simulator_Class();
    ~RL_Simulator_Class();
    void construct();
    void displaySetup(ostream&);    // display system information
    void systemMenu(ostream&);
    void editMenu(ostream&);
    void editSystem(ostream&, istream&);
    int setup(ostream&, istream&);    // setup system
    void learn(ostream&, ofstream&);
    void printSetupToFile(ofstream&);
    void genScoreFileName();
    void setNumOfGames(long);
    void setExplorationRate(double);
};

#endif SIMULATOR

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// simulator.cpp - implementation file
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <iomanip.h> // for setw()
#include <strstream.h>
#include <time.h>
#include "rlstd.h"
#include "simulator.h"

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Constructor
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
RL_Simulator_Class::RL_Simulator_Class()
{
    display = 0;
    rewardWin = 1.0;
    rewardLoss = -1.0;
    rewardTie = 0.0;
    rewardNone = 0.0;
    restore = 0; // create new network, default
    tableOrNN = NEURAL_NET; // Hash table or neural network approximation
    learning = Q_LEARNING; // Sarsa or Q-learning
    stepSize = 0.2; // default step-size for update equation (alpha)
    discount = 0.8; // discount rate for update equation (gamma)
    learningRate = 0.1; // neural network learning rate
    momentum = 0.9; // neural network momentum
    exploration = 0.1; // exploration rate
    policy = EPSILON_GREEDY; // behavior policy of agent
    difficulty = MEDIUM; // opponent's difficulty level, or intelligence
    numOfHidden = 20; // number of hidden nodes for neural network
    numOfGames = 40000; // number of trials or games for learning
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// destructor
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
RL_Simulator_Class::~RL_Simulator_Class()
{
    delete agent;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// generate a file that store the score based on the setup
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::genScoreFileName()
{
    char qOrSarsa, epsilonOrSoftmax, diff;
    switch (learning)
    {
        case Q_LEARNING : qOrSarsa = 'q'; break;
        case SARSA : qOrSarsa = 's'; break;
    }
    switch (policy)
    {
        case EPSILON_GREEDY : epsilonOrSoftmax = 'e'; break;
        case SOFTMAX : epsilonOrSoftmax = 's'; break;
    }
}

```

```

}
switch (difficulty)
{
case EASY : diff = 'e'; break;
case MEDIUM : diff = 'm'; break;
case DIFFICULT : diff = 'd'; break;
}

char tmptime[20], tmpdate[20];

/* Set time zone from TZ environment variable. If TZ is not set,
 * operating system default is used, otherwise PST8PDT is used
 * (Pacific standard time, daylight savings).
 */
_tzset();

/* Display operating system-style date and time. */
_strtime( tmptime );
_strdate( tmpdate );
for (int i=0; i<20; i++)
{
    if (tmptime[i] == ':')
        tmptime[i] = '.';
    if (tmpdate[i] == '/')
        tmpdate[i] = '.';
}

ostream *ostr = new ostream;
if (tableOrNN == NEURAL_NET)
    *ostr << qOrSarsa << "n" << numOfHidden << epsilonOrSoftmax << exploration
    << "a" << stepSize << "g" << discount << "_l" << learningRate << "m"
    << momentum << diff << numOfGames << rewardWin << rewardTie << rewardLoss
    << "_" << tmpdate << tmptime << ".txt" << ends;
else *ostr << qOrSarsa << "t" << epsilonOrSoftmax << exploration << "_a"
    << stepSize << "g" << discount << diff << numOfGames << rewardWin
    << rewardTie << rewardLoss << "_" << tmpdate << tmptime << ".txt" << ends;

char *name = ostr->str(); // generate data file name based on the setup
scoreFile = name;
delete ostr;
delete name;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// construct the simulator
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::construct()
{
    if (tableOrNN == NEURAL_NET)
        agent = new RL_Framework_Class(restore, stepSize, discount,
            learningRate, momentum, exploration, policy, difficulty,
            numOfHidden, rewardWin, rewardTie, rewardLoss, rewardNone,
            display, tableOrNN);

    else if (tableOrNN == HASH_TABLE)

```

```

    agent = new RL_Framework_Class(stepSize, discount, exploration, policy,
                                   difficulty, rewardWin, rewardTie, rewardLoss,
                                   rewardNone, display);
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// display the setup to screen
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::displaySetup(ostream& out)
{
    out << "Learning algorithm ";
    if (learning == Q_LEARNING)
        out << "Q-learning" << endl;
    else if (learning == SARSA)
        out << "Sarsa" << endl;

    out << "Behavior Policy ";
    if (policy == EPSILON_GREEDY)
        out << "epsilon-greedy" << endl;
    else out << "softmax" << endl;

    if (tableOrNN == NEURAL_NET)
    {
        out << "Neural Network      " << numOfHidden << " hidden nodes" << endl
            << "Restore weight          " << restore << endl
            << "Learning rate             " << learningRate << endl
            << "Momentum                  " << momentum << endl;
    }
    else out << "Hash Table          " << endl;

    out << "Step size          " << stepSize << endl
        << "Discount rate     " << discount << endl
        << "Exploration rate  " << exploration << endl
        << "Reward for win    " << rewardWin << endl
        << "Reward for tie    " << rewardTie << endl
        << "Reward for loss   " << rewardLoss << endl
        << "No reward         " << rewardNone << endl
        << "Opponent Difficulty ";
    if (difficulty == EASY)
        out << "easy" << endl;
    else if (difficulty == MEDIUM)
        out << "medium" << endl;
    else out << "difficult" << endl;
    out << "Number of Games    " << numOfGames << endl;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// print system setup information into a file
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::printSetupToFile(ofstream& fout)
{
    fout << "Learning algo      ";
    if (learning == Q_LEARNING)
        fout << "Q-learning" << endl;
    else if (learning == SARSA)

```



```

fout << "Behavior Policy ";
if (policy == EPSILON_GREEDY)
    fout << "epsilon-greedy" << endl;
else fout << "softmax" << endl;

if (tableOrNN == NEURAL_NET)
{
    fout << "Neural Network " << numOfHidden << " hidden nodes" << endl
        << "Restore weight " << restore << endl
        << "Learning rate " << learningRate << endl
        << "Momentum " << momentum << endl;
}
else fout << "Hash Table " << endl;

fout << "Step size " << stepSize << endl
    << "Discount rate " << discount << endl
    << "Exploration rate " << exploration << endl
    << "Reward for win " << rewardWin << endl
    << "Reward for tie " << rewardTie << endl
    << "Reward for loss " << rewardLoss << endl
    << "No reward " << rewardNone << endl
    << "Opponent Difficulty ";
if (difficulty == EASY)
    fout << "easy" << endl;
else if (difficulty == MEDIUM)
    fout << "medium" << endl;
else fout << "difficult" << endl;
fout << "Number of Games " << numOfGames << endl << endl;
}
// print the choice of menus
// - type "q 0" to use SARSA as the learning algo
// void RL_Simulator_Class::systemMenu(ostream& out)
{
    out << "c setup complete" << endl
        << "d display system setup" << endl
        << "e edit system" << endl
        << "l load system from file" << endl
        << "x exit or abort" << endl << endl;
}
// print the setup options
// void RL_Simulator_Class::editMenu(ostream& out)
{
    out << endl
        << "q Learning algorithm (0 Sarsa, 1 Q-learning" << endl
        << "b Behavior policy (0 Softmax, 1 Epsilon-greedy)" << endl
        << endl
        << "t Method to set value (0 Hash Table, 1 Neural Network" << endl
        << endl
        << "d Update equation discount (0 =< d <= 1)" << endl
        << "s Update equation step-size (0 =< s <= 1)" << endl
}

```

```

    << "l Neural network learning rate (0 =< l <= 1)" << endl
    << "m Neural network momentum (0 =< m <= 1)" << endl
    << "e Exploration rate (0 =< e <= 1)" << endl
    << "r Restore (1 true 0 false)" << endl
    << "o Opponent Difficulty" << endl
    << "+" Reward for a win" << endl
    << "=" Reward for a tie" << endl
    << "-" Reward for a loss" << endl
    << "0 No reward" << endl
    << "g Number of games (g > 0)" << endl
    << "n Score file name" << endl;
if (tableOrNN == NEURAL_NET)
{
    out << "h Number of hidden nodes" << endl;
}
out << "x exit edit menu" << endl << endl
    << "Choose the option follow by the value (eg. > d 0.25)"
    << endl << endl << endl;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// modify the setups provided by the user
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::editSystem(ostream& out, istream& in)
{
    out << "Type ? for edit help" << endl << endl
        << "> ";
    char choice;
    in >> choice;
    while (choice != 'x') // continue until exit option is chosen
    {
        switch (choice)
        {
            case '?' : editMenu(out); break;
            case 'q' : in >> learning; break;
            case 'b' : in >> policy; break;
            case 't' : in >> tableOrNN; break;
            case 'r' : in >> restore; break;
            case 'd' : in >> discount; break;
            case 's' : in >> stepSize; break;
            case 'l' : in >> learningRate; break;
            case 'm' : in >> momentum; break;
            case 'e' : in >> exploration; break;
            case 'o' : in >> difficulty; break;
            case '+' : in >> rewardWin; break;
            case '0' : in >> rewardNone; break;
            case '-' : in >> rewardLoss; break;
            case '=' : in >> rewardTie; break;
            case 'g' : in >> numOfGames; break;
            case 'n' : in >> scoreFile; break;
            case 'h' : if (tableOrNN==NEURAL_NET)
                        in >> numOfHidden; break;
            default : out << "Invalid edit option" << endl;
        }
    }
    out << "> ";
}

```

```

        in >> choice;
    }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// setup the simulation
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int RL_Simulator_Class::setup(ostream& out, istream& in)
{
    out << "Reinforcement learning system" << endl << endl;
    systemMenu(out);

    out << "Type ? for help" << endl << endl
        << "> ";
    char choice;
    in >> choice;
    while (choice != 'x' && choice != 'c')
    {
        switch (choice)
        {
            case 'c' : break;
            case '?' : systemMenu(out); break; // display system menu
            case 'd' : displaySetup(out); break; // display current system setup
            case 'e' : editSystem(out, in); break;
            default: out << "Invalid option" << endl;
        }
        out << "> ";
        in >> choice;
    }
    if (choice == 'x')
        return 0;
    else return 1;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The selected RL learning algo is called
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::learn(ostream& out, ofstream& fout)
{
    genScoreFileName();
    fout.open(scoreFile, ios::out);
    fout.flags(ios::left | ios::fixed | ios::floatfield | ios::showpoint);

    // output system setup into a file
    RL_Simulator_Class::printSetupToFile(fout);

    fout << setw(10) << "Win" << setw(10) << "Tie" << setw(10) << "Loss" << endl;

    if (tableOrNN == NEURAL_NET)
    {
        if (learning == Q_LEARNING)
            agent->Approx_Q_Learning(numOfGames, fout, out);
        else if (learning == SARSA)
            agent->Approx_Sarsa_Learning(numOfGames, fout, out);
    }
    else

```

```
{
  if (learning == Q_LEARNING)
    agent->Table_Q_Learning(numOfGames, fout, out);
  else if (learning == SARSA)
    agent->Table_Sarsa_Learning(numOfGames, fout, out);
}
fout.close();
}
////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::setNumOfGames(long games)
{
  numOfGames = games;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
// set exploration rate, only called when we want to train additional games
////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Simulator_Class::setExplorationRate(double exploration)
{
  agent->setExploration(exploration);
}
}
```

```

/////////////////////////////////////////////////////////////////
// rl.h - RLClass interface file
/////////////////////////////////////////////////////////////////
#ifndef RL
#define RL

#include <fstream.h>
#include <iostream.h>
#include "str.h"
#include "environment.h"
#include "experience.h"
#include "policy.h"
/////////////////////////////////////////////////////////////////
// Reinforcement Learning Framework Class
//
//      - The reinforcement learning framework is coded here whereby the agent
//        interacts with its environment and learn by receiving feedbacks from
//        the environment whether its decision is good or bad.
/////////////////////////////////////////////////////////////////
class RL_Framework_Class
{

private:

    RL_Policy_Class      *policy;
    RL_Environment_Class *envir;
    RL_Qvalue_Class     *Q;

    stringClass          curState;
    stringClass          nextState;

    double               alpha;    // learning rate
    double               gamma;    // discount rate
    BOOL                 net;      // flag - neural net or hash table
    void initialize();
    ofstream             gameFile; // file output learning statistic
    ofstream             playFile; // play selection file output

public:

    RL_Framework_Class(int, double, double, double, double,
        double, int, int, int, double, double, double, double, int, int);
    RL_Framework_Class(double, double, double, int, int, double,
        double, double, double, int);

    ~RL_Framework_Class()
    {
        delete policy;
        delete Q;
        gameFile.close();
        playFile.close();
    }

    void Approx_Sarsa_Learning(long int, ofstream&, ostream&);

```

```
void Approx_Q_Learning(long int, ofstream&, ostream&);
void Table_Q_Learning(long int, ofstream&, ostream&);
void Table_Sarsa_Learning(long int, ofstream&, ostream&);

void printState();
void approx_GreedyPlay(ostream&);
void table_GreedyPlay(ostream&);
void setExploration(double);
double get_alpha() { return alpha; }
double get_gamma() { return gamma; }
};

#endif RL
```

```

/////////////////////////////////////////////////////////////////
// rl.cpp - Implementation file for class reinforcementLearningClass
/////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <iomanip.h>
#include <math.h> // for exponential function
#include "random.h" // random number generator to choose who to start the game
#include "rl.h"
#include "rlstd.h"

/////////////////////////////////////////////////////////////////
// allow change of the exploration rate when user wants to train more games
/////////////////////////////////////////////////////////////////
void RL_Framework_Class::setExploration(double rate)
{
    policy->setExploration(rate);
}

/////////////////////////////////////////////////////////////////
// qLearning constructor, construct learning system
/////////////////////////////////////////////////////////////////
RL_Framework_Class::RL_Framework_Class(double stepSize, double discount,
                                         double exploration, int bPolicy, int difficulty, double
                                         rewardWin, double rewardTie, double rewardLoss,
                                         double rewardNone, int display)
{
    gameFile.open("state.txt", ios::out);
    gameFile.flags(ios::left | ios::fixed | ios::floatfield | ios::showpoint );
    playFile.open("play.txt", ios::out);
    playFile.flags(ios::left | ios::fixed | ios::floatfield | ios::showpoint );
    playFile << setw(10) << "Win" << setw(10) << "Draw" << setw(10)
        << "Loss" << setw(10) << "MSE" << endl;

    alpha = stepSize;
    gamma = discount;
    net = FALSE; // means it is using Q-table

    if (bPolicy == EPSILON_GREEDY)
        policy = new RL_Epsilon_Greedy_Class(exploration);
    else policy = new RL_Softmax_Class(exploration);

    // reward for tie is used to initialize Q-table
    Q = new RL_Hashing_Class(rewardTie, display);
    envir = new RL_Connect3_Class(difficulty, rewardWin, rewardTie,
        rewardLoss, rewardNone);
}

/////////////////////////////////////////////////////////////////
// qLearning constructor, construct learning system
/////////////////////////////////////////////////////////////////
RL_Framework_Class::RL_Framework_Class(int restore, double stepSize,
                                         double discount, double learningRate, double momentum,
                                         double exploration, int bPolicy, int difficulty,

```

```

        int numofHidden, double rewardWin, double rewardTie,
        double rewardLoss, double rewardNone,int display, int net)
{
    gameFile.open("state.txt", ios::out);
    gameFile.flags(ios::left | ios::fixed | ios::floatfield | ios::showpoint );
    playFile.open("play.txt", ios::out);
    playFile.flags(ios::left | ios::fixed | ios::floatfield | ios::showpoint );
    playFile << setw(10) << "Win" << setw(10) << "Draw" << setw(10)
        << "Loss" << setw(10) << "MSE" << endl;

    alpha = stepSize;
    gamma = discount;
    net = TRUE;      // is using Neural network

    if (bPolicy == EPSILON_GREEDY)
        policy = new RL_Epsilon_Greedy_Class(exploration);
    else policy = new RL_Softmax_Class(exploration);

    if (net == NEURAL_NET)
        Q = new RL_Neural_Network_Class(restore, learningRate, momentum,
            numofHidden, curState, 0.5, rewardNone, display);

    envir = new RL_Connect3_Class(difficulty, rewardWin,
        rewardTie, rewardLoss, rewardNone);
}

/////////////////////////////////////////////////////////////////
// initialize state to the start of a game, empty board
/////////////////////////////////////////////////////////////////
void RL_Framework_Class::initialize()
{
    curState = "      ";
    nextState = "      ";
}

/////////////////////////////////////////////////////////////////
// DESCRIPTION:      Off-Policy Q-learning where the Q-value is approximated using
//                  a neural network. Value function approximation is in
//                  accordance of Sebastian Thrun's method.
//
// PSEUDOCODE:      Initialize Q(s, a) arbitrarily
//                  Repeat (for each trial):
//                  Initialize s,a
//                  Repeat (for each step of trial):
//                  Choose a from s using policy derived from Q(e.g, e-greedy)
//                  Take action a, observe r, s'
//                  Q(s,a):= gamma * maxQ(s',a')
//                  s := s';
//                  until s is terminal
//
/////////////////////////////////////////////////////////////////
void RL_Framework_Class::Approx_Q_Learning(long int numofTrial, ostream& fout,
        ostream& out)
{
    stringClass action;

```



```

validActionClass move;
double reward, Qvalue;
double x_percent = 0.01, process;
randomNumClass random;

BOOL print = FALSE;

for (int i=0; i<numOfTrial; i++)
{
    initialize(); // reset state, or empty board
    action = ""; // reset action to blank

    if (random.randomInteger(1,2) == 1) // opponent to start
        curState = envir->chooseAction(curState, action);

    if (i % 1000 == 0)
    { // print every state transitions once every 1000 games
        print = TRUE;
        gameFile << "***** Game " << i+1 << " *****" << endl;
    }

    do // **** Approx method ****
    {
        move.generate(curState);
        action = policy->choose(*Q, move, curState);
        if (print)
            printState();
        nextState = envir->chooseAction(curState, action);
        reward = envir->feedback(nextState);
        if (envir->gameOver(nextState)==FALSE)
            Qvalue = gamma * Q->max(nextState); // Sebastian's method
        else Qvalue = reward; // end of Sebastian's method
        Q->setValue(curState, action, Qvalue);
        if (print)
            printState();
        curState = nextState;
    }
    while (envir->gameOver(curState) == FALSE);
    if (print)
        printState();
    print = FALSE;

    envir->keepScore(curState, fout);
    process = (double)i / (double)numOfTrial;
    if (process >= x_percent) // indication of % learning completed
    {
        out << int(process * 100) << "%" << endl;
        x_percent += 0.01;
    }
    if (i % 1000 == 0) // test agent's knowledge every 1000 games
        approx_GreedyPlay(out);
}
out << "100%" << endl;
}

```

```

////////////////////////////////////
// DESCRIPTION:      Off-Policy Q-learning. The Q-value is stored in hash table
//
// PSEUDOCODE:      Initialize Q(s, a) arbitrarily
//                   Repeat (for each trial):
//                   Initialize s to start state
//                   Repeat (for each step of trial):
//                       Choose a from s using policy derived from Q(e.g, e-greedy)
//                       Take action a, observe r, s'
//                       Q(s,a):= Q(s,a) + alpha[r + gamma.Q(s',a')-Q(s,a)]
//                       s := s'
//                   until s is terminal
//
////////////////////////////////////
void RL_Framework_Class::Table_Q_Learning(long int numOfTrial,
                                           ostream& fout, ostream& out)
{
    stringClass action;
    validActionClass move;
    double reward, error, Qvalue;
    double x_percent = 0.01, process;
    randomNumClass random;
    double maxQ;

    BOOL print = FALSE;
    error = 0.0;      // new trial from Sebastian's book
    for (int i=0; i<numOfTrial; i++)
    {
        initialize(); // reset state, or empty board
        action = ""; // reset action to blank
        if (random.randomInteger(1,2) == 1) // opponent to start
            curState = envir->chooseAction(curState, action);

        if ((i % 1000)==0) // for debug purposes
        {
            print = TRUE;
            gameFile << "***** Game " << i+1 << " *****" << endl;
        }

        do // ***** TABLE Q-learning *****
        {
            move.generate(curState);
            action = policy->choose(*Q, move, curState);
            if (print)
                printState();
            nextState = envir->chooseAction(curState, action);
            reward = envir->feedback(nextState);
            if (envir->gameOver(nextState)==FALSE)
                maxQ = Q->max(nextState);
            else maxQ = 0.0;
            Qvalue = Q->getValue(curState, action);
            Qvalue += alpha * (reward + (gamma * maxQ) - Qvalue);
        }
    }
}

```

```

        Q->setValue(curState, action, Qvalue);
        curState = nextState;
    }
    while (envir->gameOver(curState) == FALSE);

    if (print)
        printState();
    print = FALSE;

    envir->keepScore(curState, fout);
    process = (double)i / (double)numOfTrial;
    if (process >= x_percent)
    {
        out << int(process * 100) << "%" << endl;
        x_percent += 0.01;
    }
    if (i % 1000 == 0) // test agent's knowledge every 1000 games
        table_GreedyPlay(out);
}
out << "100%" << endl;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION:      On-Policy Q-learning where the Q-value is approximated using
//                  a neural network. Value function approximation is in
//                  accordance of Sebastian Thrun's method.
//
// PSEUDOCODE:      Initialize Q(s, a) arbitrarily
//                  Repeat (for each trial):
//                  Initialize s,a
//                  Repeat (for each step of trial):
//                  Take action a, observe r
//                  Choose a' from s using policy derived from Q(e.g, e-greedy)
//                  Q(s,a):= gamma * maxQ(s',a')
//                  s := s'; a := a';
//                  until s is terminal
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Framework_Class::Approx_Sarsa_Learning(long int numOfTrial,
                                                ostream& fout, ostream& out)
{
    stringClass action, nextAction;
    validActionClass move;
    double reward;
    double x_percent = 0.01, process, nextQ;
    randomNumClass random;

    BOOL print = FALSE;

    for (int i=0; i<numOfTrial; i++) // for each game
    {
        initialize();
        action = "";

```

```

if (random.randomInteger(1,2) == 1) // opponent to start
    curState = envir->chooseAction(curState, action);
move.generate(curState);
action = policy->choose(*Q, move, curState); // take action a

if (i % 1000 == 0)
{ // print every state transitions once every 1000 games
    print = TRUE;
    gameFile << "***** Game " << i+1 << " *****" << endl;
}

// ***** Approx Sarsa *****
do // for each move of a game
{
    if (print)
        printState();

    nextState = envir->chooseAction(curState, action);
    reward = envir->feedback(nextState);

    if (envir->gameOver(nextState)==FALSE)
    {
        move.generate(nextState);
        nextAction = policy->choose(*Q, move, nextState); // choose a'
        // Sebastian's method
        nextQ = gamma * Q->getValue(nextState, nextAction);
    }
    else nextQ = reward;

    Q->setValue(curState, action, nextQ);
    if (print)
        printState();
    curState = nextState; // s := s'
    action = nextAction; // a := a'
}
while (envir->gameOver(curState) == FALSE); // until s is terminal

if (print)
{
    printState();
    print = FALSE;
}

envir->keepScore(curState, fout);
process = (double)i / (double)numOfTrial;
if (process >= x_percent)
{
    out << int(process*100) << "%" << endl;
    x_percent += 0.01;
}
if (i % 1000 == 0)
    approx_GreedyPlay(out);
}
out << "100%" << endl;

```

```

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION:      On-Policy Q-learning. The Q-value is stored in hash table
//
// PSEUDOCODE:      Initialize Q(s, a) arbitrarily
//                   Repeat (for each trial):
//                   Initialize s,a
//                   Repeat (for each step of trial):
//                   Take action a, observe r
//                   Choose a from s' using policy derived from Q(eg. e-greedy)
//                   Q(s,a):=Q(s,a) + alpha[r + gamma.Q(s',a')-Q(s,a)]
//                   a := a'; s := s';
//                   until s is terminal
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Framework_Class::Table_Sarsa_Learning(long int numOfTrial,
                                              ostream& fout, ostream& out)
{
    stringClass action, nextAction;
    validActionClass move;
    double reward, Qvalue;
    double x_percent = 0.01, process, nextQ;
    randomNumClass random;
    BOOL print = FALSE;

    for (int i=0; i<numOfTrial; i++) // for each game
    {
        initialize();
        action = "";
        if (random.randomInteger(1,2) == 1) // opponent to start
            curState = envir->chooseAction(curState, action);
        move.generate(curState);
        action = policy->choose(*Q, move, curState); // take action a

        if (i% 1000 == 0)
        { // print every state transitions once every 1000 games
            print = TRUE;
            gameFile << "***** Game " << i+1 << " *****" << endl;
        }

        // ***** Table Sarsa Learning *****
        do // for each move of a game
        {
            if (print)
                printState();

            nextState = envir->chooseAction(curState, action);
            reward = envir->feedback(nextState);

            Qvalue = Q->getValue(curState, action);
            if (envir->gameOver(nextState)==FALSE)
            {
                move.generate(nextState);
            }
        }
    }
}

```

```

        nextAction = policy->choose(*Q, move, nextState); // choose a'
        nextQ = Q->getValue(nextState, nextAction);
    }
    else nextQ = 0.0;

    Qvalue += alpha * (reward + (gamma * nextQ) - Qvalue);
    Q->setValue(curState, action, Qvalue);

    if (print)
        printState();

    curState = nextState; // s := s'
    action = nextAction; // a := a'
}
while (envir->gameOver(curState) == FALSE); // until s is terminal

if (print)
{
    printState();
    print = FALSE;
}

envir->keepScore(curState, fout);
process = (double)i / (double)numOfTrial;
if (process >= x_percent)
{
    out << int(process*100) << "%" << endl;
    x_percent += 0.01;
}
if (i % 1000 == 0) // test agent's knowledge every 1000 games
    table_GreedyPlay(out);
}
out << "100%" << endl;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Print out state to analyse the decision selected by the agent...
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Framework_Class::printState()
{
    int i;
    gameFile
        << curState[0] << "|" << curState[1]
        << "|" << curState[2] << endl
        << "--+-" << " ";

    for (i=0; i<3; i++)
        if (curState[i] == ' ')
            gameFile << setw(15) << Q->getValue(curState, i);
        else gameFile << setw(15) << " ";

    gameFile
        << endl << curState[3] << "|" << curState[4]
        << "|" << curState[5] << " ";
}

```

```

for (i=3; i<6; i++)
    if (curState[i] == ' ')
        gameFile << setw(15) << Q->getValue(curState, i);
    else gameFile << setw(15) << " ";

gameFile << endl << "-+-+-" << " ";
for (i=6; i<9; i++)
    if (curState[i] == ' ')
        gameFile << setw(15) << Q->getValue(curState, i);
    else gameFile << setw(15) << " ";

gameFile
    << endl << curState[6] << "|" << curState[7]
    << "|" << curState[8] << endl << endl;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Agent play greedily for 100 games. Only greedy play allows unbiased
// assessment of the agent's ability to learn the task.
//
// This function is called if Q-value is stored in hash table
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Framework_Class::table_GreedyPlay(ostream& out)
{
    stringClass action, nextAction;
    validActionClass move;
    randomNumClass random;
    double reward, nextQ;
    int win=0, draw=0, loss=0;
    double meanSquaredError, sumMSE=0.0, error=0.0, target, output;
    int numOfDecision=0;

    for (int i=0; i<100; i++) // for each game
    {
        initialize();
        action = "";
        if (random.randomInteger(1,2) == 1) // opponent to start
            curState = envir->chooseAction(curState, action);
        move.generate(curState);
        action = policy->exploit(*Q, move, curState); // take action a

        // ***** Table Sarsa Learning *****
        do // for each move of a game
        {
            nextState = envir->chooseAction(curState, action);
            reward = envir->feedback(nextState);
            if (envir->gameOver(nextState)==FALSE)
            {
                move.generate(nextState);
                nextAction = policy->choose(*Q, move, nextState); // choose a'
                nextQ = Q->getValue(nextState, nextAction);
            }
            else nextQ = 0.0;
            output = Q->getValue(curState, action);

```

```

    target = reward + (gamma * nextQ);
    meanSquaredError = 0.5 * pow((target - output), 2);
    sumMSE += meanSquaredError;

    curState = nextState; // s := s'
    action = nextAction; // a := a'
    numOfDecision++;
}
while (envir->gameOver(curState) == FALSE); // until s is terminal

if (envir->feedback(nextState) == 1)
    win++;
else if (envir->feedback(nextState) == -1)
    loss++;
else draw++;
}
// mean square error analysis
meanSquaredError = sumMSE / double(numOfDecision);
out << "Win(" << win << " ), Draw(" << draw << " ), Loss(" << loss << " ), "
    << meanSquaredError << endl;
playFile << setw(10) << win << setw(10) << draw << setw(10)
    << loss << setw(10) << meanSquaredError << endl;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Agent play greedily for 100 games. Only greedy play allows unbiased
// assessment of the agent's ability to learn the task.
//
// This function is called if Q-value is approximated using neural network
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Framework_Class::approx_GreedyPlay(ostream& out)
{
    stringClass action, nextAction;
    validActionClass move;
    randomNumClass random;

    int win=0, draw=0, loss=0;
    double meanSquaredError, sumMSE=0.0, error=0.0, target, output;
    int numOfDecision=0;

    for (int i=0; i<100; i++) // for each game
    {
        initialize();
        action = "";
        if (random.randomInteger(1,2) == 1) // opponent to start
            curState = envir->chooseAction(curState, action);
        move.generate(curState);
        action = policy->exploit(*Q, move, curState); // take action a

        // ***** Table Sarsa Learning *****
        do // for each move of a game
        {
            nextState = envir->chooseAction(curState, action);
            if (envir->gameOver(nextState)==FALSE)

```



```

    {
        move.generate(nextState);
        nextAction = policy->choose(*Q, move, nextState); // choose a'
        // Sebastian's method
        target = gamma * Q->getValue(nextState, nextAction);
    }
    else target = envir->feedback(nextState);
    output = Q->getValue(curState, action);
    meanSquaredError = 0.5 * pow((target - output), 2);
    sumMSE += meanSquaredError;

    curState = nextState; // s := s'
    action = nextAction; // a := a'
    numOfDecision++;
}
while (envir->gameOver(curState) == FALSE); // until s is terminal

if (envir->feedback(nextState) == 1)
    win++;
else if (envir->feedback(nextState) == -1)
    loss++;
else draw++;
}
// mean square error analysis
meanSquaredError = sumMSE / double(numOfDecision);
out << "Win(" << win << "), Draw(" << draw << "), Loss(" << loss << "), "
    << meanSquaredError << endl;
playFile << setw(10) << win << setw(10) << draw << setw(10)
    << loss << setw(10) << meanSquaredError << endl; // print to file
}

```

```

/////////////////////////////////////////////////////////////////
// policy.h - interface file
/////////////////////////////////////////////////////////////////
#include "str.h"
#include "random.h"
#include "experience.h"

#ifndef POLICY
#define POLICY

/////////////////////////////////////////////////////////////////
// Generate valid play or decision based on the state
/////////////////////////////////////////////////////////////////
class validActionClass
{
private:
    int count;    // number of valid action selection
    stringClass action[9];
    stringClass convToStr(int);
public:
    validActionClass() { count=0; }
    int getCount() const { return count; }
    stringClass get(int i) const { return action[i]; }
    void generate(const stringClass&);
};
/////////////////////////////////////////////////////////////////
// Policy Class - agent's policy or strategy in decision making
//
// Default policy is now epsilon greedy
/////////////////////////////////////////////////////////////////
class RL_Policy_Class
{
protected:
    randomNumClass rng;
    int numOfGreedyAction;
    double explorationRate;
public:
    RL_Policy_Class(double expRate) { explorationRate = expRate;
        numOfGreedyAction=0; }
    stringClass exploit(RL_Qvalue_Class&, const validActionClass&,
        const stringClass&);
    virtual stringClass explore(RL_Qvalue_Class&, const validActionClass&,
        const stringClass&) { return "NULL"; }
    virtual stringClass choose(RL_Qvalue_Class&, const validActionClass&,
        const stringClass&);
    virtual double getExploration() { return explorationRate; }
    virtual void setExploration(double e) { explorationRate=e;}
};
/////////////////////////////////////////////////////////////////
// Epsilon Greedy... inherit most functions from Policy_Class
/////////////////////////////////////////////////////////////////
class RL_Epsilon_Greedy_Class : public RL_Policy_Class
{

```

```
public:
    RL_Epsilon_Greedy_Class(double epsilon) : RL_Policy_Class(epsilon) {}
    virtual stringClass explore(RL_Qvalue_Class&, const validActionClass&,
                               const stringClass&);
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Softmax_Class or simulated annealing or boltzmann distribution
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class RL_Softmax_Class : public RL_Policy_Class
{
public:
    RL_Softmax_Class(double tau) : RL_Policy_Class(tau) {}
    virtual stringClass choose(RL_Qvalue_Class&,
                               const validActionClass&, const stringClass&);
    virtual stringClass explore(RL_Qvalue_Class&,
                               const validActionClass&, const stringClass&);
};

#endif POLICY
```

```

/////////////////////////////////////////////////////////////////
// policy.cpp - implementation file
/////////////////////////////////////////////////////////////////
#include <stdlib.h>
#include <string.h>
#include <math.h>      // for exponential function in softmax
#include "policy.h"
/////////////////////////////////////////////////////////////////
// convert the valid action from character to stringClass
/////////////////////////////////////////////////////////////////
stringClass validActionClass::convToStr(int i)
{
    stringClass temp;
    char num[3], *s;
    s = num;
    num[0] = '0' + i;
    num[1] = '\0';
    temp = s;
    return temp;
}
/////////////////////////////////////////////////////////////////
// Generate the legal decisions based on the tic tac toe state
/////////////////////////////////////////////////////////////////
void validActionClass::generate(const stringClass& state)
{
    count = 0;
    for (int i=0; i<9; i++)
        if (state[i] == ' ')
            action[count++] = convToStr(i);
}
/////////////////////////////////////////////////////////////////
// Greedy policy. Choose the best action available
/////////////////////////////////////////////////////////////////
stringClass RL_Policy_Class::exploit(RL_Qvalue_Class& value,
                                     const validActionClass& action, const stringClass& state)
{
    stringClass tempAction;
    stringClass *tie = new stringClass[action.getCount()];
    int numOfTie = 0;
    if (action.getCount() > 0)
    {
        tempAction = action.get(0);
        tie[0] = action.get(0);
    }
    else
    {
        cerr << "No available action" << endl;
        exit(0);
    }
    double greatest, tempVal;
    greatest = value.getValue(state, tempAction);

    for (int i=1; i<action.getCount(); i++)
    {

```

```

tempAction = action.get(i);
tempVal = value.getValue(state, tempAction);
if (tempVal > greatest)
{
    greatest = tempVal;
    numOfTie = 0;
    tie[numOfTie] = action.get(i);
}
else if (tempVal == greatest)
    tie[++numOfTie] = action.get(i);
}
if (numOfTie == 0)
    return tie[numOfTie];
else
{
    int tieBreak = rng.randomInteger(0, numOfTie);
    return tie[tieBreak];
}
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Default policy is epsilon greedy. If epsilon is 0.1 that means the agent is
// exploring 10 percent of the time. The rest are all greedy moves.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
stringClass RL_Policy_Class::choose(RL_Qvalue_Class& value, const
                                   validActionClass& action, const stringClass& state)
{
    numOfGreedyAction++;
    double greedyness = (double)(1.0/numOfGreedyAction);
    stringClass chosen;
    if (greedyness <= explorationRate)
    {
        chosen = explore(value, action, state);
        numOfGreedyAction = 0;
    }
    else chosen = exploit(value, action, state);
    return chosen;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// make random decisions
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
stringClass RL_Epsilon_Greedy_Class::explore(RL_Qvalue_Class& value,
                                             const validActionClass& action,
                                             const stringClass& state)
{
    int random = rng.randomInteger(0, action.getCount()-1);
    return action.get(random);
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// explorationRate is a positive parameter called the temperature. High
// temperatures causes the actions to be all (nearly) equi-
// probable. Low temperatures cause a greater difference in
// selection probability for actions that differ in their
// value estimates.
//

```

```

// The most common soft-max method uses a Gibbs or Boltzmann
// distribution. It chooses action a on the (t+1)st play with
// probability
//      policy t+1(a) = (e^(Qt(a)/tao))/(summ (e^Qt(b)/tao))
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
stringClass RL_Softmax_Class::explore(RL_Qvalue_Class& value, const
                                     validActionClass& action, const stringClass& state)
{
    double preference, HIGHEST;
    stringClass tempAction;
    stringClass *tie = new stringClass[action.getCount()];
    stringClass ties;

    int numOfTie = 0;
    if (action.getCount() < 0)
    {
        cerr << "No available action" << endl;
        exit(0);
    }
    double summ = 0.0;
    for (int i=0; i<action.getCount(); i++)
    {
        tempAction = action.get(i);
        summ += exp(value.getValue(state, tempAction) / explorationRate);
    }
    tempAction = action.get(0);
    HIGHEST = exp(value.getValue(state, tempAction) / explorationRate) / summ;

    for (i=1; i<action.getCount(); i++)
    {
        tempAction = action.get(i);
        preference = exp(value.getValue(state, tempAction)/explorationRate) / summ;

        if (preference > HIGHEST)
        {
            HIGHEST = preference;
            numOfTie = 0;
            tie[numOfTie] = action.get(i);      // reset tie action
        }
        else if (preference == HIGHEST)
            tie[++numOfTie] = action.get(i);    // add tie action to list
    }

    // handle tie if there exist any
    if (numOfTie == 0)
        return tie[numOfTie];
    else
    {
        int tieBreak = rng.randomInteger(0, numOfTie);
        return tie[tieBreak];
    }
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// environment.h - interface file
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#ifndef ENVIRONMENT
#define ENVIRONMENT

#include <fstream.h>
#include "rlstd.h"
#include "str.h"
#include "random.h"

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// RL_Environment_Class definition
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class RL_Environment_Class
{
protected:
    int gameCount;
    int numOfWin;
    int numOfLoss;
    int numOfTie;
    double rewardWin;
    double rewardTie;
    double rewardLoss;
    double rewardNone;

public:
    RL_Environment_Class(double w, double t, double l, double n)
        { gameCount=0; numOfWin=0; numOfLoss=0; numOfTie=0;
          rewardWin = w; rewardTie = t; rewardLoss = l; rewardNone = n; }
    virtual ~RL_Environment_Class();

    virtual stringClass chooseAction(const stringClass&, const stringClass&)=0;
    virtual double feedback(const stringClass&)=0;
    virtual BOOL gameOver(const stringClass&)=0;
    virtual void keepScore(const stringClass&, ofstream&)=0;
    virtual stringClass chooseRandomly(const stringClass&, const stringClass&)=0;
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Connect3 == Tic Tac Toe : Class definition
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class RL_Connect3_Class : public RL_Environment_Class
{
private:
    enum { Xwins=1, Owins=-1, Tie=2, None=0 };
    enum { X=1, O=-1 };
    int vals[19683]; // 3^9 -- not all positions are accesible, however
    int difficulty;
    int player;
    int type;
    randomNumClass random;

public:
    RL_Connect3_Class(int, double, double, double, double);

```

```
virtual ~RL_Connect3_Class();
int boardToInt(const stringClass&);
int testWon(const stringClass&);
int minimax(const stringClass&, int, int);
stringClass chooseAction(const stringClass&, const stringClass&);
BOOL gameOver(const stringClass&);
double feedback(const stringClass&);
void keepScore(const stringClass&, ofstream&);
stringClass chooseRandomly(const stringClass&, const stringClass&);
};

#endif ENVIRONMENT
```



```

/////////////////////////////////////////////////////////////////
// environment.cpp - implementation file
/////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>
#include "environment.h"
/////////////////////////////////////////////////////////////////
// default destructor
/////////////////////////////////////////////////////////////////
RL_Environment_Class::~RL_Environment_Class()
{
}
/////////////////////////////////////////////////////////////////
// default destructor
/////////////////////////////////////////////////////////////////
RL_Connect3_Class::~RL_Connect3_Class()
{
}
/////////////////////////////////////////////////////////////////
// Print score every 100 games into a file
/////////////////////////////////////////////////////////////////
void RL_Connect3_Class::keepScore(const stringClass& state, ofstream& fout)
{
    gameCount++;
    switch(testWon(state))
    {
        case Xwins: numOfWin++; break;
        case Owins: numOfLoss++; break;
        case Tie: numOfTie++; break;
        default: fout << "The game is not ended, big error <keepScore()>!" << endl;
    }
    if (gameCount == 100)
    {
        fout << setw(10) << numOfWin << setw(10) << numOfTie
            << setw(10) << numofLoss << endl;
        gameCount = 0;
        numOfWin=0;
        numofLoss=0;
        numofTie=0;
    }
}
/////////////////////////////////////////////////////////////////
// constructor - the agent is playing X and the opponent is using O
/////////////////////////////////////////////////////////////////
RL_Connect3_Class::RL_Connect3_Class(int diff, double w, double t,
                                     double l, double n) : RL_Environment_Class(w, t, l, n)
{
    srand(time(NULL));
    player = X;
    difficulty = diff;
}
/////////////////////////////////////////////////////////////////
// return true if game is over or vice versa

```

```

/////////////////////////////////////////////////////////////////
BOOL RL_Connect3_Class::gameOver(const stringClass& state)
{
    int over;
    over = RL_Connect3_Class::testWon(state);
    switch(over)
    {
        case None: return FALSE;
        default: return TRUE;
    }
}

/////////////////////////////////////////////////////////////////
// feedback that the agent receives concerning the agent's decision
/////////////////////////////////////////////////////////////////
double RL_Connect3_Class::feedback(const stringClass& state)
{
    int status;
    status = RL_Connect3_Class::testWon(state);
    switch(status)
    {
        case Owins: return rewardLoss;
        case Xwins: return rewardWin;
        case Tie   : return rewardTie;
        default   : return rewardNone;
    }
}

/////////////////////////////////////////////////////////////////
// maps the board to a number that used as an index entry in an array
/////////////////////////////////////////////////////////////////
int RL_Connect3_Class::boardToInt(const stringClass& board)
{
    int i, out = 0, exp = 1;
    for (i=0; i<9; i++, exp *= 3)
    {
        switch(board[i])
        {
            case ' ': // zero
                break;
            case 'X': // one
                out += exp * 1;
                break;
            case 'O': // two
                out += exp * 2;
            }
        }
    }
    return out;
}

/////////////////////////////////////////////////////////////////
// test whether board configuration is a winning one..
/////////////////////////////////////////////////////////////////
int RL_Connect3_Class::testWon(const stringClass& board)
{
    int i, flag=0;
    stringClass b(board);

```

```

if ((b[0] == 'X' && b[1] == 'X' && b[2] == 'X') ||
    (b[3] == 'X' && b[4] == 'X' && b[5] == 'X') ||
    (b[6] == 'X' && b[7] == 'X' && b[8] == 'X') ||
    (b[0] == 'X' && b[3] == 'X' && b[6] == 'X') ||
    (b[1] == 'X' && b[4] == 'X' && b[7] == 'X') ||
    (b[2] == 'X' && b[5] == 'X' && b[8] == 'X') ||
    (b[0] == 'X' && b[4] == 'X' && b[8] == 'X') ||
    (b[2] == 'X' && b[4] == 'X' && b[6] == 'X'))
return Xwins;

if ((b[0] == 'O' && b[1] == 'O' && b[2] == 'O') ||
    (b[3] == 'O' && b[4] == 'O' && b[5] == 'O') ||
    (b[6] == 'O' && b[7] == 'O' && b[8] == 'O') ||
    (b[0] == 'O' && b[3] == 'O' && b[6] == 'O') ||
    (b[1] == 'O' && b[4] == 'O' && b[7] == 'O') ||
    (b[2] == 'O' && b[5] == 'O' && b[8] == 'O') ||
    (b[0] == 'O' && b[4] == 'O' && b[8] == 'O') ||
    (b[2] == 'O' && b[4] == 'O' && b[6] == 'O'))
return Owins;

// test for tie
for (i=0; i<9; i++)
{
    if (b[i] == ' ')
        flag=1;
}
if (!flag)
    return Tie;
return None; // game's not over
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Minimax algorithm
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int RL_Connect3_Class::minimax(const stringClass& state, int pl, int depth)
{
    stringClass board(state);
    int best, val;
    int i, index;
    index = boardToInt(board);
    if ((vals[index] % 10) >= depth)
        return vals[index] - (vals[index] % 10); // since 0 is a valid value

    // we don't want use the memorized value if it is shallower than
    // we're allowed to go--in fact, this doesn't matter since the memoization
    // goes away with each move, but otherwise it would.

    if (depth > difficulty)
        return 0;

    if ((val = testWon(board)) != None)
    {
        switch(val)
        {

```

```

        case Xwins:
        case Owins:
            return 1000 * val;
        case Tie:
            return 0;
    }
}
best = -pl * 1000000;
for (i=0; i<9; i++)
{
    if (board[i] == ' ')
    {
        board[i] = (pl == X) ? 'X' : 'O';
        val = minimax(board, -pl, depth+1);
        if (val * pl > best * pl)
            best = val;
        board[i] = ' ';
    }
}
vals[index] = best + depth;
return best;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// find move based on the state of the game using the minimax algo
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
stringClass RL_Connect3_Class::chooseAction(const stringClass& state,
                                             const stringClass& action)
{
    stringClass board(state);
    stringClass tempAction(action);
    int act;
    if (tempAction != "") // if action is "", it means it is the start of game
    {
        act = (int)tempAction;
        if (player == X)
            board[act] = 'X';
        else board[act] = 'O';
    }
    stringClass nextState(board);

    if (!gameOver(nextState))
    {
        int best, val, besti[9], i, bestcount = 0;

        // find the computer's move, not the player's
        int pl = -player;
        best = -pl * 1000000;
        for (i=0; i<9; i++)
        {
            if (board[i] == ' ')
            {
                board[i] = (pl == X) ? 'X' : 'O';
                val = minimax(board, -pl, 1);
                if (val == best)

```

```

        {
            best = val;
            besti[bestcount++] = i;
        }
        else if (val * pl > best * pl) {
            best = val;
            bestcount = 0;
            besti[bestcount++] = i;
        }
        board[i] = ' ';
    }
}
if (bestcount > 0)
{
    i = rand()%bestcount;
    nextState[besti[i]] = '0';
}
}
return nextState;
}
// like the name suggest, make a random move
stringClass RL_Connect3_Class::chooseRandomly(const stringClass& state,
                                               const stringClass& action)
{
    stringClass board(state);
    stringClass tempAction(action);
    int act;
    if (tempAction != "") // if action is "", it means it is the start of game
    {
        act = (int)tempAction;
        if (player == X)
            board[act] = 'X';
        else board[act] = '0';
    }
    stringClass nextState(board);
    int valid[9], vcounter=0;

    if (!gameOver(nextState))
    {
        for (int i=0; i<9; i++) // looking for winning moves
            if (nextState[i] == ' ')
                valid[vcounter++] = i; // store valid move
        if (vcounter > 0)
            i = valid[random.randomInteger(0, vcounter-1)]; // random moves
        nextState[i] = '0';
    }
    return nextState;
}
}

```

```

/////////////////////////////////////////////////////////////////
// experience.h - interface file
/////////////////////////////////////////////////////////////////
#ifndef EXPERIENCE
#define EXPERIENCE
#include <fstream.h>
#include "rlstd.h"
#include "backprop1.h"
#include "str.h"
/////////////////////////////////////////////////////////////////
// The parent class that implement the Q-values
//   - Q value is either stored in a hash table (RL_Hashing_Class) or
//   - approximated using a neural network trained by backpropagation algorithm
//     (RL_Neural_Network_Class).
//
/////////////////////////////////////////////////////////////////
const long TTT_STATE = 25001;      // hash table number of rows, must be prime
const long TTT_ACTION = 9;        // hash table number of columns
const long R = 24991;             // a variable of the hash function given below

class RL_Qvalue_Class
{
protected:
    fstream net;
    double value;
    double initValue;
    int displayStateInfo;
public:
    RL_Qvalue_Class(double, int);

    virtual ~RL_Qvalue_Class() {};
    virtual double getValue(const stringClass&, const stringClass&) = 0;
    virtual void setValue(const stringClass&, const stringClass&, double)=0;
    double getValue(const stringClass&, int);
    void setValue(const stringClass&, int, double);
    virtual double max(const stringClass&) = 0;
};

/////////////////////////////////////////////////////////////////
// Hash Table Class definition
//   - implement double hashing from Mark Allen Weiss's book
//   - hash table is written to an output file to reduce memory requirement
//   - hash table is State (X) x Action (Y), a two dimensional array file
//   -  $h(x) = R - (x \bmod R)$  is the first hash
//   - if collision happens, then probe  $h(x)$ ,  $2h(x)$ , ..so on.(double hash)
//
/////////////////////////////////////////////////////////////////
class RL_Hashing_Class : public RL_Qvalue_Class
{
private: // Attributes
    stringClass state[TTT_STATE];
    stringClass fileName;
    fstream stateAction; // state action pair Q-value hash table file
    int key;
};

```

```

    int distance; // distance for rehashing if collision occurs
    int hashValue; // the value mapped from input
    long int collision; // number of collisions
    int actionLength; // length of a stored Q-value in the file
    int stateLength; // length of one row
    long int cellPosInFile; // cell position in the hash table
private: // Operations
    int hash(const stringClass&);
    int doubleHash(int);
    void insert(const stringClass&);
    BOOL find(const stringClass&);
    void initStateAction();
public: // Operations
    RL_Hashing_Class(double, int);
    virtual ~RL_Hashing_Class();
    int getKey();
    virtual double getValue(const stringClass&, const stringClass&);
    virtual void setValue(const stringClass&, const stringClass&, double);
    virtual double max(const stringClass&);
};
////////////////////////////////////////////////////////////////////
// Neural network abstraction
// - the key operations are stored in backprop1.h
// - the source code of the backpropagation algo in backprop1.h is from
// Roger's book Object Oriented Neural Networks in C++
//
////////////////////////////////////////////////////////////////////
class RL_Neural_Network_Class : public RL_Qvalue_Class
{
// 2 binary numbers to represent a tic-tac-toe square.
// The network is a 3 layer 18:H:9 architecture.
enum { inSize = 18, outSize = 9, layer = 3 };

private:
    double input[inSize];
    double output[outSize];
    Backprop_Network *BPnet; // the neural network trained using backprop
    Pattern *data[4];

    void binaryInput(const stringClass&, int);

public:
    RL_Neural_Network_Class(int, double, double, int, const stringClass&,
        double, double, int);

    virtual ~RL_Neural_Network_Class();
    virtual double getValue(const stringClass&, const stringClass&);
    virtual void setValue(const stringClass&, const stringClass&, double);
    virtual double max(const stringClass&);
    void initialize(const stringClass&, double);

};

#endif EXPERIENCE

```

```

/////////////////////////////////////////////////////////////////
// experience.cpp - implementation file
/////////////////////////////////////////////////////////////////
#include "experience.h"
#include "random.h"
#include <iomanip.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
/////////////////////////////////////////////////////////////////
// constructor
/////////////////////////////////////////////////////////////////
RL_Qvalue_Class::RL_Qvalue_Class(double n, int d)
{
    initValue = n;
    value = (double)0.0;
    displayStateInfo = d;
}
/////////////////////////////////////////////////////////////////
// return state-action Q-value.
/////////////////////////////////////////////////////////////////
double RL_Qvalue_Class::getValue(const stringClass& s, int a)
{
    char act[3];
    act[0] = a + '0';
    act[1] = '\0';
    stringClass strAct = act;
    return getValue(s, strAct);    // virtual function
}
/////////////////////////////////////////////////////////////////
// set state-action Q-value
/////////////////////////////////////////////////////////////////
void RL_Qvalue_Class::setValue(const stringClass& s, int a, double Q)
{
    char act[2];
    act[0] = a + '0';
    act[1] = '\0';
    stringClass strAct = act;
    setValue(s, strAct, Q);    // virtual function
}
/////////////////////////////////////////////////////////////////
// Hash table constructor. q.txt is the file Q-table. used to reduce memory req.
/////////////////////////////////////////////////////////////////
RL_Hashing_Class::RL_Hashing_Class(double n, int d) : RL_Qvalue_Class(n, d)
{
    collision = 0;
    key = 0;
    hashValue = 0;

    fileName = "q.txt";
    RL_Hashing_Class::initStateAction();
    collision = 0;
}

```



```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// destructor
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
RL_Hashing_Class::~RL_Hashing_Class()
{
    stateAction.close();
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// map the input (state) into a number, the hashing function
//  $h(x) = R - (x \bmod R)$ 
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int RL_Hashing_Class::hash(const stringClass& s)
{
    hashValue = 0;
    int powerOf = 0;
    for (int i=TTT_ACTION-1; i>=0; i--)
        hashValue += s[i] * (int)pow(2, powerOf++);

    key = hashValue % TTT_STATE;
    distance = R - (hashValue % R);
    return key;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// double hashing. Probe it i * distance if there is a collision
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int RL_Hashing_Class::doubleHash(int i)
{
    key += i * distance;    // probe at i*distance
    while (key >= TTT_STATE)
        key -= TTT_STATE;
    collision++;
    return key;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// find the entry (Q-value) of state s
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
BOOL RL_Hashing_Class::find(const stringClass& s)
{
    stringClass tempState(s);
    int index = hash(tempState);
    BOOL found = FALSE;
    int i=1;

    // if entry in state table is not NULL, continue
    if (state[index] != "\0")
    {
        do
        {
            if (state[index] != tempState)
                index = doubleHash(i++);
            else found = TRUE;
        }
        while ((state[index] != "\0") && !found);
    }
}

```

```

else found = FALSE;      // NULL, no matching data in state table

return found;
}
// insert an action-value pair (Q-value)
void RL_Hashing_Class::insert(const stringClass& s)
{
    int index;
    stringClass tempState(s);
    index = hash(tempState);
    int i=1;
    while (state[index] != "\0")    // not NULL
        index = doubleHash(i++);
    state[index] = tempState;
}
// return hash key
int RL_Hashing_Class::getKey()
{
    return key;
}
// initialize the hash table file
void RL_Hashing_Class::initStateAction()
{
    stateAction.open(fileName, ios::out|ios::in);
    stateAction.flags(ios::left);
    stateAction.setf(ios::fixed, ios::floatfield);
    // use stateAction because not enough memory

    for (int j=0; j<TTT_STATE; j++)
    {
        for (int i=0; i<TTT_ACTION; i++)
        {
            stateAction << setfill('0') << setw(12) << 0.001;
            if (i != TTT_ACTION)
                stateAction << ' ';
        }
        stateAction << endl;
    }
    stateAction.seekg(0, ios::beg);
    actionLength = 0;
    char ch;
    do    // length of an action in the file
        actionLength++;
    while ((ch=stateAction.get())!=' ');

    // find out the length of STATE * ACTION space
    stateAction.seekg(0, ios::beg);
    char q[200];

```

```

stateAction.getline(q, 200, '\n');
stateLength = strlen(q)+2;      // plus 2 because of \0 and \n
}
/////////////////////////////////////////////////////////////////
// return a hash table Q-value
/////////////////////////////////////////////////////////////////
double RL_Hashing_Class::getValue(const stringClass& s, const stringClass& a)
{
    int stateKey;
    stringClass tempState(s);
    if (find(tempState)==FALSE)
        insert(tempState);

    stateKey = getKey();

    stringClass temp(a);
    int action = (int)temp;

    cellPosInFile = stateKey * stateLength + action * actionLength;
    stateAction.seekg(cellPosInFile, ios::beg);
    stateAction >> value;
    return value;
}
/////////////////////////////////////////////////////////////////
// change a Q-value in the hash table
/////////////////////////////////////////////////////////////////
void RL_Hashing_Class::setValue(const stringClass& s,
                                const stringClass& a, double Q)
{
    stringClass tempState(s);

    int stateKey;
    if (find(tempState)==TRUE)
        stateKey = getKey();
    else {
        insert(tempState);
        stateKey = getKey();
    }
    stringClass temp(a);
    int action = (int)temp;

    cellPosInFile = stateKey * stateLength + action * actionLength;
    stateAction.seekg(cellPosInFile, ios::beg);
    stateAction << setfill('0') << setw(12) << Q;
}
/////////////////////////////////////////////////////////////////
// the maximum Q-value, Q-value is state-action pair value,
// parameter provided the state, so .. check all actions for the maximum
/////////////////////////////////////////////////////////////////
double RL_Hashing_Class::max(const stringClass& s)
{
    stringClass tempState(s);
    char chAction[2];
    stringClass strAction;

```

```

double temp, max=(double)-1.0;
BOOL exist = FALSE;
for (int action=0; action<9; action++)
{
    if (tempState[action] == ' ')
    {
        exist = TRUE;
        chAction[0] = '0' + action;
        chAction[1] = '\0';
        strAction = chAction;
        temp = getValue(tempState, strAction);
        if (temp > max)
            max = temp;
    }
}
if (exist)
    return max;
else return 0.0;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// neural network constructor
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
RL_Neural_Network_Class::RL_Neural_Network_Class(int restore, double a, double
        g, int hiddenSize, const stringClass& state, double value,
        double n, int d) : RL_Qvalue_Class(n,d)
{
    srand(123);
    double learning_rate = (double)a;
    double momentum = (double)g;
    memset(input, 0, inSize*sizeof(double));
    memset(output, 0, outSize*sizeof(double));
    if (restore) // restore weights from weight.txt
        BPnet = new Backprop_Network("weight.txt");
    else // create a new network
        BPnet = new Backprop_Network(learning_rate, momentum, layer,
            inSize, hiddenSize, outSize);}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// save the neural network weight before it quits...
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
RL_Neural_Network_Class::~RL_Neural_Network_Class()
{
    // save network weight
    ofstream outfile("weight.txt");
    BPnet->Save(outfile);
    outfile.close();
    delete BPnet;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// setup input for state based on the tictactoe board..
// " " -> 00 "X" -> 10 "O"->01
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void RL_Neural_Network_Class::binaryInput(const stringClass& s, int action)
{
    stringClass State(s);

```

```

int j=0;
memset(input, 0, inSize*sizeof(double));
memset(output, 0, outSize*sizeof(double));

for (int i=0; i<9; i++)
{
    switch(State[i])
    {
        case ' ':
            input[j++] = 0.0;
            input[j++] = 0.0;
            break;
        case 'X':
            input[j++] = 1.0;
            input[j++] = 0.0;
            break;
        case '0':
            input[j++] = 0.0;
            input[j++] = 1.0;
            break;
    }
}
}

/////////////////////////////////////////////////////////////////
// return the value of an approximated value that is scaled back to range [-1,1]
// to be used in the value function
/////////////////////////////////////////////////////////////////
double RL_Neural_Network_Class::getValue(const stringClass& s, const stringClass& a)
{
    stringClass tempState(s);
    stringClass tempAction(a);
    int action = (int)tempAction;
    binaryInput(tempState, action);

    data[0] = new Pattern(inSize, outSize, 0, input, output);

    BPnet->Set_Value(data[0]);    // Set Input Node Values
    BPnet->Run();                // Forward Pass

    double value;
    value = BPnet->Get_Value(action); // 0.0 <= value <=1.0;
    delete data[0];
    return 2.0*value - 1.0;
}

/////////////////////////////////////////////////////////////////
// Q-value [-1, 1] is scaled down to range [0,1] because of the sigmoid function
/////////////////////////////////////////////////////////////////
void RL_Neural_Network_Class::setValue(const stringClass& s ,
                                       const stringClass& a, double Q)
{
    stringClass original(s);
    stringClass tempAction(a);
    int action = (int)tempAction;

```

```

binaryInput(original, action);
data[0] = new Pattern(inSize, outSize, 0, input, output);

BPnet->Set_Value(data[0]);      // Set Input Node Values
BPnet->Run();                    // Forward Pass
for (int i=0; i<outSize; i++)
    output[i] = BPnet->Get_Value(i);    // 0.0 <= value <=1.0;

output[action] = (Q+1.0)*0.5;
data[1] = new Pattern(inSize, outSize, 1, input, output);

BPnet->Set_Value(data[1]); // Set Input Node Values
BPnet->Run();                // Forward Pass
BPnet->Set_Error(data[1]); // Set Desired Output in output layer
BPnet->Learn();                // Backward Pass
delete data[0];
delete data[1];
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// return maximum Q-value of state s
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
double RL_Neural_Network_Class::max(const stringClass& s)
{
    double tempVal, max = (double)-1.0;
    stringClass strAction;
    char cAct[3], *pAct;
    pAct = cAct;
    BOOL availAction = FALSE;
    for (int action=0; action<9; action++)
    {
        if (s[action] == ' ')
        {
            availAction = TRUE;
            cAct[0] = '0' + action;
            cAct[1] = '\0';
            strAction = pAct;
            tempVal = RL_Neural_Network_Class::getValue(s, strAction);
            if (tempVal > max)
            {
                max = tempVal;
            }
        }
    }
    if (availAction == TRUE)
        return max;
    else return 0.0;    // terminal
}

```

VITA

Kean Giap Lim

Candidate for the Degree of  
Master of Science

Thesis: REINFORCEMENT LEARNING IN GAME PLAYING

Major Field: Computer Science

Biographical:

Personal Data: Born in Kuala Lumpur, Malaysia, on September 5, 1973, son of Meow Hwa and Yew Peng Lim.

Education: Graduated from Sekolah Menengah Dato' Lokman, Kampung Pandan, Kuala Lumpur in December 1990; received Bachelor of Science degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma in December 1995. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 1998.

Experience: Employed by Oklahoma State University, the Computer Center for Integrated Manufacturing as a research assistant, 1996 to 1998; Oklahoma State University, Department of Biosystem Engineering as a graduate assistant, 1997 to 1998.

Professional Membership: Association for Computing Machinery.