ISSN: 2150-8402 Volume 2, Number 2, May 2010

Scientific Research

Journal of Intelligent Learning Systems and Applications

Editor-in-Chief: Prof. Xin Xu

ISSN: 2150-8402

www.scirp.org/journal/jilsa

Journal Editorial Board

ISSN: 2150-8402 (Print), 2150-8410 (Online)

http://www.scirp.org/journal/jilsa

J. Intelligent Learning Systems & Applications, **2010, 2, 55-118**
Published Online May 2010 in SciRes (http://www.SciRP.org/journal/jilsa/)

TABLE OF CONTENTS

Volume 2 Number 2 May 2010

Journal of Intelligent Learning Systems and Applications (JILSA)

Journal Information

SUBSCRIPTIONS

The *Journal of Intelligent Learning Systems and Applications* (Online at Scientific Research Publishing, www.SciRP.org) is published quarterly by Scientific Research Publishing, Inc., USA.

Subscription rates:

Print: $50 per issue.
To subscribe, please contact Journals Subscriptions Department, E-mail: sub@scirp.org

SERVICES

Advertisements
Advertisement Sales Department, E-mail: service@scirp.org

Reprints (minimum quantity 100 copies)
Reprints Co-ordinator, Scientific Research Publishing, Inc., USA.
E-mail: sub@scirp.org

COPYRIGHT

PRODUCTION INFORMATION

For manuscripts that have been accepted for publication, please contact:
E-mail: jilsa@scirp.org

J. Intelligent Learning Systems & Applications, 2010, 2, 55-56
doi:10.4236/jilsa.2010.22008 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

Editorial: Special Section on Reinforcement Learning and Approximate Dynamic Programming

Approximate dynamic programming (ADP) is to compute near-optimal solutions to Markov decision problems (MDPs) with large or continuous spaces. In recent years, the research works on ADP have been brought together with the reinforcement learning (RL) community [1-4]. RL is a machine learning framework for solving sequential decision making problems that can also be modeled as the MDP formalism. The common objective of RL and ADP is to develop efficient algorithms for sequential decision making under uncertain complex conditions. Therefore, there are many potential applications of RL and ADP in real-world problems such as autonomous robots, intelligent control, resource allocation, network routing, etc.

This special section of JILSA focuses on key research problems emerging at the junction of RL and ADP. After a rigorous reviewing process, three papers were accepted for publication in this special section.

The first paper by M. A. Wiering [5] focuses on the applications of reinforcement learning with value function approximation in game playing. In the paper, three different schemes were studied for learning to play Backgammon with temporal difference learning. The three training schemes include: 1) self-play, 2) playing against an expert program, and 3) viewing experts play against each other. Extensive experimental results using temporal difference methods with neural networks were provided to compare the three learning schemes. It was illustrated that the drawback of learning from experts is that the learning program has few chances for exploration. The results also indicate that observing an expert play is the worst method and learning by playing against an expert seems to be the best strategy.

The second paper by J. H. Zaragoza, and E. F. Morales [6] proposed a relational reinforcement learning approach with continuous actions, called TS-RRLCA, which is based on the combination of behavioral cloning and locally weighted regression. The TS-RRLCA approach includes two main stages to learn continuous action policy for robots in partially known environments. The first stage is to develop a relational representation of robot states and actions and the rQ-learning algorithm is applied with behavioral cloning so that optimized control policies with discrete actions can be obtained efficiently. In the second stage, the learned policy is transformed into a relational policy with continuous actions through a Locally Weighted Regression (LWR) process. The proposed method was successfully applied to a simulated and a real service robot for navigation and following tasks with different conditions.

The combination of reinforcement learning or approximate dynamic programming with learning from demonstration is studied in the third paper [7]. A learning strategy was proposed to generate a control field for a mobile robot in an unknown and uncertain environment, which integrates learning, generalization, and exploration into a unified architecture. Some Simulation results were provided to evaluate the performance of the proposed method.

Although RL and ADP provide efficient ways for developing machine intelligence in a trial-and-error manner, the incorporation of human intelligence is important for the successful applications of RL and ADP. In this special section on RL and ADP, all the three papers studied the relationships between machine intelligence and human intelligence in different aspects. The results of the first paper demonstrate that an expert program for game playing will be very helpful to develop computer programs using RL [5]. The usage of relational RL to incorporate human examples was investigated in the second paper [6]. In the third paper [7], the method of learning from human demonstration was employed to generate initial control field for an autonomous mobile robots. Therefore, the results in this special section will be good references for future research in related topics.

At last, I would like to thank all of the authors and reviewers who have made contributions to this special section.

Xin Xu
Editor-in-Chief,
JILSA

REFERENCES

[1] F. Y. Wang, H. G. Zhang and D. R. Liu, “Adaptive Dynamic Programming: An Introduction,” *IEEE Computational Intelligence Magazine*, May 2009, pp. 39-47.

[2] W. B. Powell, “Approximate Dynamic Programming: Solving the Curses of Dimensionality,” Wiley, Princeton,

NJ, 2007.

[3] R. S. Sutton and A. G. Barto, "Reinforcement Learning: an Introduction," MIT Press, Cambridge, MA, 1998.

[4] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-Dynamic Programming. Belmont," Athena Scientific, MA, 1996.

[5] M. A. Wiering, "Self-play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning," *Journal of Intelligent Learning Systems and Applications*, Vol. 2, 2010, pp. 55-66.

[6] J. H. Zaragoza and E. F. Morales, "Relational Reinforcement Learning with Continuous Actions by Combining Behavioural Cloning and Locally Weighted Regression," *Journal of Intelligent Learning Systems and Applications*, Vol. 2, 2010, pp. 67-77.

[7] D. Goswami and P. Jiang, "Experience Based Learning Controller," *Journal of Intelligent Learning Systems and Applications*, Vol. 2, 2010, pp. 78-83.

J. Intelligent Learning Systems & Applications, **2010, 2: 57-68**
doi:10.4236/jilsa.2010.22009 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning

Marco A. Wiering*

Department of Artificial Intelligence, University of Groningen, Groningen, Netherlands.
Email: m.a.wiering@rug.nl

Received October 22nd, 2009; revised January 10th, 2010; accepted January 30th, 2010.

ABSTRACT

A promising approach to learn to play board games is to use reinforcement learning algorithms that can learn a game position evaluation function. In this paper we examine and compare three different methods for generating training games: 1) *Learning by self-play,* 2) *Learning by playing against an expert program, and* 3) *Learning from viewing experts play against each other. Although the third possibility generates high-quality games from the start compared to initial random games generated by self-play, the drawback is that the learning program is never allowed to test moves which it prefers. Since our expert program uses a similar evaluation function as the learning program, we also examine whether it is helpful to learn directly from the board evaluations given by the expert. We compared these methods using temporal difference methods with neural networks to learn the game of backgammon.*

Keywords: *Board Games, Reinforcement Learning, TD(λ), Self-Play, Learning From Demonstration*

1. Introduction

The success of the backgammon learning program TD-Gammon of Tesauro (1992, 1995) was probably the greatest demonstration of the impressive ability of machine learning techniques to learn to play games. TD-Gammon used reinforcement learning [1,2] techniques, in particular temporal difference (TD) learning [2,3], for learning a backgammon evaluation function from training games generated by letting the program play against itself. This has led to a large increase of interest in such machine learning methods for evolving game playing computer programs from a randomly initialized program (*i.e.*, initially there is no a priori knowledge of the game evaluation function, except for a human extraction of relevant input features). Samuel (1959, 1967) pioneered research in the use of machine learning approaches in his work on learning a checkers program. In his work he already proposed an early version of temporal difference learning for learning an evaluation function.

For learning to play games, value function based reinforcement learning (or simply reinforcement learning) or evolutionary algorithms are often used. Evolutionary algorithms (EAs) have been used for learning to play backgammon [4], checkers [5], and Othello [6] and were quite successful. Reinforcement learning has been applied to learn a variety of games, including backgammon [7,8], chess [9,10], checkers [11,12,13], and Go [14]. Other machine learning approaches learn an opening book, rules for classifying or playing the endgame, or use comparison training to mimic the moves selected by human experts. We will not focus on these latter approaches and refer to [15] for an excellent survey of machine learning techniques applied to the field of game-playing.

EAs and reinforcement learning (RL) methods concentrate on evolving or learning an evaluation function for a game position and after learning choose positions that have the largest utility or value. By mapping inputs describing a position to an evaluation of that position or input, the game program can choose a move using some kind of look-ahead planning. For the evaluation function many function approximators can be used, but commonly weighted symbolic rules (a kind of linear network), or a multi-layer perceptron that can automatically learn non-linear functions of the input is used.

A difference between EAs and reinforcement learning algorithms is that the latter usually have the goal to learn the exact value function based on the long term reward (e.g., a win gives 1 point, a loss –1, and a draw 0), whereas EAs directly search for a policy which plays well without learning or evolving a good approximation of the result of a game. Learning an evaluation function with reinforcement learning has some advantages such as better fine-tuning of the evaluation function once it is quite good and the possibility to learn from single moves without playing an entire game. Finally, the evaluation function allows feedback to a player and can in combination with multiple outputs for different outcomes also be used for making the game-playing program play more or less aggressive.

In this paper we study the class of reinforcement learning methods named temporal difference (TD) methods. Temporal difference learning [3,7] uses the difference between two successive positions for backpropagating the evaluations of the successive positions to the current position. Since this is done for all positions occurring in a game, the outcome of a game is incorporated in the evaluation function of all positions, and hopefully the evaluation functions improves after each game. Unfortunately there is no convergence proof that current RL methods combined with non-linear function approximators such as feed-forward neural networks will find or converge to an optimal value function.

For learning a game evaluation function for mapping positions to moves (which is done by the agent), there are the following three possibilities for obtaining experiences or training examples; 1) Learning from games played by the agent against itself (learning by self-play), 2) Learning by playing against a (good) opponent, 3) Learning from observing other (strong) players play games against each other. The third possibility might be done by letting a strong program play against itself and let a learner program learn the game evaluation function from observing these games or from database games played by human experts.

Research Questions. In this paper we compare different methods for acquiring and learning from training examples. We pose ourselves the following research questions:

1) Which method combined with temporal difference learning results in the best performance after a fixed number of games? Is observing an expert player, playing against an expert, or self-play the best method?

2) When the learning program immediately receives accurate evaluations of encountered board positions, will it then learn faster than when it uses its initially randomized function approximator and TD-learning to get the board evaluations?

3) Is a function approximator with more trainable parameters more efficient for learning to play the game of backgammon than a smaller representation?

4) Which value for λ in TD (λ) works best for obtaining the best performance after a fixed number of games?

Outline. This paper first describes game playing programs in section 2. Section 3 describes reinforcement learning algorithms. Then section 4 presents experimental results with learning the game of backgammon for which the above mentioned three possible methods for generating training games are compared. Section 5 concludes this paper.

2. Game Playing Programs

Game playing is an interesting control problem often consisting of a huge number of states, and therefore has inspired research in artificial intelligence for a long time. In this paper we deal with two person, zero-sum, alternative move games such as backgammon, Othello, draughts, Go, and chess. Furthermore, we assume that there is no hidden state such as in most card games. Therefore our considered board games consist of:

1) A set of possible board positions.

2) A set of legal moves in a position.

3) Rules for carrying out moves.

4) Rules for deciding upon termination and the result of a game.

A game playing program consists of a move generator, a look-ahead algorithm, and an evaluation function. The move generator just generates all legal moves, possibly in some specific order (taking into account some priority). The look-ahead algorithm deals with inaccurate evaluation functions. If the evaluation function would be completely accurate, look-ahead would only need to examine board positions resulting from each legal move. For most games an accurate evaluation function is very hard to make, however. Therefore, by looking ahead many moves, positions much closer to the end of a game can be examined and the difference in evaluations of the resulting positions is larger and therefore the moves can be more easily compared. A well known method for looking ahead in games is the Minimax algorithm, however faster algorithms such as alpha-beta pruning, Negascout, or principal variation search [16,17] are usually used for good game playing programs.

If we examine the success of current game playing programs, such as Deep Blue which won against Kasparov in 1997 [18], then it relies heavily on the use of very fast computers and look-ahead algorithms. Deep Blue can compute the evaluation of about 1 million positions in a second, much more than a human being who examines less than 100 positions in a second. Also draughts playing programs currently place emphasis on look-ahead algorithms for comparing a large number of positions. Expert backgammon playing programs only use 3-ply look-ahead, however, and focus therefore much more on the evaluation function.

Board games can have a stochastic element such as backgammon. In backgammon dice are rolled to determine the possible moves. Although the dice are rolled before the move is made, and therefore for a one-step look-ahead the dice are no computational problem, this makes the branching factor for computing possible positions after two or more moves much larger (since then look-ahead needs to take into account the 21 outcomes of the two dice). This is the reason that looking ahead many moves in stochastic games is infeasible for human experts or computers. For this Monte Carlo simulations [19] can still be helpful for evaluating a position, but due to the stochasticity of these games, many games have to be simulated.

On the other hand, we argue that looking ahead is not very necessary due to the stochastic element. Since the evaluation function is determined by dice, the evaluation function will become smoother since a position's value is the average evaluation of positions resulting from all dice rolls. In fact, in backgammon it often does not matter too much whether some single stone or field occupied by 2 or more stones are shifted one place or not. This can be again explained by the dice rolls, since different dice in similar positions can results in a large number of equal subsequent positions. Looking ahead multiple moves for backgammon may be helpful since it combines approximate evaluations of many positions, but the variance may be larger. A search of 3-ply is commonly used by the best backgammon playing programs [7,8].

This is different with e.g. chess or draughts, since for these games (long) tactical sequences of moves can be computed which let a player win immediately. Therefore, the evaluations of many positions later vary significantly and are more easily compared. Furthermore, for chess or draughts moving a piece one position can make the difference between a winning and losing position. Therefore the evaluation function is much less smooth (evaluations of close positions can be very different) and harder to learn. We think that the success of learning to play backgammon [8] relies on this smoothness of the evaluation function. It is well known that learning smooth functions requires less parameter for a machine learning algorithm and therefore faster search for a good solution and better generalization.

In the next section we will explain how we can use TD methods for learning to play games. After that the results of using TD learning for learning the game of Backgammon using different strategies for obtaining training examples will be presented.

3. Reinforcement Learning

Reinforcement learning algorithms are able to let an agent learn from its experiences generated by its interaction with an environment. We assume an underlying Markov decision process (MDP) which does not have to be known to the agent. A finite MDP is defined as; 1) The state-space $S = \{s^1, s^2, \ldots, s^n\}$, where $s_t \in S$ denotes the state of the system at time t; 2) A set of actions available to the agent in each state A(s), where $a_t \in A(s_t)$ denotes the action executed by the agent at time t; 3) A transition function P (s, a, s') mapping state action pairs s, a to a probability distribution of successor states s'; 4) A reward function R(s, a, s') which denotes the average reward obtained when the agent makes a transition from state s to state s' using action a, where r_t denotes the (possibly stochastic) reward obtained at time t; 5) A discount factor $0 \leq \gamma \leq 1$ which discounts later rewards compared to immediate rewards.

3.1 Value Functions and Dynamic Programming

In optimal control or reinforcement learning, we are interested in computing or learning an optimal policy for mapping states to actions. We denote an optimal deterministic policy as $\pi^*(s) \rightarrow a^*|s$. It is well known that for each MDP, one or more optimal deterministic policies exist. An optimal policy is defined as a policy that receives the highest possible cumulative discounted rewards in its future from all states.

In order to learn an optimal policy, value-function based reinforcement learning [1,2,3] uses value functions to summarize the results of experiences generated by the agent in the past. We denote the value of a state $V^\pi(s)$ as the expected cumulative discounted future reward when the agent starts in state s and follows a particular policy π:

$$V^\pi(s) = E\left(\sum_{i=0} \gamma^i r_i \,|s_0 = s, \pi\right)$$

The optimal policy is the one which has the largest state-value in all states. It is also well-known that there exists a recursive equation known as the Bellman optimality equation [20] which relates a state value of the optimal value function to other optimal state values which can be reached from that state using a single local transition:

$$V^*(s) = \sum_{s'} P(s, \pi^*(s), s')\,(R(s, \pi^*(s), s') + \gamma V^*(s'))$$

Value iteration can be used for computing the optimal V-function. For this we repeat the following update many times for all states:

$$V^{k+1}(s) = \max_a \sum_{s'} P(s, a, s')\,(R(s, a, s') + \gamma V^k(s'))$$

The agent can then select optimal actions using:

$$\pi^*(s) = \arg\max_a \sum_{s'} P(s, a, s')\,(R(s, a, s') + \gamma V^*(s'))$$

3.2 Reinforcement Learning

Although dynamic programming algorithms can be efficiently used for computing optimal solutions for particular MDPs, they have some problems for more practical applicability; 1) The MDP should be known a-priori; 2) For large state-spaces the computational time would become very large; 3) They cannot be directly used in continuous state-action spaces.

Reinforcement learning algorithms can cope with these problems; first of all the MDP does not need to be known a-priori, all that is required is that the agent is allowed to interact with an environment which can be modeled as an MDP; secondly, for large or continuous state-spaces, an RL algorithm can be combined with a function approximator for learning the value function. When combined with a function approximator, the agent does not have to compute state-action values for all possible states, but can concentrate itself on parts of the state-space where the best policies lead into.

There are a number of reinforcement learning algorithms, the first one known as temporal difference learning or TD(0) [3] computes an update of the state value function after making a transition from state s_t to state s_{t+1} and receiving a reward of r_t on this transition by using the temporal difference learning rule:

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

where $0 < \alpha \leq 1$ is the learning rate (which is treated here as a constant, but should decay over time for convergence proofs). Although it does not compute action-value functions, it can be used to learn the value function of a fixed policy (policy-evaluation). Furthermore, if combined with a model of the environment, the agent can use a learned state value function to select actions:

$$\pi(s) = \text{argmax}_a \quad \sum_{s'} P(s, a, s')(R(s, a, s') + \gamma V(s'))$$

It is possible to learn the V-function of a changing policy that selects greedy actions according to the value function. This still requires the use of a transition function, but can be used effectively for e.g. learning to play games [7,8].

There exists a whole family of temporal difference learning algorithms known as TD(λ)-algorithms [3] which are parameterized by the value λ which makes the agent look further in the future for updating its value function. It has been proved [21] that this complete family of algorithms converges under certain conditions to the same optimal state value function with probability 1 if tabular representations are used. The TD(λ)-algorithm works as follows. First we define the TD(0)-error of $V(s_t)$ as:

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

TD(λ) uses a factor $\lambda \in [0, 1]$ to discount TD-errors of future time steps:

$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t^{\lambda}$$

where the TD(λ)-error δ_t^{λ} is defined as

$$\delta_t^{\lambda} = \sum_{i=0} (\gamma\lambda)^i \delta_{t+i}$$

Eligibility traces. The updates above cannot be made as long as TD errors of future time steps are not known. We can compute them incrementally, however, by using eligibility traces [3,22]. For this we use the update rule:

$$V(s) = V(s) + \alpha \delta_t e_t(s)$$

for all states, where $e_t(s)$ is initially zero for all states and updated after every step by:

$$e_t(s) = \gamma\lambda e_{t-1}(s) + \eta_t(s)$$

where $\eta_t(s)$ is the indicator function which returns 1 if state s occurred at time t, and 0 otherwise. A faster algorithm to compute exact updates is described in [23]. The value of λ determines how much the updates are influenced by events that occurred much later in time. The extremes are TD(0) and TD(1) where (online) TD(1) makes the same updates as Monte Carlo sampling. Although Monte Carlo sampling techniques that only learn from the final result of a game do not suffer from biased estimates, the variance in updates is large and that leads to slow convergence. A good value for λ depends on the length of an epoch and varies between applications, although often a value between 0.6 and 0.9 works best.

3.3 Reinforcement Learning with Neural Networks

To learn value functions for problems with many state variables, there is the curse of dimensionality; the number of states increases exponentially with the number of state variables, so that a tabular representation would quickly become infeasible in terms of storage space and computational time. Also when we have continuous states, a tabular representation requires a good discretization which has to be done a-priori using knowledge of the problem, and a fine-grained discretization will also quickly lead to a large number of states. Therefore, instead of using tabular representations it is more appropriate to use function approximators to deal with large or continuous state spaces.

There are many function approximators available such as neural networks, self-organizing maps, locally weighted learning, and support vector machines. When we want to combine a function approximator with reinforcement learning, we want it to learn fast and online after each experience, and be able to represent continuous functions. Appropriate function approximators combined with reinforcement learning are therefore feed-forward neural networks [24].

In this paper we only consider fully-connected feed-forward neural networks with a single hidden layer. The architecture consist of one input layer with input units (when we refer to a unit, we also mean its activation): $I_1, \ldots, I_{|I|}$, where $|I|$ is the number of input units, one hidden layer H with hidden units: $H_1, \ldots, H_{|H|}$, and one output layer with output units: $O_1, \ldots, O_{|O|}$. The network has weights: w_{ih} for all input units I_i to hidden units H_h, and weights: w_{ho} for all hidden H_h to output units O_o. Each hidden unit and output unit has a bias b_h or b_o with a constant activation of 1. The hidden units most often use sigmoid activation functions, whereas the output units use linear activation functions.

Forward propagation. Given the values of all input units, we can compute the values for all output units with forward propagation. The forward propagation algorithm looks as follows:

1) Clamp the input vector I by perceiving the environment.

2) Compute the values for all hidden units $H_h \in H$ as follows:

$H_h = \sigma(\sum_i w_{ih} I_i + b_h)$, where $\sigma(x)$ is the Sigmoid function: $\sigma(x) = 1/(1+e^{-x})$.

3) Compute the values for all output units $O_o = \sum_h w_{ho} H_h + b_o$.

Backpropagation. For training the system we can use the back-propagation algorithm [25]. The learning goal is to learn a mapping from the inputs to the desired outputs D_o for which we update the weights after each example. For this we use backpropagation to minimize the squared error measure:

$$E = ½\sum_o (D_o - O_o)^2$$

To minimize this error function, we update the weights and biases in the network using gradient descent steps with learning rate α. We first compute the delta values of the output units (for a linear activation function):

$$\delta_O(o) = (D_o - O_o)$$

Then we compute the delta values of all hidden units (for a sigmoid activation function):

$$\delta_H(h) = \sum_o \delta_O(o) w_{ho} H_h(1 - H_h)$$

Then we change all hidden-output weights and output bias values:

$$w_{ho} = w_{ho} + \alpha\delta_O(o)H_h;\ b_o = b_o + \alpha\delta_O(o)$$

And finally we change all input-hidden weights and hidden bias values:

$$w_{ih} = w_{ih} + \alpha\delta_H(h)I_i;\ b_h = b_h + \alpha\delta_H(h)$$

Offline TD-methods. All we need is a desired output and then backpropagation can be used to compute weight updates to minimize the error-function on every different example. To get the desired output, we can simply use offline temporal difference learning [26] which waits until an epoch has ended and then computes desired values for the different time-steps. For learning to play games this is useful, since learning from the first moves will not immediately help to play the rest of the game better. In this paper we used the offline TD(λ) method which provides the desired values for each board position, taking into account the result of a game and the prediction of the result by the next state. The final position at time-step T is scored with the result r_T of the game, *i.e.* a win for white (= 1), a win for black (= –1) or a draw (= 0).

$$V'(s_T) = r_T \qquad (1)$$

The desired values of the other positions are given by the following function:

$$V'(s_t) = \gamma V(s_{t+1}) + r_t + \lambda\gamma(V'(s_{t+1}) - V(s_{t+1}))$$

After this, we use $V'(s_t)$ as the desired value of state s_t and use back-propagation to update all weights. In Backgammon, we used a minimax TD-rule for learning the game evaluation function. Instead of using an input that indicates which player is allowed to move, we always reverted the position so that white was to move. In this case, evaluations of successive positions are related by $V(s_t) = -V(s_{t+1})$. Without immediate reward and a discount factor of 1, the minimax TD-update rule becomes:

$$V'(s_t) = -V(s_{t+1}) + \lambda(V(s_{t+1}) - V'(s_{t+1}))$$

4. Experiments with Backgammon

Tesauro’s TD-Gammon program learned after about 1,000,000 games to play at human world class level, but already after 300,000 games TD-Gammon turned out to be a good match against the human grand-master Robertie. After this TD-Gammon was enhanced by a 3-ply look-ahead strategy that made it even stronger. Currently, TD-Gammon is still probably the best backgammon playing program in the world, but other programs such as BGBlitz from Frank Berger or Fredrik Dahl’s Jellyfish also rely on neural networks as evaluation functions and obtained a very good playing level. All of these programs are much better than Berliner’s backgammon playing program BKG [27] which was implemented using human designed weighted symbolic rules to get an evaluation function.

4.1 Learning an Expert Backgammon Program

We use an expert backgammon program against which we can train other learning programs and which can be used for generating games that can be observed by a learning program. Furthermore, in later experiments we can evaluate the learning programs by playing test-games against this expert. To make the expert player we used TD-learning combined with learning from self-play using hierarchical neural network architecture. This program was trained by playing more than 1 million games against itself. Since the program was not always improving by letting it play more training games, we tested the program after each 10,000 games for 5,000 test games against the best previous saved version. Then we recorded the score for each test and the weights of the network architecture with the highest score were saved. Then after each 100,000 games we made a new opponent which was the previous network with the highest score over all tests and this program was also used as learning program and further trained by self-play while testing it against the previous best program. This was repeated until there was no more progress, *i.e.* the learning program was not able to significantly beat the previous best learned program anymore. This was after more than 1,000,000 training games.

Architecture. We used modular neural network architecture, since different strategic positions require different knowledge for evaluating the positions [28]. Therefore we used a neural network architecture consisting of the following 9 neural networks for different strategic position classes, and we also show how many learning examples these networks received during training this architecture by self-play:

1) One network for the endgame; all stones are in the inner-board for both players or taken out (10.7 million examples).

2) One network for the racing game or long endgame; the stones can not be beaten anymore by another stone (10.7 million examples).

3) One network for positions in which there are no stones on the bar or stones in the first 6 fields for both players (1.9 million examples).

4) One network if the player has a prime of 5 fields or more and the opponent has one piece trapped by it (5.5 million examples).

5) One network for back-game positions where one player has a significant pip-count disadvantage and at least three stones in the first 6 fields (6.7 million examples).

6) One network for a kind of holding game; the player has a field with two stones or more or one of the 18, 19, 20, or 21 points (5.9 million examples).

7) One network if the player has all its stones further than the 8 point (3.3 million examples).

8) One network if the opponent has all its stones further than the 8 point (3.2 million examples).

9) One default network for all other positions (34.2 million examples).

For each position which needs to be evaluated, our symbolic categorization module uses the above rules to choose one of the 9 networks to evaluate (and learn) a position. The rules are followed from the first category to the last one, and if no rule applies then the default category and network is used.

Input features. Using this modular design, we also used different features for different networks. E.g., the endgame network does not need to have inputs for all fields since all stones have been taken out or are in the inner-board of the players. For the above mentioned neural network modules, we used different inputs for the first (endgame), second (racing game), and other (general) categories. The number of inputs for them is:

1) For the endgame we used 68 inputs, consisting of 56 inputs describing raw input information and 12 higher level features.

2) For the racing game (long endgame) we used 277 inputs, consisting of the same 68 inputs as for the endgame, another 192 inputs describing the raw board information, and 17 additional higher level features.

3) For the rest of the networks (general positions) we used 393 inputs consisting of 248 inputs describing raw board information and 145 higher level features including for example the probabilities that stones can be hit by the opponent in the next move.

For the neural networks we used 7 output units in which one output learned on the average result and the other six outputs learned a specific outcome (such as winning with 3, 2, or 1 point or losing with 3, 2, or 1 point). The good thing of using multiple output units is that there is more learning information going in the networks. Therefore the hidden units of the neural networks need to be useful for storing predictive information for multiple related subtasks, possibly resulting in better representations [29]. For choosing moves, we combined the average output with the combined outputs of the other output neurons to get a single board position evaluation. For this we took the average of the single output (with a value between –3 and 3) and the combined value of the other outputs times their predicted probability values. Each output unit only learned from the same output unit in the next positions using TD-learning (so the single output only learned from its own evaluations of the next positions). Finally, the number of hidden units (which use a sigmoid activation function) was 20 for the endgame and long endgame, and 40 for all other neural networks. We call the above described network architecture the large neural network architecture and trained it by self-play using TD(λ) learning with a learning rate of 0.01, a discount factor γ of 1.0, and a value for λ of 0.6. After learning we observed that the 2 different evaluation scores were always quite close and that the 6 output units usually had a combined activity close to 1.0 with only sometimes small negative values (such as –0.002) for single output units if the probability of the result was 0, which only have a small influence on the evaluation of a position.

Now we obtained an expert program, we can use it for our experiments in analyzing the results of new learners that train by self-play, train by playing against this expert, or learn by viewing games played by the expert against itself.

4.2 Experiments with Learning Backgammon

We first made a number of simulations in which 200,000 training games were used and after each 5,000 games we played 5,000 test games between the learner and the expert to evaluate the learning program. Because these simulations took a lot of time (several days for one simulation), they were only repeated two times for every setup.

The expert program was always the same as described before. For the learning program we also made use of a smaller architecture consisting of three networks; one for the endgame of 20 hidden units, one for the long endgame (racing game) of 20 hidden units, and one for the other board positions with 40 hidden units. We also used a larger network architecture with the same three networks, but with 80 hidden units for the other board positions, and finally we used an architecture with 20, 20, 40 hidden units with a kind of radial basis activation function: $H_j = e^{-(\sum W_{ij} I_i + b_j)^2}$. These architectures were trained by playing training games against the expert. We also experimented with a small network architecture that learns by self-play or by observing games played by the expert against itself.

Because the evaluation scores fluctuate a lot during the simulation, we smoothed them a bit by replacing the evaluation of each point (test after n games) by the average of it and its two adjacent evaluations. Since we used 2 simulations, each point is therefore an average of 6 evaluations obtained by testing the program 5,000 games against the expert (without the possibility of doubling the cube). For all these experiments we used extended backpropagation [30] and TD(λ)-learning with a learning rate of 0.01 and an eligibility trace factor λ of 0.6 that gave the best results in preliminary experiments. **Figures 1** and **2** show the obtained results.

First of all, it can be noted that the neural network architecture with RBF like activation functions for the hidden units works much worse. Furthermore, it can be seen that most other approaches work quite well and reach equity of almost 0.5. **Table 1** shows that all architectures, except for the architecture using RBF neurons, obtained an equity higher than 0.5 in at least one of

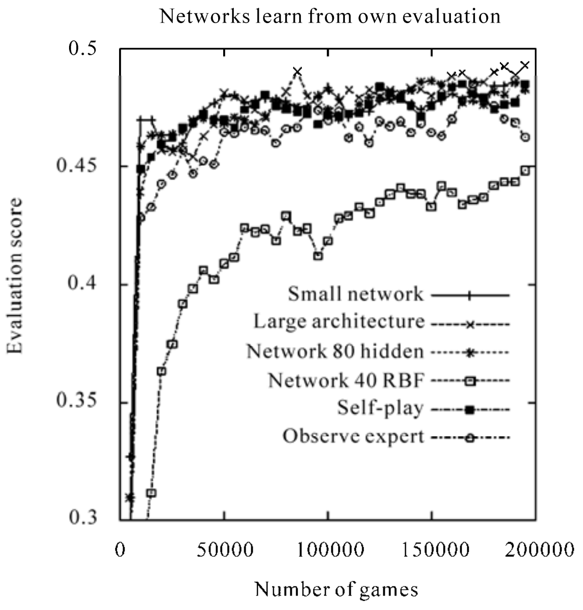


Figure 1. Results for different architectures from learning against the expert, and the small architecture that learns by self-play or by observing games of the expert

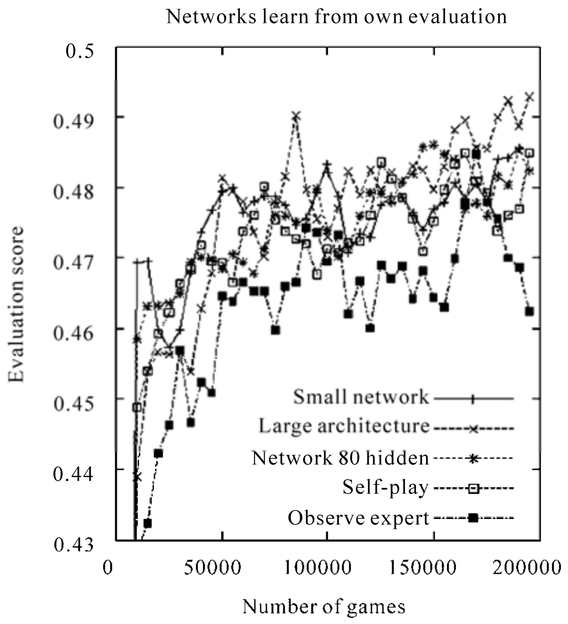


Figure 2. Results for different architectures from learning against the expert, and the small architecture that learns by self-play or by observing games of the expert. More detailed plot without the architecture with RBF hidden units

Table 1. Results for the different methods as averages of 6 matches of 5,000 games played against the expert. Note that the result after 5,000 games is the average of the tests after 100, 5000, and 10000 games

Architecture	5000	100,000	175,000	Max after	Max eval
Small Network	0.327	0.483	0.478	190,000	0.508
Large architecture	0.290	0.473	0.488	80,000	0.506
Network 80 hidden	0.309	0.473	0.485	155,000	0.505
Network 40 RBF	0.162	0.419	0.443	120,000	0.469
Small network Self-play	0.298	0.471	0.477	200,000	0.502
Small network Observing	0.283	0.469	0.469	110,000	0.510

the 80 tests. Testing these found solutions 10 times for 5000 games against the expert indicated that their playing strengths were equal. If we take a closer look at **Figure 2**, we can see that the large architecture with many module finally performs a bit better than the other approaches and that learning by observing the expert reaches a slightly worse performance.

Smaller simulations. We also performed a number of smaller simulations of 15,000 training games where we tested after each 500 games for 500 testing games. We repeated these simulations 5 times for each neural network architecture and method for generating training games. Because there is an expert available with the same kind of evaluation function, it is also possible to learn with TD-learning using the evaluations of the expert itself. This is very similar to supervised learning, although the agent generates its own moves (depending on the method for generating games). In this way, we can analyze what the impact of bootstrapping on an initially bad evaluation function is compared to learning immediately from outputs for positions generated by a better evaluation function. Again we used extended back-propagation [30] and TD(λ) with a learning rate of 0.01 and set λ = 0.6.

In **Figure 3**, we show the results of the smaller architecture consisting of three networks with 20, 20, and 40 hidden units. We also show the results in **Figure 4** where we let the learning programs learn from evaluations given by the expert program, but for which we still use TD-learning on the expert's evaluations with λ = 0.6 to make training examples.

The results show that observing the expert play and learning from these generated games progress slower and reach slightly worse results within 15,000 games if the program learns from its own evaluation function. In **Figure 4** we can see faster learning and better final results if the programs learn from the expert's evaluations (which is like supervised learning), but the differences are not very large compared to learning from the own evaluation function. It is remarkable that good performance

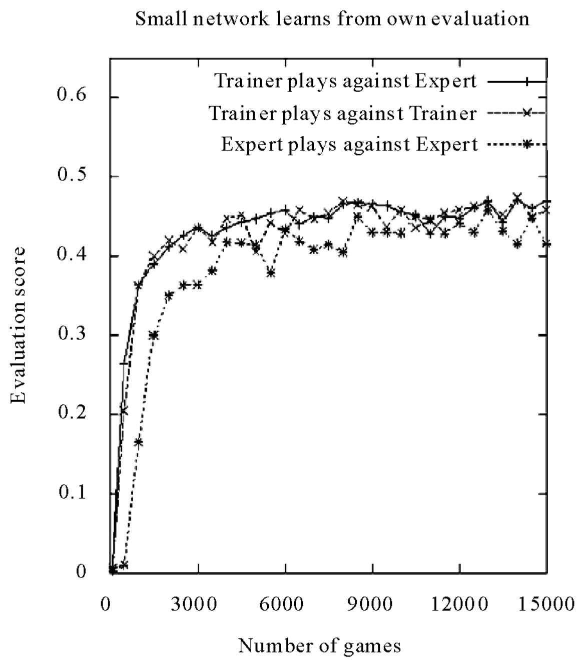


Figure 3. Results for the small architecture when using a particular method for generating games. The evaluation on which the agent learns is its own

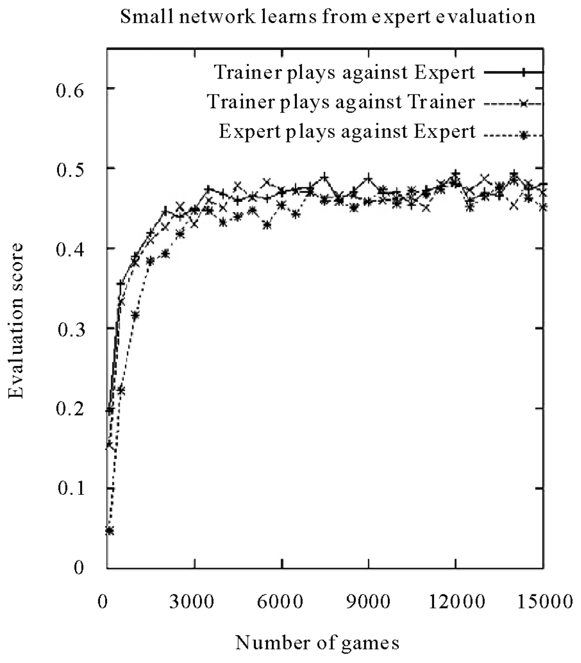


Figure 4. Results when the expert gives the evaluations of positions

has already been obtained after only 5,000 training games.

In **Table 2** we can see that if we let the learning program learn from games played against the expert, in the beginning it almost always loses (its average test-result or equity after 100 training games is 0.007), but already after 500 training games the equity has increased to an average value of 0.26. We can conclude that the learning program can learn its evaluation function by learning from the good positions of its opponent. This good learning performance can be attributed to the minimax TD-learning rule, since otherwise always losing will quickly result in a simple evaluation function that always returns a negative result. However, using the minimax TD-learning rule, the program does not need to win many games in order to learn the evaluation function. Learning by self-play performs almost as good as learning from playing against the expert. If we use the expert's evaluation function then learning progresses much faster in the beginning, although after 10,000 training games almost the same results are obtained. Learning by observing the expert playing against itself progresses slower and reaches worse results if the learning program learns from its own evaluation function. If we look at the learning curve, we can still see that it is improving however.

We repeated the same simulations for the large architecture consisting of 9 modules. The results are shown in **Figures 5** and **6**. The results show that learning with the large network architecture progresses much slower, which can be explained by the much larger number of

Table 2. Results for the three different methods for generating training games with learning from the own or the expert's evaluation function. The results are averages of 5 simulations

Method	Eval-function	100	500	1000	5000	10,000
Self-play	Own	0.006	0.20	0.36	0.41	0.46
Self-play	Expert	0.15	0.33	0.38	0.46	0.46
Against expert	Own	0.007	0.26	0.36	0.45	0.46
Against expert	Expert	0.20	0.35	0.39	0.47	0.47
Observing expert	Own	0.003	0.01	0.16	0.41	0.43
Observing expert	Expert	0.05	0.22	0.32	0.45	0.46

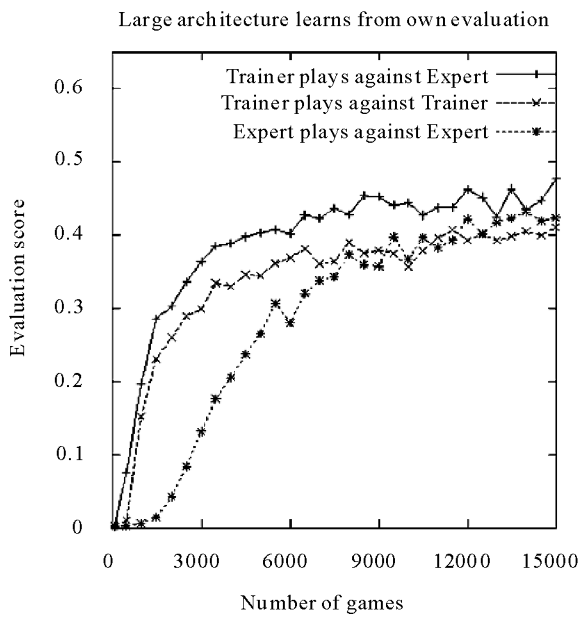


Figure 5. Results for the large architecture when using a particular method for generating games. The evaluation on which the agent learns is its own

parameters which need to be trained and the fewer examples for each individual network. The results also show that learning from observing the expert play against itself performs worse than the other methods, although after 15,000 games this method also reaches quite high equities, comparable with the other methods. The best method for training the large architecture is when games are generated by playing against the expert. **Figure 6** shows faster progress if the expert's evaluations are used.

Effect of λ. Finally, we examine what the effect of different values for λ is when the small architecture learns by playing against the expert. We tried values for λ of 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0. When using λ = 1 we needed to use a smaller learning-rate, since otherwise initially the weights became much too large. Therefore we used a learning rate of 0.001 for λ = 1.0 and a learning rate of 0.01 for the other values for λ. **Figure 7** shows the results averaged over 5 simulations. It can be seen that a λ-value of 1.0 works much worse and that values of 0.6 or 0.8 perform the best. **Table 3** shows the results after 100, 500, 1000, 5000, and 10,000 games. We can see that higher values of λ initially result in faster learning which

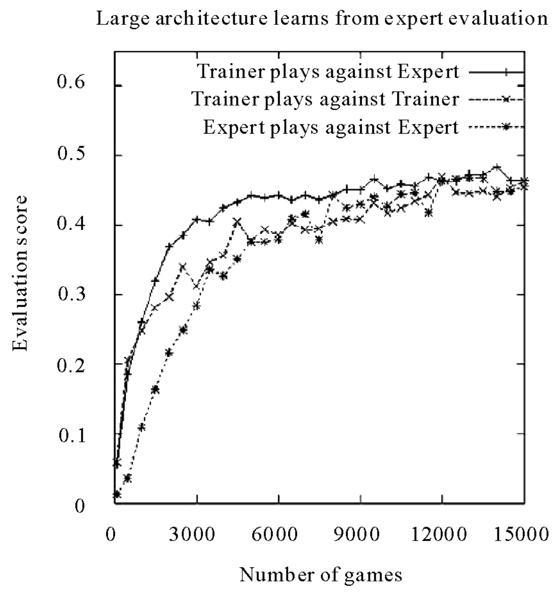


Figure 6. Results for the large architecture when using a particular method for generating games. Results when the expert gives the evaluations

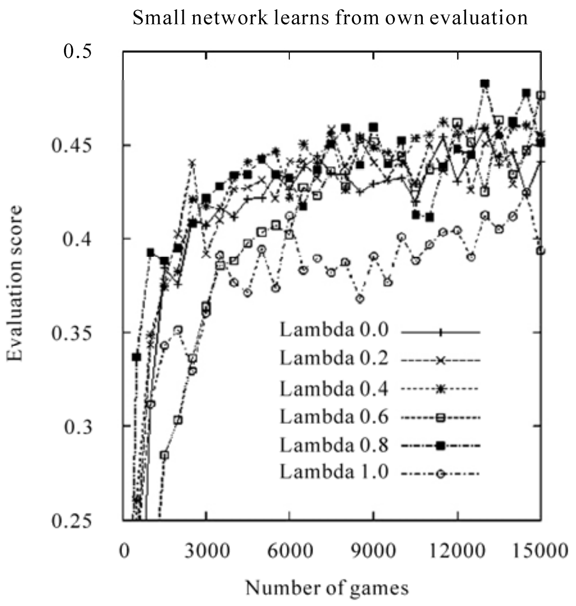


Figure 7. Results for the small architecture when using different values for λ. The games are generated by self-play

Table 3. Results for different values of λ when the small architecture learns against the expert

λ	100	500	1000	5000	10,000
0.0	0.004	0.13	0.31	0.42	0.43
0.2	0.002	0.24	0.34	0.43	0.45
0.4	0.002	0.26	0.35	0.44	0.44
0.6	0.007	0.26	0.36	0.45	0.46
0.8	0.06	0.34	0.39	0.44	0.45
1.0	0.12	0.23	0.31	0.39	0.40

can be explained by the fact that bootstrapping from the initially random evaluation function does not work too well and therefore larger eligibility traces are profitable. After a while λ values between 0.2 and 0.8 perform all similarly.

4.3 Discussion

Learning a good evaluation function for backgammon with temporal difference learning appears to succeed very well. Already within few thousands of games which can be played in less than one hour a good playing level is learned with equity of around 0.45 against the expert program. We expect this equity to be similar to a human player who regularly plays backgammon. The results show that learning by self-play and by playing against the expert obtain the same performance.

Learning by observing an expert play progresses approximately two or three times slower than the other methods. In our current experiments the learning program observed another program that still needed to select moves. Therefore there was no computational gain in generating training games. However, if we would have used a database, then in each position also one-step look-ahead would not be needed. Since the branching factor for a one-step look-ahead search is around 16 for backgammon, we would gain 94% of the computational time for generating and learning from a single game. Therefore learning from database games could still be advantageous compared to learning by self-play or playing against an expert. A problem of using a (small) database is that overfitting the evaluation function may occur. This may be solved by combining this approach with learning by self-play. In the large experiment, the learning behavior of the method that learns by observing the expert is a bit more fluctuating, but it still obtained equity a bit larger than 0.5 during one of the test-games in the large experiment and additional tests indicated that its playing strength at that point was equal to the expert player.

We also noted that training large architectures initially takes longer which can be simply explained by the larger number of parameters which need to be learned and fewer examples for individual modules. After training for a longer time, such bigger architectures can reach higher performance levels than smaller architectures. We note that since the agent learns on the same problem as on which it is tested, in these cases overfitting does not occur. A large value for λ (larger than 0.8) initially helps to improve the learning speed, but after some time smaller values for λ (smaller than 0.8) perform better. An annealing schedule for λ may therefore be useful. Finally we observed in all experiments that the learning programs are not always improving by playing more games. This can be explained by the fact that there is no convergence guarantee for RL and neural networks. Therefore testing the learning program against other fixed programs on a regular basis is necessary to be able to save the best learning program. It is interesting to note the similarity to evolutionary algorithms evolving game playing programs which also use tests. However, we expect that temporal difference learning and gradient descent is better for fine-tuning the evaluation function than a more randomized evolutionary search process**.**

Another approach that receives a lot of attention in recent RL research and good results for particular control problems is kernel-based least policy iteration (LSPI) learning [31]. However, it is unlikely that RBF kernels will generalize well to the huge state space of backgammon and that therefore kernel based LSPI is not likely to be successful. In fact, we implemented Support vector machines with RBF kernels for the game of Othello, and this showed indeed that RBF kernels are not good for games involving huge state-spaces. For this sigmoid functions are needed, but they are difficult to use as kernels, since they require a lot of structural design. The use of neural networks with sigmoid activation functions is therefore the current method of choice for difficult games.

5. Conclusions

In this paper different strategies for obtaining training examples for learning game evaluation functions have been examined. The possible advantage of playing against or observing an expert, namely that games are initially played at a high level was not clearly shown in the experimental results. We will now return to our research questions and answer them here.

1) Question 1. Which method combined with temporal difference learning results in the best performance after a fixed number of games? Is observing an expert player, playing against an expert, or self-play the best method?

Answer. The results indicate that observing an expert play is the worst method. The reason can be that the learning program is never actively involved in playing and therefore can not learn to penalize particular moves that it may have overestimated. Learning by playing against an expert seems to be the best strategy. Another approach that could be useful is learning from the expert combined with learning by self-play.

2) Question 2. When the learning program immediately receives accurate evaluations of encountered board positions, will it then learn faster than when it uses its initially randomized function approximator and TD-learning to estimate the board evaluations?

Answer. Initially, learning goes much faster when accurate evaluations are given. However, after 10,000 training games, the disadvantage of the initially randomized function approximator has almost disappeared.

3) Question 3. Is a function approximator with more trainable parameters more efficient for learning to play the game of backgammon than a smaller representation?

Answer. Yes, in general the larger function approximators obtain better performance levels, although in the beginning they learn at a slower rate. Since the agent is tested on exactly the same problem as on which it is trained (different from supervised learning), overfitting does not occur in reinforcement learning.

4) Question 4. Which value for λ in TD(λ) works best for obtaining the best performance after a fixed number of games?

Answer. Initially larger values for λ result in a faster learning rate. However, the final performance is best for intermediate values of λ around 0.6. It should be noted that this observation is quite problem specific.

Future work. Although in this paper it was demonstrated that learning from observing an expert is not profitable to learn to play backgammon, we also mentioned some advantages of using an expert or a database. Advantages of learning from experts are that the system does not explore the whole huge state-space and that in some applications it is a safer method for obtaining experiences than learning by trial-and-error. Furthermore, learning game evaluation functions from databases has the advantage that no look-ahead during game-play is necessary.

Learning from experts or databases can also be used for other applications, such as learning in action or strategic computer games for which human games played with a joystick can be easily recorded. Furthermore, for therapy planning in medicine, databases of therapies may be available and could therefore be used for learning policies. For robotics, behavior may be steered by humans and these experiences can be recorded and then learned by the robot [32]. Thus, we still think that learning from observing an expert has many advantages and possibilities for learning control knowledge, although care should be taken that the learner tries out its own behavior during learning.

REFERENCES

[1] L. P. Kaelbling, M. L. Littman and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, Vol. 4, 1996, pp. 237-285.

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

[3] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, Vol. 3, 1988, pp. 9-44.

[4] J. B. Pollack and A. D. Blair, "Why Did TD-Gammon Work," In: D. S. Touretzky, M. C. Mozer and M. E. Hasselmo, Ed., *Advances in Neural Information Processing Systems* 8, MIT Press, Cambridge, MA, 1996, pp. 10-16.

[5] D. B. Fogel, "Evolving a Checkers Player without Relying on Human Experience," *Intelligence*, Vol. 11, No. 2, 2000, pp. 20-27.

[6] D. E. Moriarty, "Symbiotic Evolution of Neural Networks in Sequential Decision Tasks," PhD thesis, Department of Computer Sciences, The University of Texas at Austin, USA, 1997.

[7] G. Tesauro, "Practical Issues in Temporal Difference Learning," In: D. S. Lippman, J. E. Moody and D. S. Touretzky, Ed., *Advances in Neural Information Processing Systems* 4, Morgan Kaufmann, San Mateo, CA, 1992, pp. 259-266.

[8] G. J. Tesauro, "Temporal Difference Learning and TD-Gammon," *Communications of the ACM*, Vol. 38, 1995, pp. 58-68.

[9] S. Thrun, "Learning to Play the Game of Chess," In: G. Tesauro, D. Touretzky and T. Leen, Ed., *Advances in Neural Information Processing Systems* 7, Morgan Kaufmann, San Fransisco, CA, 1995, pp. 1069-1076.

[10] J. Baxter, A. Tridgell and L. Weaver, "Knightcap: A Chess Program that Learns by Combining TD(λ) with Minimax Search," Technical report, Australian National University, Canberra, 1997.

[11] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal on Research and Development*, Vol. 3, No. 3, 1959, pp. 210-229.

[12] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers II—Recent Progress," *IBM Journal on Research and Development*, Vol. 11, No. 6, 1967, pp. 601-617.

[13] J. Schaeffer, M. Hlynka and V. Hussila, "Temporal Difference Learning Applied to a High-Performance Game," *In Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, WA, USA, 2001, pp. 529-534.

[14] N. N. Schraudolph, P. Dayan and T. J. Sejnowski, "Temporal Difference Learning of Position Evaluation in the Game of Go," In: J. D. Cowan, G. Tesauro and J. Alspector, Ed., *Advances in Neural Information Processing Systems*, Morgan Kaufmann, San Francisco, CA, 1994, pp. 817-824.

[15] J. Furnkranz, "Machine Learning in Games: A Survey," In: J. Furnkranz and M. Kubat, Ed., *Machines that learn to Play Games*, Nova Science Publishers, Huntington,

NY, 2001, pp. 11-59.

[16] A. Plaat, "Research Re: search and Re-search," PhD thesis, Erasmus University Rotterdam, Holland, 1996.

[17] J. Schaeffer, "The Games Computers (and People) Play," *Advances in Computers*, Vol. 50, 2000, pp. 189-266.

[18] J. Schaeffer and A. Plaat, "Kasparov versus Deep Blue: The Re-match," *International Computer Chess Association Journal*, Vol. 20, No. 2, 1997, pp. 95-102.

[19] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," *Proceedings of the fifth International Conference on Computers and Games*, Turin, Italy, 2006, pp. 72-83.

[20] R. Bellman, "Dynamic Programming," Princeton University Press, USA, 1957.

[21] J. N. Tsitsiklis, "Asynchronous Stochastic Approximation and Q-learning," *Machine Learning*, Vol. 16, 1994, pp. 185-202.

[22] A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike Adaptive Elements that Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 13, 1983, pp. 834-846.

[23] M. A. Wiering and J. H. Schmidhuber, "Fast Online Q(λ)," *Machine Learning*, Vol. 33, No. 1, 1998, pp. 105-116.

[24] C. M. Bishop, "Neural Networks for Pattern Recognition," Oxford University, New York, 1995.

[25] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning Internal Representations by Error Propagation," In: D. E. Rumelhart and J. L. Mcclelland, Ed., *Parallel Distributed Processing*, MIT Press, USA, 1986, pp. 318-362.

[26] L.-J. Lin, "Reinforcement Learning for Robots Using Neural Networks," PhD thesis, Carnegie Mellon University, Pittsburgh, 1993.

[27] H. Berliner, "Experiences in Evaluation with BKG—A Program that Plays Backgammon," *In Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 1, 1977, pp. 428-433.

[28] J. A. Boyan, "Modular Neural Networks for Learning Context-Dependent Game Strategies," Master's thesis, University of Chicago, USA, 1992.

[29] R. Caruana and V. R. de Sa, "Promoting Poor Features to Supervisors: Some Inputs Work Better as Outputs," In: M. C. Mozer, M. I. Jordan and T. Petsche, Ed., *Advances in Neural Information Processing Systems* 9, Morgan Kaufmann, San Mateo, CA, 1997, pp.246-252.

[30] A. Sperduti and A. Starita, "Speed up Learning and Network Optimization with Extended Backpropagation," *Neural Networks*, Vol. 6, 1993, pp. 365-383.

[31] X. Xu, D. Hu and X. Lu, "Kernel-Based Least Squares Policy Iteration for Reinforcement Learning," *IEEE Transactions on Neural Networks*, Vol. 18, No. 4, 2007, pp. 973-992.

[32] W. Smart and L. Kaelbling, "Effective Reinforcement Learning for Mobile Robots," *Proceedings of the IEEE International Conference on Robotics and Automation*, Washington, DC, USA, 2002, pp. 3404-3410.

J. Intelligent Learning Systems & Applications, 2010, 2: 69-79
doi:10.4236/jilsa.2010.22010 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

Relational Reinforcement Learning with Continuous Actions by Combining Behavioural Cloning and Locally Weighted Regression

Julio H. Zaragoza, Eduardo F. Morales

National Institute of Astrophysics, Optics and Electronics, Computer Science Department, Tonantzintla, México.
Email: {jzaragoza, emorales}@inaoep.mx

Received October 30th, 2009; revised January 10th, 2010; accepted January 30th, 2010.

ABSTRACT

Reinforcement Learning is a commonly used technique for learning tasks in robotics, however, traditional algorithms are unable to handle large amounts of data coming from the robot's sensors, require long training times, and use discrete actions. This work introduces TS-RRLCA, a two stage method to tackle these problems. In the first stage, low-level data coming from the robot's sensors is transformed into a more natural, relational representation based on rooms, walls, corners, doors and obstacles, significantly reducing the state space. We use this representation along with Behavioural Cloning, i.e., traces provided by the user; to learn, in few iterations, a relational control policy with discrete actions which can be re-used in different environments. In the second stage, we use Locally Weighted Regression to transform the initial policy into a continuous actions policy. We tested our approach in simulation and with a real service robot on different environments for different navigation and following tasks. Results show how the policies can be used on different domains and perform smoother, faster and shorter paths than the original discrete actions policies.

Keywords: *Relational Reinforcement Learning, Behavioural Cloning, Continuous Actions, Robotics*

1. Introduction

Nowadays it is possible to find service robots for many different tasks like entertainment, assistance, maintenance, cleanse, transport, guidance, etc. Due to the wide range of services that they provide, the incorporation of service robots in places like houses and offices has increased in recent years. Their complete incorporation and acceptance, however, will depend on their capability to learn new tasks. Unfortunately, programming service robots for learning new tasks is a complex, specialized and time consuming process.

An alternative and more attractive approach is to show the robot how to perform a task, rather than trying to program it, and let the robot to learn the fine details of how to perform the task. This is the approach that we follow on this paper.

Reinforcement Learning (*RL*) [1] has been widely used and suggested as a good candidate for learning tasks in robotics, e.g., [2-9]. This is mainly because it allows an agent, *i.e.*, the robot, to "autonomously" develop a control policy for performing a new task while interacting with its environment. The robot only needs to know the goal of the task, *i.e.*, the final state, and a set of possible actions associated with each state.

The use and application of traditional *RL* techniques however, has been hampered by four main aspects: 1) vast amount of data produced by the robot's sensors, 2) large search spaces, 3) the use of discrete actions, and 4) the inability to re-use previously learned policies in new, although related, tasks.

Robots are normally equipped with laser range sensors, rings of sonars, cameras, etc., all of which produce a large number of readings at high sample rates creating problems to many machine learning algorithms.

Large search spaces, on the other hand, produce very long training times which is a problem for service robots where the state space is continuous and a description of a state may involve several variables. Researchers have proposed different strategies to deal with continuous state and action spaces, normally based on a discretization of the state space with discrete actions or with function approximation techniques. However, discrete actions produce unnatural movements and slow paths for a robot and function approximation techniques tend to be computationally expensive. Also, in many approaches, once a policy has been learned to solve a particular task, it cannot be re-used on similar tasks.

In this paper, *TS-RRLCA* (*Two-Stage Relational Reinforcement Learning with Continuous Actions*), a two-stage method that tackles these problems, is presented. In the first stage, low-level information from the robot's sensors is transformed into a relational representation to characterize a set of states describing the robot's environment. With these relational states we applied a variant of the *Q-learning* algorithm to develop a relational policy with discrete actions. It is shown how the policies learned with this representation framework are transferable to other similar domains without further learning. We also use Behavioural Cloning [10], *i.e.*, human traces of the task, to consider only a subset of the possible actions per state, accelerating the policy learning process and obtaining a relational control policy with discrete actions in a few iterations. In the second stage, the learned policy is transformed into a relational policy with continuous actions through a fast Locally Weighted Regression (*LWR*) process.

The learned policies were successfully applied to a simulated and a real service robot for navigation and following tasks with different scenarios and goals. Results show that the continuous actions policies are able to produce smoother, shorter, faster and more similar paths to those produced by humans than the original relational discrete actions policies.

This paper is organized as follows. Section 2 describes related work. Section 3 introduces a process to reduce the data coming from the robot's sensors. Section 4 describes our relational representation to characterize states and actions. Sections 5 and 6 describe, respectively, the first and second stages of the proposed method. Section 7 shows experiments and results, Section 8 presents some discussion about our method and the experimental results, and Section 9 concludes and suggests future research directions.

2. Related Work

There is a vast amount of literature describing RL techniques in robotics. In this section we only review the most closely related work to our proposal.

In [8] a method to build relational macros for transfer learning in robot's navigation tasks is introduced. A macro consists of a finite state machine, *i.e.*, a set of nodes along with rulesets for transitions and action choices. In [11], a proposal to learn relational decision trees as abstract navigation strategies from example paths in presented. These two approaches use relational representations to transfer learned knowledge and use training examples to speed up learning, however, they only consider discrete actions.

In [9], the authors introduced a method that temporarily drives a robot which follows certain initial policy while some user commands play the role of training input to the learning component, which optimizes the autonomous control policy for the current task. In [2], a robot is tele-operated to learn sequences of state-action pairs that show how to perform a task. These methods reduce the computational costs and times for developing its control scheme, but they use discrete actions and are unable to transfer learned knowledge.

An alternative to represent continuous actions is to approximate a continuous function over the state space. The work developed in [12] is a Neural Network coupled with an interpolation technique that approximates *Q-values* to find a continuous function over all the search space. In [13], the authors use *Gaussian Processes* for learning a probabilistic distribution for a robot navigation problem. The main drawback of these methods is the computational costs and the long training times as they try to generate a continuous function over all of the search space.

Our method learns, through a relational representation, relational discrete actions policies able to transfer knowledge between similar domains. We also speed up and simplify the learning process by using traces provided by the user. Finally we use a fast *LWR* to transform the original discrete actions policy into a continuous actions policy. In the following sections we describe in detail the proposed method.

3. Natural Landmarks Representation

A robot senses and returns large amounts of data readings coming from its sensors while performing a task. In order to produce a smaller set of meaningful information TS-RRLCA uses a process based on [14,15] In [14] the authors described a process able to identify three kinds of natural landmarks through laser sensor readings: 1) discontinuities, defined as an abrupt variation in the measured distance of two consecutive laser readings (**Figure 1(a)**), 2) walls, identified using the Hough transform (**Figure 1(c)**), and 3) corners, defined as the location where two walls intersect and form an angle (**Figure 1(d)**). We also add obstacles identified through sonars and defined as any detected object within certain range (**Figure 1(e)**).

A natural landmark is represented by a tuple of four attributes: (*DL*, *θL*, *A*, *T*). *DL* and *θL* are, respectively, the relative distance and orientation from the landmark to the robot. *T* is the type of landmark: *l* for left discontinuity, *r* for right discontinuity (see **Figure 1(b)**), *c* for corner, *w* for wall and *o* for obstacle. *A* is a distinctive attribute and its value depends on the type of landmark; for discontinuities *A* is depth (*dep*) and for walls *A* is its

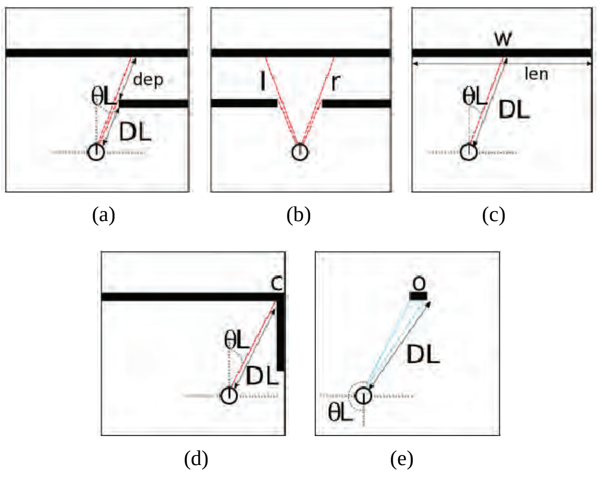


Figure 1. Natural landmarks types and associated attributes, (a) discontinuity detection; (b) discontinuity Types; (c) wall detection; (d) corner detection; (e) wall detection

length (*len*), for all of the other landmarks the *A* attribute is not used.

In [15] the data from laser readings is used to feed a clustering-based process which is able to identify the robot's actual location such as room, corridor and/or intersection (the location where rooms and corridors meet). **Figure 2** shows examples of the resulting location classification process.

Table 1 shows an example of the data after applying these processes to the laser and sonar readings from **Figure 3**. The robot's actual location in this case is *in-room*.

The natural landmarks along with the robot's actual location are used to characterize the relational states that describe the environment.

4. Relational Representations for States and Actions

A relational representation for states and actions has the advantage that it can produce relational policies that can be re-used in other, although similar, domains without any further learning. The idea it to represent states as sets of properties that can be used to characterize a particular situation which may be common to other states. For example, suppose the robot has some predicates that are able to recognize a room from its sensors' readings. If the robot has learned a policy to exit a room, then it can apply it to exit any recognizable room regardless of the current environment.

A relational state (r-state) is a conjunction of first order predicates. Our states are characterized by the following predicates which receive as parameters a set of values such as those shown in **Table 1**.

1) *place*: This predicate returns the robot's location, which can be in-room, in-door, in-corridor and in-intersection.

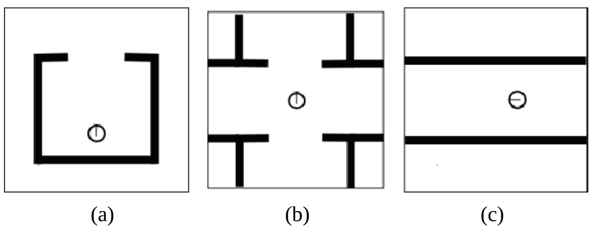


Figure 2. Locations detected through a clustering processes, (a) room; (b) intersection; (c) corridor

Table 1. Identified natural landmarks from the sensor's readings from Figure 3

N	*DL*	*θL*	*A*	*T*
1	0.92	-17.60	4.80	*r*
2	1.62	-7.54	3.00	*l*
3	1.78	17.60	2.39	*l*
4	0.87	-35.70	1.51	*w*
5	4.62	-8.55	1.06	*w*
6	2.91	-6.54	1.88	*w*
7	1.73	23.63	0.53	*w*
8	2.13	53.80	2.38	*w*
9	5.79	-14.58	0.00	c
10	2.30	31.68	0.00	c
11	1.68	22.33	0.00	c
12	1.87	-170.00	0.00	*o*
13	1.63	-150.00	0.00	*o*
14	1.22	170.00	0.00	*o*
15	1.43	150.00	0.00	*o*

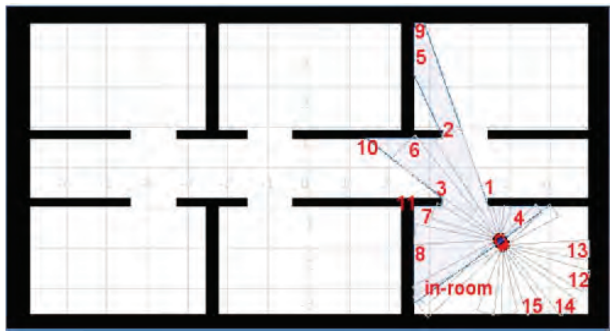


Figure 3. Robot sensing its environment through laser and sonar sensors and corresponding natural landmarks

2) *doors-detected*: This predicate returns the orientation and distance to doors. A door is characterized by identifying a right discontinuity (*r*) followed by a left discontinuity (*l*) from the natural landmarks. The door's orientation angle and distance values are calculated by averaging the values of the right and left discontinuities angles and distances. The discretized values used for door orientation are: *right* (door's angle between –67.5° and –112.5°), *left* (67.5° to 112.5°), *front* (22.5° to –22.5°), *back* (157.5° to –157.5°), *right-back* (–112.5° to –157.5°), *right-front* (–22.5° to –67.5°), *left-back* (112.5° to 157.5°) and *left-front* (22.5° to 67.5°). The discretized values used for distance are: *hit* (door's distance between 0 m and 0.3 m), *close* (0.3 m to 1.5 m), *near* (1.5 m to 4.0 m) and *far* (door's distance > 4.0 m).

For example, if the following discontinuities are obtained from the robot's sensors (shown in **Table 1**: [0.92, –17.60, 4.80, *r*], [1.62, –7.54, 3.00, *l*]), the following predicate is produced:

doors-detected ([*front*, *close*, –12.57, 1.27])

This predicate corresponds to the orientation and distance descriptions of a detected door (shown in **Figure 3**), and for every pair of right and left discontinuities a list with these orientation and distance descriptions is generated.

3) *walls-detected*: This predicate returns the length, orientation and distance to walls (type *w* landmarks). Possible values for wall's length are: *small* (length between 0.15 m and 1.5 m), *medium* (1.5 m to 4.0 m) and *large* (wall's size or length > 4.0 m). The discrete values used for orientation and distance are the same as with doors and the same goes for predicates *corners-detected* and *obstacles-detected* described below.

4) *corners-detected*: This predicate returns the orientation and distance to corners (type *c* landmarks).

5) *obstacles-detected*: This predicate returns the orientation and distance to obstacles (type *o* landmarks).

6) *goal-position*: This predicate returns the relative orientation and distance between the robot and the current goal. Receives as parameter the robot's current position and the goal's current position, though a trigonometry process, the orientation and distance values are calculated and then discretized as same as with doors.

7) *goal-reached*: This predicate indicates if the robot is in its goal position. Possible values are *true* or *false*.

The previous predicates tell the robot if it is in a room, a corridor or an intersection, detect walls, corners, doors, obstacles and corridors and give a rough estimate of the direction and distance to the goal. Analogous to r-states, r-actions are conjunctions of the following first order logic predicates that receive as parameters the odometer's speed and angle readings.

8) *go*: This predicate returns the robot's actual moving action. Its possible values are *front* (speed > 0.1 m/s), *nil* (–0.1 m/s < speed < 0.1 m/s) and *back* (speed < –0.1 m/s).

9) *turn*: This predicate returns the robot's actual turning angle. Its possible values are *slight-right* (–45° < angle < 0°), *right* (–135° < angle ≤ –45°), *far-right* (angle ≤ –135°), *slight-left* (45° > angle > 0°), *left* (135° > angle ≥ 45°), *far-left* (angle ≥ 135°) and *nil* (angle = 0°).

Table 2 shows an r-state-r-action pair generated with the previous predicates which corresponds to the values from **Table 1**. As can be seen, some of the r-state predicates (doors, walls, corners and obstacles detection) besides returning the nominal descriptions; they also return the numerical values of every detected element. The r-action predicates also return the odometer's speed and the robot's turning angle. These numerical values are used in the second stage of the method as described in Section 6. The discretized or nominal values, *i.e.*, the r-states and r-actions descriptions, are used to learn a relational policy through *rQ-learning* as described below.

5. TS-RRLCA First Stage

TS-RRLCA starts with a set of human traces of the task that we want the robot to learn. A trace $T_k = \{f_{k1}, f_{k2}, \ldots, f_{kn}\}$ is a log of all the odometer, laser and sonar sensor's readings of the robot while it is performing a particular task. A trace-log is divided in frames; every frame is a register with all the low-level values of the robot's sensors (f_{kj} = {laser1 = 2.25, laser2 = 2.27, laser3 = 2.29, …, sonar1 = 3.02, sonar2 = 3.12, sonar3 = 3.46, …, speed = 0.48, angle = 87.5}) at a particular time.

Once a set of traces ($T_1, T_2, \ldots, T_m$) has been given to TS-RRLCA, every frame in the traces, is transformed

Table 2. Resulting r-state-r-action pair from the values in Table 1

r-state	*r-action*
Place (*in-room*),	*go* (*nil*, 0.0),
doors-detected ([[*front, close*, –12.57, 1.27]]),	*turn* (*right*, 92).
walls-detected ([[*right-front, close, medium*, –35.7, 0.87], [*front, far, small*, –8.55, 4.62], [*front, near, medium*, –6.54, 2.91], [*left-front, near, small*, 23.63, 1.73], [*left-front, near, medium*, 53.80, 2.13]]), *corners-detected* ([[*front, far*, –14.58, 5.79], [*front, near*, 31.68, 2.30], [*left-front, near*, 22.33, 1.68]]), *obstacles-detected* ([[*back, near*, –170.00, 1.87], [*right-back, near*, –150.00, 1.63], [*back, close*, 170.00, 1.22], [*left-back, close*, 150.00, 1.43]]),	
goal-position ([*right-front, far*]),	
goal-reached (*false*).	

into natural landmarks along with the robot's location. This transformed frames are given to the first order predicates to evaluate the set of relations, *i.e.*, generate the corresponding r-state and r-action (as the one shown in **Table 2**). By doing this, every frame from the traces corresponds to an r-state-r-action pair and every one of these pairs is stored in a database (*DB*).

Algorithm 1 gives the pseudo-code for this Behavioural Cloning (*BC*) approach. At the end of this *BC* approach, the *DB* contains r-state-r-action pairs corresponding to all the frames in the set of traces.

As the traces correspond to different examples of the same task and as they might have been generated by different users, there can be several r-actions associated to the same r-state. *RL* is used to develop a control policy that selects the best r-action in each r-state.

5.1 Relational Reinforcement Learning

The *RL* algorithm selects the r-action that produces the greatest expected accumulated reward among the possible r-actions in each r-state. Since we only used information from traces only a subset of all the possible r-actions, for every r-state, are considered which significantly reduces the search space. In a classical reinforcement learning framework a set of actions (*A*) is predefined for all of the possible states (*S*). Every time the agent reaches a new state, it must select one action from all of the possible actions in *S* to reach a new state. In our *RL* approach when the robot reaches a new r-state, it chooses one action from a subset of r-actions performed in that r-state in the traces.

In order to execute actions, each time the robot reaches an r-state, it retrieves from the *DB* the associated r-actions. It chooses one according to its policy and the associated nominal value of the selected r-action is transformed into one of the following values:

1) For the predicate *go*, if the description of the r-action is *front* the corresponding value is 0.5 m/s, for *back* the corresponding value is –0.5 m/s, and for *nil* the value is 0.0 m/s.

2) For the predicate *turn* the values are: *slight-right* = –45°, *right* = –90°, *far-right* = –135°, *slight-left* = 45°, *left* = 90°, *far-left* = 135° and *nil* = 0°.

Once the r-action has been chosen and executed the robot gets into a new r-state and the previous process is repeated until reaching a final r-state.

Algorithm 2 gives the pseudo-code for this *rQ-learning* approach. This is very similar to the *Q-learning* algorithm, except that the states and actions are characterized by relations.

By using only the r-state-r-action pairs from the traces (stored in the *DB*) our policy generation process is very fast and thanks to our relational representation, policies can be transferred to different, although similar office or

Algorithm 1. Behavioural cloning algorithm

```
Require: T_1, T_2, …T_n: Set of n traces with examples of the task the robot has to learn.
Ensure: DB: r-state-r-action pairs database.
  for i = 1 to n do
      k ← number of frames in the trace i
      for j = 1 to k do
          Transform frame_ij (frame j from trace i) into their corresponding natural landmarks and into the corresponding robot's location.
          Use the natural landmarks and the robot's location to get the corresponding r-state (through the first order predicates).
          Use the robot's speed and angle to get the corresponding r-action.
          DB ← DB ∪ {r-state, r-action}. % Each register in DB contains an r-state with its corresponding r-action
      End for
End for
```

Algorithm 2. rQ-learning algorithm

```
Require: DB, r-state-r-action pairs database.
Ensure: function Q: discrete actions relational control policy.
  Initialize Q ($S_t$, $A_t$) arbitrarily
  Repeat
    $s_t$ ← robot's sensors readings values.
    Transform $s_t$ into its corresponding natural landmarks and into the corresponding robot's location.
    $S_t$ ← r-state ($s_t$)% Use those natural landmarks and the robot's location to get the corresponding r-state (through the first order predicates).
    for each step of the episode do
        Search the r-state ($S_t$) description in DB.
        for each register in DB which contains the r-state ($S_t$) description do
            Get its corresponding r-actions
        End for
        Select an r-action $A_t$ to be executed in $S_t$ trough an action selection policy (e.g., ε-greedy).
        Execute action $A_t$, observe $r_{t+1}$ and $s_{t+1}$
        Transform $s_{t+1}$ into its corresponding natural landmarks and into the corresponding robot's location.
        $S_{t+1}$ ← r-state ($s_{t+1}$)% Use those natural landmarks and the robot's location to get the corresponding r-state (through the first order predicates).
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(r_{t+1} + \gamma max A_{t+1} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
        $S_t \leftarrow S_{t+1}$
    End for
until $S_t$ is terminal
```

house-like environments. In the second stage, this discrete actions policy is transformed into a continuous actions policy.

6. TS-RRLCA Second Stage

This second stage refines the coarse actions from the previously generated discrete actions policy. This is achieved using Locally Weighted Regression (*LWR*).

The idea is to combine discrete actions' values given by the policy obtained in the first stage with the action's values previously observed in the traces. This way the robot follows the policy, developed in the first stage, but the actions are tuned through a *LWR* process. What we do is to detect the robot's actual r-state, then, for this r-state the previously generated discrete actions policy determines the action to be executed (**Figure 4(a)**). Before performing the action, the robot searches in the *DB* for all the registers that share this same r-state description (**Figure 4(b)**). Once found, the robot gets all of the numeric orientation and distance values from these registers. This orientation and distance values are used to perform a triangulation process. This process allows us to estimate the relative position of the robot from previous traces with respect to the robot's actual position. Once this position has been estimated, a weight is assigned to the previous traces action's values. This weight depends on the distance of the robot from the traces with respect to the actual robot's position (**Figure 4(c)**). These weights are used to perform the *LWR* that produces continuous r-actions (**Figure 4(d)**).

The triangulation process is performed as follows. The robot *R* in the actual r-state (**Figure 5(a)**), senses and detects elements *E* and *E'* (which can be a door, a corner, a wall, etc.). Each element has a relative distance (*a* and

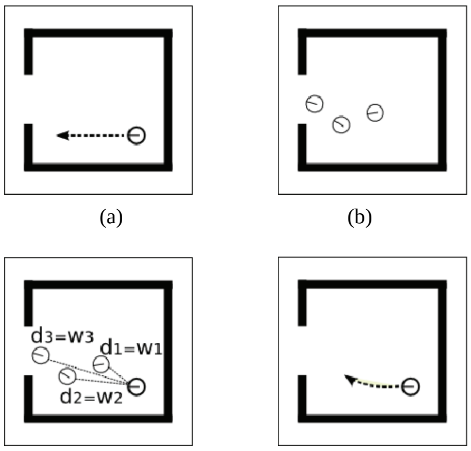
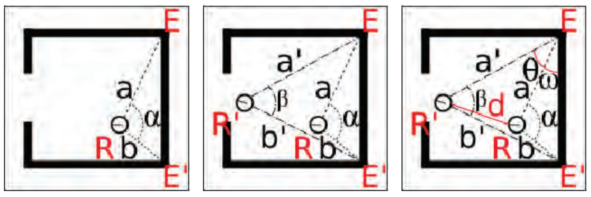


Figure 4. Continuous actions developing process, (a) r-state and corresponding r-action; (b) a trace segment; (c) distances and weights; (d) resulting continuous action



(a) (b) (c)

Figure 5. Triangulation process, (a) R robot's r-state and identified elements; (b) R'robot from traces; (c) elements to be calculated

b) and a relative angle with respect to *R*. The angles are not directly used in this triangulation process, what we use is the absolute difference between these angles (α). The robot reads from the *DB* all the registers that share the same r-state description, *i.e.*, that have the same r-state discretized values. The numerical angle and distance values associated with these *DB* registers correspond to the relative distances (*a*' and *b*') from the robot *R*' in a trace relative to the same elements *E* and *E'*, and the corresponding angle β (**Figure 5(b)**). In order to know the distance between *R* and *R*' (*d*) through this triangulation process, Equations (1)-(4) are applied.

$$\overline{EE'} = \sqrt{a^2 + b^2 - 2ab\cos(\alpha)} : \textit{Distance between E and E'}. \quad (1)$$

$$\theta + \omega = \arcsin\left(a' / \overline{EE'}\right) : \textit{Angle between a' and } \overline{EE'}. \quad (2)$$

$$\omega = \arcsin(a / \overline{EE'}) : \textit{Angle between a and } \overline{EE'}. \quad (3)$$

$$d = \sqrt{a^2 + a'^2 - 2aa'\cos(\theta)} : \textit{Distance between R and R'}. \quad (4)$$

These four equations give the relative distance (*d*) between *R* and *R'*. Once this value is calculated, a kernel is used to assign a weight (*w*). This weight is multiplied by the speed and angle values of the *R'* robot's r-action. The resulting weighted speed and angle values are then added to the *R* robot's speed and angle values. This process is applied to every register read from the *DB* whose r-state description is the same as *R* and is repeated every time the robot reaches a new r-state.

To summarize this process, each time the robot reaches an r-state and chooses an r-action according to the learned policy; it retrieves from the *DB* all the registers that share the same r-state. It uses the numerical values of the retrieved r-states to evaluate the relative distance of the position of the robot in a trace to the position of the robot in the actual r-state. Once all the distance values (*di*) are calculated we apply a Gaussian kernel (Equation (5)) to obtain a weight w_i. We tried different kernels, e.g., Tricubic kernel, and results were better with

Gaussian kernel but further tests are needed.

$$w_i(d_i) = \exp(-d_i^2): \textit{ Gaussian kernel.} \quad (5)$$

Then, every weight w_i is multiplied by the corresponding speed and angle values ($w_i \times speed_{DBi}$ and $w_i \times angle_{DBi}$) of the r-state-r-action pairs retrieved from the *DB*. The resulting values are added to the discrete r-action (rA_t = {*disc_speed*, *disc_angle*}) values of the policy obtained in the first stage in order to transform this discrete r-action into a continuous action (Equations (6) and (7)) that is finally executed by the robot. This process is performed in real-time every time the robot reaches a new r-state.

$$continuous\_speed = disc\_speed + \{w_1 \times speed_{DB1}\} + \{w_2 \times speed_{DB2}\} + \ldots + \{w_n \times speed_{DBn}\}: \textit{LWR for developing the continuous speed.} \quad (6)$$

$$continuous\_angle = disc\_angle + \{w_1 \times angle_{DB1}\} + \{w_2 \times angle_{DB2}\} + \ldots + \{w_n \times angle_{DBn}\}: \textit{LWR for developing the continuous angle.} \quad (7)$$

The weights are directly related to the distances between the robots in the actual r-state to the r-states to the robot in the human traces stored in the *DB*. The closer the human traces registers are to the robot's actual position, the higher the influence they have in transforming the discrete action into a continuous action.

The main advantage of our approach is the simple and fast strategy to produce continuous actions policies that, as will be seen in the following section, are able to produce smoother and shorter paths in different environments.

7. Experiments

For testing purposes, two types of experiments were performed:

1) Learning Curves: In these experiments we compared the number of iterations it takes our method *TS-RRLCA* to learn a policy against classical Reinforcement Learning (*RL*) and against the *rQ-learning* algorithm (shown in Algorithm 2) without using Behavioural Cloning approach, which we will refer to as Relational Reinforcement Learning (*RRL*).

2) Performance: In these experiments we compared the performance of the policies learned through *TS-RRLCA* with discrete actions against the policies learned through *TS-RRLCA* with continuous actions. Particularly we tested: How close the tasks are to the tasks performed by the user and how close the tasks are from obstacles in the environment.

3)Execution times.

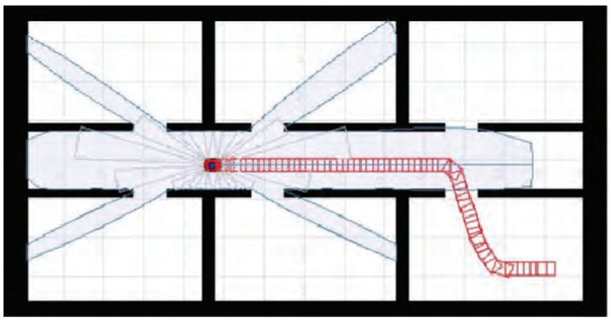
These experiments were carried out in simulation (*Player/Stage* [16]) and with a real robot which is an *ActivMedia GuiaBot* (www.activrobots.com).

Both robots (simulated and real) are equipped with a 180° front laser sensor and an array of four back sonars (located at –170°, –150°, 150° and 170°).

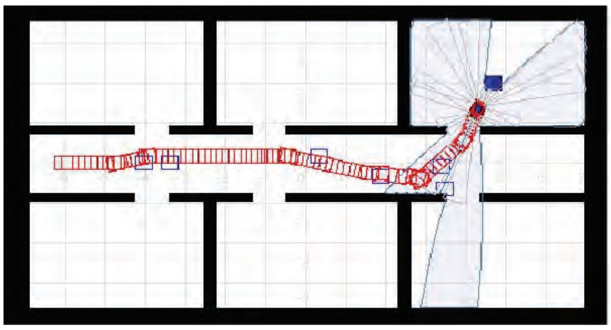
The laser range is 8.0 m and for the sonars is 6.0 m. The tasks in these experiments are "navigating through the environment" and "following an object".

The policy generation process was carried out in the map shown in **Figure 6** (Map 1 with size 15.0 m × 9.0 m). For each of the two tasks a set of 20 traces was generated by the user. For the navigation tasks, the robot and the goal's global position (for the *goal-position* predicate) were calculated using the work developed in [14]. For the following tasks we used a second robot which orientation and angle were calculated through the laser sensor. **Figure 6** shows an example of navigation and a following trace.

To every set of traces, we applied our approach to abstract the r-states and induce the subsets of relevant r-actions. Then, *rQ-learning* was applied to learn the policies. For generating the policies, *Q-values* were initialized to –1, $\varepsilon = 0.1$, $\gamma = 0.9$ and $\alpha = 0.1$. Positive reinforcement, *r* (+100) was given when reaching a goal (within 0.5 *m*), negative reinforcement (–20) was given when the robot hits an element and no reward value was given otherwise (0).



(a)



(b)

Figure 6. Traces examples, (a) navigation trace; (b) following trace

7.1 Learning Curves

Our method (*TS-RRLCA*) was compared in the number of iterations it takes to develop a control policy, against classical reinforcement learning (*RL*) and against the *rQ-learning* algorithm described in Algorithm 2, considering all the possible r-actions (the 8 r-actions, shown in Section 4) per r-state (*RRL*).

For developing the "navigating through the environment" policy with *RL* we discretized the state and action space as follows: the training Map 1, depicted in **Figure 6**, was divided in states of 25 cm^2. Since this map's size is 15 m × 9 m, the number of states is 2,160. In every state, one of the next 8 actions can be chosen to get into a new state which gives a total of 17,280 state-action pairs (This set of 8 actions correspond to the set of 8 r-actions we used in our *rQ-learning* algorithm).

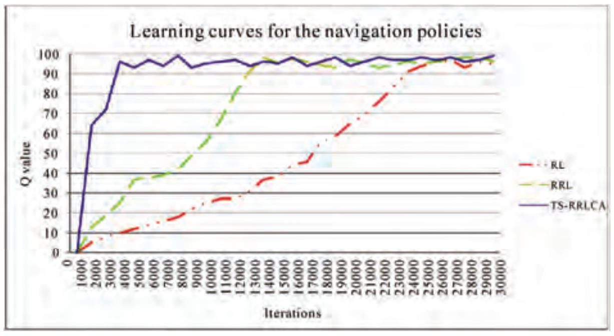
1) *front*: robot goes forward 25 cm.
2) *back*: robot goes back 25 cm.
3) *slight-right*: robot turns –45°.
4) *right*: robot turns –90°.
5) *far-right*: robot turns –135°.
6) *slight-left*: robot turns 45°.
7) *left*: robot turns 90°.
8) *far-left*: robot turns 135°.

For developing the navigation policy with *RRL* we have 655 r-states with 8 possible r-actions for each r-state, this gives a total of 5,240 possible r-state-r-action pairs. The number of r-states corresponds to the total number of r-states in which the training map can be divided.

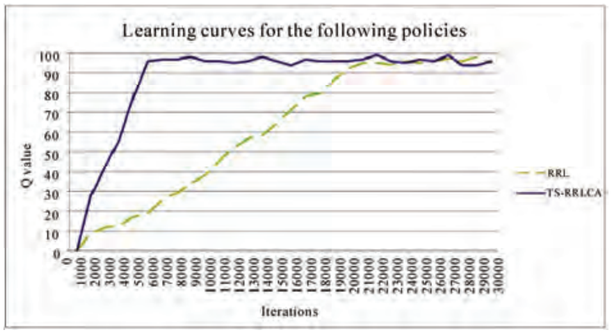
For developing the navigation policy with *TS-RRLCA* we used 20 navigation traces from which 934 r-state-r-action pairs were obtained. As can be seen, by using our Behavioural Cloning approach we significantly reduced the number of state-action pairs to consider in the learning process.

In each trace, every time our program performed a robot's sensors reading, which includes laser, sonars and odometer, we first transformed the laser and sonar readings into natural landmarks (as described in Section 3). These natural landmarks are sent to the predicates to generate the corresponding r-state, the corresponding r-action is generated by using the odometer's readings (as described in Section 4). This gives an r-state-r-action pair such as the one shown in **Table 2**.

Figure 7(a) shows the learning curves of *RL*, *RRL* and *TS-RRLCA* for a navigation policy. They show the accumulated Q-values every 1,000 iterations. As can be seen from this figure, the number of iterations for developing an acceptable navigation policy with *TS-RRLCA* is very low when compared to *RRL* and is significantly lower when compared to *RL*. It should be noted that the navigation policy learned with *RL* only works for going to a single destination state while the policies learned with our relational representation can be used to reach several destination places in different environments.



(a)



(b)

Figure 7. Learning curves comparison, (a) learning curves for the navigation policies; (b) learning curves for the following policies

For developing the "following an object" policy, the number of r-state-r-action pairs using our relational representation (*RRL*) is 3,149, while the number of r-state-r-action pairs using the same representation but with behavioural cloning (*TS-RRLCA*) is 1,406, obtained from 20 traces. For the following policy we only compared our approach against *RRL*.

Figure 7(b) shows the learning curves of these two methods. As can be seen the number of iterations that our method needs to generate an acceptable following policy is much lower than *RRL*.

To generate the continuous actions policies, *LWR* was applied using the Gaussian kernel for estimating weights. In the next section we compare the traces performed with the discrete actions policy with those using continuous actions.

7.2 Performance Tests

Once the policies were learned, experiments were executed in the training map with different goal positions and in two new and unknown environments for the robot (Map 2 shown in **Figure 8** with size 20.0 m × 15.0 m and Map 3, shown **Figure 9**, which corresponds to the real robot's environment whose size is 8.0 m × 8.0 m). A total of 120

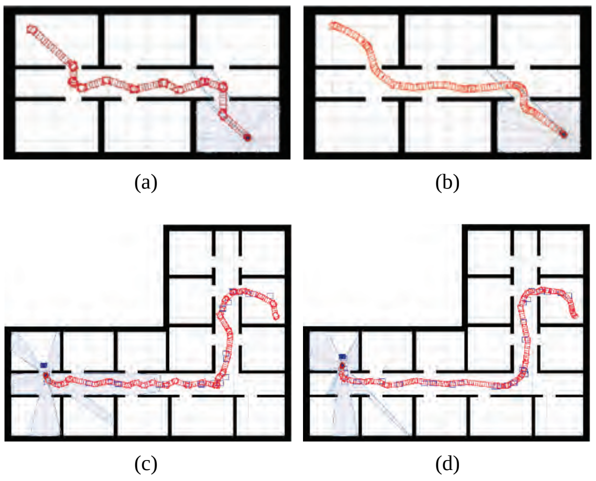


Figure 8. Navigation and following tasks performed with the policies learned with *TS-RRLCA*, (a) navigation task with discrete actions, Map 1; (b) navigation task with continuous actions, Map 1; (c) following task with discrete actions, Map 2; (d) following task with continuous actions, Map 2

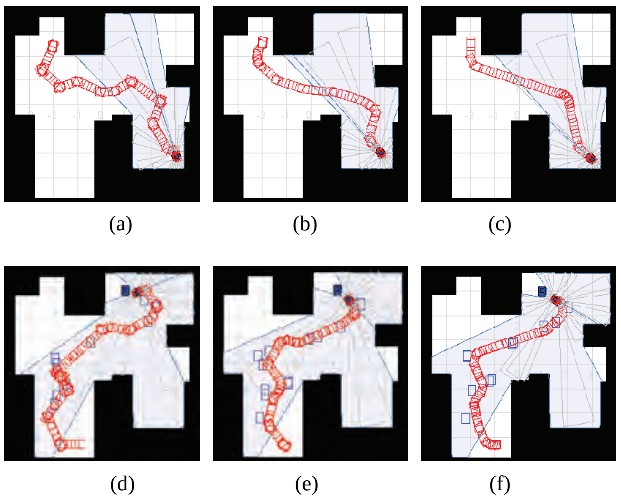


Figure 9. Navigation and following tasks examples from Map 3, (a) navigation task with discrete actions; (b) navigation task with continuous actions; (c) navigation task performed by user; (d) following task with discrete actions; (e) Following task with continuous actions; (f) following task performed by user

experiments were performed: 10 different navigation and 10 following tasks in each map, each of these tasks were executed first with the discrete actions policy from the first stage and then with the continuous actions policy from the second stage. Each task has a different distance to cover and required the robot to traverse through different places. The minimum distance was 2 m. (Manhattan distance), and it was gradually increased up to 18 m.

Figure 8 shows navigation (on the top) and a following task (on the bottom) performed with discrete and continuous actions policies respectively.

Figure 9 shows navigation and a following task performed with the real robot, with the discrete and with the continuous actions policy.

As we only use the r-state-r-action pairs from the traces developed by the user in Map 1 (as the ones shown in **Figure 6**), when moving the robot to the new environments (Map 2 and Map 3), sometimes, it was not able to match the new map's r-state with one of the previously visited states by the user in the traces examples. So when the robot reached an unseen r-state, it asked the user for guidance. Through a joystick, the user indicates the robot which r-action to execute in the unseen r-state and the robot saves this new r-state-r-action pair in the *DB*. Once the robot reaches a known r-state, it continues its task. As the number of experiments increased in these new maps, the number of unseen r-states was reduced. **Table 3** shows the number of times the robot asked for guidance in each map and with each policy.

Figure 10(a) shows results in terms of the quality of the performed tasks with the real robot. This comparison is made against tasks performed by humans (For **Figures 10(a)**, **10(b)** and **11**, the following acronyms are used, NPDA: Navigation Policy with Discrete Actions, NPCA: Navigation Policy with Continuous Actions, FPDA: Following Policy with Discrete Actions and FPCA: Following Policy with Continuous Actions).

All of the tasks performed in the experiments with the real robot, were also performed by a human using a joystick (**Figures 9(c)** and **9(f)**), and logs of the paths were saved. The graphic shows the normalized quadratic error between these logs and the trajectories followed by the robot with the learned policy.

Figure 10(b) shows results in terms of how closer the robot gets to obstacles. This comparison is made using the work developed in [17]. In that work, values were given to the robot accordingly to its proximity to objects or walls. The closer the robot is to an object or wall the higher cost it is given. Values were given as follows: if the robot is very close to an object (between 0 m and 0.3 m) a value of –100 is given, if the robot is close to an object (between 0.3 m and 1.0 m) a value of –3 is given, if the robot is near an object (between 1.0 m and 2.0 m) a value of –1 is given, otherwise a value of 0 is given. As can be seen in the figure, quadratic error and penalty values for continuous actions policies are lower than those with discrete actions.

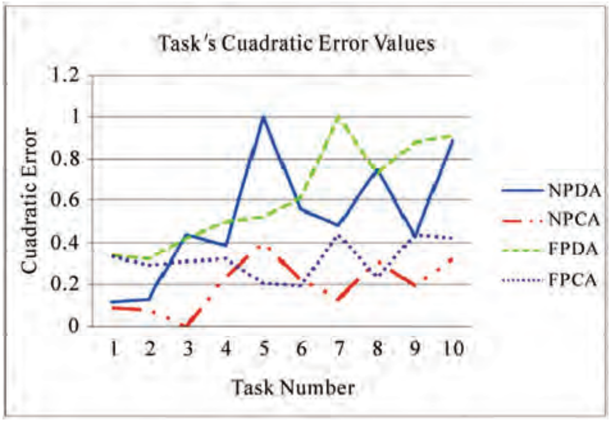
Policies developed with this method allow a close-to-human execution of the tasks and tend to use the available free space in the environment.

7.3 Execution Times

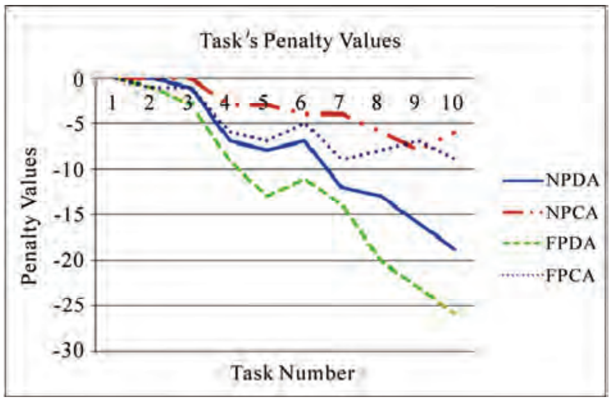
Execution times with the real robot were also registered. We compared the time that takes to the robot to perform a tasks with discrete actions against tasks performed with

Table 3. Number of times the robot asked for guidance in the experiments

Policy type	Map 1	Map 2	Map 3	Total
Navigation	2	6	14	22
Following	7	15	27	49



(a)



(b)

Figure 10. Navigation and following results of the tasks performed by the real robot, (a) quadratic error values; (b) penalty values

continuous actions. Every navigating or following experiment, that we carried out, was performed first with discrete actions and then with continuous actions.

As can be seen in **Figure 11**, continuous actions policies execute faster paths than the discrete actions policy despite our triangulation and *LWR* processes.

8. Discussion

In this work, we introduced a method for teaching a robot how to perform a new task from human examples. Experimentally we showed that tasks learned with this method and performed by the robot are very similar to

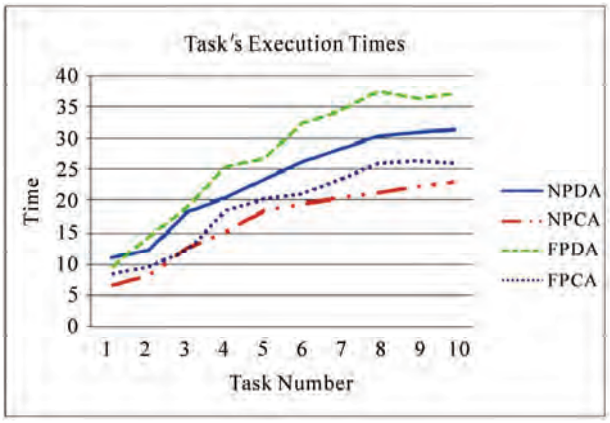


Figure 11. Execution times results

those tasks when performed by humans. Our two-stage method learns, in the first stage, a rough control policy which, in the second stage, is refined, by means of Locally Weighted Regression (*LWR*), to perform continuous actions. Given the nature of our method we can not guaranteed to generate optimal policies. There are two reasons why this can happen: 1) the actions performed by the user in the traces may not part of the optimal policy. In this case, the algorithm will follow the best policy given the known actions but will not be able to generate an optimal policy. 2) The *LWR* approach can take the robot to states that are not part of the optimal policy, even if they are smoother and closer to the user's paths. This has not represented a problem in the experiments that we performed.

With the Behavioural Cloning approach we observed around a 75% reduction in the state-action space. This reduction depends on the traces given by the user and on the training environment. In a hypothetical optimal case, where a user always performs the same action in the same state, the system only requires to store one action per state. This, however, is very unlikely to happen due to the continuous state and action space and the uncertainty in the outcomes of the actions perform with a robot.

9. Conclusions and Future Work

In this paper we described an approach that automatically transformed in real-time low-level sensor information into a relational representation. We used traces provided by a user to constraint the number of possible actions per state and use a reinforcement learning algorithm over this relational representation and restricted state-action space to learn in a few iterations a policy. Once a policy is learned we used *LWR* to produce a continuous actions policy in real time. It is shown that the learned policies with continuous actions are more similar to those performed by users (smoother), and are safer and faster than the policies obtained with discrete actions. Our relational policies are expressed in terms of more natural descriptions, such as rooms, corridors, doors, walls, etc., and can be re-used for different tasks and on different house or office-like environments. The policies were learned on a simulated environment and later tested on a different simulated environment and on an environment with a real robot with very promising results.

There are several future research directions that we are considering. In particular, we would like to include an exploration strategy to identify non-visited states to complete the traces provided by the user. We are also exploring the use of voice commands to indicate the robot which action to take when it reaches an unseen state.

10. Acknowledgements

We thank our anonymous referees for their thoughtful and constructive suggestions. The authors acknowledge to CONACyT the support provided through the grant for MSc. studies number 212418 and in part by CONACyT project 84162.

REFERENCES

[1] C. Watkins, “Learning from Delayed Rewards,” PhD Thesis, University of Cambridge, England, 1989.

[2] K. Conn and R. A. Peters, “Reinforcement Learning with a Supervisor for a Mobile Robot in a Real-World Environment,” *International Symposium on Computational Intelligence in Robotics and Automation*, Jacksonville, FI, USA, June 20-23, 2007, pp. 73-78.

[3] E. F. Morales and C. Sammut, “Learning to Fly by Combining Reinforcement Learning with Behavioural Cloning,” *Proceedings of the Twenty-First International Conference on Machine Learning*, Vol. 69, 2004, pp. 598-605.

[4] J. Peters, S. Vijayakumar and S. Schaal, “Reinforcement Learning for Humanoid Robotics,” *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots*, Karlsruhe, Germany, September 2003, pp. 29-30.

[5] W. D. Smart, “Making Reinforcement Learning Work on Real Robots,” Department of Computer Science at Brown University Providence, Rhode Island, USA, 2002.

[6] W. D. Smart and L. P. Kaelbling, “Effective Reinforcement Learning for Mobile Robots,” *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, Washington, DC, USA, 2002, pp. 3404-3410.

[7] R. S. Sutton and A. G. Barto, “Reinforcement Learning: An introduction,” MIT Press, Cambridge, MA, 1998.

[8] L. Torrey, J. Shavlik, T. Walker and R. Maclin, “Relational Macros for Transfer in Reinforcement Learning,” *Lecture Notes in Computer Science*, Vol. 4894, 2008, pp. 254-268.

[9] Y. Wang, M. Huber, V. N. Papudesi and D. J. Cook, “User-guided Reinforcement Learning of Robot Assistive Tasks for an Intelligent Environment,” *Proceedings of the* 2003 *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Las Vegas, NV, 2003, pp. 27-31.

[10] I. Bratko, T. Urbancic and C. Sammut, “Behavioural Cloning of Control Skill,” In: R. S. Michalski, I. Bratko and M. Kubat, Ed., *Machine Learning and Data Mining*, John Wiley & Sons Ltd., Chichester, 1998, pp. 335-351.

[11] A. Cocora, K. Kersting, C. Plagemann, W. Burgard and L. De Raedt, “Learning Relational Navigation Policies,” *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Beijing, China, October 9-15, 2006, pp. 2792-2797.

[12] C. Gaskett, D. Wettergreen and A. Zelinsky, “Q-learning in Continuous State and Action Spaces,” *In Australian Joint Conference on Artificial Intelligence*, Australia, 1999, pp. 417-428.

[13] F. Aznar, F. A. Pujol, M. Pujol and R. Rizo, “Using Gaussian Processes in Bayesian Robot Programming,” *Lecture notes in Computer Science*, Vol. 5518, 2009, pp. 547-553.

[14] S. F. Hernández and E. F. Morales, “Global Localization of Mobile Robots for Indoor Environments Using Natural Landmarks,” *IEEE Conference on Robotics, Automation and Mechatronics*, Bangkok, September 2006, pp. 29-30.

[15] J. Herrera-Vega, “Mobile Robot Localization in Topological Maps Using Visual Information,” Masther’s thesis (to be publised), 2010.

[16] R. T. Vaughan, B. P. Gerkey and A. Howard, “On Device Abstractions for Portable, Reusable Robot Code,” *Proceedings of the* 2003 *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3, 2003, pp. 2421-2427.

[17] L. Romero, E. F. Morales and L. E. Sucar, “An Exploration and Navigation Approach for Indoor Mobile Robots Considering Sensor’s Perceptual Limitations,” *Proceedings of the IEEE International Conference on Robotics and Automation*, Seoul, Korea, May 21-26, 2001, pp. 3092-3097.

J. Intelligent Learning Systems & Applications, **2010, 2: 80-85**
doi:10.4236/jilsa.2010.22011 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

An Experience Based Learning Controller

Debadutt Goswami, Ping Jiang*

Department of Computing, University of Bradford, Bradford, UK.
Email: p.jiang@bradford.ac.uk

Received November 10th, 2009; revised April 10th, 2010; accepted April 20th, 2010.

ABSTRACT

The autonomous mobile robots must be flexible to learn the new complex control behaviours in order to adapt effectively to a dynamic and varying environment. The proposed approach of this paper is to create a controller that learns the complex behaviours incorporating the learning from demonstration to reduce the search space and to improve the demonstrated task geometry by trial and corrections. The task faced by the robot has uncertainty that must be learned. Simulation results indicate that after the handful of trials, robot has learned the right policies and avoided the obstacles and reached the goal.

Keywords: *Mobile Robots, Learning from Demonstration, Neural Network Control, Reinforcement Learning*

1. Introduction

Most of the autonomous mobile robots or the human controlled mobile robots working in the industrial, domestic and entertainment environment lack the flexibility to learn and perform the new complex tasks. In this paper we consider the problem of motion control of an autonomous mobile robot using the control field generated by a neural controller. The proposed system enables the autonomous mobile robot to learn, modify and adapt its skills efficiently in order to react adequately in complex and unstructured environment. To incorporate the precision and flexibility in robots, we incorporated three very common natural approaches that, human uses to learn and execute any task, 1) Learning from demonstration 2) Generalization of the learned task 3) Reinforcement of the task by trial and error [1,2]. The designing of a motion controller depends on the kinematics constraint of a mobile robot and the complexity and structure of a task, and is very difficult to design whereas the proposed controller can easily and quickly learn the complex controls with some simple demonstrations and few trials. The major advantage of our technique is that any mobile robot can be taught to move in an unknown environment using the generated control field as the general control law. The control law is independent of both physical and virtual sensors. The use of domain knowledge has a great significance in reducing the learning time and the use of trial and error learning method improves the learning continuously [3]. The learning from demonstration has reduced the robot programming and made it very simple to use [4,5]. The reinforcement learning has already been used in many navigational and reactive problems [6,7,8]. The ability of reinforcement learning to interact with the environment and the domain knowledge from demonstrations has made it easier to tackle uncertainty. Due to the simplicity to operate and flexibility to learn uncertainty, the proposed controller can be operated by the non-professionals who are not skilled enough to control and program the sophisticated mobile robots in the complex industrial environment [3,9].

The paper is presented in following manner. Section II presents the previous works related to the field. Section III presents the controller architecture and the proposed algorithm. Section IV presents the simulation results. Section V presents conclusions and discusses the work further.

2. Related Works

We have incorporated two different research areas: Learning from demonstration and Reinforcement Learning, which bears some relation to a number of previous works. The importance of human robot interaction and the need of modern artificial intelligence are described in [10] that focus on creating new possibilities for the flexible and interacting robot from the engineering point of view and from the humanistic point of view. The paper incorporates these ideas and focuses on flexible learning controller of a robot which can learn and adapt the changes in the environment very easily. The proposed methodology differs from [3,11,12] and [5] in two fundamental aspects. Firstly our approach learns from demonstration, which acts as initial policy to perform the desired task and builds a model of partially observable environment. Secondly, the methodology is offline and online computations are done on the basis of generated control field and the previous experience. The proposed controller initializes the weights of the controller from the demonstrated points of the task whereas in [11] the learner is initially empty. The proposed controller utilizes *Q*-values to avoid uncertainty where as [3] and [11] used fuzzy behaviour as a reflex to act into a new situation. The aim is to generate an abstract description of the task reflecting user's intention and modelling the problem solution offline. In this context we also discuss the issues of evaluation, tuning, suitable generalization and execution of the elementary skills [9,13-15]. Instead of using radial basis as a function approximation [3,12], we used inverse distance interpolation. It works like local approximators. We also used learning through feedback, which improves the learning and generalization continuously.

3. Controller Architecture

The proposed learning controller can be represented as recurrent neural network architecture. The controller has basically 3 layers, input layer, hidden layer and output layer and a reinforcement controller which is connected to the output layer and the hidden layer. $S \in R^N$ and $A \in R^N$ are the sensor input and output. The weight is updated when the sensor and the controller try to learn an unknown model from sensors to actuators, *i.e.*

$$A = f(S) \tag{1}$$

Input Layer—Input layer consists of several neural input nodes that represent some state in the state space *i.e.* *S*. The sensor input is propagated through the hidden layer from the input layer to the output layer. Each neural node in the input layer is connected to all the neural nodes in the hidden layer.

Hidden Layer—Hidden layer is composed of several computational neurons. The output of the hidden neurons is the distance between the input layer neuron and the weight similar to a prototype fuzzy law. The weight W_i in the hidden layer neuron represents the centre *i.e.* the demonstrated data points. Each neural node of the input layer is connected to all the neural nodes of the hidden layer. The output of the hidden layer codes the distance of the input neurons from the hidden layer neurons.

Output layer—The output layer neuron computes the inverse weighted sum of the output v_i of the hidden layer neurons *i.e.*

$$A = \sum_{i=1}^{N} W_{oi} \times v_i \tag{2}$$

The output layer neuron uses inverse distance interpolation to update the weights W_{oi} of the output layer.

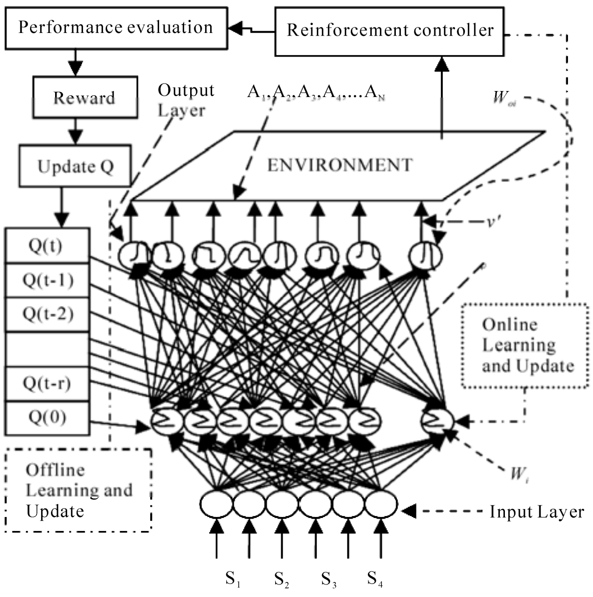


Figure 1. Neural network control architecture

The advantage of using this technique is that, the estimated data point has the influence of all the neighbouring data points depending on the distance from each prototype rule. This is the simplest generalized technique and mathematically expressed as

$$v' = \left(\sum_{i=1}^{N(v')} \frac{1}{d^p} \times v_i \right) \Bigg/ \left(\sum_{i=1}^{N(v')} \frac{1}{d^p} \right) \tag{3}$$

where, v' is the estimated control action in term of a sensor state S'; v_i is a prototyped action of the hidden layer neuron at state S; d^p is the distance between *S* and *S'*; *p* is the power; *N*(*v'*) is the number of data points in the effective window of *v'*. Generally the distance can be calculated by simple Euclidean distance formula. The above generalization technique only acts as a function approximation that reproduces the generalized control field based on the distance from the existing prototype rule. But to tackle uncertainty and to produce an improved control field, the robot needs to incorporate a weight that represents the significance of the previous learning history and incorporates continuous learning experiences. The idea is to improve the learning by trial and error. So *Q*-learning [16] is incorporated, which produces *Q*-matrix based on the scalar reward *R*. The *Q*-value determines the new policy to achieve the goal. Therefore in the neural network architecture, the output layer is connected to the reinforcement controller which in turn is connected to the hidden layer. Depending on the achievement of the desired goal, the reinforcement reward *R* is given to the robot by the reinforcement controller. The reward *R* updates the *Q*-matrix and the new *Q*-value is used to generate a new control field. The reinforcement controller follows the online exploration and offline learning policy [17]. So the *Q*-value is updated in following two cases:

Case 1: Online exploration—During online exploration the robot uses *Q*-matrix to avoid uncertainty like unknown obstacles. Therefore, whenever the robot finds uncertainty in its next course of action, the control switches from the generalized control field to *Q*-matrix and selects an action with other maximum *Q*-value from its effective window that corresponds to a policy with next higher probability to reach the goal and avoids the uncertainty. This earns an online reward *R* and updates the *Q*-value online in that time step. The change in the action in that time step is used as a prototype rule *i.e.* center in the hidden layer.

Case 2: Offline learning policy—In the end of the trial during offline learning policy, the reinforcement reward *R* is given offline to the robot. The reward *R* updates the corresponding *Q*-values of the trajectory based on the final payoff. After updating *Q*-value, the change is applied to the neurons in the hidden layer of the network architecture. So another factor that influences the output of the hidden layer is the *Q*-value.

The *Q*-value is then used as a measure to calculate the relevance of the data points in terms of usefulness of that particular task geometry in achieving the goal. The change in the *Q*-matrix influences all the neighbouring data points in the control field. This is called the generalization. After that the control field is transferred back to the robot. In every trial *Q*-matrix is updated and reflects the continuous learning. The steps are repeated until we get the desired control field. This technique of learning is known as online exploration and offline policy learning. The reward scheme for online exploration and offline policy learning is as follows:

$$R = \begin{cases} +1 \text{ If the goal is achieved or obstacle is aovided} \\ 0 \text{ If no obstacle or no goal} \\ -1 \text{ If failed to achieve goal or unable to aovid obstacle} \end{cases} \tag{4}$$

The reward function can represented as:

$$R = R_{uncertainty} + R_{goal} \tag{5}$$

In online exploration R_{goal} is set to zero and $R_{uncertainty}$ is awarded online to avoid obstacles on its path whereas in offline learning policy R_{goal} is awarded offline and $R_{uncertainty}$ is set to zero.

The *Q* update rule can be described in two steps.

Step 1*:* Give Reward *R* using (4) and (5) and update the weight Q_w using the weight update Equation.

$$\begin{aligned} &Q_w(s,a) \\ &= Q_w(s,a) + \alpha \times \left[R(s,a) + \lambda \times Q_w(s^{'},a^{'}) - Q_w(s,a) \right] \end{aligned} \tag{6}$$

Here $Q_w(s,a)$ is the weight matrix to be updated; (*s*, *a*) is the current state and action; $R(s,a)$ is the reward matrix; $Q_w(s^{'},a^{'})$ is the future state with the highest Q_w value; α is the learning rate; λ is the discount factor in the range [0, 1]. The weight update Equation (6) is inherited from the traditional *Q*-learning algorithm [16].

Step 2*:* Estimate the data points v' based on the Q_w calculated in step 1. The new estimate updates the previously generated control field and generates a new generalized control field. To implement our idea we modify (3) by introducing Q_w.

$$v' = \left(\sum_{i=1}^{N(v')} \frac{Q_w}{d^p} \times v_i \right) \bigg/ \left(\sum_{i=1}^{N(v')} \frac{Q_w}{d^p} \right) \tag{7}$$

Here $\frac{Q_w}{d^p}$ is the weight. A small Q_w value represents the small contribution of the point in accomplishing the goal where as a large Q_w value signifies large contribution of the point in accomplishing the goal. Q_w is inversely related to the distance, which signifies that the Q_w will have higher value when the estimated data point is closer to the good trajectories. The trajectories with lower Q_w value have less contribution in the overall generalization.

In the proposed context behaviour is improved based on scalar rewards from a critic. It does not require a person to program the desired actions in different situations. However several difficulties stand in the way. Firstly a robot using the basic reinforcement-learning framework might require an extremely long time to converge to the right track and to acquire the right action. In our framework learning from demonstration is used that reduces the search space to achieve an adequate control and it does not need any programming of the desired behaviour. The demonstration provides the domain knowledge, that gives hint to perform the desired task and reduces knowledge base significantly. Through trial and error the knowledge base can be improved further [18,19]. Secondly reinforcement learning is basically applied on the task where the state of the environment is completely observable, but in the real world tasks most of the knowledge is incomplete and inaccurate. The online exploration and offline learning policy explores the unknown environment with the help of generalized control field and tackles the uncertainty by selecting the appropriate control action which signifies maximum experience *i.e.* maximum *Q*-value in the effective window in achieving the desired goal. The trajectory generated by it changes the control field and tackles the uncertainty in the environment.

Ideally for a real robot any learning algorithm should be feasible and efficient to perform the complex computations, keeping in mind the robots internal configuration under consideration. A robot has limited memory and limited computational ability. Almost all the cases of practical interest consist of far more state observation pairs than could possibly entered into the memory. The robot is even typically unable to perform enough computation per time step to fully use it. Especially in the real world experiences, which consist of large number of action and observation pairs and at each step the robot needs to memorize and learn the action-observation pairs. Thus a good approach is to compute and store a limited number of data online which are relevant and learning the policy offline. In the proposed algorithm the old data can be used and new experiences can be collected, which are somewhat characteristic for the task. The best thing is the experience replay and the assessment of the performance of the previous experiences. Another important step is the generalization of the learned policy. The policy learned in this way keeps the learning continuous. This kind of learning is generally called as learning from experience. For a robot controller, it is also important to reduce the real time computation. The robot takes action on the generated control field only. Hence learning is made continuous and the computational overhead is also reduced. On the other hand the exploration is done online. The robot decides between exploration and exploitation based on the number of times an action has been chosen. The robot generally chooses action with the highest state action value with high probability, which is known as exploitation. If the robot oscillates in a particular region then exploitation is stopped and control switches to exploration to avoid oscillation and finds a new path. During exploration the selection of next state is made random. This approach has a major advantage of not being influenced by the limited knowledge only *i.e.* to avoid local optima. During exploration the robot can take the control actions which were never taken before and hence exploring more of the unobserved environment. The balance between exploration and exploitation is necessary, because exploitation is helping the robot to avoid uncertainty and to reach the goal where as exploration is avoiding all the oscillation and unnecessary iterations. So this methodology has a very positive approach to learn any new skill quickly. The robot can be restrained easily to adapt to environment changes and trained and learned improving the performance all the time.

The proposed algorithm can be summarized as below:

1) Build the control field by demonstrating the task, *i.e.* initializing the weights in the hidden layer with the demonstrated data points and imitating the relevant knowledge required to achieve the task.

2) Based on the demonstrated control field, using inverse distance weight interpolation, generalize the field using (7).

3) Transfer the generalized control field to the robot and do some trials to achieve the goal. Do some exploration also. If all the trials are successful in achieving the goal then the demonstrated control field is perfect and does not need any changes. If in trials, at some point the robot hits or senses any obstacle then give reward $R_{uncertainty}$ to the point using (4) and (5) and move towards the point that has next highest *Q*-value, without any obstacle. The next higher *Q*-value represents the next most experienced and favorable point heading towards the goal. Then update the *Q*-values using (6). This will generate a new path to reach the goal. After the goal has been reached, generalize the control field offline using (7). The control field will have influence from the points updated. This will reproduce an updated control field.

4) After reaching the goal, assign final reward R_{goal} to the robot using (4) and (5). This reward represents the successful completion of the desired task avoiding all the obstacles and is made offline. Update all the *Q*-values of the updated control trajectory generated in step 3 using (6).

5) Generalize the control field with the new updates using (7). The updated field represents the new control field. Repeat from Step 3 to 5 until the robot learns the desired task.

4. Simulation Results

In simulation we build a discrete workspace with a dimension of 20 × 20. The workspace contains a goal and obstacles, represented by the ‘G’ and the rectangles. Robot has a sensor of 3 × 3 neighbourhoods which means it can sense obstacles around its 8 neighbouring cells. The arrows represent the direction of each point in the field. The simulation is done based on the proposed algorithm in section 3. The robot was initially demonstrated to reach the goal. Based on the demonstration initial control field was built (**Figure 2**). The simulation considered two different tasks with different uncertain complexities with same initial control field.

Task 1: The robot had to avoid an obstacle and reach the goal.

Task 2: The robot had to avoid obstacles and pass through the door and reach the goal.

In both the cases robot completed the tasks in few trials avoiding all the uncertainties. The final control fields for task 1 and task 2 are shown in **Figures 3** and **4**.

The learning history of the controller for both the cases is described in **Figures 5-7**. The broken line curve and the solid line curve represents first and second task.

In **Figure 6** it can be observed that number of hidden neurons increases with the number of trials. Increase in hidden layer neurons indicates the increase in number

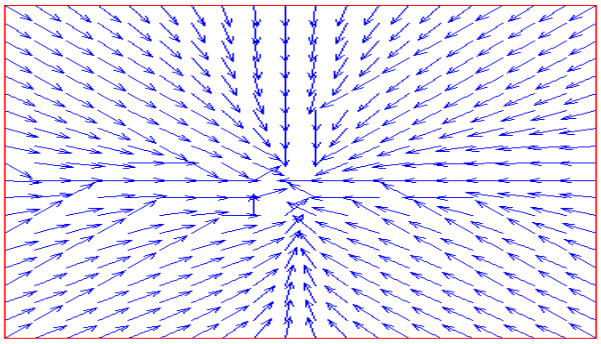


Figure 2. Demonstrated control field

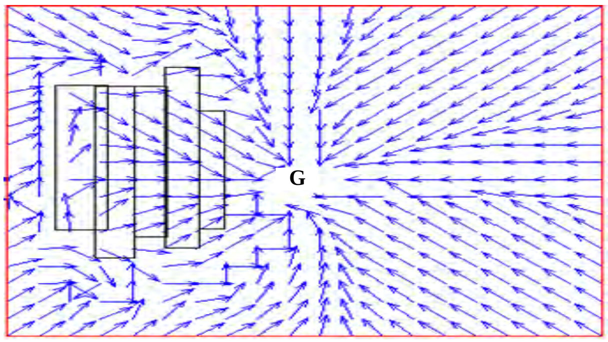


Figure 3. Final control field of task 1

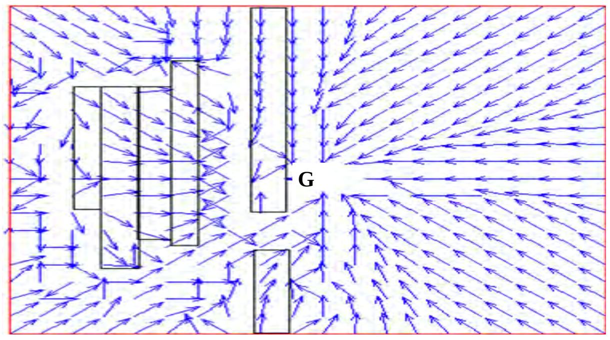


Figure 4. Final control field of task 2

centers or prototype rules in the network. It clearly indicates the learning in each trial. **Figure 7** displays the plotting of the total number reinforcements used in each trial. The total reinforcement can be defined as the sum of immediate reinforcement the learner receives till the robot reaches the goal and the offline reinforcement the learner receives after reaching the goal. It is clear in **Figure 7** that after trial 3 the controller did not use any reinforcements to complete the tasks. The controller needed only three trials 1-3 to learn a new control law to avoid uncertainty in both tasks. To see the reliability of the control field we chose different starting points. The robot was successful in completing the tasks from all the points.

Another observation is that the number of reinforcement received (**Figure 7**) and the number of steps taken

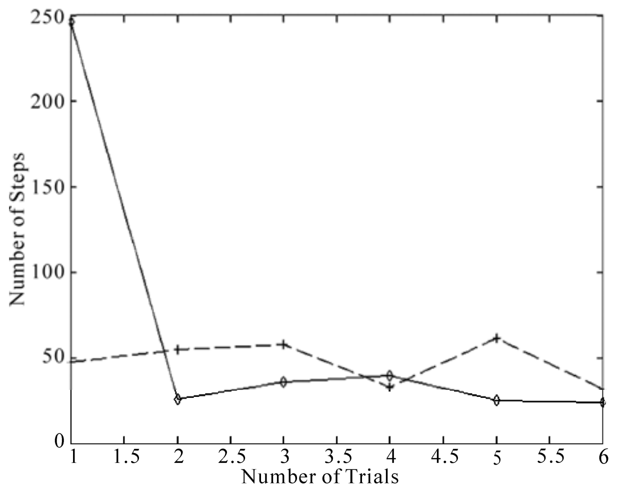


Figure 5. Number of steps taken to reach goal in each trial

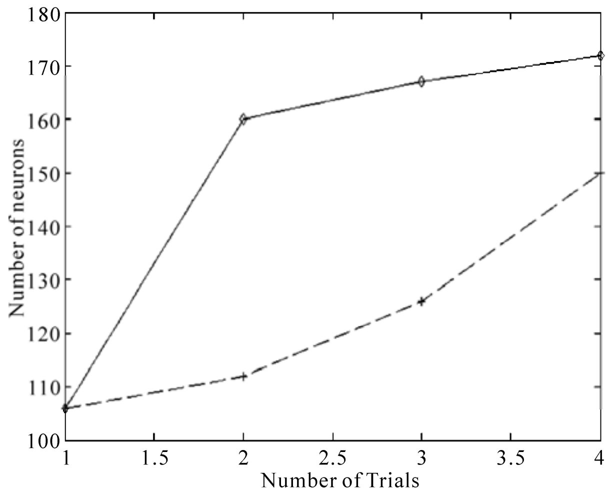


Figure 6. Number of hidden layer neurons used in each trial

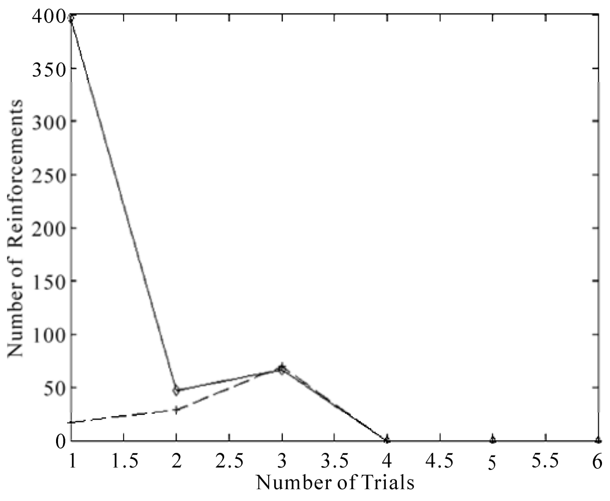


Figure 7. Number of reinforcements given in each trial

(**Figure 5**) in trial 1 in task 2 (solid line curve) is very high. It indicates that, in task 2 the maximum exploration was done in trial 1 where as in task 1 exploration was increasing with trials. Therefore with increase in exploration reinforcement is also increasing. Furthermore, it was during first three trials the robot attained reinforcements (**Figure 7**) and a sharp growth in network (**Figure 6**).

5. Conclusions

This paper presented a learning strategy to generate a control field for a mobile robot in an unknown and uncertain environment, which integrates learning, generalization, exploration and offline computation into a unified architecture. After the learning, a robot can approach the goal by following the control field. The important lessons learned from the implementation included 1) imitation of very accurate and exact action sequence is not necessary [15]; 2) a prior knowledge is required to plan a model of the task to support rapid learning; 3) generalization is improved by improving the learning policies; 4) simple method like inverse distance is adequate to generalize the task; 5) offline learning is an important method for real time applications to avoid the large online computations; 6) online exploration is required to explore other possibilities to perform a task and to improve the quality of learning; 7) balance between exploration and exploitation improves the learning policies, which reduces the learning time significantly; 8) the robot can learn from few demonstrations but it effects the learning speed. The major disadvantage of this method is the use of position information which is not always accurate in real robots due to inaccurate sensor information due to rotational or translational error caused by the slippage between robot's wheel and the floor.

REFERENCES

[1] M. N. Nicolescus and M. J. Mataric, "Natural Methods for Robot Task Learning: Instructive Demonstrations, Generalization and Practice," *In Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, July 14-18, 2003, pp. 241-248.

[2] M. N. Nicolescus and M. J. Mataric, "Learning and Interacting in Human—Robot Domains," *IEEE Transactions on systems, Man and cybernetics—Part A*: *Systems and Humans*, Vol. 31, No. 5, 2001, pp. 419-430.

[3] G. Hailu,"Symbolic Structures in Numeric Reinforcement for Learning Optimum Robot Trajectory," *Robotics and Automation Systems*, Vol. 37, No. 1, 2001, pp. 53-68.

[4] D. C. Bentivegna, C. G. Atkeson and G. Cheng, "Learning Tasks from Observation and Practice," *Robotics and Automation Systems*, Vol. 47, No. 2-3, 2004, pp. 163-169.

[5] G. Hailu and G. Sommer, "Learning by Biasing," *IEEE International Conference on Robotics and Automation*, Leuvem, Belgium, 1998, pp. 16-21.

[6] J. R. Millan, "Reinforcement Learning of Goal Directed Obstacle Avoiding Reaction Strategies in an Autonomous Mobile Robot," *Robotics and Autonomous Systems*, Vol. 15, No. 4, 1995, pp. 275-299.

[7] A. Johannet and I. Sarda, "Goal-Directed Behaviours by Reinforcement Learning," *Neurocomputing*, Vol. 28, No. 1-3, 1990, pp. 107-125.

[8] P. M. Bartier and C. P. Keller, "Multivariate Interpolation to Incorporate Thematic Surface Data Using Inverse Distance Weighting (IDW)," *Computers & Geosciences*, Vol. 22, No. 7, 1996, pp. 795-799.

[9] H. Friedrich, M. Kaiser and R. Dillmann, "What Can Robots Learn from Humans," *Annual Reviews in Control*, Vol. 20, 1996, pp. 167-172.

[10] S. Thrun, "An Approach to Learning Mobile Robot Navigation," *Robotics and Automation Systems*, Vol. 15, 1995, pp. 301-319.

[11] M. Kasper, G. Fricke, K. Steuernagel and E. von Puttkamer, "A Behaviour Based Learning Mobile Robot Architecture for Learning from Demonstration," *Robotics and Automation Systems*, Vol. 34, No. 2-3, 2001, pp. 153-164.

[12] W. Ilg and K. Berns, "A Learning Architecture Based on Reinforcement Learning for Adaptive Control of the Walking Machine LAURON," *Robotics and Automation Systems*, Vol. 15, No. 4, 1995, pp. 321-334.

[13] H. Friedrich, M. Kaiser and R. Dillmann, "PBD-The Key to Service Robot Programming," AAAI Technical Report SS-96-02, American Association for Artificial Intelligence, America, 1996.

[14] H. Friedrich, M. Kaiser and R. Dillmann, "Obtaining Good Performance from a Bad Teacher," *In Programming by Demonstration vs. Learning from Examples Workshop at ML'*95, Tahoe, 1995.

[15] R. Dillmann, M. Kaiser and A. Ude, "Acquisition of Elementary Robot Skills from Human Demonstration," *In International Symposium on Intelligent Robotics Systems*, Pisa, Italy, 1995, pp. 185-192.

[16] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," The MIT Press Cambridge, Massachusetts, 1998.

[17] B. Bakker, V. Zhumatiy, G. Gruener and J. Schmidhuber, "A Robot that Reinforcement-Learns to Identify and Memorize Important Previous Observations," *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, Las Vegas, USA, October 27-31, 2003, pp. 430-435.

[18] A. Billard, Y. Epars, S. Calinon, S. Schaal and G. Cheng, "Discovering Optimal Imitation Strategies," *Robotics and Automation Systems*, Vol. 47, No. 2-3, 2004, pp. 69-77.

[19] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," Second Edition, Prentice Hall, New Jersey, USA, 2002.

J. Intelligent Learning Systems & Applications, **2010, 2: 86-96**
doi:10.4236/jilsa.2010.22012 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

Machine Learning Algorithms and Their Application to Ore Reserve Estimation of Sparse and Imprecise Data

Sridhar Dutta[1], Sukumar Bandopadhyay[2], Rajive Ganguli[3], Debasmita Misra[4]

[1]Mining Engineer and MineSight Specialist, Mintec, Inc. & MineSight® Applications, Tucson, USA; [2]Mining Engineering, Fairbanks, USA; [3]Mining Engineering, University of Alaska Fairbanks, Fairbanks, USA; [4]Geological Engineering, University of Alaska Fairbanks, Fairbanks, USA.
Email: sridhar.dutta@mintec.com, {sbandopadhyay, rganguli}@alaska.edu, ffdm1@uaf.edu

Received November 2nd, 2009; revised January 6th, 2010; accepted January 15th, 2010.

ABSTRACT

Traditional geostatistical estimation techniques have been used predominantly by the mining industry for ore reserve estimation. Determination of mineral reserve has posed considerable challenge to mining engineers due to the geological complexities of ore body formation. Extensive research over the years has resulted in the development of several state-of-the-art methods for predictive spatial mapping, which could be used for ore reserve estimation; and recent advances in the use of machine learning algorithms (MLA) have provided a new approach for solving the problem of ore reserve estimation. The focus of the present study was on the use of two MLA for estimating ore reserve: namely, neural networks (NN) and support vector machines (SVM). Application of MLA and the various issues involved with using them for reserve estimation have been elaborated with the help of a complex drill-hole dataset that exhibits the typical properties of sparseness and impreciseness that might be associated with a mining dataset. To investigate the accuracy and applicability of MLA for ore reserve estimation, the generalization ability of NN and SVM was compared with the geostatistical ordinary kriging (OK) method.

Keywords: *Machine Learning Algorithms, Neural Networks, Support Vector Machine, Genetic Algorithms, Supervised*

1. Introduction

Estimation of ore reserve is essentially one of the most important platforms upon which a successful mining operation is planned and designed. Reserve estimation is a statistical problem and involves determination of the value (or quantity) of the ore in unsampled areas from a set of sample data (usually drill-hole samples) X_1, X_2, X_3, …. X_n collected at specific locations within a deposit. During this process, it is assumed that the samples used to infer the unknown population or underlying function responsible for the data are random and independent of each other. Since the accuracy of grade estimation is one of the key factors for effective mine planning, design, and grade control, estimation methodologies have undergone a great deal of improvement, keeping pace with the advancement of technology. There are a number of methodologies [1-6] that can be used for ore reserve estimation. The merits and demerits associated with these methodologies determine their application for a particular scenario. The most common and widely used methods are the traditional geostatistical estimation techniques of kriging. Typically, the previously mentioned criteria of randomness and independence among the samples are rarely observed. The samples are correlated spatially, and this spatial relationship is incorporated in the traditional geostatistical estimation procedure. The resulting information is contained in a tool known as the "variogram function," which describes both graphically and numerically the continuity of mineralization within a deposit. The information can also be used to study the anisotropies, zones of influence, and variability of ore grade values in the deposit. Although kriging estimators find a wide range of application in several fields, their estimation ability depends largely on the quality of usable data. Usable data applies to the presence of good and sufficient data to map the spatial correlation structure. Their performance is also appreciably better when a linear relationship exists between the input and output patterns. In real life, however, this is extremely unlikely. Even though there are a number of kriging versions, such as log-normal kriging and indicator kriging that apply certain specific transformations to capture nonlinear relationships, they may not be efficient enough to capture the broad nature of spatial nonlinearity.

Modernization and recent developments in computing technologies have produced several machine learning algorithms (MLA), for example, neural networks (NN) and support vector machines (SVM), that operate nonlinearly. These artificial MLA learn the underlying functional relationship inherently present in the data from the samples that are made available to them. The attractiveness of these nonlinear estimators lies in their ability to work in a black-box manner. Given sufficient data and appropriate training, they can learn the relationship between input patterns (such as coordinates) and output patterns (such as ore grades) in order to generalize and interpolate ore grades for areas between drill holes. With this approach, no assumptions must be made about factors or relationships of ore grade spatial variations, such as linearity, between boreholes.

This study investigated ore-reserve estimation capabilities of NN and SVM in the Nome gold deposit under data-sparse conditions. The performance of these MLA is validated by comparing results with the traditional ordinary kriging (OK) technique. Several issues pertaining to model development are also addressed. Various estimation errors, namely, root mean square error (RMSE), mean absolute error (MAE), bias or mean error (ME), and Pearson's correlation coefficient, were used as mea-sures to assess the relative performance of all the models.

2. Nome Gold Deposit and Data Sparseness

The Nome district is located on the south shore of the Seward Peninsula roughly at latitude 64°30'N and longitude 165°30'W. It is 840 km west of Fairbanks and 860 km northwest of Anchorage (**Figure 1**). Placer gold at Nome was discovered in 1898. Gold and antimony have been produced from lode deposits in this district, and tungsten concentrates have been produced from residual material above the scheelite-bearing lodes near Nome. Other valuable metals, including iron, copper, bismuth, molybdenum, lead, and zinc, are also reported in the Nome district.

[7] and [8] studied the Nome ore deposit and presented an excellent summary regarding its origins by chronicling their exploration and speculating on the chronology of events in the complex regional glacial history that allowed the formation and preservation of the deposit. Apart from the research just mentioned, several independent agencies have carried out exploration work in this area over the last few decades. **Figure 2** shows the composition of the offshore placer gold deposit. Altogether, around 3500 exploration drill holes have been made available by the various sampling explorations in the 22,000-acre Nome district. The lease boundary is arbitrarily divided into nine blocks named Coho, Halibut, Herring, Humpy, King, Pink, Red, Silver, and Tomcod. These blocks represent a significant gold resource in the Nome area that could be mined economically.

The present study was conducted in the Red block of the Nome deposit. Four hundred ninety-seven drill-hole samples form the data used for the investigation. Although the length of each segment of core sample collected from bottom sediment of the sea floor varied considerably, the cores were sampled at roughly 1 m intervals. On average, each hole was drilled to a depth of 30 m underneath the sea floor. Even though a database compiled from the core samples of each drill hole was made available, an earlier study by [9] revealed that most of the gold is concentrated within the top 5 m of bottom sediment of the sea floor. As a result, raw drill-hole samples were composited of the first 5 m of sea floor bottom sediment. These drill-hole composites were eventually used for ore grade modeling using NN, SVM, and Geostatistics.

Preliminary statistical analysis conducted on drill-hole composites from the Red block displayed a significantly large grade variation, with a mean and standard deviation of 440.17 mg/m^3 and 650.58 mg/m^3, respectively. The coefficient of variation is greater than one, which indicates the presence of extreme values in the dataset. Spatial variability of the dataset was studied and characterized through a variography study. **Figure 3** presents a spatial plot showing an omni-directional variogram for gold concentration in the data set. From the variogram plot, it can be observed that there is a small amount of the regional component. Large proportions of spatial variability occur from the nugget effect, indicating the presence of a poor spatial correlation structure in the deposit over the study area. Poor spatial correlation, in general, tends to suggest that prediction accuracy for this deposit might not be reliable. Hence, borehole data are sparse for reserve estimation, considering the high spatial variation of ore grade that is commonly associated with placer gold deposit. A histogram plot of the gold data is presented in **Figure 4.** The histogram plot illustrates that the gold values are positively skewed. A log-normal distribution may be a suitable fit to the data. Visual portrayal of the histogram plot also reveals that the gold datasets are composed of a large proportion of low values and a small proportion of extremely high values. Closer inspection of the spatial distribution of high and low gold-grade values portrays a distinct spatial characteristic of the deposit. For example, the high values do not exhibit any regular trend. Instead, one or two extremely

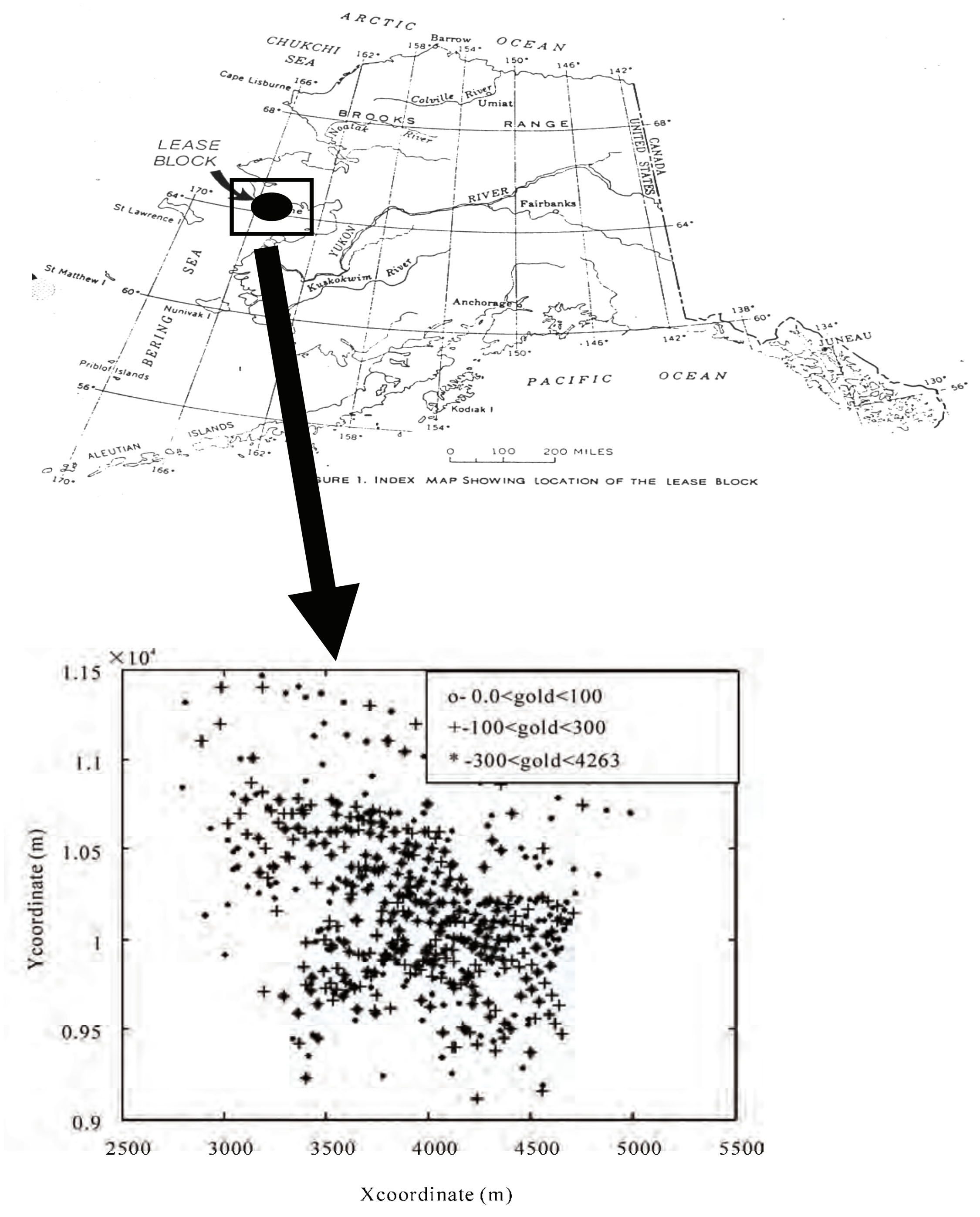


Figure 1. Location plot of the studied area

high values occasionally occur in a mix of low values. This may pose a particular difficulty in ore grade modeling, since the pattern of occurrence of extremely high values is somewhat unpredictable.

As it is discernable that the available gold data are sparse and exhibit a low level of spatial correlation, spatial modeling of these datasets is complex. Prediction accuracy may be further reduced if the problem of sparse data is not addressed. Prediction accuracy not only depends on the type of estimation method chosen but also, largely, on the model data subsets on which the model is built. Since learning models are built by exploring and capturing similar properties of the various data subsets, these data subsets should be statistically similar to each

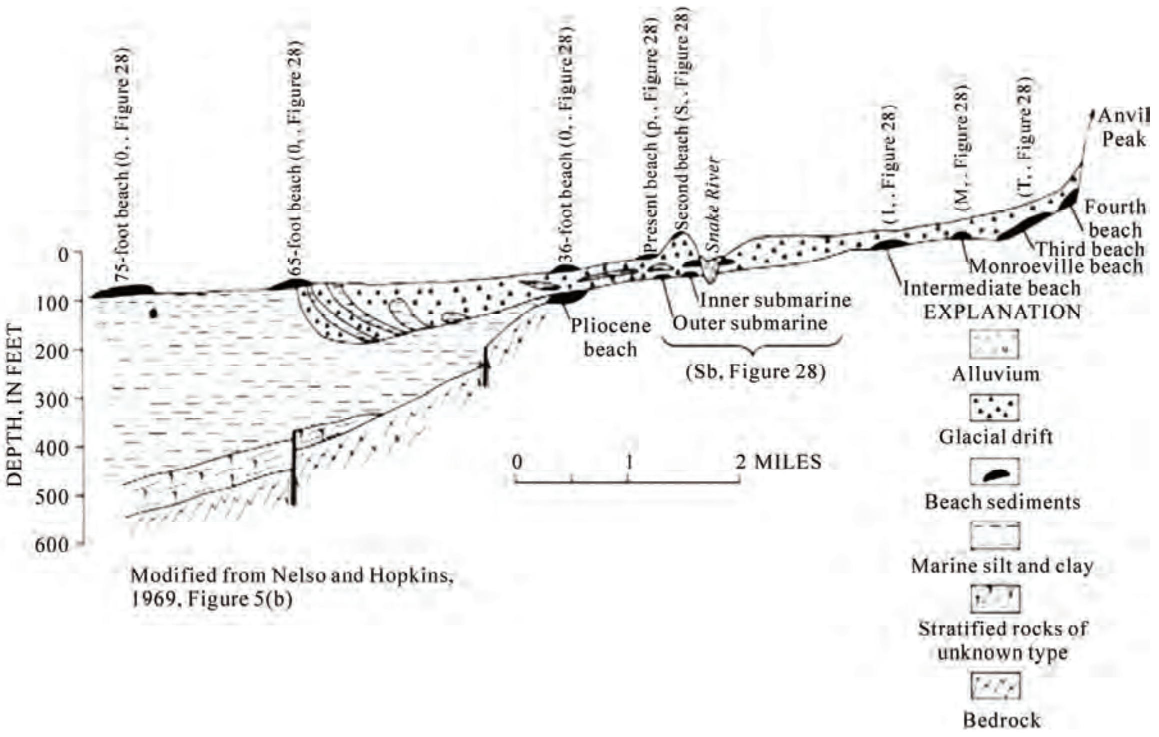


Figure 2. Offshore placer gold deposit

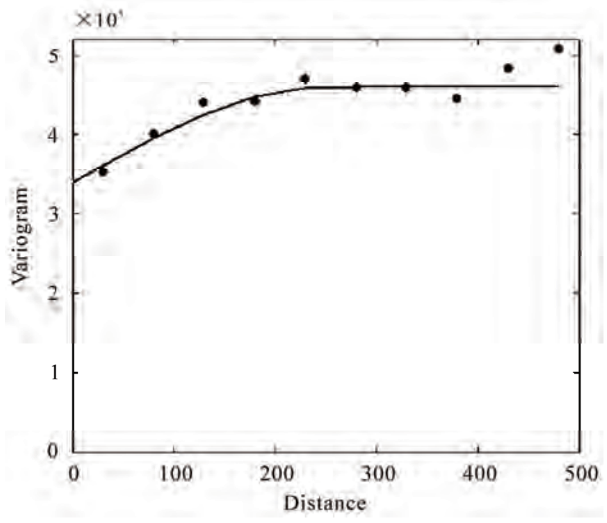


Figure 3. Omni-directional variogram plot

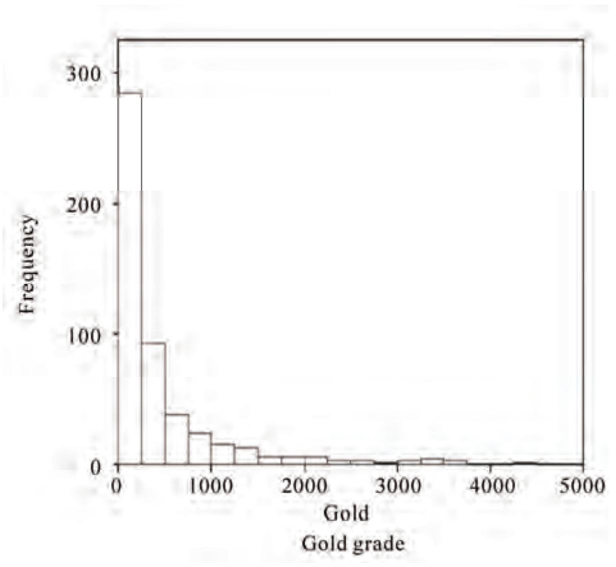


Figure 4. Histogram plot Red block

other and should reflect the statistical properties of the entire dataset. Statistical similarity ensures that the comparisons made for the model built on the training dataset and tested on the prediction dataset are logical [10,11]. Traditionally used practices of random division of data might fail to achieve the desired statistical properties when data are sparse and heterogeneous. Due to sparseness, limited data points categorized into data subsets by random division might result in dissimilarity of the data subsets [12].Consequently, overall model performance will be decreased. In order to demonstrate the severity of data sparseness in random data division, a simulation study was conducted using the Nome datasets. One hundred random data divisions were generated, in which sample members for training, calibration, and validation subsets were chosen randomly. The reason for the choice of three data subsets is presented in Section 3.0.1. Statistical similarity tests of the three data subsets, using analysis of variance (ANOVA) and Wald tests were conducted. Data division was based on a consideration of all the attributes associated with the deposit, namely, the *x*-coordinate, *y*-coordinate, water-table depth, and gold concentration. The results of the simulation study made obvious the fact that almost one-quarter of the data divisions are bad during random division of data due to the existing sparseness. This figure can be regarded as quite significant. The unreliability of random data division is further explored through inspection of a bad data division. A statistical summary for one of the arbitrarily selected random data divisions for the dataset is presented in **Table 1**. From the table, it is seen that both the mean and standard deviation values are significantly different. Therefore, careful subdivision of data during model development is essential. Various methodologies were investigated in this regard for proper data subdivision under such a modeling framework, including the application of genetic algorithms (GA) [3,5,13,14] and Kohonen networks [11,14]. A detailed description of the theory and working principle of these methodologies can be found in any NN literature [15,16].

3. Nome Gold Reserve Estimation

When estimating ore grade, northing, easting, and water-table depth were considered as input variables, and gold grade associated with drill-hole composites up to a depth of 5 m of sea floor was considered an output variable. The next few sections describe the application of NN and SVM to ore reserve estimation, along with various issues that arose while using NN and SVM for ore reserve modeling.

3.1 NN for Grade Estimation

Neural networks form a computational structure inspired by the study of biological neural processing. This structure exhibits certain brain-like capabilities, including perception, pattern recognition, and pattern prediction in a variety of situations. As with the brain, information processing is done in parallel using a network of “neurons.” As a result, NN have capabilities that go beyond algorithmic programming and work exceptionally well for nonlinear input-output mapping. It is this property of nonlinear mapping that makes NN appealing for ore grade estimation.

There is a fundamental difference in the principles of OK and NN. While OK utilizes information from local samples only, NN utilize information from all of the samples. Ordinary Kriging is regarded as a local estimation technique, whereas NN are global estimation techniques. If any nonlinear spatial trend is present in a deposit, it is expected that the NN will capture it reasonably well. The basic mechanisms of NN have been discussed at length in the literature [15,17]. A brief discussion of NN theory is presented below to provide an overview of the topic.

In NN, information is processed through several interconnected layers, where each layer is simply represented by a group of elements designated as neurons. Basic NN architecture is made of an input layer consisting of inputs, one or more *hidden layers* consisting of a number of neurons, and the output layer consisting of outputs. Typical network architecture, having three layers, is presented in **Figure 5**. Note that while the input layer and the output layer have a fixed number of elements for a given problem, the number of elements in the hidden layer is arbitrary. The basic functioning of NN involves a manipulation of the elements in the input layer and the hidden layer by a weighing function to generate network output. The goodness of the resulting output (how realistic it is) depends upon how each element in the layers is weighted to capture the underlying phenomenon. As it is apparent that the weights associated with the interconnections largely decide output accuracy, they must be determined in such a way as to result in minimal error. The process of determination of weights is called *learning* or *training* during which, depending upon the output, NN adjust weights iteratively based on their contribution to the error. This process of propagating the effect of the error onto all the weights is called *back-propagation*. It is during the process of learning that NN map the patterns pre-existing in the data by reflecting the changes in data fluctuations in a spatial coordinate. The sample dataset for a given deposit is used for this purpose. Therefore, given the spatial coordinates and other relevant attributes as input and the grade attribute as output, NN will be able to generate a mapping function through a set of connection weights between the input and output. Hence, output, O, of a neural network can be regarded as a function of inputs, X, and connection weights, W: $O = \varphi\ (X)$, where φ is a mapping function. Training of NN involves finding a good mapping function that maps the input-output patterns correctly. This is done, as previously described, by adjusting connection weights between the neurons of a network, using a suitable learning algorithm while simultaneously fixing the network architecture and activation function.

An additional criterion for optimization of the NN architecture is to choose the network with minimal generalization error. The main goal of NN modeling is not to generate an exact fit to the training data, but to generalize a model that will represent the underlying characteristics of a process. A simple model may result in poor generalization, since it can not take into account all the intricacies present in the data. On the other hand, a too-complex

Table 1. Statistical summary of one of the random divisions for the Red dataset

Attribute	Mean				Standard Deviation			
	Overall	Training	Calibration	Validation	Overall	Training	Calibration	Validation
X	3941.8	3947.7	3838.2	4032.6	456.54	436.89	505.50	425.5
Y	10198	10174	10350	10097	469.75	483.19	487.22	384.06
Gold	440.17	297.75	781.77	385.00	650.58	353.31	10340	475.12
WTD	8.4845	8.94	6.6242	9.414	5.2063	5.1679	5.3559	4.69

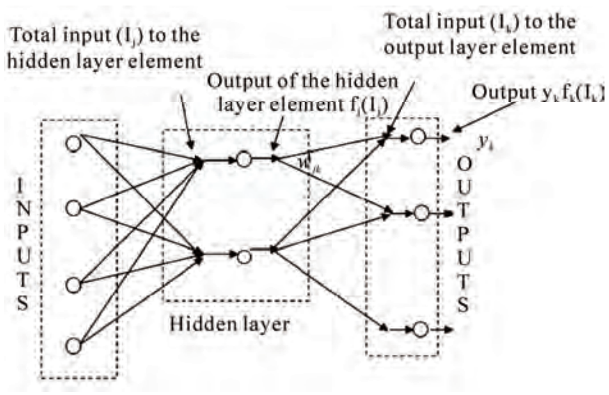

Total input (I_j) to the hidden layer element
Output of the hidden layer element $f_j(I_j)$
Total input (I_k) to the output layer element
Output $y_k f_k(I_k)$
y_k
w_{jk}
INPUTS
Hidden layer
OUTPUTS

Figure 5. A typical neural network architecture

model is flexible enough to fit data with anomalies or noise. Therefore, complexity of a model should be well matched to improve generalization properties of the data. Past research has been devoted to improving the generalization of NN models, including techniques such as regularization, quick-stop training, and smoothing, and combining several learning models using various ensemble techniques like bragging and boosting [1,18]. In the present study, a quick-stop training method is employed to improve the NN model generalization. Quick-stop training is based on the notion that generalization performance varies over time as the network adapts during training [15]. Using this method, the dataset is split into three subsets: training, calibration, and validation. The network actually undergoes training on the training set. However, the decision to stop the training is made on the network's performance in the calibration set. The error for the training set decreases monotonically with an increasing number of iterations; however, the error for the calibration set decreases monotonically to a minimum, and then starts to increase as the training continues. A typical profile of the training error and the calibration error of a NN model is presented in **Figure 6**. This observed behavior occurs because, unlike the training data, the calibration data are not used to train the network. The calibration data are simply used as an independent measure of model performance. Thus, it is possible to stop over-training or under-training by monitoring the performance of the network on the calibration subset, and then stop the training when the calibration error starts increasing. In order to make a valid model-performance measurement, the training, calibration, and validation subsets should have similar statistical properties. Thus, the members of the data in the training, calibration, and validation subsets should be selected in such a way that the three data subsets acquire similar statistical properties. Once the data subsets are obtained, a NN model is developed based on the NN architecture and learning rule to generate model outputs.

3.2 NN Grade Estimation Results

The Levenberg-Marquardt backpropagation (LMBP) learning algorithm was used in conjunction with slab architecture, as shown in **Figure 7**, for NN modeling. The hidden layer consisted of 12 neurons. The number of hidden neurons chosen was based on the minimum generalization errors of NN models while experimenting with a different number of hidden nodes in the hidden slabs. A MATLAB code was developed for conducting all the studies associated with NN modeling. The model datasets were obtained by following an integrated approach using data segmentation and GA. Data segmentation involved the division of data into three prime segments of high-, medium-, and low-grade gold concentrations. This division was based on a visual inspection of the histogram plot. **Figure 8** presents a schematic diagram of data segmentation and the GA approach. After data segmentation, GA was applied in each of the segments: segment 1, segment 2, and segment 3. The members of the training, calibration, and the validation datasets were selected using GA from each of the segments. Once the members for the training, calibration, and validation data were chosen, the selected members from the segments were appended together to form respective subsets. **Table 2** presents the summary statistics of the mean and standard deviation values for all variables of the three data subsets and the entire dataset. Observe that the mean and standard deviation values are similar for all the data subsets. The histogram plots of the three subsets

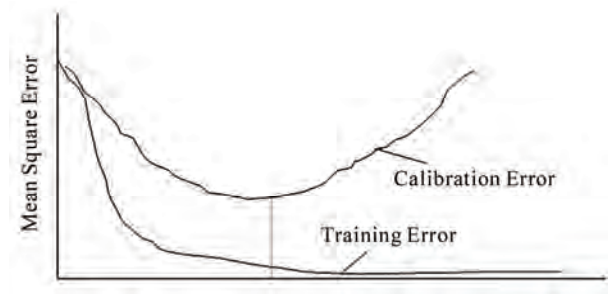


Figure 6. A typical profile of training and calibration error of a NN model [16]

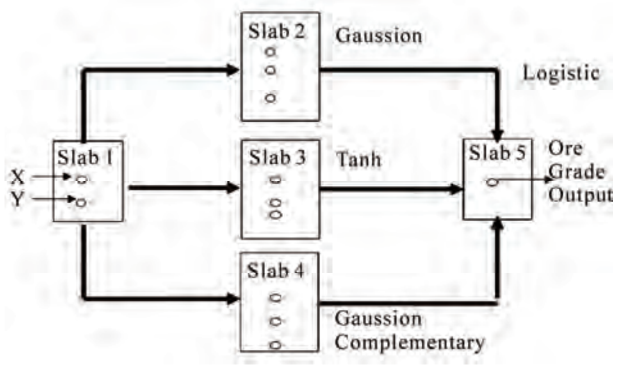


Figure 7. Slab architecture for NN modeling

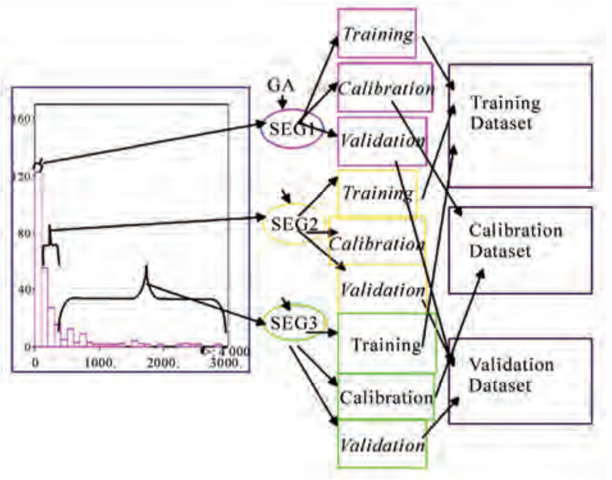


Figure 8. Data segmentation and genetic algorithms for data divisions

and the entirety of the nine datasets are presented in **Figure 9**. From the figure, it can be seen that all the data subsets assume an almost identical shape to that of the overall dataset, and that the skewness of the data in the three subsets is preserved. **Table 3** presents summary statistics of the generalization performance of the NN model for the Red dataset, while **Figure 10** presents a scatterplot of the actual values versus predicted values of the validation data subset for the Red block.

3.3 SVM for Ore Grade Estimation

The SVM method is based on statistical learning theory (SLT) and performing structural risk minimization (SRM). Popularly known as support vector regression (SVR) for its regression abilities, the SVR not only has a solid mathematical background but also is robust to noise in measurements [19-21]. Support vector regression acquires knowledge from the training data by building a model, during which the expected risk, R, is approximated and minimized by the empirical risk, R_{emp}. This process always involves a generalization error bound and is given by

$$R(h) \le R_{emp}(h) + \Omega(h) \qquad (1)$$

where R is the bound on the testing error, R_{emp} is the empirical risk on the training data, and Ω is the confidence term that depends on the complexity of the modeling function. Though a brief explanation of how the SVR approach works is described below, interested readers are referred to [20-22]. Given the training dataset $\{(\boldsymbol{x_i}, y_i), i = 1, 2, \ldots, L\}$, where $\boldsymbol{x_i}$ is the input variable and y_i is the output variable, the idea of SVR is to develop a linear regression hyperplane expressed in Equation (2), which allows, at most, ε deviation for the true values, y_i, in the training data (see **Figure 11**) and at the same time searches for a solution that is as flat as possible [21].

$$f(x) = W_O^T \phi(x) + b \qquad (2)$$

where W_o is the optimum weight vector, b is the bias, and $\phi(x)$ is a mapping function used to transform the input variable in the input space to a higher dimensional feature space. This transformation allows the handling of any nonlinearity that might exist in the data. The desired flatness is obtained by seeking a small w [21]. In reality, however, a function that approximates all the $(\boldsymbol{x_i}, y_i)$ pairs with ε precision may not be feasible. Slack variables ε_i, ε_i^* [23] are introduced in such cases that allow the incorporation of some amount of error (see **Figure 11**). The problem of obtaining a small w and at the same time restricting the errors to, at most, ε deviation after introducing the slack variables can be obtained by solving the following convex quadratic optimization problem:

$$\text{minimize} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l}(\varepsilon_i + \varepsilon_i^*)$$

$$\text{subject to} \quad \begin{aligned} & y_i - (w^t\phi(x) - b) \le \varepsilon + \varepsilon_i \\ & (w^t\phi(x) + b) - y_i \le \varepsilon + \varepsilon_i^* \\ & \varepsilon_i, \varepsilon_i^* \ge 0 \end{aligned} \qquad (3)$$

In Equation (3), both the empirical risk, realized by the training error $\Sigma(\varepsilon_i + \varepsilon_i^*)$, as well as the confidence term,

Table 2. Statistical summary of data division using GA (Red)

Attribute	Mean				Standard Deviation			
	Overall	Training	Calibration	Validation	Overall	Training	Calibration	Validation
X	3941.8	3950	3935.6	3931.6	456.54	456.6	457.6	458.6
Y	10198	10194	10218	10186	469.75	471.2	467.2	472.4
Gold	440.17	461.99	418.46	418.59	650.58	673.97	627.89	628.8
WTD	8.4845	8.38	8.63	8.54	5.2063	5.23	5.19	5.19

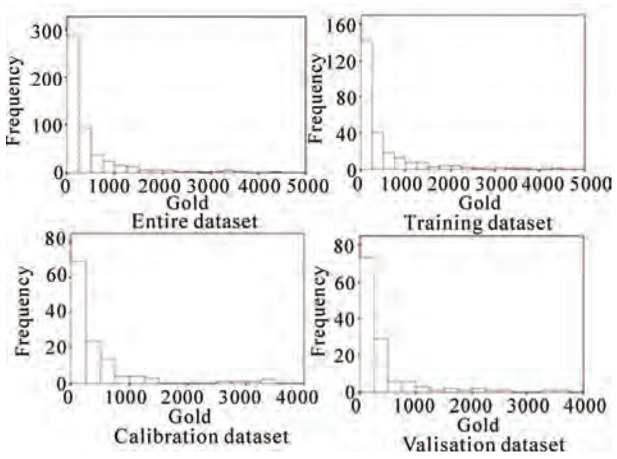


Figure 9. Histogram plot of Red dataset

Table 3. Generalization performance of the models for the Red block

Statistics	SVM	NN	OK
Mean Error	-8.1	-1.30	33.54
Mean Absolute Error	341.2	351.50	353.02
Root Mean Squared Error	563.13	564.34	565.23
R^2	0.234	0.191	0.193

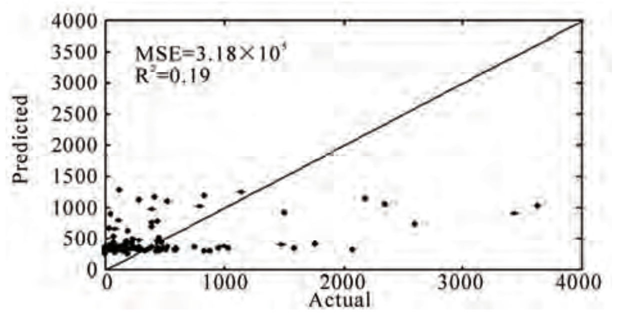


Figure 10. Actual vs. predicted for the validation data of Red block

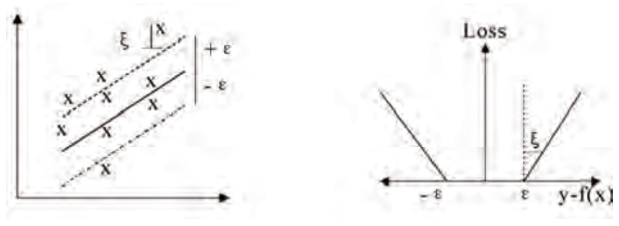


Figure 11. The soft margin loss settings for linear SVM [21]

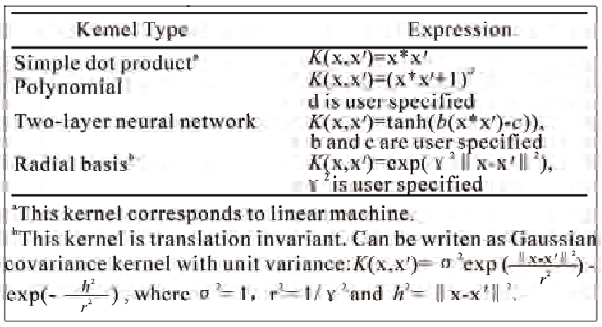
realized by the $||w||^2$ term (expressed by Equation (1)) are minimized. An optimum hyperplane is obtained by solving the above minimization problem employing the Kharush-Kuhn-Tucker (KKT) conditions [24] which results in minimum generalization error. The final formulation to obtain the SVR model predictions is given by

$$\begin{aligned} f(x) &= \sum_{i=1}^{l}(\alpha_i - \alpha_i^*)\phi(x_i)\phi(x) + b \\ &= \sum_{i=1}^{l}(\alpha_i - \alpha_i^*)K(x_i, x) + b \end{aligned} \tag{4}$$

where α_i, α_i^* are the weights corresponding to individual input patterns, $K(x_i, x)$ is a user-defined kernel function, and b is the bias parameter. **Figure 12** presents a list of commonly used kernels. The most commonly used kernel function is an isotropic Gaussian RBF defined by

$$K(x_i, x) = e^{\frac{-||x_i - x||^2}{2\sigma^2}} \tag{5}$$

where σ is the kernel bandwidth. The solution of this optimization problem might result in zero weight for some input patterns and non-zero weight for the rest. The patterns with zero weight are redundant and are insignificant to the model structure. On the other hand, input patterns with non-zero weights are termed *support vectors* (SV), and they are vital to obtaining model predictions. As the number of support vectors increases, so does model complexity. The main parameters that influence SVR model performance are the C, σ, and ε. Parameter C is a trade-off between empirical risk and the weight vector norm $||w||$. It decreases empirical risk but, at the same time, increases model complexity, which



Kemel Type	Expression
Simple dot product[a]	$K(x,x')=x*x'$
Polynomial	$K(x,x')=(x*x'+1)^d$ d is user specified
Two-layer neural network	$K(x,x')=\tanh(b(x*x')-c))$, b and c are user specified
Radial basis[b]	$K(x,x')=\exp(\gamma^2 \| x-x' \|^2)$, γ^2 is user specified

[a]This kernel corresponds to linear machine.
[b]This kernel is translation invariant. Can be writen as Gaussian covariance kernel with unit variance: $K(x,x')=\sigma^2\exp(\frac{\| x-x' \|^2}{r^2})$-$\exp(-\frac{h^2}{r^2})$, where $\sigma^2=1$, $r^2=1/\gamma^2$ and $h^2=\| x-x' \|^2$.

Figure 12. Commonly used kemels in SVM

deteriorates model generalization. Parameter ε defines the width of the insensitivity zone, and controls the number of SV. The effect of increase in ε is a decrease in the number of SV, which results in smoothing of the final solution during modeling of noisy data. Similarly, note from Equation (5) that a higher value of kernel width, σ, has a smoothing effect on the solution. Optimal values of these parameters can be obtained by a grid-search procedure.

3.4 SVM Grade Estimation Results

Out of the numerous options available for the choice of kernel function, a RBF kernel was selected, and the recommendations of [20] and [25] were considered carefully in the development of the SVM-based model. As per recommendations, the input data were first scaled assuming a uniform distribution. In other words, the scaled value of an attribute was calculated using the maximum and minimum values of the attribute. The drill holes were used to estimate SVM parameters, the cost function (C), the radial basis kernel parameter (σ), and the error margin (ε). Optimal SVM parameters were determined based on a K-fold cross-validation technique applied to the training dataset. The K-fold cross-validation approach splits the available data into more or less K equal parts. Of the K parts of the data, only K-1 parts of the data were used to find the SVM estimate and calculate the error of the fitted model, and for predicting the k^{th} part of the data as part of the validation process. The procedure was then repeated for $k = 1, 2, \ldots, K$, and the selection of parameters was based on the minimum prediction error estimates over all K parts. The value of K is based on the shape of a "learning curve" [26], which is a plot of the training error versus the training size. For given SVM parameters, the training errors are calculated by progressively estimating the SVM model for increased training data size, thereby constituting the learning curve. From the learning curve, an optimum training size (or in other words, the number of folds, K) can be obtained where the error is minimal. In this study, the optimum value of K was found to be 10. Once the value of K is obtained, the SVM model is trained using K-fold cross validation. Training and testing involves a thorough grid search for optimal C and σ values. Thus, unlike NN, where training involves passing a dataset through hidden layers to optimize the weights, optimal training of SVM involves estimation of parameters C and σ through a grid search such that the error is minimized. The optimum values for C and σ for the Red block was found to be 0.53 and 9.5. These values are depicted by the troughs and flat regions of the error surface in **Figure 13**. Once the optimal values for the SVM model parameters were determined, the model was tested for its generalization ability on validation datasets for the Red block. **Figure 14** shows the performance of the SVM model in predicting gold grade for the Red block, while performance statistics are presented in **Table 3**.

4. Summary and Conclusions

Nome gold reserve estimation is challenging because of the geologic complexity associated with placer gold deposits and because of sparse drill holes. Each drill hole contains information on northing (Y-coordinate), easting (X-coordinate), water-table depth, and gold grade in mg/m^3, as well as other relevant information. For grade estimation, northing, easting, and water-table depth were considered input variables, and gold grade was considered an output variable. Gold grade up to a 5 m sea floor depth, were considered. The gold grade associated with the Nome deposit Red block was estimated using two MLA—the NN method and the SVM method—and their performance were compared using the traditional geostatistical OK technique. Various issues involved in the use of these techniques for grade estimation were discussed. Based on the results from this study, the SVM-based model produced better estimates as compared with the other two methods. However, the improvement was only marginal, which may be due to the presence of extreme data values. The various criteria used to compare model performance were the mean error (ME), the mean absolute error (MAE), the root mean squared error (RMSE), and the coefficient of determination (R^2). Generally, a model with less error and high R^2 is considered a better fit. Since the improvements were only marginal, a summary statistic was developed to compare the three models. This summary statistic, termed the *skill value*, is an entirely subjective measurement, expressed by Equation (6) [14,27-29]. Numerous skill measures can be devised; however, the one proposed in this study considers the ME, MAE, and RMSE equally, and applies scaling to the R^2 so that it is of the same order of magnitude as the others. Note that the lower the skill value, the better the method is. In this way, various methods can be ranked

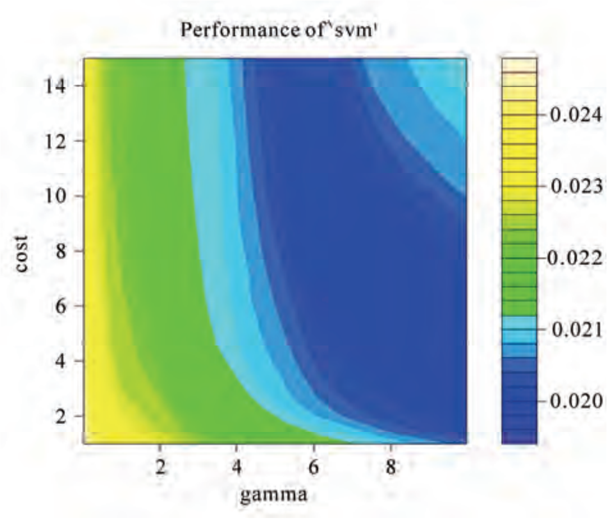


Figure 13. Effect of cost and kernel width variation on error (Red)

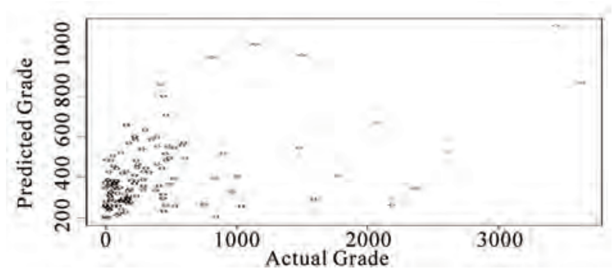


Figure 14. Scatterplot for actual vs. predicted grade (Red)

Table 4. Model performances based on skill values

Statistics	SVM	NN	OK
Skill Value	989.03	998.03	1032.49
Rank	01	02	03

based on their skill values, that is, their overall performance on the prediction dataset.

$$\text{'skill value'} = \text{abs (ME)} + \text{MAE} + \text{RMSE} + 100 \times (1 - R^2) \quad (6)$$

Table 4 presents skill values and ranks for the various methods that were used on the prediction dataset. It can be seen from the table that the MLA performed significantly better than the traditional kriging method. The difference in the skill values is mainly due to the high variation in the R^2 (**Table 3**).

REFERENCES

[1] S. Dutta, D. Mishra, R. Ganguli and B. Samanta, "Investigation of two Neural Network Ensemble Methods for the Prediction of Bauxite Ore Deposit," *Proceedings of the* 6th *International Conference on Information Technology*, Bhubaneswar, December 22-25, 2003.

[2] S. Dutta, R. Ganguli and B. Samanta, "Comparative Evaluation of Radial Basis Functions and Kriging for Ore Grade Estimation," 32nd *International Symposium of the application of Computers and Operation research in Mineral Industry*, Arizona, USA, 2005, pp. 203-211.

[3] S. Dutta, D. Misra, R. Ganguli, B. Samanta and S. Bandopadhyay, "A Hybrid Ensemble Model of Kriging and Neural Network for Ore Grade Estimation," *International Journal of Surface Mining, Reclamation and Environment*, Vol. 20, No. 1, 2006a, pp. 33-46.

[4] S. Dutta, S. Bandopadhyay and B. Samanta, "Support Vector Machines—An Emerging Technique for Ore Reserve Estimation," *Proceedings of the Sixth International Symposium on Information Technology Applied to Mining* (*CD*), Peruvian Institute of Mining Engineers, 2006b.

[5] B. Samanta, S. Bandopadhyay, R. Ganguli and S. Dutta, "A Comparative Study of the Performance of Single Neural Network vs. Adaboost Algorithm Based Combination of Multiple Neural Networks for Mineral Resource Estimation," *Journal of South African Institute of Mining and Metallurgy*, Vol. 105, No. 4, 2005a, pp. 237-246.

[6] B. Samanta, R. Ganguli and S. Bandopadhyay, "Comparing the Predictive Performance of Neural Networks with Ordinary Kriging in a Bauxite Deposit," *Transactions of Institute of Mining and Metallurgy, Section A, Mining Technology*, Vol. 114, No. 3, 2005b, pp. 129-139.

[7] D. M. Hopkins and L. MacNeil, "Dredged Area: Alaska Division of Mines and Minerals," 1960.

[8] P. C. Rusanowski, "Nome Offshore Gold Placer Project: Nova," Natural Resources Corp., Alaska, 1994.

[9] J. Ke, "Neural Network Modeling for Placer Ore Grade Spatial Variability," Ph.D. dissertation, University of Alaska Fairbanks, Fairbanks, 2002.

[10] G. J. Bowden, H. R. Maier and G. C. Dandy, "Optimal Division of Data for Neural Network Models in Water Resources Application," *Water Resources Research*, Vol. 38, No. 2, 2002, pp. 1-11.

[11] S. Yu, R. Ganguli, D. E. Walsh, S. Bandopadhyay and S. L. Patil, "Calibration of On-line Analyzers Using Neural Networks," *Mining Engineering*, Vol. 56, No. 9, 2003, pp. 99-102.

[12] R. Ganguli and S. Bandopadhyay, "Dealing with Sparse Data Issues in a Mineral Industry Neural Network Application," *Proceedings Computer Applications in the Mineral Industry* (*CAMI*), Calgary, Alberta, Canada, 2003, pp. 1-5.

[13] B. Samanta, S. Bandopadhyay and R. Ganguli, "Data Segmentation and Genetic Algorithms for Sparse Data Division in Nome Placer Gold Grade Estimation Using Neural Network and Geostatistics," *Exploration and Mining Geology*, Vol. 11, 2004, pp. 69-76.

[14] S. Dutta, "Predictive Performance of Machine Learning Algorithms for Ore Reserve Estimation in Sparse and Imprecise Data," Ph.D. dissertation, University of Alaska

Fairbanks, Fairbanks, 2006.

[15] M. T. Hagan, H. B. Demuth and M. Beale, "Neural Network Design," PWS Publishing Company, Boston, MA, 1995.

[16] S. Haykins, "Neural Networks: A Comprehensive Foundation," 2nd Edition, Prentice Hall, New Jersey, 1999.

[17] C. M. Bishop, "Neural Networks for Pattern Recognition," Clarendon Press, Oxford, 1995.

[18] S. Dutta and R. Ganguli, "Application of Boosting Algorithm in Neural Network Based Ash Measurement Using Online Ash Analyzers," 32nd *International Symposium of the Application of Computers and Operation Research in Mineral Industry*, Arizona, USA, 2005b.

[19] V. Kecman, "Learning and Soft Computing: Support Vector Machines, Neural Network and Fuzzy Logic Models," MIT Publishers, USA, 2000.

[20] V. Kecman, "Support Vector Machines Basics—An Introduction Only," University of Auckland, School of Engineering Report, New Zealand, 2004.

[21] A. J. Smola and B. Scholkopf, "A Tutorial on Support Vector Regression," *Statistics and Computing*, Vol. 14, No. 3, 2004, pp. 199-222.

[22] V. Vapnik, "Statistical Learning Theory," John Wiley and Sons, New York, 1998.

[23] C. Cortes, and V. Vapnik, "Support Vector Networks," *Machine Learning*, Vol. 20, No. 3, 1995, pp. 273-297.

[24] R. E. Miller, "Optimization-Foundations and Applications," John Wiley and Sons, New York, 2000.

[25] C. C. Chang and C. Lin, "LIBSVM: A Library for Support Vector Machines," 2001. Internet Available: http://www.csie.ntu.edu.tw/~cjlin/libsvm

[26] T. Hastie, R. Tibshirani and J. Friedman, "The Elements of Statistical Learning Theory—Data Mining, Inference and Prediction," Springer, New York, 2001.

[27] G. Dubois, "European Report on Automatic Mapping Algorithms for Routine and Emergency Monitoring Data," Office for Official Publications of the European Communities, Luxembourg, 2005.

[28] S. Dutta, R. Ganguli and B. Samanta, "Investigation of two Neural Network Methods in an Automatic Mapping Exercise," *Journal of Applied GIS*, Vol. 1, No. 2, 2005a, pp. 1-19.

[29] S. Dutta, R. Ganguli and B. Samanta, "Investigation of two Neural Network Methods in an Automatic Aapping Exercise," In: G. Dubois, Ed., *European Report on Automatic Mapping Algorithms for Routine and Emergency Monitoring Data*. *Report on the Spatial Interpolation Comparison* (*SIC* 2004) *Exercise*, Office for Official Publications of the European Communities, Luxembourg, 2005c.

J. Intelligent Learning Systems & Applications, **2010, 2: 97-109**
doi:10.4236/jilsa.2010.22013 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

Design of Hybrid Fuzzy Neural Network for Function Approximation

Amit Mishra[1]*, Zaheeruddin[1]

[1]Jamia Millia Islamia (A Central University), Department of Electrical Engineering, New Delhi, India; *Jaypee Institute of Engineering and Technology, Madhya Pradesh, India.
Email: amitutk@ gmail.com

Received November 21st, 2009; revised April 25th, 2010; accepted April 30th, 2010.

ABSTRACT

In this paper, a hybrid Fuzzy Neural Network (FNN) system for function approximation is presented. The proposed FNN can handle numeric and fuzzy inputs simultaneously. The numeric inputs are fuzzified by input nodes upon presentation to the network while the Fuzzy rule based knowledge is translated directly into network architecture. The connections between input to hidden nodes represent rule antecedents and hidden to output nodes represent rule consequents. All the connections are represented by Gaussian fuzzy sets. The method of activation spread in the network is based on a fuzzy mutual subsethood measure. Rule (hidden) node activations are computed as a fuzzy inner product. For a given numeric o fuzzy input, numeric outputs are computed using volume based defuzzification. A supervised learning procedure based on gradient descent is employed to train the network. The model has been tested on two different approximation problems: sine-cosine function approximation and Narazaki-Ralescu function and shows its natural capability of inference, function approximation, and classification.

Keywords: *Cardinality, Classifier, Function Approximation, Fuzzy Neural System, Mutual Subsethood*

1. Introduction

The conventional approaches to system modeling that are based on mathematical tools (*i.e.* differential equations) perform poorly in dealing with complex and uncertain systems. The basic reason is that, most of the time; it is very difficult to find a global function or analytical structure for a nonlinear system. In contrast, fuzzy logic provides an inference morphology that enables approximate human reasoning capability to be applied in a fuzzy inference system. Therefore, a fuzzy inference system employing fuzzy logical rules can model the quantitative aspects of human knowledge and reasoning processes without employing precise quantitative analysis.

In recent past, artificial neural network has also played an important role in solving many engineering problems. Neural network has advantages such as learning, adaptation, fault tolerance, parallelism, and generalization. Fuzzy systems utilizing the learning capability of neural networks can successfully construct the input output mapping for many applications. The benefits of combining fuzzy logic and neural network have been explored extensively in the literature [1-3].

The term neuro-fuzzy system (also neuro-fuzzy methods or models) refers to combinations of techniques from neural networks and fuzzy system [4-8]. This never means that a neural network and a fuzzy system are used in some kind of combination, but a fuzzy system is created from data by some kind of (heuristic) learning method, motivated by learning procedures used in neural networks. The neuro-fuzzy methods are usually applied, if a fuzzy system is required to solve a problem of function approximations or special case of it, like, classification or control [9-12] and the otherwise manual design process should be supported and replaced by an automatic learning process.

Here, the attention has been focused on the function approximation and classification capabilities of the subsethood based fuzzy neural model (subsethood based FNN). This model can handle simultaneous admission of fuzzy or numeric inputs along with the integration of a fuzzy mutual subsethood measure for activity propagation. A product aggregation operator computes the strength of firing of a rule as a fuzzy inner product and works in conjunction with volume defuzzification to generate numeric outputs. A gradient descent framework allows the model to fine tune rules with the help of numeric data.

The organization of the paper is as follows: Section 2 presents the architectural and operational detail of the model. Sections 3 and 4 demonstrate the gradient descent learning procedure and the applications of the model to the task of function approximation respectively. Finally, the Section 5 concludes the paper.

2. Architectural and Operational Detail

To develop a fuzzy neural model, following issues are important to be discussed:

1) Signal transmission at input node.

2) Signal transmission method (similarity measures) from input nodes to rule nodes.

3) Method for activity aggregation at rule nodes.

4) Signal computation at output layer.

5) Learning technique and its mathematical formulation used in model.

The proposed architecture of subsethood based Fuzzy neural network is shown in **Figure 1**. Here x_1 to x_m and x_{m+1} to x_n are numeric and linguistic inputs respectively. Each hidden node represents a rule, and input-hidden node connection represents fuzzy rule antecedents. Each hidden-output node connection represents a fuzzy rule consequent. Fuzzy set corresponding to linguistic levels of fuzzy *if-then* rules are defined on input and output UODs and are represented by symmetric Gaussian membership functions specified by a center and spread. The center and spread of fuzzy weights w_{ij} from input nodes i to rule nodes j are shown as c_{ij} and σ_{ij} of a Gaussian fuzzy set and denoted by $w_{ij} = (c_{ij}, \sigma_{ij})$. As this model can handle simultaneous admission of numeric as well as fuzzy inputs, Numeric inputs are first fuzzified so that all outputs transmitted from the input layer of the network are fuzzy. Now, since the antecedent weights are also fuzzy, this requires the design of a method to transmit a fuzzy signal along a fuzzy weight. In this model signal transmission along the fuzzy weight is handled by calculating the mutual subsethood. A product aggregation operator computes the strength of firing at rule node. At output layer the signal computation is done with volume defuzzification to generate numeric outputs (y_1 to y_p). A gradient descent learning technique allows the model to fine tune rules with the help of numeric data.

2.1 Signal Transmission at Input Nodes

Since input features $x_1, \ldots, x_n$ can be either numeric and linguistic, there are two kinds of nodes in the input layer. Linguistic nodes accept a linguistic input represented by fuzzy sets with a Gaussian membership function and modeled by a center c_i and spread σ_i. These linguistic inputs can be drawn from pre-specified fuzzy sets as shown in **Figure 2**, where three Gaussian fuzzy sets have been defined on the universe of discourse (UODs) [–1, 1]. Thus, a linguistic input feature x_i is represented by the pair (c_i, σ_i). No transformation of inputs takes place at linguistic nodes in the input layer. They merely transmit the fuzzy input forward along antecedent weights.

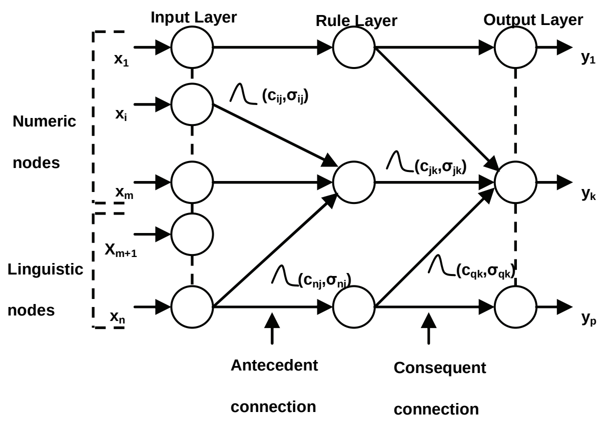


Figure 1. Architecture of subsethood based FNN model

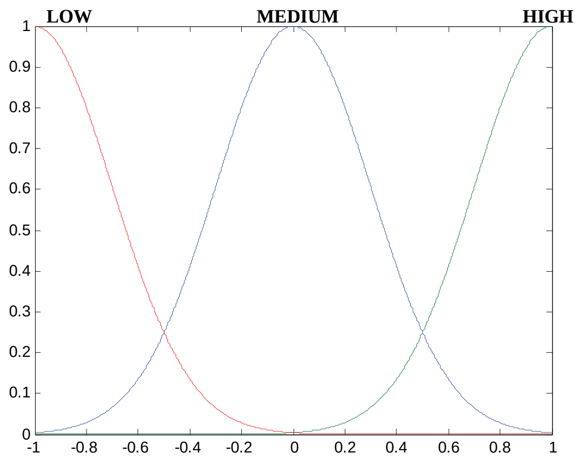


Figure 2. Fuzzy sets for fuzzy inputs

Numeric nodes accept numeric inputs and fuzzify them into Gaussian fuzzy sets. The numeric input is fuzzified by treating it as the centre of a Gaussian membership function with a heuristically chosen spread. An example of this fuzzification process is shown in **Figure 3**, where a numeric feature value of 0.3 has been fuzzified into a Gaussian membership function centered at 0.3 with spread 0.35. The Gaussian shape is chosen to match the Gaussian shape of weight fuzzy sets since this facilitates subsethood calculations detailed in Section 2.2. Therefore, the signal from a numeric node of the input layer is represented by the pair (c_i, σ_i). Antecedent connections uniformly receive signals of the form (c_i, σ_i). Signals ($S(x_i) = (c_i, \sigma_i)$) are transmitted to hidden rule nodes through fuzzy weights also of the form (c_{ij}, σ_{ij}), where single subscript notation has been adopted for the input sets and the double subscript for the weight sets.

2.2 Signal Transmission from Input to Rule Nodes (Mutual Subsethood Method)

Since both the signal and the weight are fuzzy sets, being

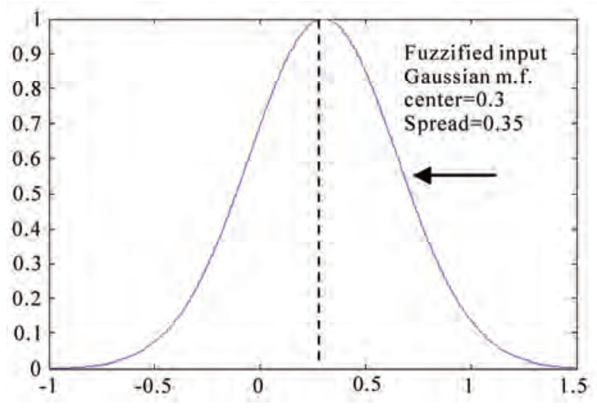


Figure 3. Fuzzification of numeric input

represented by Gaussian membership function, there is a need to quantify the net value of the signal transmitted along the weight by the extent of overlap between the two fuzzy sets. This is measured by their *mutual subsethood* [13].

Consider two fuzzy sets A and B with centers c_1, c_2 and spreads σ_1, σ_2 respectively. These sets are expressed by their membership functions as:

$$a(x) = e^{-((x-c_1)/\sigma_1)^2} \tag{1}$$

$$b(x) = e^{-((x-c_2)/\sigma_2)^2} \tag{2}$$

The cardinality $C(A)$ of fuzzy set A is defined by

$$C(A) = \int_{-\infty}^{\infty} a(x)dx = \int_{-\infty}^{\infty} e^{-((x-c_1)/\sigma_1)^2} dx \tag{3}$$

Then the mutual subsethood $\varepsilon(A,B)$ of fuzzy sets A and B measures the extent to which fuzzy set A equals fuzzy set B can be evaluated as:

$$\varepsilon(A,B) = \frac{C(A \cap B)}{C(A) + C(B) - C(A \cap B)} \tag{4}$$

Further detail on the mutual subsethood measure can be found in [13]. Depending upon the relative values of centers and spreads of fuzzy sets A and B, the following four different cases of nature of overlap arise:

Case 1: $c_1 = c_2$ having any values of σ_1 and σ_2.

Case 2: $c_1 \neq c_2$ and $\sigma_1 = \sigma_2$.

Case 3: $c_1 \neq c_2$ and $\sigma_1 > \sigma_2$.

Case 4: $c_1 \neq c_2$ and $\sigma_1 < \sigma_2$.

In case 1, the two fuzzy sets do not cross over-either one fuzzy set belongs completely to the other or two fuzzy sets are identical. In case 2, there is exactly one cross over point, whereas in cases 3 and 4, there are exactly two crossover points. An example of case 4 type overlap is shown in **Figure 4**.

To calculate the crossover points, by setting $a(x) = b(x)$, the two cross over points h_1 and h_2 yield as,

$$h_1 = \frac{c_1 + \frac{\sigma_1}{\sigma_2} c_2}{1 + \frac{\sigma_1}{\sigma_2}} \tag{5}$$

$$h_2 = \frac{c_1 - \frac{\sigma_1}{\sigma_2} c_2}{1 - \frac{\sigma_1}{\sigma_2}} \tag{6}$$

These values of h_1 and h_2 are used to calculate the mutual subsethood $\varepsilon(A,B)$ based on $C(A \cap B)$, as defined in (4).

Symbolically, for a signal $s_i = S(x_i) = (c_i, \sigma_i)$ and fuzzy weight $w_{ij} = (c_{ij}, \sigma_{ij})$, the mutual subsethood is

$$\varepsilon_{ij} = \varepsilon(s_i, w_{ij}) = \frac{C(s_i \cap w_{ij})}{C(s_i) + C(w_{ij}) - C(s_i \cap w_{ij})} \tag{7}$$

As shown in **Figure 5**, in the subsethood based FNN model, a fuzzy input signal is transmitted along a fuzzy weight that represents an antecedent connection. The transmitted signal is quantified ε_{ij}, which denotes the mutual subsethood between the fuzzy signal $S(x_i)$ and fuzzy weight (c_{ij}, σ_{ij}) and can be computed using (4).

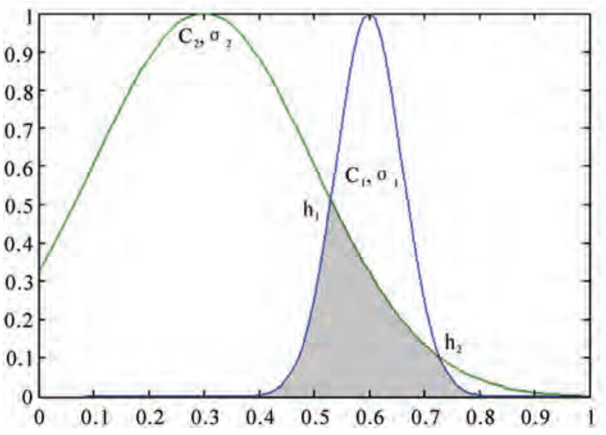


Figure 4. Example of overlapping: $c_1 > c_2$ and $\sigma_1 < \sigma_2$

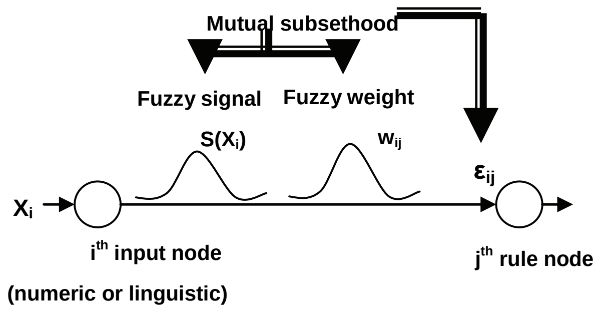


Figure 5. Fuzzy signal transmission

The expression for cardinality can be evaluated for each of the four cases in terms of standard error function *erf*(*x*) represented as (8).

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \qquad (8)$$

The case wise expressions for $C(s_i \cap w_{ij})$ are given in Appendix (1).

2.3 Activity Aggregation at Rule Nodes (Product Operator)

The net activation z_j of the rule node j is a product of all mutual subsethoods known as the fuzzy inner product can be evaluated as

$$z_j = \prod_{i=1}^{n} \varepsilon_{ij} = \prod_{i=1}^{n} \varepsilon(S(x_i), w_{ij}) \qquad (9)$$

The inner product in (9) exhibits some properties: it is bounded between 0 and 1; monotonic increasing; continuous and symmetric. The signal function for the rule node is linear.

$$S(z_j) = z_j \qquad (10)$$

Numeric activation values are transmitted unchanged to consequent connections.

2.4 Output Layer Signal Computation (Volume Defuzzification)

The signal of each output node is determined using standard volume based centroid defuzzification [13]. The activation of the output node is y_k, and V_{jk}'s denote consequent set volumes, then the general expression of defuzzification is

$$y_k = \frac{\sum_{j=1}^{q} z_j c_{jk} V_{jk}}{\sum_{j=1}^{q} z_j V_{jk}} \qquad (11)$$

The volume V_{jk} is simply the area of consequent fuzzy sets which are represented by Gaussian membership function. From (11), the output can be evaluated as

$$y_k = \frac{\sum_{j=1}^{q} z_j c_{jk} \sigma_{jk}}{\sum_{j=1}^{q} z_j \sigma_{jk}} \qquad (12)$$

The signal of output node k is $S(y_k) = y_k$.

3. Supervised Learning (Gradient Descent Algorithm)

The subsethood based linguistic network is trained by supervised learning. This involves repeated presentation of a set of input patterns drawn from the training set. The output of the network is compared with the desired value to obtain the error, and network weights are changed on the basis of an error minimization criterion. Once the network is trained to the desired level of error, it is tested by presenting a new set of input patterns drawn from the testing set.

3.1 Update Equations for Free Parameters

Learning is incorporated into the subsethood-linguistic model using the gradient descent method. A squared error criterion is used as a training performance parameter. The squared error e^t at iteration t is computed in the standard way

$$e^t = \frac{1}{2} \sum_{k=1}^{p} (d_k^t - S(y_k^t))^2 \qquad (13)$$

where d_k^t is the desired value at output node k, and the error evaluated over all p outputs for a specific pattern k. Both the centers and spreads $c_{ij}, c_{jk}, \sigma_{ij}$ and σ_{jk} of antecedents and consequent connections are modified on the basis of update equations given as follows:

$$c_{ij}^{t+1} = c_{ij}^t - \eta \frac{\partial e^t}{\partial c_{ij}^t} + \alpha \Delta c_{ij}^{t-1} \qquad (14)$$

where η is the learning rate, α is the momentum parameter, and

$$\Delta c_{ij}^{t-1} = c_{ij}^t - c_{ij}^{t-1} \qquad (15)$$

3.2 Partial Derivative Evaluation

The expressions of partial derivatives required in these update equations are derived as follows:

For the error derivative with respect to consequent centers

$$\frac{\partial e}{\partial c_{jk}} = \frac{\partial e}{\partial y_k} \frac{\partial y_k}{\partial c_{jk}} = -(d_k - y_k) \frac{z_j \sigma_{jk}}{\sum_{j=1}^{q} z_j \sigma_{jk}} \qquad (16)$$

and the error derivative with respect to the consequent spreads

$$\frac{\partial e}{\partial \sigma_{jk}} = \frac{\partial e}{\partial y_k} \frac{\partial y_k}{\partial \sigma_{jk}} = -(d_k - y_k) \frac{z_j c_{jk} \sum_{j=1}^{q} z_j \sigma_{jk} - z_j \sum_{j=1}^{q} z_j c_{jk} \sigma_{jk}}{(\sum_{j=1}^{q} z_j \sigma_{jk})^2} \qquad (17)$$

The error derivatives with respect to antecedent centers and spreads involve subsethood derivatives in the chain and are somewhat more involved to evaluate. Specifically, the error derivative chains with respect to antecedent centers and spreads are given as following,

$$\frac{\partial e}{\partial c_{ij}} = \sum_{k=1}^{p} \frac{\partial e}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial \varepsilon_{ij}} \frac{\partial \varepsilon_{ij}}{\partial c_{ij}} = \sum_{k=1}^{p} -(d_k - y_k) \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial \varepsilon_{ij}} \frac{\partial \varepsilon_{ij}}{\partial c_{ij}} \tag{18}$$

$$\frac{\partial e}{\partial \sigma_{ij}} = \sum_{k=1}^{p} \frac{\partial e}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial \varepsilon_{ij}} \frac{\partial \varepsilon_{ij}}{\partial \sigma_{ij}} = \sum_{k=1}^{p} -(d_k - y_k) \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial \varepsilon_{ij}} \frac{\partial \varepsilon_{ij}}{\partial \sigma_{ij}} \tag{19}$$

and the error derivative chains with respect to input feature spread is evaluated as

$$\frac{\partial e}{\partial \sigma_j} = \sum_{j=1}^{q} \sum_{k=1}^{p} \frac{\partial e}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial \varepsilon_{ij}} \frac{\partial \varepsilon_{ij}}{\partial \sigma_i} = \sum_{j=1}^{q} \sum_{k=1}^{p} -(d_k - y_k) \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial \varepsilon_{ij}} \frac{\partial \varepsilon_{ij}}{\partial \sigma_i} \tag{20}$$

where

$$\frac{\partial y_k}{\partial z_j} = \frac{\sigma_{jk} c_{jk} \sum_{j=1}^{q} z_j \sigma_{jk} - \sigma_{jk} \sum_{j=1}^{q} z_j c_{jk} \sigma_{jk}}{\left(\sum_{j=1}^{q} z_j \sigma_{jk}\right)^2} = \sigma_{jk} \left[\frac{c_{jk} \sum_{j=1}^{q} z_j \sigma_{jk} - \sum_{j=1}^{q} z_j c_{jk} \sigma_{jk}}{\left(\sum_{j=1}^{q} z_j \sigma_{jk}\right)^2} \right] = \frac{\sigma_{jk} (c_{jk} - y_k)}{\sum_{j=1}^{q} z_j \sigma_{jk}} \tag{21}$$

and

$$\frac{\partial z_j}{\partial \varepsilon_{ij}} = \prod_{i=1,\, i \neq j}^{n} \varepsilon_{ij} \tag{22}$$

The expressions for antecedent connection, mutual subsethood partial derivatives $\frac{\partial \varepsilon_{ij}}{\partial c_{ij}}$ and $\frac{\partial \varepsilon_{ij}}{\partial \sigma_{ij}}$ are obtained by differentiating (7) with respect to c_{ij}, σ_{ij} and σ_i as in (23), (24) and (25).

$$\frac{\partial \varepsilon_{ij}}{\partial c_{ij}} = \left[\frac{\left(\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} \left(\sqrt{\pi}(\sigma_i + \sigma_{ij}) - C(s_i \cap w_{ij}) \right) \right) - \left(\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} C(s_i \cap w_{ij}) \right)}{\left(\sqrt{\pi}(\sigma_i + \sigma_{ij}) - C(s_i \cap w_{ij}) \right)^2} \right] \tag{23}$$

$$\frac{\partial \varepsilon_{ij}}{\partial \sigma_{ij}} = \left[\frac{\left(\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_{ij}} (\sqrt{\pi}(\sigma_{ij} + \sigma_i) - C(s_i \cap w_{ij})) \right) - \left(\left(\sqrt{\pi} - \frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_{ij}} \right) C(s_i \cap w_{ij}) \right)}{(\sqrt{\pi}(\sigma_{ij} + \sigma_i) - C(s_i \cap w_{ij}))^2} \right] \tag{24}$$

$$\frac{\partial \varepsilon_{ij}}{\partial \sigma_i} = \left[\frac{\left(\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_i} (\sqrt{\pi}(\sigma_{ij} + \sigma_i) - C(s_i \cap w_{ij})) \right) - \left(\left(\sqrt{\pi} - \frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_i} \right) C(s_i \cap w_{ij}) \right)}{(\sqrt{\pi}(\sigma_{ij} + \sigma_i) - C(s_i \cap w_{ij}))^2} \right] \tag{25}$$

In these equations, the calculation of $\partial C(s_i \cap w_{ij}) / \partial c_{ij}$, $\partial C(s_i \cap w_{ij}) / \partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij}) / \partial \sigma_i$ are required which depends on the nature of overlap of the input feature fuzzy set and weight fuzzy set. The case wise expressions are demonstrated in Appendix (2).

4. Function Approximation

Function approximation involves determining or learning the input-output relations using numeric input-output data. Conventional methods like linear regression are useful in cases where the relation being learnt, is linear or quasi-linear. For nonlinear function approximation multilayer neural networks are well suited to solve the problem but with the drawback of their black box nature and heuristic decisions regarding network structure and tunable parameters. Interpretability of learnt knowledge in conventional neural networks is a severe problem. On the other hand, function approximation by fuzzy systems employs the concept of dividing the input space into sub regions, and for each sub region a fuzzy rule is defined thus making the system interpretable.

The performance of the fuzzy system depends on how finally the sub regions are generated. The practical imitation arises with fuzzy systems when the input variables are increased and the number of fuzzy rules explodes leading to the problem known as the curse of dimensionality. It is now well known that both fuzzy system and neural network are universal function approximators and can approximate functions to any arbitrary degree of accuracy [13,14]. Fuzzy neural system also has capability of approximating any continuous function or modeling a system [15-17].

There are two broad applications of function approximation-prediction and interpretation. In this paper, the work has been done on applications of function approximation related to prediction. In prediction, it is expected that, in future, new observations will be encountered for which only the input values are known, and the goal is to predict a likely output value for each case. The function estimate obtained from the training data through the learning algorithm is used for this purpose.

The subsethood based linguistic network proposed in the present paper has been tested on two different approximation problem: *sine-cosine* function approximation and *Narazaki-Ralescu* function [18].

4.1 Sine-Cosine Function

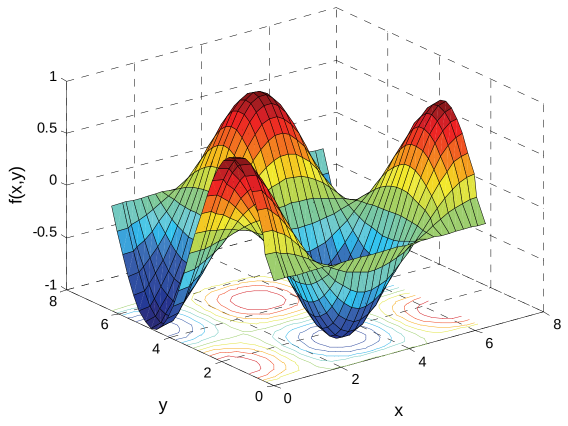
The learning capabilities of the proposed model can be demonstrated by approximating the *sine-cosine* function given by

$$f(x, y) = \sin(x)\cos(y) \tag{26}$$

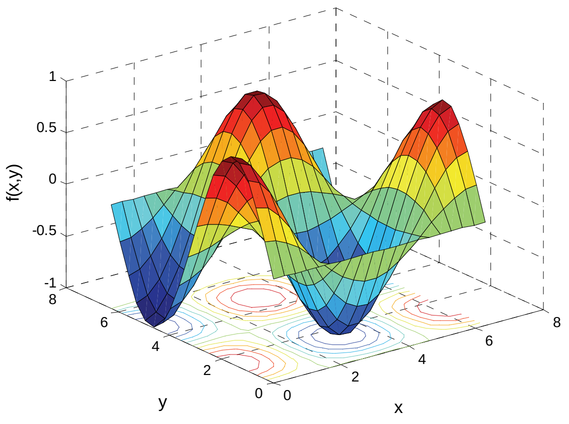
for the purpose of training the network the above function was described by 900 sample points, evenly distributed in a 30 × 30 grid in the input cross-space [0, 2π] × [0, 2π]. The model is tested by another set of 400 points evenly distributed in a 20 × 20 grid in the input cross-space [0, 2π] × [0, 2π]. The mesh plots of training and testing patterns are shown in **Figure 6**. For training of the model, the centers of fuzzy weights between the input layer and rule layer are initially randomized in the range [0, 2π] while the centers of fuzzy weights between rule layer and output layer are initially randomized in the range [–1, 1]. The spreads of all the fuzzy weights and the spreads of input feature fuzzifiers are initialized randomly in range [0.2, 0.9].

The number of free parameters that subsethood based FNN employs is straightforward to calculate: one spread for each numeric input; a center and a spread for each antecedent and consequent connection of a rule. For this function model employs a 2-*r*-1 network architecture, where *r* is the number of rule nodes. Therefore, since each rule has two antecedents and one consequent, an *r*-rule FNN system will have 6*r* + 2 free parameters.

Model was trained for different number of rules— 5, 10, 15, 20, 30 and 50. Simulations were performed with different learning schedules given in **Table 1** to study the effect of learning parameters on the performance of model.



(a)



(b)

Figure 6. (a) Mesh plot and counters of 900 training patterns; (b) mesh plot and counters of 400 testing

Table 1. Details of different learning schedules used for simulation studies

Learning Schedule	Details
LS = 0.2	η and α are fixed to 0.2
LS = 0.1	η and α are fixed to 0.2
LS = 0.01	η and α are fixed to 0.2
LS = 0.001	η and α are fixed to 0.2

(η-learning rate and α-momentum)

The root mean square error, evaluated for both training and testing patterns, is given as

$$RMSE_{trn} = \sqrt{\frac{\sum_{training\ \ patterns}(desired - actual)^2}{number\ of\ training\ patterns}} \tag{27}$$

$$RMSE_{test} = \sqrt{\frac{\sum_{testing\ \ patterns}(desired - actual)^2}{number\ of\ testing\ patterns}} \tag{28}$$

In order to visualize the surface obtained from the test set after training the function *f*(*x*, *y*) = sin(*x*) cos(*y*) for 250 epochs the three dimensional plots of the function is generated. **Figure 7** illustrates surface plots of the function and the error surface for different values of rule counts with learning schedule as LS = 0.01. It can be observed that a model of mere 5 rules seems to be coarsely approximating the given function. The error is more where the slope of the function changes in that region. Thus, increasing the number of rules generates better approximated surface as can be observed as shown in **Figure 7**. As per the observation shown in **Table 2**, we can conclude that for learning schedule LS = 0.2 or higher and with small rule count the subsethood model is unable to train, resulting in oscillations in error trajectories shown as **Figure 8**. This may occur due to the improper selection of learning parameters (learning rate (η) and momentum (α)) and number of rules. But with same learning parameters and higher rule counts like 30 and 50 rules model produces good approximation.

The observations for fuzzy neuro model drawn in the above experiments can be summarized as the following:

1) As the number of rules increases the approximation performance of model improves to a certain limit.

2) For higher learning rates and momentum with lower rule counts the model is unable to learn. In contrast if the learning rate and momentum are kept to small values a smooth decaying trajectory is obtained even for small rule counts.

3) In general, Model works fairly well even for simple learning schemes by keeping the learning rate and momentum fixed to small values.

4) Most of the learning is achieved in a small number of epochs.

4.2 Narazaki and Ralescu Function

The function is expressed as follows,

$$y(x) = 0.2 + 0.8(x + 0.7\sin(2\pi x)), \quad 0 \le x \le 1 \qquad (29)$$

and the plot of the function is shown in **Figure 9**.

The system architecture used for approximating single input-output function is 1-*r*-1, where *r* is the number of rule nodes. The tunable parameters that model employs for this application is calculated to be as, one spread for one input, and a center and a spread for each antecedent and consequent connection of rule. As each rule has one antecedent and one consequent, *r* rule architecture will have 4*r* + 1 free parameters. The model is trained using 21 training patterns. These patterns were generated at intervals of 0.05 in range [0, 1]. Thus, the training patterns are of the form:

$$(0, y(0)), (0.05, y(0.05)), \ldots, (1, y(1)) \qquad (30)$$

The evaluation was done using 101 test data taken at intervals of 0.01. The training and test sets generated are

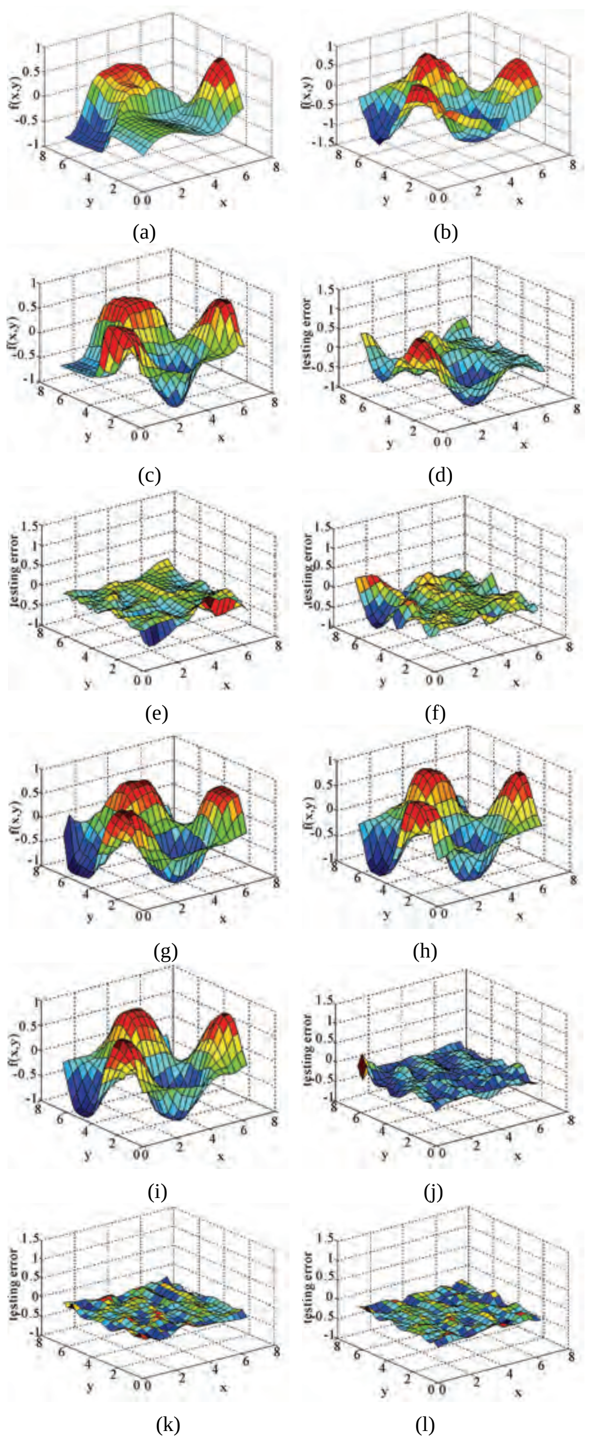


Figure 7. ***f*(*x*, *y*) surface plot and their corresponding testing error surface after 250 epochs for different rule counts with learning schedule as LS = 0.01, (a) f(x, y) surface for 5 rules; (b) f(x, y) surface for 10 rules; (c) f(x, y) surface for 15 rules; (d) error surface for 5 rules; (e) error surface for 10 rules; (f) error surface for 15 rules; (g) f(x, y) surface for 20 rules; (h) f(x, y) surface for 30 rules; (i) f(x, y) surface for 50 rules; (j) error surface for 20 rules; (k) error surface for 30 rules; (l) error surface for 50 rules**

Table 2. Root mean square errors for different rule count and learning schedules for 250 epochs

Rules	LS = 0.2		LS = 0.1	
	$RMSE_{trn}$	$RMSE_{test}$	$RMSE_{trn}$	$RMSE_{test}$
5	0.4306	0.5928	0.3464	0.6210
10	0.1851	0.3144	0.2745	0.3239
15	0.0897	0.1125	0.1250	0.1746
20	0.0631	0.1518	0.0811	0.1026
30	0.0418	0.0522	0.0518	0.0615
50	0.0316	0.0323	0.0219	0.0452
Rules	**LS = 0.01**		**LS = 0.001**	
	$RMSE_{trn}$	$RMSE_{test}$	$RMSE_{trn}$	$RMSE_{test}$
5	0.3352	0.4080	0.3428	0.3567
10	0.1758	0.1997	0.2194	0.2783
15	0.1419	0.1516	0.2771	0.2954
20	0.0972	0.1247	0.1446	0.1432
30	0.0645	0.0735	0.1135	0.1246
50	0.0336	0.0354	0.0336	0.0354

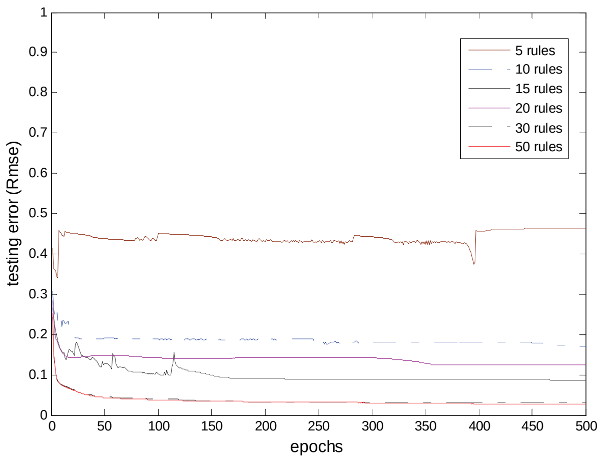
mutually exclusive. Two performance indices *J*1 and *J*2 as defined in [18], used for evaluation are given below:

$$J1 = 100 \times \frac{1}{21} \sum_{training\ data} \frac{|actual\ output - desired\ output|}{desired\ output} \qquad (31)$$

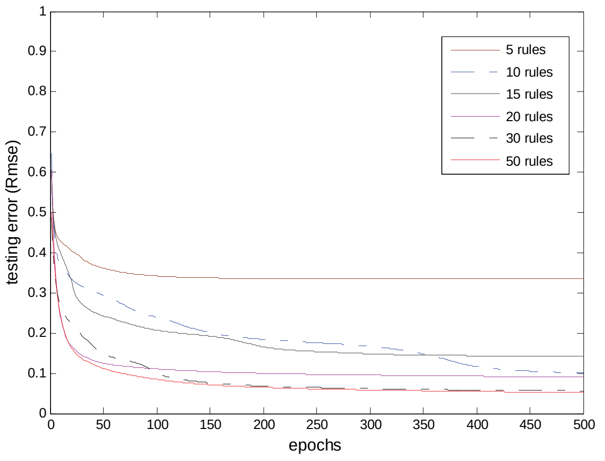
$$J2 = 100 \times \frac{1}{101} \sum_{test\ data} \frac{|actual\ output - desired\ output|}{desired\ output} \qquad (32)$$

Experiments were conducted for different rule counts, using a learning rate of 0.01 and momentum of 0.01 throughout the learning procedure. **Table 3** summarizes the performance of model in terms of indices *J*1 and *J*2 for rule counts 3 to 6. It is evident from the performance measures that for 5 or 6 rules the approximation accuracy is much better than that for 3 or 4 rules. In general up to a certain limit, as the number of rules grows, the performance of model improves.

Table 4 compares the test accuracy performance index *J*2 for different models along with the number of rules and tunable parameters used to achieve it. With five rules our model obtained *J*1 = 0.9467 and *J*2 = 0.7403 as better than all other schemes. From the above results, it can be infer that subsethood-based FNN shows the ability to approximate function with good accuracy in comparison with other models.



(a)



(b)

Figure 8. Error trajectories for different rules and learning schedule (a) LS = 0.2, (b) LS = 0.01

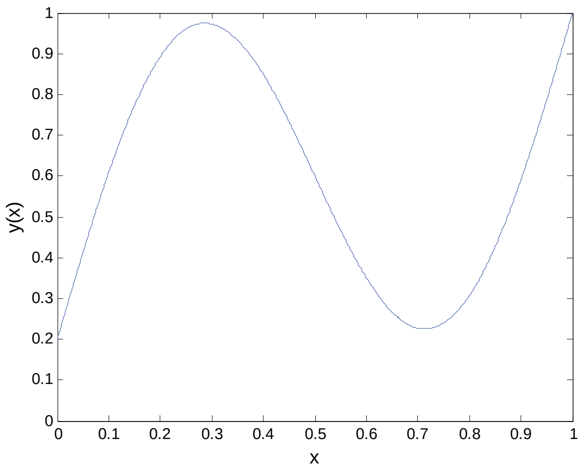


Figure 9. Narazaki-Ralescu function

Table 3. Subsethood based FNN performance for Narazaki-Ralescu's function

Number of Rules	Trainable Parameter	Training Accuracy (J1%)	Testing Accuracy (J2%)
3	13	2.57	1.7015
4	17	1.022	0.7350
5	21	0.9468	0.7403
6	25	0.6703	0.6595

Table 4. Performance comparison of subsethood based FNN with other methods for Narazaki-Ralescu's function

Methods and reference	Number of rules	Trainable Parameters	Testing Accuracy (J2%)
FuGeNeSys [19]	5	15	0.856
Lin and Cunningham III [20]	4	16	0.987
Narazaki and Ralescu [18]	Na	12	3.19
Subsethood based FNN	3	13	1.7015
Subsethood based FNN	5	21	0.7403

5. Conclusions

In this paper the proposed subsethood based Fuzzy Neural Network model has now proved to be a universal function approximator. The model is tested on two different applications and found suitable for any function approximation problem empirically. The applications of function approximation are diverse and include the domain of physics, economics, control, planning, forecasting, machine learning and image compression. In future work authors shall incorporate concepts of fuzzy neural function approximator in image compression.

REFERENCES

[1] C. T. Lin and C. S. G. Lee, "Neural-Network-Based Fuzzy Logic Control and Decision System," *IEEE Transactions on Computers*, Vol. 40, No. 12, 1991, pp. 1320-1336.

[2] J. M. Keller, R. R. Yager and H. Tahani, "Neural Network Implementation of Fuzzy Logic," *Fuzzy Sets and Systems*, Vol. 45, No. 5, 1992, pp. 1-12.

[3] S. Horikawa, T. Furuhashi and Y. Uchikawa, "On Fuzzy Modeling Using Fuzzy Neural Networks with the Back Propagation Algorithm," *IEEE transactions on Neural Networks*, Vol. 3, No. 5, 1992, pp. 801-806.

[4] D. Nauck and R. Kruse, "A Neuro-Fuzzy Method to Learn Fuzzy Classification Rules from Data," *Fuzzy Sets and Systems*, Vol. 89, No. 3, 1997, pp. 277-288.

[5] J. S. R. Jang, "ANFIS: Adaptive-Network-Based Fuzzy Inference System," *IEEE Transactions on Systerms*, Vol. 23, 1993, pp. 665-685.

[6] S. Mitra and S. K. Pal, "Fuzzy Multilayer Perceptron, Inferencing and Rule Generation," *IEEE Transactions on Neural Networks*, Vol. 6, No. 1, 1995, pp. 51-63.

[7] W. L. Tung and C. Quek, "A Mamdani-Takagi-Sugeno Based Linguistic Neural-Fuzzy Inference System for Improved Interpretability-Accuracy Representation," *IEEE International Conference on Fuzzy Systems*, Jeju Island, August 2009, pp. 367-372.

[8] W. L. Tung and C. Quek, "eFSM-A Novel Online Neural-Fuzzy Semantic Memory model," *IEEE Transactions on Neural Networks*, Vol. 21, No. 1, 2010, pp. 136-157.

[9] P. K. Simpson, "Fuzzy Min-Max Neural Networks-Part 1: Classification," *IEEE Transactions on Neural Networks*, Vol. 3, No. 5, 1992, pp. 776-786.

[10] P. K. Simpson, "Fuzzy Min-Max Neural Networks-Part 2: Clustering," *IEEE Transaction on Fuzzy Systems*, Vol. 1, No. 1, 1992, pp. 32-45.

[11] R.-J. Wai and Z.-W. Yang, "Adaptive Fuzzy Neural Network Control Design via a T-S Fuzzy Model for a Robot Manipulator Including Actuator Dynamics," *IEEE Transactions on System*, *Man and Cybernetics-Part B*, Vol. 38, No. 5, 2008, pp. 1326-1346.

[12] P. Sandeep and S. Kumar, "Subsethood Based Adaptive Linguistic Networks for Pattern Classification," *IEEE Transaction on System, man and cybernetics-part C*: *application and reviews*, Vol. 33, No. 2, 2003, pp. 248-258.

[13] B. Kosko, "Fuzzy Engineering," Prentice-Hall, Englewood Cliffs, New Jersey, 1997.

[14] K. Hornik, "Approximation Capabilities of Multilayer Feed forward Networks are Universal Approximators," *IEEE Transaction on Neural Networks*, Vol. 2, No. 5, 1989, pp. 359-366.

[15] B. Kosko, "Fuzzy Systems as Universal Approximators," *IEEE Transactions on computers*, Vol. 43, No. 11, 1994, pp. 1329-1333.

[16] C. T. Lin and Y. C. Lu, "A Neural Fuzzy System with Linguistic Teaching Signals," *IEEE Transactions Fuzzy Systems*, Vol. 3, No. 2, 1995, pp. 169-189.

[17] L. X. Wang and J. M. Mendel, "Generating Fuzzy Rules from Numerical Data, with Application," Technical Report 169, USC SIPI, University of Southern California, Los Angeles, January 1991.

[18] H. Narazaki and A. L. Ralescu, "An Improved Synthesis Method for Multilayered Neural Networks Using Qualitative Knowledge's," *IEEE Transactions Fuzzy Systems*, Vol. 1, No. 2, 1993, pp. 125-137.

[19] M. Russo, "FuGeNeSys-a Fuzzy Genetic Neural System for Fuzzy Modeling," *IEEE Transactions Fuzzy Systems*, Vol. 6, No. 3, 1993, pp. 373-388.

[20] Y. Lin and G. A. Cunningham, "A New Approach to Fuzzy-Neural System Modeling," *IEEE Transactions Fuzzy Systems*, Vol. 3, No. 2, 1995, pp. 190-198.

Appendix

1. Expressions for $C(s_i \cap w_{ij})$

The expression for cardinality can be evaluated in terms of the standard error function *erf* (*x*) given in (8). The case wise expressions for $C(s_i \cap w_{ij})$ for all four possibilities identified in Section (2.2) are as follows.

Case 1—$C_i = C_{ij}$: If $\sigma_i < \sigma_{ij}$, the signal fuzzy set s_i completely belongs to the weight fuzzy set w_{ij}, and the cardinality $C(s_i \cap w_{ij}) = C(s_i)$

$$\begin{aligned} C(s_i \cap w_{ij}) = C(s_i) &= \int_{-\infty}^{\infty} e^{-((x-c_i)/\sigma_i)^2} dx \\ &= \sigma_i \frac{\sqrt{\pi}}{2}[erf(\infty) - erf(-\infty)] \\ &= \sigma_i \sqrt{\pi}. \end{aligned} \tag{33}$$

Similarly, $C(s_i \cap w_{ij}) = C(w_{ij})$ if $\sigma_i > \sigma_{ij}$ and $C(s_i \cap w_{ij}) = \sigma_{ij}\sqrt{\pi}$. If $\sigma_i = \sigma_{ij}$, the two fuzzy sets are identical. Summarizing these three sub cases, the values of cardinality can be shown as (34).

$$C(s_i \cap w_{ij}) = \begin{cases} C(s_i) = \sigma_i \sqrt{\pi}, & if\ \sigma_i < \sigma_{ij} \\ C(w_{ij}) = \sigma_{ij}\sqrt{\pi}, & if\ \sigma_i > \sigma_{ij} \\ C(s_i) = C(w_{ij}) = \sigma_i\sqrt{\pi} = \sigma_{ij}\sqrt{\pi}, & if\ \sigma_i < \sigma_{ij.} \end{cases} \tag{34}$$

Case 2—$C_i \neq C_{ij}$, $\sigma_i = \sigma_{ij}$: In this case there will be exactly one cross over point h_1. Assuming $c_{ij} > c_i$, the cardinality $C(s_i \cap w_{ij})$ can be evaluated as

$$\begin{aligned} C(s_i \cap w_{ij}) &= \int_{-\infty}^{h_1} e^{-((x-c_{ij})/\sigma_{ij})^2} dx + \int_{h_1}^{\infty} e^{-((x-c_i)/\sigma_i)^2} dx \\ &= \sigma_i \frac{\sqrt{\pi}}{2}\left[1 + erf\left(\frac{(h_1 - c_{ij})}{\sigma_{ij}}\right)\right] \\ &+ \sigma_i \frac{\sqrt{\pi}}{2}\left[1 - erf\left(\frac{(h_1 - c_i)}{\sigma_i}\right)\right] \end{aligned} \tag{35}$$

If $c_{ij} < c_i$, the expression for cardinality $C(s_i \cap w_{ij})$ is

$$\begin{aligned} C(s_i \cap w_{ij}) &= \int_{-\infty}^{h_1} e^{-((x-c_i)/\sigma_i)^2} dx + \int_{h_1}^{\infty} e^{-((x-c_{ij})/\sigma_{ij})^2} dx \\ &= \sigma_i \frac{\sqrt{\pi}}{2}\left[1 + erf\left(\frac{(h_1 - c_i)}{\sigma_i}\right)\right] \\ &+ \sigma_i \frac{\sqrt{\pi}}{2}\left[1 - erf\left(\frac{(h_1 - c_{ij})}{\sigma_{ij}}\right)\right] \end{aligned} \tag{36}$$

Case 3—$C_i \neq C_{ij}$, $\sigma_i < \sigma_{ij}$: In this case, there will be two crossover points h_1 and h_2, as calculated in (5) and (6). Assuming $h_1 < h_2$ and $c_{ij} > c_i$, the cardinality $C(s_i \cap w_{ij})$ can be evaluated as

$$\begin{aligned} C(s_i \cap w_{ij}) &= \int_{-\infty}^{h_1} e^{-((x-c_i)/\sigma_i)^2} dx \\ &+ \int_{h_1}^{h_2} e^{-((x-c_{ij})/\sigma_{ij})^2} dx \\ &+ \int_{h_2}^{\infty} e^{-((x-c_i)/\sigma_i)^2} dx \\ &= \sigma_i \frac{\sqrt{\pi}}{2}\left[1 + erf\left(\frac{(h_1 - c_i)}{\sigma_i}\right)\right] \\ &+ \sigma_i \frac{\sqrt{\pi}}{2}\left[1 - erf\left(\frac{(h_1 - c_i)}{\sigma_i}\right)\right] \\ &+ \sigma_{ij} \frac{\sqrt{\pi}}{2}\left[erf\left(\frac{(h_2 - c_{ij})}{\sigma_{ij}}\right)\right. \\ &\left. - erf\left(\frac{(h_1 - c_{ij})}{\sigma_{ij}}\right)\right] \end{aligned} \tag{37}$$

if $c_{ij} < c_i$, the expression for $C(s_i \cap w_{ij})$ is identical to (37).

Case 4—$C_i \neq C_{ij}$, $\sigma_i > \sigma_{ij}$: This case is similar to case 3 and once again there will be two cross over points h_1 and h_2, as calculated in (5) and (6). Assuming $h_1 < h_2$, and $c_{ij} > c_i$, the cardinality $C(s_i \cap w_{ij})$ can be evaluated as

$$\begin{aligned} C(s_i \cap w_{ij}) &= \int_{-\infty}^{h_1} e^{-((x-c_{ji})/\sigma_{ji})^2} dx \\ &+ \int_{h_1}^{h_2} e^{-((x-c_i)/\sigma_i)^2} dx \\ &+ \int_{h_2}^{\infty} e^{-((x-c_{ij})/\sigma_i)^2} dx \\ &= \sigma_{ij} \frac{\sqrt{\pi}}{2}\left[1 + erf\left(\frac{(h_1 - c_{ij})}{\sigma_{ij}}\right)\right] \\ &+ \sigma_{ij} \frac{\sqrt{\pi}}{2}\left[1 - erf\left(\frac{(h_1 - c_{ij})}{\sigma_{ij}}\right)\right] \end{aligned}$$

$$+\sigma_i \frac{\sqrt{\pi}}{2}\left[erf\left(\frac{(h_2-c_i)}{\sigma_i}\right) - erf\left(\frac{(h_1-c_i)}{\sigma_i}\right)\right] \quad (38)$$

if $c_{ij} < c_i$, the expression for $C(s_i \cap w_{ij})$ is identical to (38).

Corresponding expressions for $\varepsilon(s_i \cap w_{ij})$ are obtained by substituting the values of $C(s_i \cap w_{ij})$ from (34)-(38) to (7).

2. Expressions for $\partial C(s_i \cap w_{ij})/\partial c_{ij}$, $\partial C(s_i \cap w_{ij})/\partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij})/\partial \sigma_i$

As per the discussion in the Section (3.2) that, the calculation of $\partial C(s_i \cap w_{ij})/\partial c_{ij}$, $\partial C(s_i \cap w_{ij})/\partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij})/\partial \sigma_i$ is required in (23), (24) and (25), which depends on the nature of overlap. Therefore the case wise expressions are given as following:

Case 1 — $C_i = C_{ij}$: As is evident from (34), $C(s_i \cap w_{ij})$ is independent from c_{ij}, and therefore,

$$\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} = 0 \quad (39)$$

Similarly the first derivative of (34) with respect to σ_{ij} and σ_i is shown as (40) and (41).

$$\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_{ij}} = \begin{cases} \sqrt{\pi}, & if\ c_{ij} = c_i\ and\ \sigma_{ij} \le \sigma_i \\ 0, & if\ c_{ij} = c_i\ and\ \sigma_{ij} > \sigma_i \end{cases} \quad (40)$$

$$\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_i} = \begin{cases} 0, & if\ c_{ij} = c_i\ and\ \sigma_{ij} \le \sigma_i \\ \sqrt{\pi}, & if\ c_{ij} = c_i\ and\ \sigma_{ij} > \sigma_i \end{cases} \quad (41)$$

Case 2—$C_i = C_{ij}$, $\sigma_i = \sigma_{ij}$: when $c_{ij} > c_i$, the values of $\partial C(s_i \cap w_{ij})/\partial c_{ij}$, $\partial C(s_i \cap w_{ij})/\partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij})/\partial \sigma_i$ are derived by differentiating (35) as follows :

$$\begin{aligned}\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} &= \int_{-\infty}^{h_1} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx + \int_{h_1}^{\infty} \frac{\partial}{\partial c_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx \\ &= \int_{-\infty}^{h_1} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx \\ &= -e^{-((h_1-c_{ij})/\sigma_{ij})^2}\end{aligned} \quad (42)$$

$$\begin{aligned}\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_{ij}} &= \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx + \int_{h_1}^{\infty} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx \\ &= -\frac{h_1-c_{ij}}{\sigma_{ij}} e^{-((h_1-c_{ij})/\sigma_{ij})^2} + \frac{\sqrt{\pi}}{2}\left[erf\left(\frac{h_1-c_{ij}}{\sigma_{ij}}\right)+1\right]\end{aligned} \quad (43)$$

$$\begin{aligned}\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_i} &= \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_i} e^{-((x-c_{ij})/\sigma_{ij})^2} dx + \int_{h_1}^{\infty} \frac{\partial}{\partial \sigma_i} e^{-((x-c_i)/\sigma_i)^2} dx \\ &= \frac{h_1-c_i}{\sigma_i} e^{-((h_1-c_i)/\sigma_i)^2} - \frac{\sqrt{\pi}}{2}\left[erf\left(\frac{h_1-c_i}{\sigma_i}\right)-1\right]\end{aligned} \quad (44)$$

when $c_{ij} < c_i$, the values of $\partial C(s_i \cap w_{ij})/\partial c_{ij}$, $\partial C(s_i \cap w_{ij})/\partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij})/\partial \sigma_i$ are derived by differentiating (36) as follows :

$$\begin{aligned}\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} &= \int_{-\infty}^{h_1} \frac{\partial}{\partial c_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx + \int_{h_1}^{\infty} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx \\ &= \int_{h_1}^{\infty} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx \\ &= e^{-((h_1-c_{ij})/\sigma_{ij})^2}\end{aligned} \quad (45)$$

$$\begin{aligned}\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_{ij}} &= \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx + \int_{h_1}^{\infty} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx \\ &= -\frac{h_1-c_{ij}}{\sigma_{ij}} e^{-((h_1-c_{ij})/\sigma_{ij})^2} - \frac{\sqrt{\pi}}{2}\left[erf\left(\frac{h_1-c_{ij}}{\sigma_{ij}}\right)-1\right]\end{aligned} \quad (46)$$

$$\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_i} = \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_i} e^{-((x-c_i)/\sigma_i)^2} dx + \int_{h_1}^{\infty} \frac{\partial}{\partial \sigma_i} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$

$$= -\frac{h_1 - c_i}{\sigma_i} e^{-((h_1 - c_i)/\sigma_i)^2}$$
$$+ \frac{\sqrt{\pi}}{2}\left[erf\left(\frac{h_1 - c_i}{\sigma_i}\right) + 1 \right] \tag{47}$$

Case 3— $C_i \neq C_{ij}$, $\sigma_i < \sigma_{ij}$: once again, two sub cases arise similar to those of Case 2. When $c_{ij} > c_i$, the values of $\partial C(s_i \cap w_{ij}) / \partial c_{ij}$, $\partial C(s_i \cap w_{ij}) / \partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij}) / \partial \sigma_i$ are derived by differentiating (36) as follows:

$$\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} = \int_{-\infty}^{h_1} \frac{\partial}{\partial c_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx$$
$$+ \int_{h_1}^{h_2} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$+ \int_{h_2}^{\infty} \frac{\partial}{\partial c_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx \tag{48}$$
$$= \int_{h_1}^{h_2} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$= -e^{-((h_2-c_{ij})/\sigma_{ij})^2}$$
$$-e^{-((h_1-c_{ij})/\sigma_{ij})^2}$$

$$\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_{ij}} = \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx$$
$$+ \int_{h_1}^{h_2} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$+ \int_{h_2}^{\infty} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx$$
$$= \int_{h_1}^{h_2} \frac{\partial}{\partial \sigma_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$= \frac{h_1 - c_{ij}}{\sigma_{ij}} e^{-((h_1-c_{ij})/\sigma_{ij})^2}$$
$$- \frac{h_2 - c_{ij}}{\sigma_{ij}} e^{-((h_2-c_{ij})/\sigma_{ij})^2}$$
$$+ \frac{\sqrt{\pi}}{2}\left[-erf\left(\frac{(h_1 - c_{ij})}{\sigma_{ij}}\right) \right.$$
$$\left. + erf\left(\frac{(h_2 - c_{ij})}{\sigma_{ij}}\right) \right] \tag{49}$$

$$\frac{\partial C(s_i \cap w_{ij})}{\partial \sigma_i} = \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_i} e^{-((x-c_i)/\sigma_i)^2} dx$$
$$+ \int_{h_1}^{h_2} \frac{\partial}{\partial \sigma_i} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$+ \int_{h_2}^{\infty} \frac{\partial}{\partial \sigma_i} e^{-((x-c_i)/\sigma_i)^2} dx$$
$$= \int_{-\infty}^{h_1} \frac{\partial}{\partial \sigma_i} e^{-((x-c_j)/\sigma_i)^2} dx$$
$$+ \int_{h_2}^{\infty} \frac{\partial}{\partial \sigma_i} e^{-((x-c_j)/\sigma_i)^2} dx$$
$$= \frac{h_1 - c_i}{\sigma_i} e^{-((h_1-c_i)/\sigma_i)^2}$$
$$+ \frac{h_2 - c_i}{\sigma_i} e^{-((h_2-c_i)/\sigma_i)^2}$$
$$+ \frac{\sqrt{\pi}}{2}\left[\left\{ erf\left(\frac{(h_1 - c_i)}{\sigma_i}\right) + 1 \right\} \right.$$
$$\left. - \left\{ erf\left(\frac{(h_2 - c_{ij})}{\sigma_{ij}}\right) - 1 \right\} \right]. \tag{50}$$

Similarly, if $c_{ij} < c_i$

$$\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} = \int_{-\infty}^{h_1} \frac{\partial}{\partial c_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx$$
$$+ \int_{h_1}^{h_2} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$+ \int_{h_2}^{\infty} \frac{\partial}{\partial c_{ij}} e^{-((x-c_i)/\sigma_i)^2} dx \tag{51}$$
$$= \int_{h_1}^{h_2} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$
$$= -e^{-((h_2-c_{ij})/\sigma_{ij})^2}$$
$$+e^{-((h_1-c_{ij})/\sigma_{ij})^2}$$

Thus for both the cases $(c_{ij} < c_i$ or $c_{ij} > c_i)$, identical expressions for $\partial C(s_i \cap w_{ij}) / \partial c_{ij}$ are obtained. Similarly, the expressions for $\partial C(s_i \cap w_{ij}) / \partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij}) / \partial \sigma_i$ also remain same as (49) and (50) respectively in both the conditions.

Case 4— $C_i \neq C_{ij}$, $\sigma_i > \sigma_{ij}$: When $c_{ij} > c_i$, the values of $\partial C(s_i \cap w_{ij}) / \partial c_{ij}$, $\partial C(s_i \cap w_{ij}) / \partial \sigma_{ij}$ and $\partial C(s_i \cap w_{ij}) / \partial \sigma_i$ are derived by differentiating (38) as :

$$\frac{\partial C(s_i \cap w_{ij})}{\partial c_{ij}} = \int_{-\infty}^{h_1} \frac{\partial}{\partial c_{ij}} e^{-((x-c_{ij})/\sigma_{ij})^2} dx$$

$$+\int_{h_1}^{h_2}\frac{\partial}{\partial c_{ij}}e^{-((x-c_j)/\sigma_i)^2}dx$$
$$=-e^{-((h_1-c_{ij})/\sigma_{ij})^2}$$
$$+e^{-((h_2-c_{ij})/\sigma_{ij})^2} \tag{52}$$

$$\frac{\partial C(s_i\cap w_{ij})}{\partial\sigma_{ij}}=\int_{-\infty}^{h_1}\frac{\partial}{\partial\sigma_{ij}}e^{-((x-c_{ij})/\sigma_{ij})^2}dx$$
$$+\int_{h_1}^{h_2}\frac{\partial}{\partial\sigma_{ij}}e^{-((x-c_i)/\sigma_i)^2}dx$$
$$+\int_{h_2}^{\infty}\frac{\partial}{\partial\sigma_{ij}}e^{-((x-c_{ij})/\sigma_{ij})^2}dx$$
$$=\frac{h_1-c_{ij}}{\sigma_{ij}}e^{-((h_1-c_{ij})/\sigma_{ij})^2} \tag{53}$$
$$-\frac{h_2-c_{ij}}{\sigma_{ij}}e^{-((h_1-c_{ij})/\sigma_{ij})^2}$$
$$+\frac{\sqrt{\pi}}{2}\left[2+erf\left(\frac{(h_1-c_{ij})}{\sigma_{ij}}\right)\right.$$
$$\left.-erf\left(\frac{(h_2-c_{ij})}{\sigma_{ij}}\right)\right]$$

$$\frac{\partial C(s_i\cap w_{ij})}{\partial\sigma_i}=\int_{-\infty}^{h_1}\frac{\partial}{\partial\sigma_i}e^{-((x-c_{ij})/\sigma_{ij})^2}dx$$
$$+\int_{h_1}^{h_2}\frac{\partial}{\partial\sigma_i}e^{-((x-c_i)/\sigma_i)^2}dx$$
$$+\int_{h_2}^{\infty}\frac{\partial}{\partial\sigma_i}e^{-((x-c_{ij})/\sigma_{ij})^2}dx$$
$$=\frac{h_1-c_i}{\sigma_i}e^{-((h_1-c_i)/\sigma_i)^2} \tag{54}$$
$$-\frac{h_2-c_i}{\sigma_i}e^{-((h_2-c_i)/\sigma_i)^2}$$
$$+\frac{\sqrt{\pi}}{2}\left[\left\{erf\left(\frac{(h_2-c_i)}{\sigma_i}\right)\right\}\right.$$
$$\left.-\left\{erf\left(\frac{(h_2-c_{ij})}{\sigma_{ij}}\right)\right\}\right]$$

If $c_{ij} < c_i$, the expressions for $\partial C(s_i\cap w_{ij})/\partial c_{ij}$, $\partial C(s_i\cap w_{ij})/\partial\sigma_{ij}$ and $\partial C(s_i\cap w_{ij})/\partial\sigma_i$ are again the same as (52), (53) and (54) respectively.

J. Intelligent Learning Systems & Applications, 2010, 2: 110-118
doi:10.4236/jilsa.2010.22014 Published Online May 2010 (http://www.SciRP.org/journal/jilsa)

Implementation of Adaptive Neuro Fuzzy Inference System in Speed Control of Induction Motor Drives

K. Naga Sujatha, K. Vaisakh

Department of Electrical Engineering, AU College of Engineering, Andhra University, Visakhapatnam, India.
Email: vaisakh_k@yahoo.co.in

Received December 18th, 2009; revised January 6th, 2010; accepted January 15th, 2010.

ABSTRACT

A new speed control approach based on the Adaptive Neuro-Fuzzy Inference System (*ANFIS*) *to a closed-loop, variable speed induction motor* (*IM*) *drive is proposed in this paper. ANFIS provides a nonlinear modeling of motor drive system and the motor speed can accurately track the reference signal. ANFIS has the advantages of employing expert knowledge from the fuzzy inference system and the learning capability of neural networks. The various functional blocks of the system which govern the system behavior for small variations about the operating point are derived, and the transient responses are presented. The proposed* (*ANFIS*) *controller is compared with PI controller by computer simulation through the MATLAB/SIMULINK software. The obtained results demonstrate the effectiveness of the proposed control scheme.*

Keywords: *ANFIS Controller, PI Controller, Fuzzy Logic Controller, Artificial Neural Network Controller, Induction Motor Drive*

1. Introduction

Over the last three decades, variable speed drives are the most complex of all power electronic systems. Drive technology has been a confluence of many professionals from other fields, such as electrical machines, control systems and traditional power engineering. To a traditional power electronics engineer with expertise in the design of, such as thyristor phase-controlled converters, switching mode power supplies, or uninterruptible power supply systems, the technology is incomprehensible because of its complexity and multidisciplinary characteristics.

Modern variable speed drive applications require steeples control and suitable dynamic response and accuracy. These considerations have been met to a large extent in the past decade by thyristor-controlled dc machines. However, the dc machine remains expensive in relation to the types of rotating machines. For the higher power drives in industries, the lighter, less expensive, reliable simple, more robust and commutator less induction motors are desirable and these motors are being applied today to a wider range of applications requiring variable speed. Unfortunately, accurate speed control of such machines by a simple and economical means remains a difficult task. With the development of the silicon-controlled rectifier, triac and related members of the thyristor family, it has become most feasible to design variable-speed induction motor drives for a wide variety of applications. Different techniques have been used, using SCR controllers. A back-to back connected SCR' are used in series with the rotor phases to control their effective impedance [1-4]. A chopper-controlled external resistance is used to control the speed by varying the duty cycle of the chopper. A controlled rectifier is used in the rotor circuit to feed the external resistance, and by varying the firing angle, the effective rotor impedance is controlled.

Generally, variable speed drives for Induction Motor (IM) require both wide operating range of speed and fast torque response, regardless of load variations. This leads to more advanced control methods to meet the real demand. Very recently, the artificial intelligence tools, such as expert system, fuzzy logic and neural network are showing impact on variable frequency drives.

They are applied to important fields such as variable speed drives, control systems, signal processing, and system modeling. Artificial Intelligent systems, means those systems that are capable of imitating the human reasoning process as well as handling *quantitative* and *qualitative* knowledge. It is well known that the intelligent systems, which can provide human like expertise such as domain knowledge, uncertain reasoning, and adaptation to a noisy and time-varying environment, are important in tackling practical computing problems. ANFIS has gain a lot of interest over the last few years as a powerful technique to solve many real world problems. Compared to conventional techniques, they own the capability of solving problems that do not have algorithmic solution. Neural networks and fuzzy logic technique are quite different, and yet with unique capabilities useful in information processing by specifying mathematical relationships among numerous variables in a complex system, performing mappings with degree of imprecision, control of nonlinear system to a degree not possible with conventional linear systems [5-11]. To overcome the drawbacks of Neural networks and fuzzy logic, Adaptive Neuro-Fuzzy Inference System (ANFIS) was proposed in this paper. The ANFIS is, from the topology point of view, an implementation of a representative fuzzy inference system using a Back Propagation neural network structure.

The purpose of this paper is to present a general method for estimating both the nature of the dynamic response and the values of the significant parameters and operating constraints of typical induction machines controlled by SCR controllers [12,13]. The dynamic behavior of a closed-loop speed-control system with delta-connected SCR's in the rotor is discussed. The various functional blocks of the feedback system which governs the system behavior for small variations about the operating point are derived, and responses for speed perturbations are obtained analytically and simulated.

2. State Space Approach

A Set of nonlinear differential equations can describe the behavior of the induction motor [14-16]. If a complete solution of the dynamic behavior of the induction machine is desired, these equations must be solved in detail. By linerarizing these questions about a steady state operating condition, the resulting equations in state form can describe the dynamics, and provide the future state and output of the system.

Perturbations in reference voltage or firing angle and load torque leads to changes in rotor speed. The analytical results used to investigate these speed changes are obtained considering the various previous functional blocks, where the different input and output variables are denoted by X_1, X_2, X_3 and X_4. These variables are defined as follows:

$$X_1 = \Delta\omega,\ X_2 = \Delta V,\ X_3 = \Delta V_c \text{ and } X_4 = \Delta\alpha \qquad (1)$$

The differential equations, which govern the small variations about the operating point, are written in terms of the above variables and representing in matrix form in Equation (2), where

$$\dot{X} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^T,\ u = \begin{bmatrix} \Delta T_L & \Delta V_R \end{bmatrix}^T = \begin{bmatrix} u_1 & u_2 \end{bmatrix}^T$$

3. System Description

The system consists of a slip-ring induction motor with three equal external resistances, each connected to the rotor phase and three delta-connected phase-controlled SCR's placed at the open star point of the rotor as shown in **Figure 1**.

In variable speed ac induction motor drives, a continuous monitoring or control of slip speed or slip frequency is required. A permanent magnet tachogenerator is mounted on the rotor shaft to provide a dc signal proportional to the rotor speed to the feedback control circuit.

The block diagram of the feedback control scheme of the induction motor is shown in **Figure 2**.

The induction motor stator is supplied with constant voltage, constant frequency supply. The rotor speed is controlled and adjusted by advancing or retarding the firing angle α of the SCRs. The tachogenerator output voltage proportional to the rotor speed and is compared

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} \left(\frac{K_G K_5}{T_G} - \frac{1}{T_G}\right) & 0 & 0 & \frac{K_G K_4}{T_G} \\ \frac{K_1}{T_1} & -\frac{1}{T_1} & 0 & 0 \\ -\frac{K_1 K_2}{T_1} & -\left(\frac{K_2}{T_2} + \frac{K_2}{T_1}\right) & 0 & 0 \\ 0 & 0 & \frac{K_3}{T_3} & -\frac{1}{T_3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} -\frac{K_G}{T_G} & 0 \\ 0 & 0 \\ 0 & -\frac{K_2}{T_2} \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta T_L \\ \Delta V_R \end{bmatrix} \qquad (2)$$

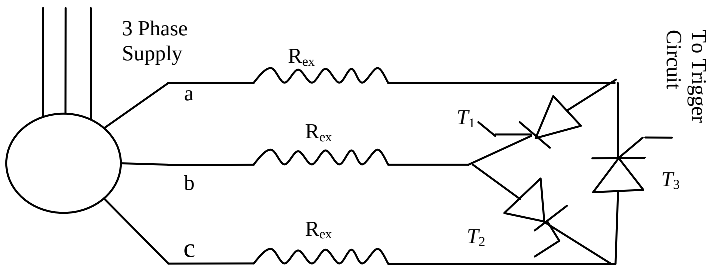


Figure 1. Schematic diagram of phase controlled SCR's in delta (Δ) configuration

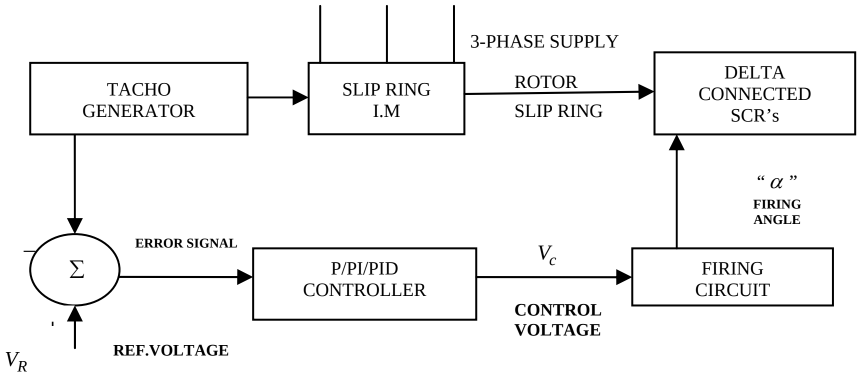


Figure 2. Block diagram of feedback system

with a fixed dc level V_R which represents the set speed. The error voltage is forwarded to the controller. The set peed is changed by varying V_R automatically or manually. The controller may be a proportional, or proportional integral or proportional integral derivation type. The function of the controller is to give the required control voltage which will adjust the firing angle to the suitable value and can be used also as a stabilizing signal if more than one controller is used.

The simulink block diagram of feedback control scheme of the induction motor is shown in **Figure 3**.

Transfer functions for the functional blocks:

The transfer functions for the various functions blocks of the feedback system are shown in **Figure 4**, and given in details as follows:

1) Tachogenerator and filter: The transfer function of this block is represented by:

$$G_1(s) = \frac{K_1}{1+ST_1} \tag{3}$$

where K_1 is the combined gain of the tachogenerator and the associated filter, and T_1 is the effective time constant of the filter.

2) Controller: The change in the output voltage of the tachogenerator is compared with the reference voltage V_R and the resultant error voltage is fed to the controller. The controller output voltage is corrected in accordance with the input change in voltage. The change in the controller output voltage is denoted as V_c. The transfer function of the proportional integral controller is:

$$G_1(s) = \frac{K_2(1+ST_2)}{ST_2} \tag{4}$$

3) Firing Circuit: The firing circuit decides the change in firing angle in accordance with the change in control voltage V_c. It consists of a ramp generator and a comparator. The ramp is synchronized with the signal available across the slip-rings of the machine. For a given change in the control voltage V_c, the change in firing angle is given by:

$$\Delta\alpha = \frac{1}{m}\Delta V_c \tag{5}$$

where *m* is the slope of the ramp. For the present study, the firing circuit transfer function can be written as

$$G_3(s) = \frac{K_3}{1+ST_3} \tag{6}$$

where K_3 is equal to l/*m*, and the time constant is equal to one half of the maximum expected delay. If the slip of the rotor at the operating point is *s*, then the time constant T_3 is given by:

$$T_3 = \frac{1}{s\times f\times 2\times 3} \tag{7}$$

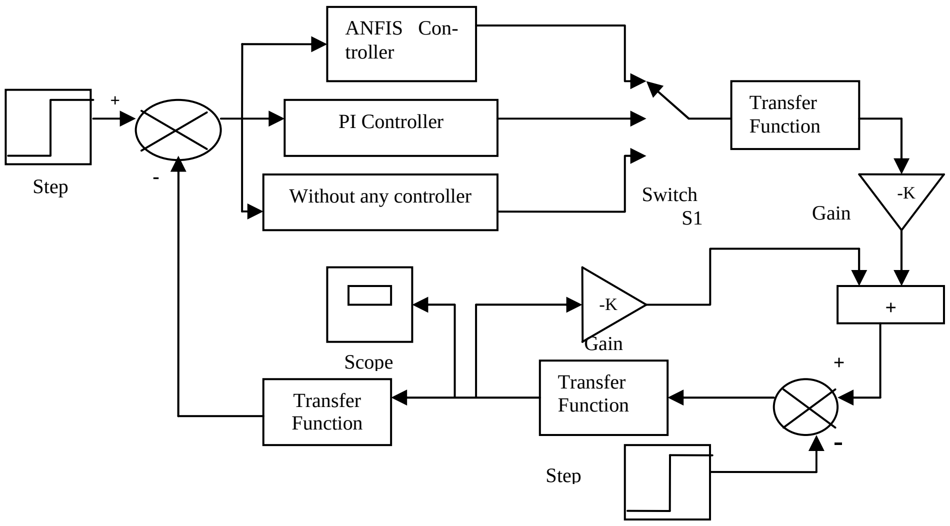


Figure 3. The simulink block diagram of feedback control scheme of the induction motor drive

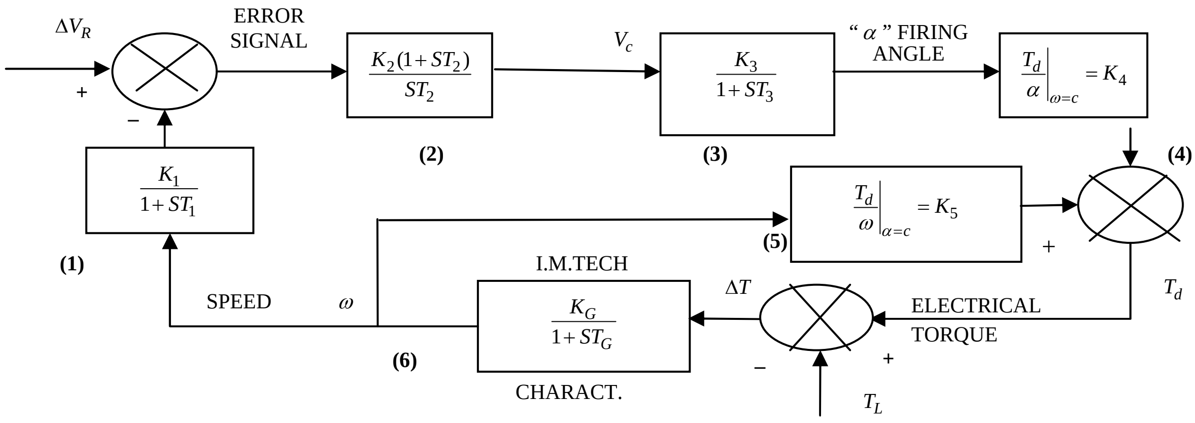


Figure 4. Functional blocks of closed-loop system

4) Induction Motor: The torque developed by the machine at a given operating point is a function of speed of the machine and the firing angle of the thyristors. The difference between the developed torque and the load torque is applied to the rotating elements. The torque developed by the machine is presented by

$$T_d = F(\omega, \alpha) \tag{8}$$

where ω is the rotor speed in rad/sec, and α is the firing angle.

For the dynamic behavior of the induction machine about any operating point for a given perturbation, the small change in the developed torque can be represented in terms of the small changes in rotor speed and firing angle as:

$$\Delta T_d = \left.\frac{\Delta T_d}{\Delta \alpha}\right|_{\omega = cons\tan t} \Delta\alpha + \left.\frac{\Delta T_d}{\Delta \omega}\right|_{\alpha = cons\tan t} \Delta\omega \tag{9}$$

or

$$\Delta T_d = K_4 \Delta\alpha + K_5 \Delta\omega \tag{10}$$

The constants K_4 and K_5 depend upon the operating point and are to be obtained from the steady-state characteristics of the system.

The resultant change in the developed torque is represented as the summation of the outputs of the two blocks (4) and (5). The change in the developed torque is compared with the change in load torque and the resultant value is forwarded to the mechanical system, whose transfer function can be expressed as:

$$G_m(s) = \frac{K_G}{1 + ST_G} \tag{11}$$

where $K_G = \frac{1}{F}$ and $T_G = \frac{J}{F}$

F is the frictional constant in N.m/rad/s, and *J* is the moment of inertia of the rotating system in $K_G \cdot \text{m}^2$.

4. ANFIS Based Speed Controller

Artificial Intelligent tools such as Fuzzy Logic and Artificial Neural Networks have shown great potential on variable frequency drives. Artificial Neural Networks are concerned with adaptive learning, nonlinear function approximation, and universal generalization; fuzzy logic with imprecision and approximate reasoning [17,18]. But they share some common shortcomings that hinder them from being used more widely. For example, neural networks, often suffer from a slow learning rate. This drawback renders neural networks less than suitable for time critical applications. Therefore, new and enhanced methods can be put forward.

The fuzzy neural network is constructed to merge fuzzy inference mechanism and neural networks into an integrated system so that their individual weaknesses are overcome. The ANFIS system determines a control action by using a neural network which implements a fuzzy inference. In this way, the prior expert's knowledge can be incorporated easily. The controller has two states, a learning state and a controlling state. In the learning state, the performance evaluation is carried out according to the feedback which represents the process state. If input-output training data is available, the performance can be assessed easily, and supervised learning can be employed.

5. Adaptive Neuro-Fuzzy Principle

The fuzzy inference commonly used in ANFIS is first order Sugeno fuzzy model because of its simplicity, high interpretability, and computational efficiency, built-in optimal and adaptive techniques. A typical architecture of an ANFIS is as shown in **Figure 5**. Among many FIS models, the Sugeno fuzzy model is the most widely applied one for its high interpretability and computational efficiency, and built-in optimal and adaptive techniques. For a first order Sugeno fuzzy model, a common rule set with two fuzzy if-then rules can be expressed as:

Rule 1: if x is A_1 and y is B_1, then $z_1 = p_1x + q_1y + r_1$

Rule 2: if x is A_2 and y is B_2, then $z_2 = p_2x + q_2y + r_2$

where A_i and B_i are the fuzzy sets in the antecedent, and p_i, q_i and r_i are the design parameters that are determined during the training process.

Layer 1: Every node in this layer contains membership functions.

$$o_i^1 = \mu_{A_i}(x), i = 1,2 \tag{12}$$

$$o_i^1 = \mu_{B_{i=2}}(y), i = 3,4 \tag{13}$$

where μ_{A_i} and μ_{B_i} can adopt any fuzzy membership function (MF).

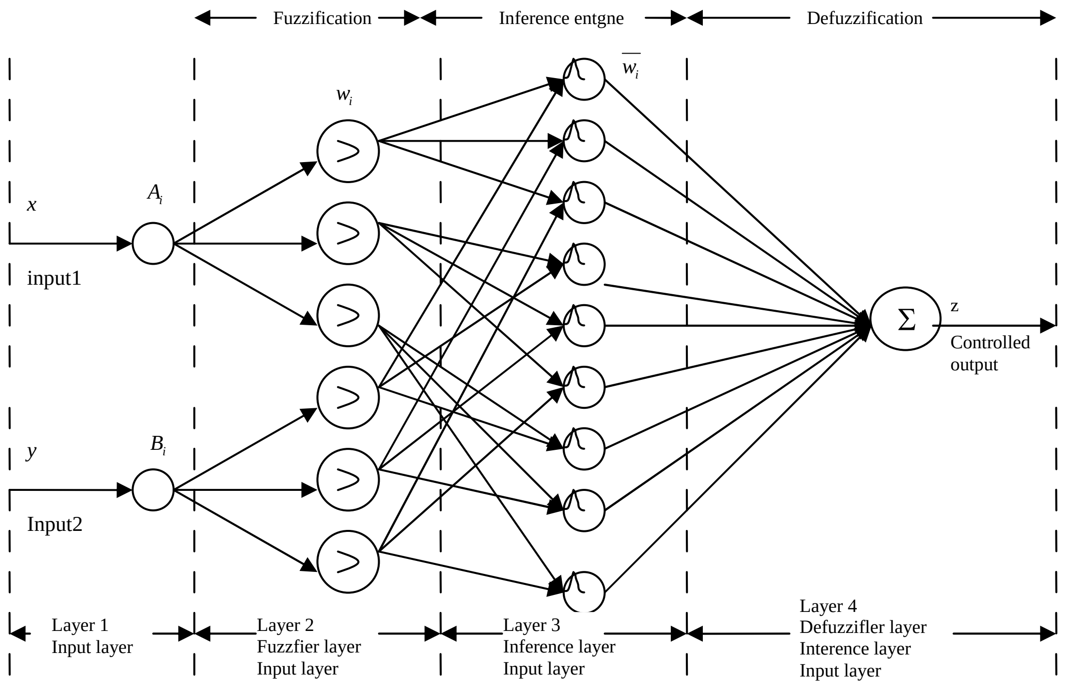


Figure 5. Adaptive neuro fuzzy structure

Layer 2: This layer chooses the minimum value of two input weights.

$$o_i^2 = w_i = \mu_{A_i}(x)\mu_{B_i}(y), i = 1,2 \tag{14}$$

Layer 3: Every node of these layers calculates the weight, which is normalized.

$$o_i^3 = \overline{w_i} = \frac{w_i}{w_1 + w_2}, i = 1,2 \tag{15}$$

where $\overline{w_i}$ is referred to as the normalized firing strengths.

Layer 4: This layer includes linear functions, which are functions of the input signals.

$$o_i^4 = \overline{w_i} z_i = \overline{w_i}(p_i x + q_i y + r_i), i = 1,2 \tag{16}$$

where $\overline{w_i}$ is the output of layer 3, and $\{p_i, q_i, r_i\}$ is the parameter set. The parameters in this layer are referred to as the consequent parameters.

Layer 5: This layer sums all the incoming signals.

$$o_i^5 = \sum_{i=1}^{2} \overline{w_i} z_i = \frac{w_1 z_1 + w_2 z_2}{w_1 + w_2} \tag{17}$$

The output *z* in **Figure 5** can be rewritten as:

$$\begin{aligned} z &= \left(\overline{w_1}x\right)p_1 + \left(\overline{w_1}y\right)q_1 + \left(\overline{w_1}\right)r_1 \\ &+ \left(\overline{w_2}x\right)p_2 + \left(\overline{w_2}y\right)q_2 + \left(\overline{w_2}\right)r_2 \end{aligned} \tag{18}$$

In this paper the normalized membership functions of input variables and output variable are shown in **Figures 6** and **7**. The Three-dimensional plot of Fuzzy Control surface is shown in **Figure 8**.

6. Simulation Results

In this paper, performance of the proposed ANFIS speed controller is evaluated and is compared with PI controller and without any controller. The controller parameters are chosen to optimize the performance criterion of the dynamic operation, and then the tuning was empirically improved. The simulation is carried out to observe the performance of the system at different load perturbations.

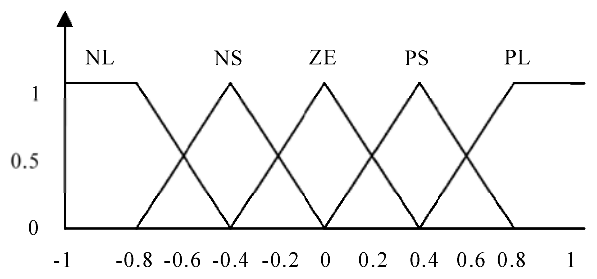


Figure 6. Triangular membership functions for input variables *e* and Δe

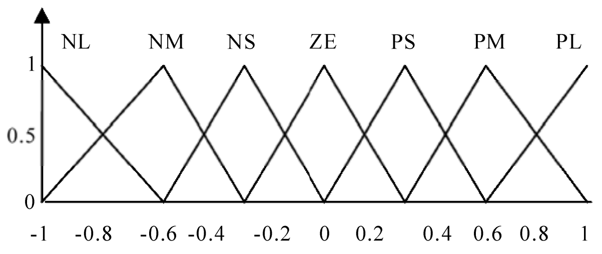


Figure 7. Triangular membership functions for output variable

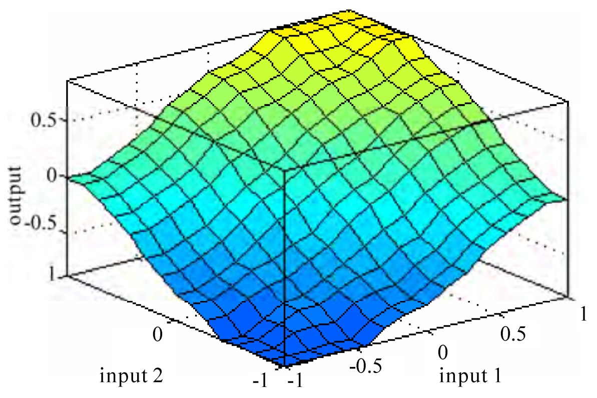


Figure 8. Three-dimensional plot of control surface

The software environment used for this simulation is Matlab ver. 7.1, with simulink package.

The change in rotor speed is due to the perturbations in reference voltage or firing angle and load torque. The analytical results used to investigate these speed changes are obtained considering the various previous functional blocks, where the different input and output variables are denoted by X_1, X_2, X_3 & X_4. The differential equations which govern the small variations about the operating point in terms of above variables are given in Equation (2).

The perturbation studies were carried out at different operating points with different system parameters (gains and time constants) which are given in Appendix. Studies are carried out at operating points with various system parameters (gains and time constants). The simulation results give the present perturbation study for step change in the load torque and reference voltage. From the **Figures 9** to **11** the starting transients are realized for ANFIS controller at different operating conditions. It can be observed from the figures that the performance of the ANFIS gives better response compared with PI controller and without any controller.

7. Conclusions

A framework for tuning and self organizing ANFIS controller has been presented. This approach has been contrasted without any controller and with PI controller. The dynamic behavior of a closed-loop, variable speed induction motor drive which uses three silicon controlled rectifiers has been studied in this paper. Transfer function blocks of the system for small variations about an operating point are derived, and the transient responses with the analytical studies have been carried out. Comparison of ANFIS controller, without any controller and with PI controller under normal operation for a given load torque and reference speed perturbations has been presented. It

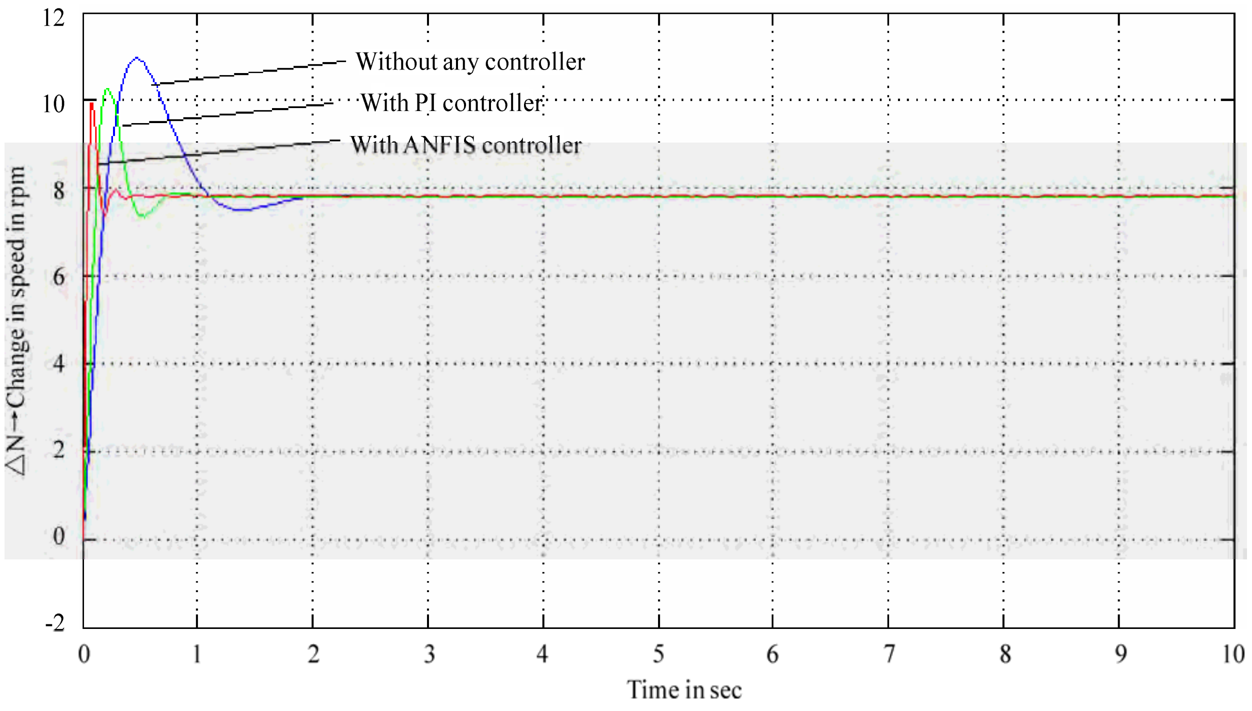


Figure 9. Variation of speed deviation at 5% load change

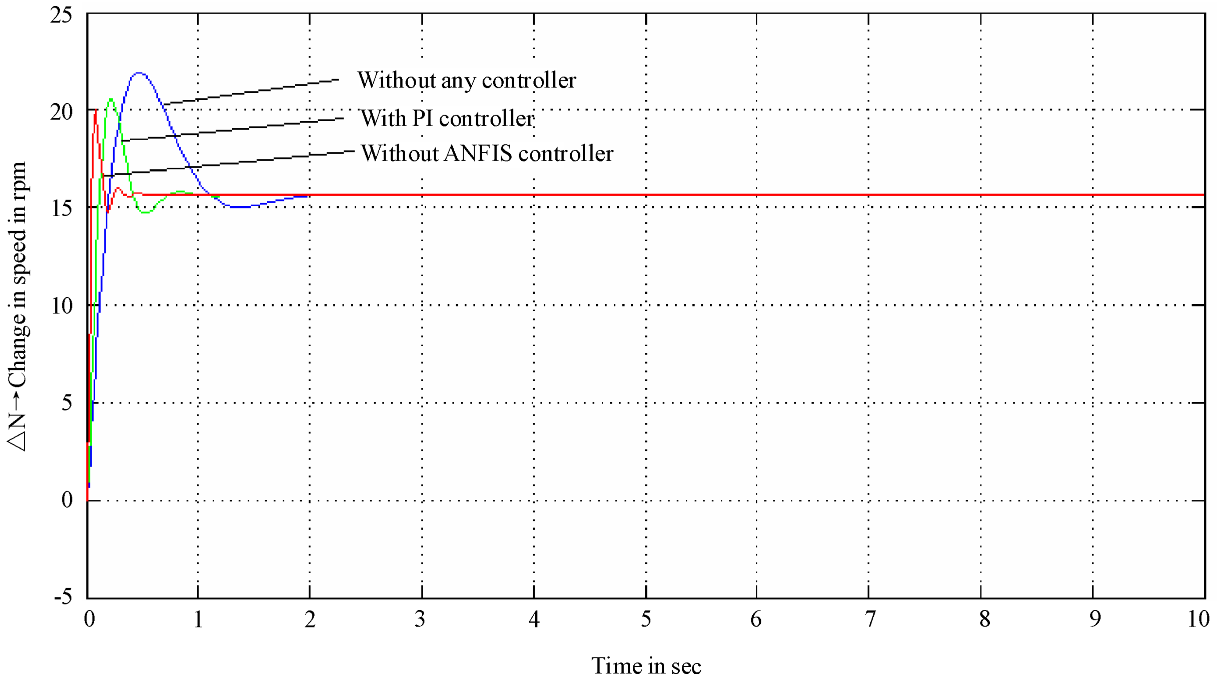


Figure 10. Variation of speed deviation at 10% load change

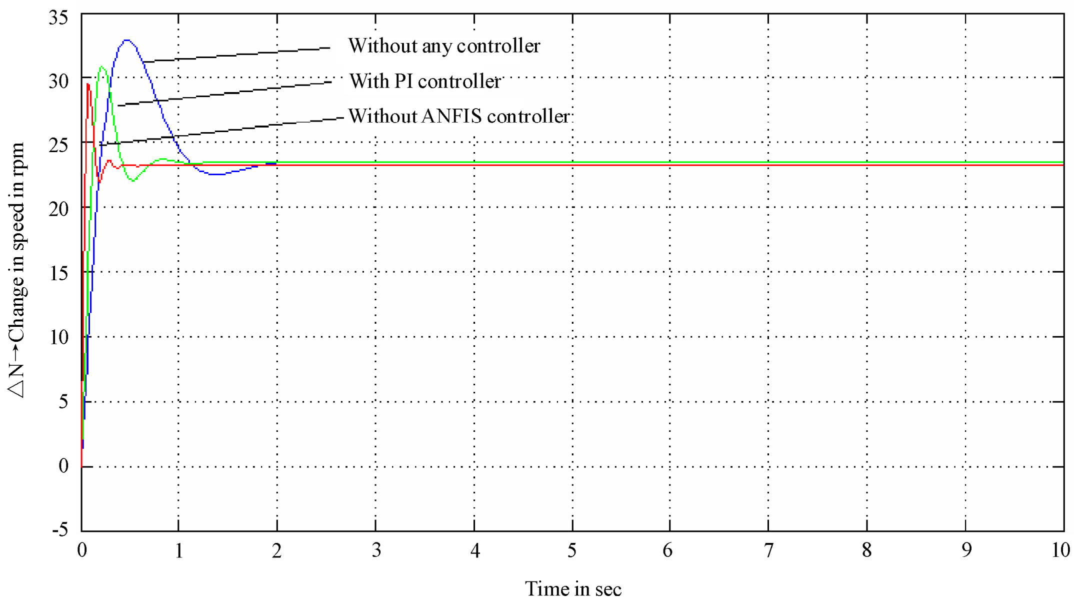


Figure 11. Variation of speed deviation at 15% load change

has been demonstrated that the proposed method gives a good response, regardless of parameter variations or external force. Simulation results have shown the capabilities of the proposed controller in tracking predetermined desired speed trajectory.

REFERENCES

[1] R. P. Basu, "A Variable Speed Induction Motor Using Thyristors in the Secondary Circuit," *IEEE Transactions on Parer Apparatus and Systems*, Vol. 90, 1971, pp. 509-514.

[2] M. Ramamoorthy and M. Arunachalam, "A Solid-State Controller for Slip Ring Induction Motors," *The IEEE Industry Applications Society Annual Meeting*, Los Angeles, California, October 2-6, 1977.

[3] M. Ramamoorthy and M. Arunachalam, "Dynamic Performance of a Closed Loop Induction Motor Speed Control System with Phase-Controlled SCR's in the Rotor," *IEEE Transactions on Industry Applications*, Vol. 15, No. 5, 1979, pp. 489-493.

[4] Y. Hsu and W. Chan, "Optimal Variable-Structure Controller for DC Motor Speed Control," *IEEE Proceedings D on Control Theory and Applications*, Vol. 131, No. 6, 1984, pp. 233-237.

[5] B. S. Zhang and J. M. Edmunds, "On Fuzzy Logic Controllers," *IEEE International Conference on Control*, Edinburg, UK, 1991, pp. 961-965.

[6] H. Ying, W. Siler and J. J. Buckley, "Fuzzy Control Theory: A nonlinear Case," *Automatica*, Vol. 26, No. 3, 1990, pp. 513-520.

[7] D. Dirankov, H. Hellendorn and M. Reinfrank, "An Introduction to Fuzzy Control," Springer-Verlag, New York, 1993.

[8] M. Maeda and S. Murakami, "A Self-Tuning Fuzzy Controller," *Fuzzy sets and Systems*, Vol.51, No. 1, 1992, pp. 29-40.

[9] T. J. Procyk and E. H. Mamdani, "A Linguistic Self-Organizing Process Controller," *Automatica*, Vol. 15, No. 1, 1979, pp. 53-65.

[10] R. Storn and K. Price, "Differential Evolution-A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces," ICSI Technical Report, March 1995.

[11] D. Karaboga and S. Okdem, "A Simple and Global Optimization Algorithm for Engineering Problems: Differential Evolution Algorithm," *Turk Journal of Electrical Engineering*, Vol. 12, No. 1, 2004, pp. 53-60.

[12] D. Borojevic, L. Garces and F. Lee, "Performance Comparison of Variable Structure Controls with PI Control for DC Motor Speed Regulator," *IEEE Industry Applications Conference*, 1984, pp. 395-405.

[13] J. Zhao and B. K. Bose, "Evaluation of Membership Functions for Fuzzy Logic Controlled Induction Motor Drive," *IEEE* 2002 28*th annual Conference of the Industrial Electronics Society*, Vol. 1, 2002, pp. 229-234.

[14] A. S. A. Farag, "State-Space Approach to the Analysis of DC Machines Controlled by SCRs," *IEEE Proceeding Publication-on the Control of Power Systems Conference*, Oklahoma, March 10-12, 1976, pp. 157-163.

[15] N. Mohan, "Electric Drives: An Integrative Approach," Minnesota Power Electronics Research & Education, Minnesota, 2003.

[16] N. Mohan, "Advanced Electric Drives: Analysis, Control and Modeling using Simulink®," Minnesota Power Electronics Research & Education, Minnesota, 2001.

[17] B. K. Bose, "Fuzzy Logic and Neural Network Applications in Power Electronics," *Proceedings of the IEEE*, Vol. 82, No. 8, 1994, pp. 1303-1323.

[18] M. G. Simoes and B. K. Bose, "Neural Network Based Estimation of Feedback Signals for Vector Controlled Induction Motor Drive," *IEEE Transactions on Industry Applications*, Vol. 31, No. 3, 1995, pp. 620-629.

Appendix

Various Gains and Time Constants used for Perturbation Study (Motor Speed 'N = 1050 rpm)

$K_1 = 0.032$	$T_1 = 0.009$
$K_2 = 0.25$	$T_2 = 0.22$
$K_3 = –60$	$T_3 = 0.01111$
$K_4 = –0.0363$	$K_5 = –0.095$
$K_5 = 40.0$	$T_G = 15.6$

TABLE OF CONTENTS

Volume 2 Number 2 May 2010