Adaptive Logic Networks and Robot Control

Allen G. Supynuk
University Computing Systems

William W. Armstrong
Department of Computing Science

University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

The equations of motion of a tree of rigid links connected by hinges are well understood. While relatively efficient closed-form solutions exist, they do not allow us to compute much faster than real time on current systems. Adaptive logic networks (ALNs) have the potential to approximate the quantities describing motion (after training on representative sample data) in a few tens of logic gate propagation delays (ie. in nanoseconds).

1 Introduction

Our main focus in this paper is on high-speed simulation of autonomous robot motion using adaptive logic networks. High speed simulation could be a useful basis for real-time trajectory planning in cases where an analytical solution to the planning problem is too time-consuming to be useful. Canny has shown [Cann88] that dynamic robot motion planning in an environment of moving obstacles is a hard task if the robot's velocity is constrained. Even in two dimensions, if the obstacles are convex polygons moving without rotation, motion planning is NP-hard. Hence, some heuristic approach to motion planning with such obstacles is appropriate.

The method for achieving high-speed simulation proposed in this paper is based on fast prediction of an approximation to the motion by means of adaptive logic networks (ALNs). In hardware form, trained ALNs can be implemented as combinational logic circuits, capable of predicting approximate values defining future motion in time proportional to the depth of the circuit. For the size of circuits used, a completely parallel hardware implementation would compute an iteration step of the simulation in much less than a microsecond. Based on iterated faster-than-real-time prediction of what would happen given certain applied torques or other values which can be controlled, the torques etc. can be changed until the predicted solution is satisfactory.

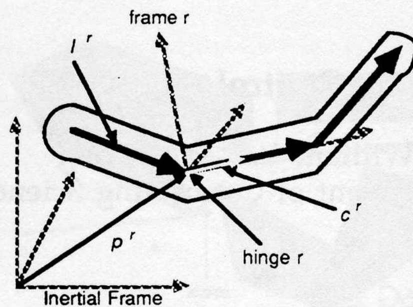
In the field of computer animation, simulation of motion using correct dynamics was investigated by Wilhelms [Wilh87]. A fast recursive method for the solution of equations of motion was described by Armstrong and Green [Arms85]. Lake and Green [Lake90] had some success in modeling ballroom dancing motions using the recursive technique, however, there was no attempt to alleviate the planning of collision-free motion in the presence of other moving dancers. Baraff [Bara90] studied some problems of collision between bodies with twice-differentiable surfaces using an iterative trial and error approach similar to the one we are proposing. The forces generated by contact were modeled and it was suggested that the solution of equations of motion in the presence of friction is NP-hard, and that heuristic methods might be required. Despite the development of a parallel version of the recursive technique, [Arms87] the recursive solution method is still far from the goal of being much faster than real time on today's machines, and so is incapable of supporting numerous trial simulations forming the basis of a heuristic method for solving motion planning problems in the presence of moving obstacles.

Our proposed ALN method is independent of the solution method of the equations of motion, since only the step of generating training data for the adaptive logic networks requires the solution. This allows one to use slower methods of solution that can incorporate more generality, such as one-degree-of-freedom hinges, without paying any penalty in speed.

2 Equations of Motion Simulator

For a complete description of the recursive solution of, and variables used by, the equations of motion simulator used to produce both the training and testing data for the adaptive logic networks see Armstrong and Green [Arms85].

The model represents objects as a tree structure (that is, there are no loops) of flexibly linked objects. The point where an object joins its parent is called its proximal hinge. Orthonormal moving three dimensional frames are attached to each object at its proximal hinge. There is also a fixed, non-rotating inertial frame. The model calculates position, velocity, acceleration, angular velocity, and angular acceleration based on external forces and torques and control torques at the hinges.



The inertial frame and some selected variables

The model uses the following quantities. Lower-case letters denote scalars and vectors; uppercase denote matrices. Superscripts denote the link number. Each link other than the root (link 1) has one proximal hinge connecting it to its parent:

Scalars

- m^r : the mass of link r

Quantities in the inertial frame

- a_G : the acceleration of gravity
- p^r : the position vector of the hinge of link r which joins it to its parent (the proximal hinge of r)
- v^r : the velocity of the proximal hinge of link r
- f_E^r : an external force acting on link r at the proximal hinge
- g_E^r : an external torque acting on link r

Quantities in the frame of link r

- a^r : the acceleration of the proximal hinge of link r
- ω^r : the angular velocity of link r
- $\dot{\omega}^r$: the rate of change of ω^r . This would be the angular acceleration if ω^r was represented in the inertial frame.
- c^r : the vector from the proximal hinge of link r to the center of mass of link r
- f^r : the force which link r exerts on its parent at the proximal hinge
- g^r : the control torque which link r exerts on its parent at the proximal hinge
- J^r : the moment of inertia matrix of link r about its proximal hinge

Quantities in the frame of the parent of link r

- l^r : the vector from the proximal hinge of the parent of link r to the proximal hinge of link r (a constant vector in this frame)

Rotation Matrices

- R^r : converts vector representations in the frame of link r to their representations in the frame of the parent link

For a description of which variables were selected for learning by the ALN predictor and why, see section 4.1 - *Variables Learned.*

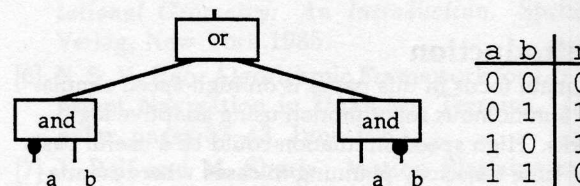
3 Adaptive Logic Networks (ALNs)

Adaptive logic networks [Arms79, Arms 90, Arms91] are (conceptually) binary trees. Each interior node calculates one of four binary functions: **and, right, left,** and **or.** The input to the tree (a sequence of 0's and 1's) is fed in through the leaves. Negation (logical not of an input) can happen only at a leaf. Each tree calculates one bit of the result.

a	b	a and b	a right b	a left b	a or b
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	0	1	1
1	1	1	1	1	1

Table of functions at nodes of adaptive logic networks

For example, the following adaptive logic network calculates the function on the right:



a	b	r
0	0	0
0	1	1
1	0	1
1	1	0

An adaptive logic network and the function it implements (• denotes negation)

3.1 ALNs vs. Neural Networks

This section explores the relationship between adaptive logic networks and the most popular mainstream neural networks [Hech90].

Each node of the usual neural networks trained by the back-propagation algorithm typically calculates a function of the form:

$$\frac{1}{1 + e^{-\gamma \cdot \sum_{i=0}^{n} w_i \cdot x_i}}$$

where w_i $(0 \le i \le n)$ are weights and x_i $(1 \le i \le n)$ are the inputs (possibly an output from another node), w_0 is a bias (constant after training), x_0 is a constant 1, and γ is a constant that controls the slope of the sigmoid. As γ goes towards infinity, this can be better approximated by

$$f(x) = \begin{cases} 1, & \sum_{i=0}^{n} w_i \cdot x_i \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

To get an adaptive logic network from this model restrict w_i to the set {1,2}, x_i to the set {0,1}, and set w_0 to -2, x_0 to 1. This gives:

$$-2.1 + w_1.x_1 + w_2.x_2 \geq 0 \Leftrightarrow w_1.x_1 + w_2.x_2 \geq 2$$

If we set w_1 and w_2 to 1 our output will be 1 if and only if both x_1 and x_2 are 1, which is equivalent to **and**. Setting w_1 and w_2 to 2 give us **or**, $w_1 = 1$ and $w_2 = 2$ gives us **right**, and $w_1 = 2$ and $w_2 = 1$ gives us **left**. That is, adaptive logic networks are special cases of linear threshold networks.

3.2 Parsimonious or Lazy Evaluation of ALNs

We almost never have to calculate all the node functions of a tree. Once we know one input to a node we have a 50% chance of knowing what the output is. (For **and** and **or** this is called McCarthy or lazy evaluation in the programming languages literature. Since "lazy" has negative connotations, and since "parsimony" has already been used in neural network literature [Meis90], we use the term *Parsimony* or *Parsimonious Evaluation.*)

To calculate the relative savings over the whole tree, consider the function ns(d), the number of signals a node that is d levels from being a leaf has to have in order to determine its result. Clearly,

$$ns(d) = \begin{cases} 1, & d = 0 \\ ns(d-1) + 0.5 * ns(d-1), & d > 0 \end{cases}$$

That is, we need to calculate ns(d-1) nodes to know one signal, and then have a 50% chance of needing to calculate another ns(d-1) nodes to know the other signal. Expanding a few terms of this recurrence relation quickly leads to the closed form solution: $ns(d) = \left(\frac{3}{2}\right)^d$. A tree of height d has $2^{d+1}-1$ nodes (including the leaves), so the average relative savings are:

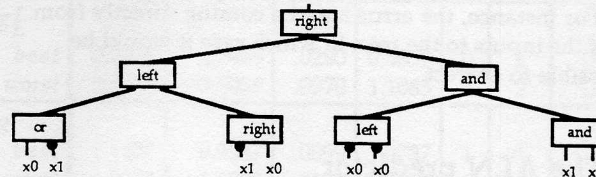
$$\frac{ns(d)}{2^{d+1}-1} = \frac{\left(\frac{3}{2}\right)^d}{2^{d+1}-1} \cong \frac{3^d}{2^{2d+1}} = \frac{1}{2}\left(\frac{3}{4}\right)^d$$

This results in an exponential speedup in the evaluation of an ALN; the larger the tree, the smaller the fraction of the tree that is likely to need evaluation.

3.3 How ALNs Learn

This section outlines the method which adaptive logic networks use to learn functions. For a more complete description see the papers by Bochman and Armstrong [Boch74], and Armstrong [Arms79, Arms90].

Each ALN is a binary tree, initially set up with random functions at each of the interior nodes. The k binary inputs to the tree and their k binary complements are randomly connected to the leaves of the tree. (Note that the number of leaves should be at least twice k so that each input and its complement can appear on a leaf. It should preferably be several times k so that useful pairings occur close together on the tree.) For example:



An ALN with 8 leaves, two inputs (x0 and x1), one output

Now suppose that an input is presented to the tree for training, so we know what the expected output is. First we check to see if the answer is correct. If it is, the current behavior is reinforced. If not, the behavior is discouraged. To adjust behavior we recursively traverse the tree looking for nodes that might be responsible for the output. Note that if we determine a node is not responsible, then none of its children are responsible either.

Since all four functions (**and, or, left, right**) produce the same value when their input is (0,0) or (1,1), the adaptation consists of adjusting responsible nodes whose inputs are either (0,1) or (1,0). For this purpose, a counter associated with (0,1) and another associated with (1,0) are implemented in each node. If the desired output is 1 the relevant counter is incremented; if 0, decremented. The counters are bounded; trying to increment or decrement past their limit has no effect. In effect, each counter tries to keep track if it is expected to produce a 1 or a 0 when it is responsible. The following table shows how the counters induce a function on the node:

(0,1) ctr	(1,0) ctr	output on receiving (0,0)	(0,1)	(1,0)	(1,1)	induced function
<0	<0	0	0	0	1	**and**
<0	≥ 0	0	0	1	1	**left**
≥ 0	<0	0	1	0	1	**right**
≥ 0	≥ 0	0	1	1	1	**or**

The problem occurs, of course, in the assigning of responsibility to a node. If the wrong node is deemed responsible, learning will be either hindered or blocked. A simple form of heuristic responsibility is [Arms90, p 5]:

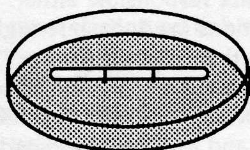
1) the root is always *heuristically responsible*

2) **if** a node is *heuristically responsible* **and** one of its input signals is not equal to the desired network output **then** that input signal is called an *error*
3) the child on the <u>opposite</u> side to an *error* is *heuristically responsible*
4) **if** the output of a heuristically responsible node would change if the output of a child changed, that child is *heuristically responsible*
5) no other nodes are *heuristically responsible*

The motivation behind this form of heuristic responsibility is that an erroneous input signal (*error*) means that the tree on the other side needs to try harder to compensate. For instance, the error may be coming directly from one of the inputs to the tree, in which case it would be impossible to correct.

4 The ALN predictor

Our goal was to use adaptive logic networks to predict the results of the model used in section 2 *Equations of Motion Simulator*. An existing program, *DynaTree*, was used that implemented this system for a three-segment "worm" moving (rolling, jumping, falling) around in a circular walled arena [Arms85]. Since the environment is concave, the only points that need to be considered to evaluate the non-holonomic constraints are the two end points and the two joints of the worm:



The worm in its lair

As section 2 attests, the model has many variables. Which of these should be chosen for the predictor? The following section outlines the choices made and the motivation behind them.

4.1 Variables Learned

The brute force approach of learning all 18 variables, most of which are vectors and matrices, was quickly rejected. A biologically motivated decision was made not to include any variable that described the inherent properties of the worm itself. For example, the mass of each segment (m^r) was not included since you and I do not use the weight of our arm when deciding how to move it. Instead, we just move our arm around until we learn that "pushing that hard makes it go that fast." Similarly the acceleration of gravity (a_G), the position vector of the links (p^r), the position of the center of mass relative to the hinge (c^r), the forces (f^r) between the links, the relative position of the links (l^r), and the moment of inertia matrix (J^r)were not included.

The variables chosen were the velocity (v^r), acceleration (a^r), angular velocity (ω^r), its rate of change ($\dot{\omega}^r$), torques (g^r) at each of the three hinges (the model has a hinge at the head as well as at the interior joints), the external forces (f_E^r) and torques (g_E^r), and the angle at each of the two joints in terms of roll and pitch (derived from R^r).The constraining force (f^r) does not play a role in controlling the worm. For consistency, and in keeping with an unsubstantiated expectation of the biological model, all variables were translated into the frame of the link.

Six of these variables are three dimensional vectors for each of the 3 links, the roll and pitch are two numbers for each of two links, and the torque is a three dimensional vector at each of the two interior hinges. That is, a total of 64 numbers are both fed into and predicted by ALNs.

4.2 Quantizing the Variables

To determine the range of values each variable took on during the course of execution, the simulator was changed to determine the minimum, maximum, mean, and standard deviation. All available configuration files for *DynaTree* were then run, giving the following results:

Quantity	*min*	*max*	*n vars*
a^r	-500	500	9
f_E^r	-3000	4000	9
g_E^r	-3000	3000	9
g^r	-150	150	6
(R^r)	-1.57	1.57	4
ω^r	-35	35	9
$\dot{\omega}^r$	-1500	2000	9
v^r	-18	11	9

All variables were quantized to 214 levels and coded into binary vectors of size 23 bits.

4.3 Other ALN Parameters

Having 64 real variables each quantized into 23 bits, gives $64 \cdot 23 = 1472$ input bits. Since both the input bit and its complement must be fed in, this requires a tree with at least $1472 \cdot 2 = 2944$ leaves. Since it is highly desirable to have room for at least two copies of the input, and since tree sizes that are powers of two are marginally more efficient, the initial size of each tree was set to have 8192 leaves (8191 internal nodes).

The trees were trained on 1000 input vectors for a maximum of 10 epochs (presentations of the training set). The training data was derived by instrumenting the original dynamics program and having it write out the values of all 64 variables after each time slice. A separate

program (stride) was then run to pick out every nth line and its successor from this file (we used n=9), append the successor to the end of the nth line, and write the result to a file. Hence the trees were trained with uniform samples of what the variables looked like before and after a time slice in the original program.

The trees were then tested on 9000 samples from the same data. One in nine tests was seen during training, giving some slight bias to the results.

4.4 What Can We Reasonably Expect?

Perfect prediction is an unrealistic goal for an ALN. Input values are quantized and then represented by bit vectors. An arbitrarily small change in input can lead to a different quantization level which leads to a different bit vector. Two different bit vectors differ in at least one bit, so it is not unreasonable for the output of one or more of the trees to change. Similarly, since the expected output is also a quantized value, the expected output can change by one quantization level in each case. Hence, errors of 2 or 3 quantization levels are unavoidable for many problem domains.

So, if the mean and the standard deviation suggest that 95% of the time we are within 3 quantization levels, we have an ALN that is very near optimal.

4.5 A View of the Raw Data

The complete ALN takes two to three days to train the 1472 trees on a lightly loaded Sun SPARCStation I. The test results were the output of `lf` after piping them through **histogram**, The result is a histogram of errors that says, for each variable learned, "the ALN had m errors of magnitude n."

The results were characterized by having a small number of outliers - values a long way from the mean. While few in number, each has an enormous effect on the mean. For example, the results for the third variable ($a^r[1][z]$) look like this:

6300	at correct quantization level
1854	out by 1 quantization level
343	out by 2 quantization levels
185	out by 3 quantization levels
109	out by 4 quantization levels
48	out by 5 quantization levels
31	out by 6 quantization levels
12	out by 7 quantization levels
21	out by 8 quantization levels
21	out by 9 quantization levels

The remaining 76 values were spread between errors of quantization levels from 10 through 151. Removing the last 18 values cause the mean of the number of quantization levels in error to drop from .6953 to .5795 and the standard deviation to drop from 3.4881 to 1.5434.

4.6 Results

At the end of several trials, each variable was learned to the following values of the error, measure in quantization levels:

var	*mean*	*std dev*	*std err*	*5% trim*	*me an*	IQR	*max*
a^r							
best	0.6953	3.4881	.0368	0.3332	0	1	151
worst	2.8976	9.3316	.0984	1.3563	1	2	168
f_E^r							
best	0.7422	2.8522	.0301	0.4389	0	1	161
worst	4.0970	15.2223	.1605	1.4733	1	2	181
g_E^r							
best	0.7540	2.7485	.0290	0.4857	0	1	156
worst	2.5918	9.2039	.0970	1.1065	1	2	150
g^r							
best	1.1401	0.9347	.0099	1.0737	1	1	10
worst	1.5663	1.6206	.0171	1.3957	1	2	21
R^r							
best	0.5246	3.2197	.0339	0.3900	0	1	212
worst	1.1103	8.1942	.0864	0.6654	1	1	213
ω^r							
best	1.0896	1.6230	.0171	0.9591	1	2	95
worst	2.5804	11.8330	.1247	0.9584	1	1	190
$\dot{\omega}^r$							
best	0.9749	3.2439	.0342	0.6127	0	1	162
worst	3.6872	10.5746	.1115	1.9826	1	2	190
v^r							
best	.8927	1.5956	.0168	0.7425	1	1	78
worst	2.4616	9.6765	.1020	1.0393	1	2	166

4.7 Conclusions

These results clearly show the accuracy of the ALN predictor, albeit with some caveats. The "best" mean and standard deviation for each variable are all reasonable approximations. The interesting numbers come from the 5% Trim and the IQR. These indicate in every case, that if the problem of the outliers can be solved, the ALN prediction would be very close to the best that can be expected.

How can these outliers be eliminated or reduced? Training three or five trees and doing a majority vote has improved performance of ALNs in similar circumstances [Arms90], although the increased memory requirements (3 to 5 times) is a definite drawback. Early tests on the present problem indicate no improvement with majority vote. A better method of quantizing may help reduce the number of spurious errors. Finally, the simple expedient of not allowing any variable to change value by more than 10% per iteration should both limit the effect of the errors seen, and reduce the time taken to decode the vector of binary outputs of the trees (we only need to check 20% of the steps on the path).

5 Summary

In this paper, adaptive logic networks were shown to be capable of reasonably accurate predictions of motion using a moderate amount of memory (5 megabytes). Since the original tree size was 8192 leaves on a balanced tree, if implemented in hardware, each tree would take a maximum of 13 propagation delays to compute. It is reasonable to expect that a circuit can be built that can decode a result in twice as many cycles as the width of the codewords. Since our codewords are 23 bits long, the resulting system could take on the order of $13+23 \cdot 2 = 59$ times the speed of an **and** or **or** gate's propagation delay.

6 Future Work

The fact that large errors occur at all has led to the development of other techniques for treating continuous values. Rather than have a forest of trees where each tree calculates one bit of $y = f(x_1, x_2, \ldots, x_n)$, we are experimenting with one large tree that calculates the predicate $p(y, x_1, x_2, \ldots, x_n) = 1$ iff $y < f(x_1, x_2, \ldots, x_n)$. Inputs are still quantized, but are fed in as binary numbers which the trees treat as implicit unary (base 1) numbers. (The sequence of numbers from 0 to 3 in a 3-bit unary code are: 000, 001, 011, 111.) Given x_i, the predicate p can be used to do a binary search on y for the first transition between false and true. Results obtained since submitting this paper are very promising.

Bibliography

[Arms79] Armstrong, W. W. Gecsei, J. 1979. Adaption Algorithms for Binary Tree Networks, in *IEEE Transactions on Systems, Man, and Cybernetics*, Vol 9, no. 5, May 1979. pp 276-285.

[Arms85] Armstrong, W. W., Green, M. 1985. The Dynamics of Articulated Rigid Bodies for Purposes of Animation, *Proceedings of Graphics Interface 85*. pp 407-415.

[Arms87] Armstrong, W. W., Marsland, T. A., Olafsson, M. and Schaeffer, J. 1987. Solving equations of motion on a virtual tree machine, *SIAM J. Sci. Stat. Comput.*, vol 8, no. 1, pp. s59-s72.

[Arms90] Armstrong, W. W., Liang, J., Lin, D., Reynolds, S. 1990. *Experiments using Parsimonious Adaptive Logic*, Technical Report TR 90-30, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

[Arms91] Armstrong, W. W. et al. 1991. Learning and Generalization in Adaptive Logic Networks, *Artificial Neural Networks*, Kohonen et al (Eds), North Holland, 1991.

[Bara90] Baraff, D. 1990. Curved surfaces and Coherence for non-penetrating rigid body simulation, *Computer Graphics*, vol. 24, no. 4, 1990, pp. 19 - 28.

[Boch74] Bochmann, G. V., Armstrong, W. W. 1974. Properties of Boolean Functions with a Tree Decomposition, *BIT 14*. pp 1-13.

[Cann88] John Canny, J. 1988. *The Complexity of Robot Motion Planning*, MIT Press, Cambridge MA.

[Hech90] Hecht-Nielsen, R. 1990, *Neurocomputing*, Don Mills, Ontario: Addison-Wesley.

[Lake90] Lake, R. M. 1970. *Dynamic Motion Control of an Articulated Figure*, Masters Thesis, University of Alberta, April 1990.

[Meis90] Meisel, W. S. 1990. Parsimony in Neural Networks, *Proceedings IJCNN-90-WASH-DC*, volume 1. pp 443-446.

[Supy91] Supynuk, A. 1991. *The use of Adaptive Logic Networks as Fast Predictors of Motion*, Masters Thesis, University of Alberta, 1991.

[Wilh87] Wilhelms, J. 1987. Using dynamic analysis for realistic animation of rigid bodies, *IEEE Computer Graphics and Animation*, June 1987, pp. 12 - 27.

Code Availability

The code for the ALN predictor is available via anonymous ftp from menaik.cs.ualberta.ca [129.128.4.241] in the directory pub/predictor.tar.Z. The code for DynaTree can be found in pub/dynatree.tar.Z.