Towards The Integration Of Learning Into Mainstream Computer Programming

Sebastian Thrun
School of Computer Science
Carnegie Mellon University
http://www.cs.cmu.edu/∼thrun

Abstract

This paper argues that statistical learning from examples is not only useful in nature, but also has its firm utility in conventional software development. It presents preliminary work towards an extension of C++ that makes it possible to "train" C++ programs through examples. A mobile robot implementation suggests that the development of software using these learning tools, can be an order of magnitude faster than conventional computer programming.

Introduction

Today, the vast majority of computer software is developed by hand. To develop software, a (team of) programmer(s) analyzes the problem, and develops program code by hand to solve the problem. In recent years, an alternative paradigm has gained in importance: *learning*, or *teaching/demonstration* (Friedrich *et al.* 1996; Mitchell 1997). Here a programmer feeds a computer with examples, and the computer acquires a solution by generalizing from those. Successful fielded applications of learning for software development ranges from automatic parameter tuning (nowadays common in speech recognition (Lee 1990)) to learning entire programs from scratch (e.g., as is the case in generic programming (Koza 1992; 1994)).

Let us contrast manual programming with learning from examples. Clearly, both methodologies have advantages and shortcomings. It comes at little surprise that manual programming is by far the most effective way of software development to date. Put simply, in most domains learning from scratch is hopeless, since too many training examples would be required to learn a function of the desired complexity. However, several remarkable examples exist where teaching a computer outperformed programming by hand. One of the most striking examples is Pomerleau's ALVINN (Pomerleau 1993), an autonomous land vehicle. ALVINN learned driving by watching a human drive. It could be trained by an average human driver to reliably stay on the road in as little as 10 minutes training time, using a generalizing neural network for image analysis. Coding the same function manually required several orders of magnitude more effort, as the work by Dickmanns and colleagues has demonstrated (). This comparison illustrates that learning has the potential to play a pivotal role in certain software development tasks.

The recognition that none of two approaches contrasted here—conventional programming and learning—is uniformly superior to the other, suggests that programming frameworks powerful enough to encompass both should be better than those that support only one of them. When solutions are easily hand-coded, conventional programming should be used. When easier taught by examples, exemplar-based training should be invoked. Thus, the central thesis put forward in this paper is that new programming methodologies are needed that continue to support proven methodologies, but are powerful enough to integrate learning and training as a means of software development.

The impact of such a new programming paradigm could be dramatic. Instead of developing all code by hand,

- the costs of code development could be reduced substantially (preliminary tests suggest that an order of magnitude is not unreasonable in certain applications), and
- code could modify itself at run-time to adapt to its environment. This would enable us to develop continually improving software.

However, such a paradigm also opens new questions, for many of which we currently lack good answers

- How can we effectively search through code? How can we assign credit between the target signal and the internals the code that are to be learned?
- What are good ways of modifying code so that the result is still human-comprehensible? How can the learning algorithm explain to programmers how certain code was modified, and why?
- How can a programmer signal to the computer his level of confidence in his own (or previously learned) code, so that learning does not modify code already known to be correct?
- How can we obtain formal performance guarantees with code that has been learned from examples?

- How can one debug code that has partially been learned from examples?

Obviously, answers for many of these question are unclear, as this research is in its early stage. To investigate the viability of the basic idea advocated here, we have developed a prototype, as an extension of the popular programming language C++. The bulk of this paper describes this prototype implementation. We return to the general discussion towards its end.

The Learning Language CES

Leaving Gaps

CES is an extension of C++, developed as an attempt to smoothly integrate learning into mainstream programming. The principal idea for learning in CES is to allow programmers to leave "gaps" in the code. For example, instead of specifying the exact code that transforms a set of variables A into another set of variables B, one might instead specify that *there exists a function* and define a parameter space for this function. Searching this space in pursuit of minimizing some data-defined error function is CES's approach to learning from examples.

Currently, our implementation supports continuous function approximators with differentiable parameters, such as Back-propagation-style networks. For example, the code segment

```
fa<double, double> f(oneLayerNeuronet, 5);
double x, y;
y = f.eval(x);
```

declares a neural network f with 5 hidden units that maps a (one-dimensional) real-valued variable into another one, and subsequently applies this network to map x into y. The mnemonic `fa` is short for *function approximator.* The code above specifies that the variable y can be computed from x, but leaves open the exact specifications of this computation. Instead, it suggests that a neural network can do the job. Learning now amounts to estimating the network's parameters.

This example illustrates a specific design choice in CES: For code to be trainable from examples, it has to contain explicitly marked function approximators. Learning takes only place there. Such a representation has the advantage that it clearly separates people's code from learned code. People's code is conveniently coded in C++, whereas the learned code resides in numerical parameters. The separation makes it easier for programmers to maintain control over their software. However, what has been learned is only understandable by observing the approximator's input-output behavior (unless one tried analytical tools for dissecting function approximators like neural networks (Craven & Shavlik 1994; Thrun 1995)).

Training in CES is initiated through specific training statements. An example is given by the following code segment:

```
double x, y, z, d;
  ...
y = f(x);
  ...
z.train(d);
```

Here the value of the variable y depends on the function approximator f. Let us also assume the value of z was computed using y.

The last command invokes learning. It expresses that we desire the value of z to be d. With a poorly trained function approximator f, z might be quite different from its target value. Learning is the adjustment of f's parameters so as to minimize the deviation between the actual value of z, and its target value, by modifying the parameters of the function approximator f.

Credit Assignment

The key question is: How to assign credit, or blame, between the error in the training command, and the function approximator f. CES approaches this problem in a similar way as Back-propagation: It uses *gradient descent* to adjust its parameters (Rumelhart, Hinton, & Williams 1986). In particular, let w be a parameter of the function approximator f. In the most generic case, CES automatically computes

$$\nabla_w E \quad = \quad \frac{\partial E}{\partial w} \tag{1}$$

using the LMS error function

$$E \quad = \quad \frac{1}{2}\,(z-d)^2 \tag{2}$$

It then modifies the weight w in opposite direction of the gradient

$$w \quad \longleftarrow \quad -\alpha \nabla_w E \tag{3}$$

where $\alpha > 0$ is the familiar step size in gradient descent.

To compute $\nabla_w E$, CES has to keep track of auxiliary gradients: For example, when computing y in the code segment above, CES also computes

$$\frac{\partial y}{\partial w} \tag{4}$$

Later, when computing z using y, CES notices that y is annotated with a gradient field and computes the new gradients

$$\frac{\partial z}{\partial w} \tag{5}$$

using the chain rule of differentiation—very much as in Back-propagation, just forward in time.

When finally a training statement is encountered, the weight updates are simply obtained via the product rule of differentiation:

$$\nabla_w E \quad = \quad \frac{\partial E}{\partial z}\,\frac{\partial z}{\partial w} \tag{6}$$

While this is mathematically straightforward, the reader should not dismiss the complexity in the credit assignment mechanism. Statements in C++ can be recursive; variables can be used multiple times; they can of course be part of the condition in while-loops; and values can be influenced by a whole array of function approximators—these all introduce difficulties in computing gradients. However, all these cases are semantically straightforward. In the interest of brevity we omit a discussion of the implementation details.

On a more fundamental note, the reader should also be aware of the fact in most program code, gradients would be flatly zero; and gradient descent would be inapplicable for search in weight space. To see, consider the following routine, which contains a simple if-then-else statement:

```
double f(double x){
 double y;
 if (x > 0.5)
   y = 1.0;
 else
   y = 0.0;
 return y;
}
```

Clearly, the gradient

$$\frac{\partial f(x)}{\partial x} \tag{7}$$

is 0 almost everywhere, except if $x = 0.5$ when the (right-sided) gradient is ∞. In this light, gradient descent may appear questionable a choice for parameter estimation in CES.

Probabilistic Variables

Luckily, there is a solution. The issue of non-differentiability brings us to the second idea in CES, the concept of *probabilistic computation*. We regard this concept to be equally important as the idea of learning, but we chose not to emphasize it in this paper.

In short, probabilistic variables represent *probability density functions*. They are derived from standard data types and inherit many of their properties. For example, the declaration

$$prob < double > x; \tag{8}$$

declares a probabilistic variable x over base type `double`.

What exactly is a probabilistic variable? In contract to a conventional variable, which can only take a single value, probabilistic variables can take multiple values, each weighted by a probability. For example, the assignment

$$x = 1, 0.2, 2, 0.5, 3, 0.3; \tag{9}$$

assigns to x a discrete distribution over the space $\{1, 2, 3\}$ with $Pr(x = 1) = 0.2$, $Pr(x = 2) = 0.5$, and $Pr(x = 3) = 0.3$. Many other forms of probabilistic assignments exist, such as

$$x = UNIFORM(10, 20); \tag{10}$$

which assigns to x a uniform distribution in the interval $[10, 20]$.

The utility of probabilistic variables has been discussed in length in a recent paper (Thrun 1998). Here we refer the reader to this literature. We only briefly remark that computing with probabilistic variables is analogous to conventional computing, with the understanding that a variable may take more than just one value. Since in many embedded system applications, computers are inherently uncertain, probabilistic variable offer enhanced robustness with minimum effort on the programmer's side. Thus, probabilistic variables are worthwhile in their own right, even for languages that do not integrate learning.

In the context of this paper, however, the key benefit of probabilistic variables is their utility for credit assignment in learning. They are in fact a key enabling factor for CES's credit assignment mechanisms. This is because with probabilistic variables, CES becomes differentiable in many cases where conventional code would not be.

To see, let us return the example used above to illustrate the non-differentiability of C++ code. With probabilistic variables, the corresponding code segment reads

```
prob<double> f(prob<double> x){
 prob<double> y;
 if (x > 0.5)
   y = 1.0;
 else
   y = 0.0;
 return y;
}
```

The gradient of the procedure's output with respect to its input is given by

$$\frac{\partial Pr(f(x) = 1)}{\partial Pr(x = a)} = \begin{cases} 1 & \text{if } a > 0.5 \\ -1 & \text{if } a \leq 0.5 \end{cases} \tag{11}$$

and

$$\frac{\partial Pr(f(x) = 0)}{\partial Pr(x = a)} = \begin{cases} -1 & \text{if } a > 0.5 \\ 1 & \text{if } a \leq 0.5 \end{cases} \tag{12}$$

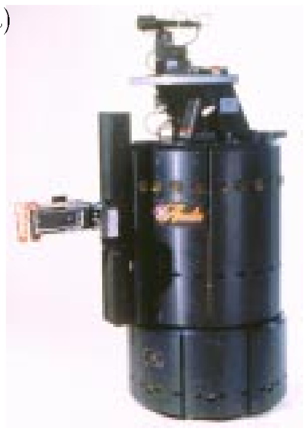
for the two legal output values, 0 and 1. Thus, the gradient does not vanish, and gradient descent leads to a modification of the weights. This brief example can be generalized to a large class of computations, as the output of a probabilistic computation is usually smooth in its input values. Thus, we have illustrated the utility of probabilistic variables in the context of the learning language extension CES.

Implementation

How important are these ideas? How good is the language extension really?

To elucidate these questions, we chose to re-implement a common reference problem in mobile robotics: a mail delivery robot. To truly investigate the utility of learning and probabilistic computation,

(a)



(b)

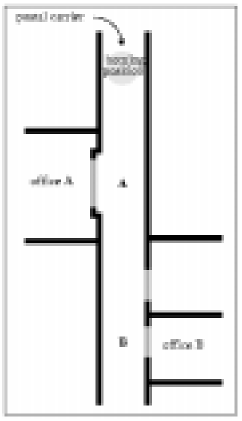


Figure 1: (a) The Real World Interface B21 robot used in our research. (b) Schematics of the robot's environment.

we programmed the robot from scratch, assuming no other infrastructure than an interface that generates sensor data (sonars, down-sampled camera images) and that accepts basic robot motion commands (motor velocities). Such interfaces are commonly found in mobile robotics.

The purpose of the implementation was to evaluate the relative advantage of a CES-like language over conventional programming languages, such as C or C++. In particular, we demanded the following functionality:

- The robot must wait in a designated parking location for a mail carrier.
- It must accept delivery commands by the carrier, communicated through visual arm gestures.
- It must reliably navigate through a populated hallway, while avoiding collisions with people and obstacles.
- It must find the delivery locations, which in our case are not recognizable from the momentary sensor readings.
- It must honk a horn in its delivery locations, and wait for person to pick up the mail.
- Finally, when all deliveries have been completed, it must return to the parking position and continue the delivery. The starting position is also not recognizable from momentary sensor readings, thus finding out that the robot is there requires careful bookkeeping.

This scenario is relatively complex. Few existing mobile robot systems are capable of providing the desired functionality. Control programs for tasks this complex usually possess in the order or 50,000 lines of code or more. In our own lab, we recently developed a large-scale software package capable of this and similar functions, with an effort of over 10 man years (Burgard *et al.* 1998) ($\sim 10^6$ lines of code). The reader should notice that our software is much more general than the scenario above. However, we believe that traditional

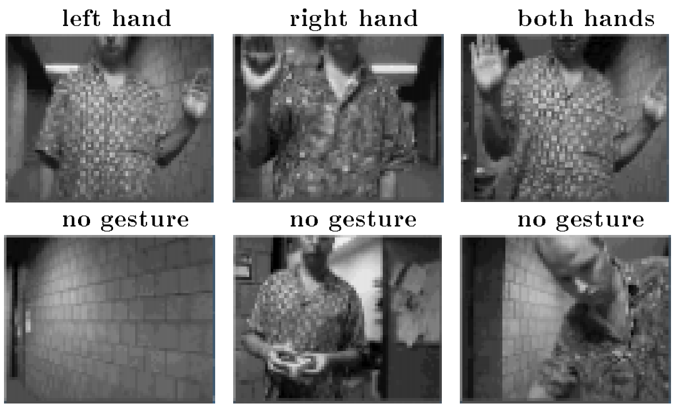


Figure 2: Positive (top row) and negative (bottom row) examples of gestures.

means of programming require significant amount of time and program code to robustly control a robot with the functionality above. Figures 1 and 2 show the robot, its environment, and examples of gestures, respectively.

Table 1 shows our final implementation in CES, capable of performing the mail delivery function robustly. As the reader quickly verifies, the entire code is only 137 lines long. Less than one day of programming time was necessary to develop this implementation. Additionally, at several points during the development the program was trained through examples, whose collection and labeling took less of 2 hours total. The program and its development is discussed in depth in (Thrun 1998).

Conclusion

This paper argues in favor of learning, or training, as an alternative means of programming computers. We briefly described a preliminary experiment using an extension of C++, called CES, that supports learning from examples. In CES, programmers can leave parametric "gaps" that are filled by specifying the desired input-output behavior of code segments through examples. A second key concept in CES, that of probabilistic variables, facilitates the credit assignment necessary to "fill these gaps." CES's built-in credit assignment mechanism uses gradient descent to translate training examples into incremental parameter updates during learning, crucially exploiting the differentiability of probabilistic variables.

Initial experiments in a credible mobile robot domain suggests a reduction of software development efforts by more than an order of magnitude. A second example was explored in (Thrun 1998), where a similar reduction was observed for programming a state-of-the-art landmark localization algorithm. While these examples are highly preliminary, they show some of the promise arising from this new programming paradigm. Based on these initial findings, we are confident that a careful integration of learning into mainstream programming can overcome significant barriers, thereby reducing costs of software development. We furthermore conjecture that additional benefits of integrating learning into program-

```
001: main(){
002:
003:   // ============= Declarations ==============
004:   fa<vector<double>,        prob<double> > netSonar(oneLayerNeuronet, 5);
005:   fa<prob<vector<double> >, prob<double> > netX(oneLayerNeuronet, 5),
006:                                            netY(oneLayerNeuronet, 5);
007:   fa<vector<double>,        prob<int> >    netLeft(oneLayerNeuronet, 5),
008:                                            netRight(oneLayerNeuronet, 5);
009:   prob<double>          alpha, alphaLocal, probRotation;
010:   prob<double>          thetaLocal, theta, transVel, rotVel;
011:   prob<double>          x, xLocal, y, yLocal, probTransl;
012:   prob<int>             coin = {{0, 0.5}, {1. 0.5}};
013:   prob<int>             gestureLeft, gestureRight;
014:   prob<vector<double >> newSonar(2);
015:   double                alphaTarget, scan[24], image[300];
016:   double                xTarget, yTarget, xGoal, yGoal, t, v;
017:   struct                { double rotation, transl; } odometryData;
018:   struct                { double x, y, dir; } stack[3];
019:   int                   targetLeft, targetRight;
020:   int                   numGoals = 0, activeGoal;
021:
022:   // ============= Initialization ==============
023:   alpha = UNIFORM1D(0.0, M_PI);
024:   theta = UNIFORM1D(0.0, M_PI);
025:   x = XHOME; y = YHOME;
026:
027:   // ============= Main Loop ==============
028:
029:   for (;;){
030:
031:     // ---------------- Localization ----------------
032:     GETSONAR(scan);                       // get sonar scan
033:     alphaLocal = netSonar.eval(scan) * M_PI;
034:     alpha = multiply(alpha, alphaLocal);
035:     probloop(alphaLocal, coin; thetaLocal){
036:       if (coin)
037:         thetaLocal = alphaLocal;
038:       else
039:         thetaLocal = alphaLocal + M_PI;
040:     }
041:     theta = multiply(theta, thetaLocal);// robot's orientation
042:     probloop(theta; newSonar){
043:       int i = int(theta / M_PI * 12.0);
044:       int j = (i + 12) % 24;
045:       if (scan[i] < 300.0) newSonar[0] = scan[i];
046:       if (scan[j] < 300.0) newSonar[1] = scan[j];
047:     }
048:     xLocal = netX.eval(newSonar);
049:     yLocal = netY.eval(newSonar);
050:     x = multiply(x, xLocal);            // robot's x coordinate
051:     y = multiply(y, yLocal);            // robot's y coordinate
052:
053:     GETODOM(&odometryData);             // get odometry data
054:     probRotation = prob<double>(odometryData.rotation)
055:        + normal1d(0.0, 0.1 * fabs(odometryData.rotation));
056:     alpha += probRotation;
057:     if (alpha <   0.0) alpha += M_PI;
058:     if (alpha >= M_PI) alpha -= M_PI;
059:     theta += probRotation;              // new orientation
060:     if (theta <  0.0)        theta += 2.0 * M_PI;
061:     if (theta >= 2.0 * M_PI) theta -= 2.0 * M_PI;
062:     theta = probtrunc(theta, 0.01);
063:     probTransl = (prob<double>) odometryData.transl
064:       + NORMAL1D(0.0, 0.1 * fabs(odometryData.transl));
065:     x = x + probTransl * cos(theta);
066:     y = y + probTransl * sin(theta);
067:     x.truncate(0.01);                   // new x coordinate
068:     y.truncate(0.01);                   // new y coordinate
069:     // ------------ Gesture Interface & Scheduler ------------
070:     GETIMAGE(image);
071:     gestureLeft  = netLeft.eval(image);
072:     gestureRight = netRight.eval(image);
073:     if (numGoals == 0){                 // wait for gesture
074:       if (double(gestureLeft) > 0.5){
075:         stack[numGoals  ].x   = XA;    // location A on stack
076:         stack[numGoals  ].y   = YA;
077:         stack[numGoals++].dir = 1.0;
078:       }
079:       if (double(gestureRight) > 0.5){
080:         stack[numGoals  ].x   = XB;    // location B on stack
081:         stack[numGoals  ].y   = YB;
082:         stack[numGoals++].dir = 1.0;
083:       }
084:       if (numGoals > 0){
085:         stack[numGoals  ].x   = XHOME; // HOME location on stack
086:         stack[numGoals  ].y   = YHOME;
087:         stack[numGoals++].dir = -1.0;
088:         activeGoal = 0;
089:       }
090:     }
091:     else if (stack[activeGoal].dir *   // reached a goal?
092:              (double(y) - stack[activeGoal].y) > 0.0){
093:       SETVEL(0, 0);                     // stop robot
094:       activeGoal = (activeGoal + 1) % depth;
095:       if (activeGoal)
096:         for (HORN(); !GETBUTTON(); );  // wait for button
097:       else
098:         numGoals = 0;
099:     }
100:
101:     else{ // ---------------- Navigation ----------------
102:       xGoal = stack[activeGoal].x;
103:       yGoal = stack[activeGoal].y;
104:       probloop(theta, x, y, xGoal, yGoal;
105:                transVel, rotVel){
106:         double thetaGoal = atan2(y - yGoal, x - xGoal);
107:         double thetaDiff = thetaGoal - theta; // location of goal
108:         if (thetaDiff < -M_PI) thetaDiff += 2.0 * M_PI;
109:         if (thetaDiff >  M_PI) thetaDiff -= 2.0 * M_PI;
110:         if (thetaDiff < 0.0)
111:           rotVel =  MAXROTVEL;         // rotate left
112:         else
113:           rotVel = -MAXROTVEL;         // rotate right
114:         if (fabs(thetaDiff) > 0.25 * M_PI)
115:           transVel = 0;                // no translation
116:         else
117:           transVel = MAXTRANSVEL;      // go ahead
118:       }
119:       v = double(rotVel);              // convert to double
120:       t = double(transVel);            // convert to double
121:       if (sonar[0] < 15.0 || sonar[23] < 15.0) t = 0.0;
122:       SETVEL(t, v);                    // set velocity
123:     }
124:
125:     // ---------------- Training ----------------
126:     GETTARGET(&alphaTarget);           // these command are
127:     alpha.train(alphaTarget);          // only enabled during
128:     GETTARGET(&xTarget);               // training. They are
129:     x.train(xTarget);                  // removed afterwards.
130:     GETTARGET(&yTarget);
131:     y.train(yTarget);
132:     GETTARGET(&targetLeft);
133:     gestureLeft.train(targetLeft);
134:     GETTARGET(&targetRight);
135:     gestureRight.train(targetRight);
136:   }
137: }
```

Table 1: The complete implementation of the mail delivery program. Line numbers have been added for the reader's convenience. Functions in capital letters (`GET_...` and `SET_...`) are part of the interface to the robot.

ming include the ability of code to adapt over its lifetime, and the reuse of program code that is almost, but not quite, right. However, these conjectures are certainly speculative, and more research is needed to substantiate the claims made in this paper.

We conclude by remarking that the vast majority of today's computers are indeed programmed tediously by hand. People and animals, in contrast, are "instructed" through much richer means, including demonstrations, explanations, environment interaction, qualitative feedback, and so on. We believe that by integrating learning into mainstream programming we can overcome existing barriers in the field of software development, thereby reducing the effort involved in software development, and facilitating the development of more robust and reusable code.

Acknowledgments

This research benefitted tremendously through discussions with Tom Mitchell, Frank Pfenning, and Sungwoo Park, which is gratefully acknowledged.

This research is sponsored by the National Science Foundation (and CAREER grant number IIS-9876136 and regular grant number IIS-9877033), and by DARPA-ATO via TACOM (contract number DAAE07-98-C-L032) and DARPA-ISO via Rome Labs (contract number F30602-98-2-0137), which is gratefully acknowledged. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of the United States Government or any of the sponsoring institutions.

References

Burgard, W.; A.B.; Cremers; Fox, D.; Hähnel, D.; Lakemeyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 1998. The interactive museum tour-guide robot. In *Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence.*

Craven, M. W., and Shavlik, J. W. 1994. Using sampling and queries to extract rules fom trained neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, 37–45. San Mateo, CA: Morgan Kaufmann.

Friedrich, H.; Münch, S.; Dillman, R.; Bocionek, S.; and Sassin, M. 1996. Robot programming by demonstration (rpd): Supporting the induction by human interaction. *Machine Learning* 23(2/3):5–46.

Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, J. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Lee, K.-F. 1990. Context-dependent phonetic hidden markov models for speaker-independent continuous speech recognition. In Waibel, A., and Lee, K.-F., eds., *Readings in Speech Recognition.* San Mateo, CA: Morgan Kaufmann Publishers, Inc. Also appeared in the IEEE Transactions on Acoustics, Speech, and Signal Processing.

Mitchell, T. 1997. *Machine Learning.* McGraw-Hill.

Pomerleau, D. 1993. *Neural Network Perception for Mobile Robot Guidance.* Boston, MA: Kluwer Academic Publishers.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning internal representations by error propagation. In Rumelhart, D. E., and McClelland, J. L., eds., *Parallel Distributed Processing. Vol. I + II.* MIT Press.

Thrun, S. 1995. Extracting rules from artificial neural networks with distributed representations. In Tesauro, G.; Touretzky, D.; and Leen, T., eds., *Advances in Neural Information Processing Systems (NIPS) 7.* Cambridge, MA: MIT Press.

Thrun, S. 1998. A framework for programming embedded systems: Initial design and results. Technical Report CMU-CS-98-142, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA.