N91-20652

An Architectural Approach to Create Self Organizing Control Systems for Practical Autonomous Robots

Helen Greiner
California Cybernetics Corporation

Introduction

For practical industrial applications, the development of trainable robots is an important and immediate objective. Therefore, we emphasize developing the type of flexible intelligence directly applicable to training. It is generally agreed upon by the AI community that the fusion of expert systems, neural networks, and conventionally programmed modules (e.g. a trajectory generator) is promising in the quest for autonomous robotic intelligence. In spite of the recent advances in all of these fields, autonomous robot development is hindered by integration and architectural problems. Some obstacles towards the construction of more general robot control systems are as follows:

1. Growth Problem- In current systems, substantial portions of the existing control software must be modified upon the addition of a new subsystem.

2. Software Generation- Currently, most software is written by people, limiting the size of code that can be created. Automatic software generation methods are premature; program writing programs are domain specific and have severe limitations.

3. Interaction with Environment- In order for the robot to properly respond to the environment, it must rely on a continuous influx of sensor data as opposed to internally stored representations. Conventional programming methods do not easily lend to massive, pipelined data processing.

4. Reliability- Most current systems are built such that single point failures cause complete system failure.

5. Resource Limitation- Current neural networks can learn most input to output functions in terms of mapping, but in case of practical problems they often take an impossibly long time to learn a function. The number of nodes or connections needed may suffer from combinatorial explosions rendering the system impossible to build.

Neural networks can be successfully applied to some of these problems. However, current implementations of neural networks are hampered by the resource limitation problem and must be trained extensively to produce computationally accurate output. Currently, there is no consensus as to the structure of an intelligent robot brain, functional break down, or interface definition. In this publication, a generalization of conventional neural nets is proposed, and an architecture is offered in an attempt to address the above problems.

Approach

The architecture that we propose consists of three components: functional groups, interfaces, and the graph describing the information flow pattern [1,2]. Each functional group performs a specific operation, and the interfaces between groups are vectors. The interconnection graph will not strongly depend on the kinematic structure of the robot. However, if a robot lacks certain sensory input, obviously the corresponding functional groups will not be present.

A functional group takes a vector as input, performs its operation, and produces an output vector. The operation of the functional group could be carried out by conventional software, hardware, or what we call a generalized network. The term generalized network describes one of the key elements in our work, and deserves detailed explanation.

A generalized network consists of two components, nodes and connections. The nodes are simply memory elements (2 byte numbers in our current implementation). The connections are able to perform mathematical operations on the node values. There is no theoretical limitation on the kind of operation that connections can perform or the number of inputs and outputs that they have (currently 16 bytes are being used). For example, a PID control servo could be a connection, where the inputs are the position setpoint and gain and the output is the commanded motor current. This method developed from a practical standpoint, to fuse advantageous properties of neural nets and table driven software. The programming is simplified because the bulk of the coding is done when the subroutine for the connection is developed. During training or operation the gains might change or connections may be created or destroyed, but this activity does not carry the risk of catastrophic software malfunction. If the task of a functional group is recognition of a situation present in sensory inputs, this group will use connections designed to best perform this task.

The architecture of the robot is defined in a hierarchical, bottom up manner, and training also occurs in this order. Each functional group is independently trained, and uses locally available information (observation of input and output vectors) to improve its behavior.

To illustrate how training occurs, we will take the example of lowest level motor control (see Figure 1). For this purpose, the sensor inputs that are directly related to motor action are separated from the rest of the sensors, and a new vector is created. A functional group is defined whose output directly drives the motors and the inputs are as follows:

- sensor vector being controlled
- a vector marking which sensor

readings should be affected
- a vector of desired sensor readings

This functional group could be realized using conventional software, if the effect of motor action is fully known to the programmer. In this case, the functional group would consist of a number of PID servos that are surrounded by conditional branches such that the servo computation is skipped if the particular sensor does not need to be affected (the enable vector). The gains in these PID servo loops would be computed based on a model of the system and modified based on observed performance. An alternative approach is to use a generalized network to carry out this control function. The tuning of the gains is automatic based on the connection's observation of the response. Assuming that the generalized network is simulated in software, the difference between it and the original software implementation is very subtle. The generalized net looks like table driven software. Later, when a custom processor is built the connection operations will be processed in parallel, making the difference more pronounced.

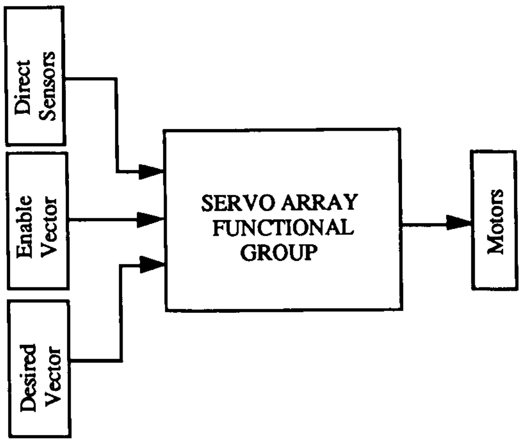

Figure 1 - Motor Servo

The advantage of using a generalized network in this instance is the relative ease of writing a list of connections. It can be seen that even this simple function of servoing low level sensor readings can be improved by various techniques that require progressively more and more computational resources. These functions can be added by adding more connections to the array.

The input to the motor servo array consists of three vectors: the direct sensor readings, the enable vector, and the desired vector. The direct sensor readings are inputs from the environment. The input nodes do not have to be physical sensor readings, nodes can be added purely to simplify later calculations. For example, in order to be able to move the tip of a robot leg along a straight trajectory in cartesian space, a new sensor node describing the x coordinate of the tip is added to the inputs. This node is calculated by conventional forward kinematic software. This is an excellent example of integrating conventional software with generalized networks. The enable vector turns individual servos on and off. This prevents servoing motors when they are not needed and can prevent two competing servos from being simultaneously active. The desired vector is a command to the motor servo group from a higher level. The objective of the motor servo group is to make the direct sensor reading as close to the desired sensor readings as possible.

The next higher level functional group is the "activity group" (see Figure 2). This group will be described in detail because it contains many elements not present in our previous example, and it has features that reappear in the higher levels. The interface between this group and the motor servo group is the desired and enable vectors which have previously been described. The input to this functional group is a vector of activities (for example, the nodes of this vector may include walking, standing, or returning to the home position), and a vector of sensory readings. The lines into and out of the activity functional group may be misleading, they in fact represent a matrix of tunable connections. The functional array contains internal nodes which all have some physical interpretation. The internal vectors are also tied together by matrices of connections. The three internal vectors used in this example are the situation vector, the vector of possible motions, and the robot motion vector. The situation vector contains nodes corresponding to certain combinations of environmental conditions. It is connected to the sensor values. A unique feature of this vector is that the nodes are competitive [3]. Strong activation of one node will inhibit activation of the others. Thus, the robot generalizes situations because a partial match of environmental conditions can cause the correct node to dominate. The next vector, the possible motions vector, contains nodes for each action such as move leg 1 up or rotate body about yaw axis. Each node is active only if the motion is possible given the current state of the robot. This prevents situations such as driving a leg while it is against a joint stop or picking up a leg when the robot's weight is on it. The last internal vector describes what motion the robot should take. Examples of nodes on this vector would be pick up leg or rotate robot body. From this vector the transformation to the desired and enable vector is straightforward.

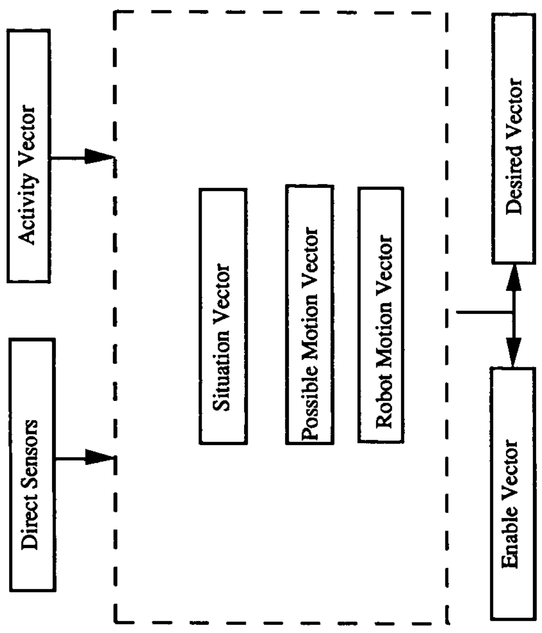

Figure 2 - Activity Functional Group

When the robot is first activated, all the connections are present. Training is a matter of the robot connections being modified to produce the correct response. Unnecessary connections are eliminated to save resources. The robot could be trained by producing random motions and seeing if any produce the correct result. However, since we know what the output vectors should be for a certain activity, another vector called a hunch is introduced. Using the hunch the robot's connections will be tuned. For example, to train the robot to walk, the node on the activity vector corresponding to walking is activated. The first hunch will activate the robot motion vector such that one leg moves forward (note: this simplified example ignores other motions that might be need to walk such as shifting body weight). The sensors at this time have caused a specific situation vector. Now unless inhibited by the possible motion vector, the connections between the active nodes on the situation and the robot motion vector will be strengthened. The next hunch may be to move one of the other legs. Again, the connections between the new situation and the motion of this leg are strengthened. This process is repeated for all the legs until if the robot is in walking mode, it has been trained what action to take given the current state of the robot. This is more valuable than simply programming the robot to move the legs sequentially because the robots actions are a function of the situation it is most nearly in.

Higher level functional groups can be added to this architecture. For example, the next level may be a "task group" in which the objective is to retrieve an object or follow a person. It is at this level that the robots begin to be useful. The bottom up approach to training of each functional group allows the higher levels to use the capabilities of the lower levels. An important point is that any improvement or additions to the lower levels improve the performance of the upper levels and don't necessitate retraining each level.

What has been described so far is one extreme of a wide spectrum of learning methods. Namely, fully hunch based learning. Learning in an intelligent system could take place totally autonomously, without the assistance of hunches. In a real learning situation, for a robot to be useful it has to simultaneously use all possible sources of information, and all beneficial learning methods. The following example will demonstrate non hunch based learning and simultaneously it will show one possible implementation of an interface between layers that facilitates smooth transition from higher level control to low level automatic execution of a task. In this learning scheme instead of behaving according to hunches the objective of learning is to maximize a scalar function called the objective function. It is assumed that the computation of this function is much simpler than carrying out the actual task. This function is either programmed into the robot by hand or somehow communicated to it. The robot control architecture generates learning as described above. To learn how to execute the task the control system has to build a list of which is the best action for every situation. The difference from the earlier case is that there is no hunch input which directly facilitates the selection of the appropriate action. The only clue as to which action is best to take is the change in the objective function. It is clearly not adequate to locally maximize the objective function with every action since several neutral or slightly adverse actions may have to be executed in a sequence before progress is made. The proposed scheme allows the robot to develop a strategy for acheiving the biggest increase in the objective function in as short a time as possible. To do this the robot builds a knowledge base that describes the consequence of its actions. This means that for every situation and every action in that situation, the robot has a prediction about what situation it will get into. Initially this data base is totally empty and the robot builds it by registering the actual sequences of situations that took place and the actions that cause them. Two distinct types of behavior are possible with this representation: goal oriented behavior and exploratory behavior. When displaying exploratory behavior the robot will try different action in situations that it has already encountered just to see the effect. On the other hand, when displaying goal oriented behavior, the robot will only chose actions which have been tried to maximize the object function as efficiently as possible. In the implementation of such a system there are two layers, reflexive and strategic. Initially, the reflexive layer is programmed with individual actions that are terminated by special situations that make the action impossible. For example, leg forward motion is terminated when the leg hits its joint limit or an obstacle. When the current motion is terminated, the reflexive layer goes idle. Detecting the idle condition the strategic layer evaluates the longterm consequence of each possible subsequent action, and choses the one deemed best in terms of the current behavior pattern (exploratory or goal oriented). Learning takes place simultaneously in both layers. The reflexive layer tries to guess what action the strategic layer will chose next. A database contains the accuracy of such guess for every situation. If the accuracy is high enough the reflexive layer will take the next action automatically (i.e. it never goes idle). In such a case the strategic layer is not involved. Learning in the strategic layer takes place by the continuous improvement of the situation action consequence database.

Conclusion

There are many advantages to creating a trainable architecture. In the introduction, obstacles towards creating a more general robot control system were listed. Now, we briefly describe how this architecture addresses these issues:

1. Growth Problem- Adding a new subsystem only effects the immediate functional group and expands it's capabilities. Addition of new sensors merely increases the number of connections in the functional array.

2. Software Generation- Software is not required to extend capabilities. Capabilities grow through training.

3. Interaction with Environment- Applicable sensory information is available at all levels of the system and the robot's action always depends on the current situation.

4. Reliability- In case of e.g. sensor failure, relevant situations are still recognized based on other sensor readings. If enhanced internal reliability is desired, the number of nodes and connections being used can be arbitrarily increased limited only by resource availability.

5. Resource Limitation- After training, the number of interconnections is reduced from O(nXn) to O(n). The connections so freed up can be reused to support learning elsewhere in the system.

We recognize that intelligent robots are a long way from being fully developed. However, practical autonomous robots can be constructed with existing technology.

References

[1] William P. Coleman et al. Modularity of Neural Network Architecture. IJCNN-90-Wash-DC, Lawrence Erlbaum Associates, Inc., 365 Broadway, Hillsdale NJ 07642, 1990.

[2] DARPA Neural Network Study. AFCEA International Press, 4400 Fair Lakes Court, Fairfax VA 22033 USA, 1990.

[3] Rumelhart, D., Hinton, G. and Williams, R. (1986). *Parallel Distributed Processing*, Vol I Cambridge, MA: MIT Press.