The Robot Builder

Volume Eleven Number Eight August 1999

Notices

- Introductory Mobile Robotics Class - 10:00AM - 12:00PM
- Business Meeting - 12:30 - 1:00
- General Meeting - 1:00 - 3:00

Distribution

If you would like to receive The Robot Builder via e-mail, contact the editor at:

apendragn@earthlink.net

Inside this Issue

Introduction to Neural Nets
Part 2

by Arthur Ed LeBouthillier

WARNING - The attorney general has concluded that this article contains mathematics that may be hazardous to your health.

In our last installment (The Robot Builder, July 1999), we introduced the basic neural net model and showed how it might be implemented in a program. In this article we will review a common learning algorithm that would modify the weights in the program provided last month.

How do we teach a neural net?

We know that the intelligence in a neural net is embodied in the connections between cells and the weights between them. In the multi-layer perceptron (MLP), the cells are connected in layers so that the input layer feeds into a hidden layer and the hidden layer feeds into the output layer. Since the MLP defines the connection structure, we are left with only determining the weights between the cells of one layer and the next. Once the proper weights are determined, we no longer need the training algorithm and we use the program described last month to use the neural net for a useful purpose.

So how do we do it? We will modify the weights of the neurons so that those neurons that properly determine the output are enhanced and those that generate the wrong output will be inhibited in accordance with the degree of the change they produce in the result. Our job will entail telling the neural net, through the learning algorithm, what the proper output should be for the given input; we must do this for all of the desired training samples. After modifying the network appropriately, the network will then be trained to provide the proper output for each of the provided inputs.

If we concentrated on what the output should be and then, going backwards towards the input, modified each layer's weights so that they produced the proper answer with the training sample, then we would have the correct answer for one training sample. We only need to do this for all samples until the difference between the desired output and the actual output is small. This is the basis of the Backpropagation learning algorithm.

The Backpropagation Algorithm

The backpropagation learning algorithm works by propagating the error in the output layer backwards towards the hidden layer and then modifies the hidden layer weights so that they reflect more accurately what the output should be for its input. The weights are only modified by the amount of change they produce in the output. It repeats this process for all test samples until the network is trained. There are complicated mathematical explanations for how this works, but we won't cover that. Suffice it to say that the algorithm treats the *n* inputs of the input layer as an *n*-dimensional surface and then heads "down hill" on that surface until the output error is minimized. It continuously modifies

(see Neural Net on page 5)

The New Millennium Remote Agent Architecture

by Arthur Ed LeBouthillier

In the movie 2001: A Space Odyssey, the spacecraft embodying HAL was essentially a large robot. It was composed of an artificial intelligent component, HAL, and the spaceship itself with all of its automatic effector systems. HAL, although a fictional program, represented an entity which was aware both of itself and events surrounding it. When there were hardware problems in the spacecraft, HAL was able to identify these problems, solve them him- or it-self or alert the crew to solve them. With its New Millennium Remote Agent (NMRA), NASA is heading in the direction of providing similar capabilities for future autonomous spacecraft.

The New Millennium Remote Agent was designed as an autonomous control and planning system enabling autonomous robotic spacecraft for space exploration. It eventually flew on the Deep Space One spacecraft and successfully demonstrated its capabilities in several tests, some of which are still ongoing. Such highly autonomous systems are vital for future space exploration because they reduce the cost of exploration while enhancing capabilities. They reduce costs because they require fewer ground-based operators, who would be required for the duration of missions which could last years, and because they allow the spacecraft to be more robust to failure because they are able to continue operating even after system failures. They are also important because they allow more autonomous spacecraft to operate at once because they require lower bandwidth to control; they do not require minute instructions for the every operation but can be given more abstract commands which are executed by the spacecraft. This is deemed to be important in the years to come because of limitations in the bandwidth of the Deep Space Network at the same time that the number of operations is expected to increase dramatically. As roboticists, it is important to review the New Millennium Remote Agent (NMRA) architecture since it provides an example of an advanced robot control architecture from which we can learn.

Deep Space One

The Deep Space One spacecraft, although utilizing many cutting-edge technologies, represents a typical spacecraft in its functional layout. It has a control system based on a radiation-hardened version of the IBM 6000 RISC processor with 128 Megabytes of RAM and 16 Megabytes of EEPROM. It has many interacting systems such as an unconventional ion thruster system, a conventional attitude control system, several scientific instruments, a communication system

(see Remote Agent on page 3)

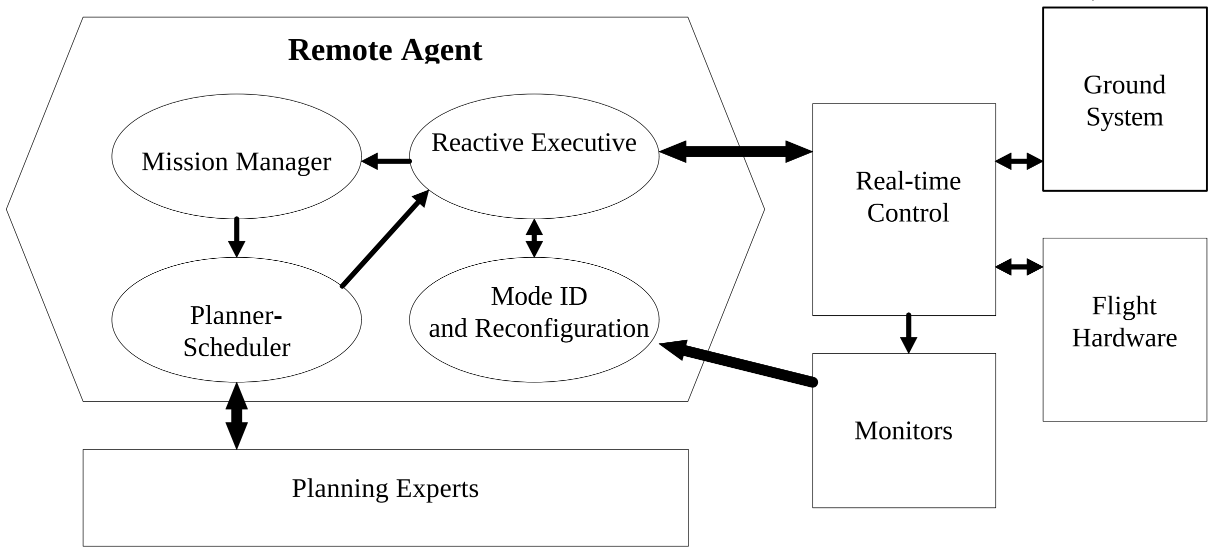

Figure 1 - The Remote Agent embedded in Deep Space One's Flight Software

Remote Agent from Page 2

which maintains communications with Earth-based controllers and several navigation sensors to identify its orientation and location in space. All of these components operate under strict energy budgets because all power is derived from solar cells or on-board fuel. Almost all systems have a minor amount of functional redundancy allowing reduced operation should some part of them fail. Operating the system despite failures is one of the major jobs of the Remote Agent.

Running on the spacecraft's computer is the Vx-Works Real-time Operating system. Operating in this operating environment are a number of standard software components used for controlling the spacecraft's attitude and thrust systems and the Remote Agent. The Deep Space One spacecraft was not designed to work solely with the Remote Agent, so it maintains a complete operating suite of software for navigation as well as navigational control.

The Remote Agent

The Remote Agent interfaces to the spacecraft control system similarly to how the standard Earth-based control system. Rather than receive direct motor-control messages from Earth, the Remote Agent deduces its activity from the goal database and performs the control duties. It essentially becomes an independent onboard mission control system to navigate and control the spacecraft in accordance with its mission directives. These mission directives can be uploaded from Earth. This saves an enormous amount of communication bandwidth since high-level abstract commands are sent instead of raw control data.

The Remote Agent (RA) consists of three major functional elements distinct from the normal control software of the Deep Space One spacecraft: the Mission Manager which performs planning and scheduling, the Reactive Executive which carries out the direct control tasks and the Mode Identification and Reconfiguration (MIR) system which performs model-based reasoning about the condition of the spacecraft, allowing it to reconfigure and control itself despite numerous faults. It is this last component that makes the Remote Agent somewhat unique in robotic control systems. The MIR system allows the RA to reason about the status of various systems and reconfigure its reactive mechanisms to continue working despite system failures.

The Mission Manager

The Mission Manager is responsible for creating short-term plans based on long-term mission goals. It does this with a Planner/Scheduler (PS) which is able to reason about time and resource constraints and generate a flexible time-constrained plan for execution by the Reactive Executive. The Remote Agent is not launched with a detailed list of operations to occur at specific times, but is given a list of goals from which a sequence of commands must be generated by the Mission Manager. The Mission Manager determines the goals which have to be achieved over a period of a week or two and passes them to its Planner/Scheduler. The plan produced by the Planner/Scheduler constrains the types of activities that must be performed at specified times but does not detail how those activities will be carried out. They state such things as requirements for star measurements to establish the spacecraft location and a rough time at which they are to occur, when engines are to be turned on for desired orientations and directions, and what activities scientific instruments should engage in. The plans are not specific step-by-step action lists, but rather represent behavior envelopes for the Reactive Executive. The planner does this using fairly-standard AI reasoning and planning techniques which consider time constraints, goal priorities and resource limitations. Action planning and resource allocation are considered simultaneously in the generation of the plans so that considerations of the effects of actions on resources can be part of the planning process. A final part of each plan details the next time that planning should occur. This allows the Mission Manager/Planner Scheduler to be shut down when it is not needed.

The Reactive Executive

It is the job of the Reactive Executive to take the plans produced by the Mission Manager and turn

(see Remote Agent on page 4)

Remote Agent from Page 3

them into specific instrument control activities. It performs process synchronization, process dependency management, hardware reconfiguration, and runtime resource management and executes fault-recovery procedures. It is able to execute and manage multiple activities simultaneously and invokes the Mission Manager's planner and the Mode Identification facilities to help it perform these duties. The Reactive Executive generates the specific control signals that control the spacecraft hardware by taking into account its knowledge of the state of various instruments and devices (i.e. current status and known problems of hardware), the goals it has for them, and the rough times activities should occur. When failures occur in the execution of a process, the executive first invokes the MIR system to attempt recovery and then can invoke the planner to generate new plans.

The Reactive Executive is based on a classical reactive execution system called RAPS. This event and goal-driven system helps ensure quick reaction loops by limiting deductive reasoning in the execution of tasks. However, this limits the ability of the executive to handle complicated problems. To handle this, the Reactive Executive requests problem solutions from a reasoning system called Livingstone which monitors system status and informs the executive of deduced solutions. Working in conjunction with Livingstone, the executive is able to robustly react to unforeseen problems by choosing an alternative execution behavior. This ability to quickly reconfigure in the face of problems makes the executive extremely robust.

Livingstone, the Mode Identification and Reconfiguration System

One of the most unique features of the Remote Agent is its ability to resolve system problems through a model-based reasoning system. Livingstone, also known as the Mode Identification and Reconfiguration (MIR) system, eavesdrops on command sent by the executive to the hardware, monitors sensor signals and deduces the current actual configuration of spacecraft systems; it then reports the actual status of the spacecraft to the executive. Livingstone maintains a complex model of all hardware systems and their internal states and keeps track of state changes and commands. In a spacecraft, weight restrictions of do not allow sensing all system conditions and so some conditions can only be deduced indirectly from their effects on other sensors.

One of Livingstone's job is to deduce the status of non-sensored conditions based on its system model and senses. It does this by maintaining state models of each piece of equipment and their interactions and deducing that certain sensory data implies a certain system state. If this state is different from expected states, it reports these anomalies to the executive. Another major job of Livingstone's is to help the executive solve problems. If the executive is informed of a problem and doesn't have an immediate solution to work around it, it will request a solution from Livingstone. Livingstone will reason about the various pieces of equipment, their status and interactions and try to deduce a new set of actions that would lead to a solution of a problem.

Livingstone does its job by an extensive concurrent state-machine model of all hardware. Each hardware device is modeled as a state machine and all interactions between machines are observed and modeled. This model is able to identify the current actual mode, identify anomalous behaviors, and the models can be used to deduce solutions to identified problems. For example, if a thrust engine valve appears stuck in the off position, Livingstone could recommend alternate valve configurations to produce the desired thrust; if a particular device does not work because it is appears stuck, Livingstone might suggest that the executive try again. This is a powerful capability which enables the Deep Space One spacecraft to identify and overcome temporary and permanent malfunctions in its equipment.

Summary

The Remote Agent represents the state of the art in autonomous spacecraft systems which demonstrates principles and techniques which can be applied to earth-bound robotic systems.

Neural Net from Page 1
the weights until the total output error for the training set is less than a given threshold.

```
Repeat
  total_error = 0
  For each training-sample input, x
    1) generate the actual output of the neural  net, y,  for this
       training sample, x, with the current weights.
    2) generate the desired output of the neural  net, t.
    3) Compute the output layer change
      a) total_error = total_error + absolute(output_layer_change)
    4) Modify the output layer weights
    5) Compute the hidden layer change
      a) total_error = total_error + absolute(hidden_layer_change)
    6) Modify the hidden layer weights next training-sample
while total_error > threshold
```

Figure 1 - the backpropagation learning algorithm

As shown in figure 1, the algorithm is simple on its surface. The problem is that each of the steps to determine the adjustment and modifying the weights gets a little bit complicated. We'll review each of these steps. Prior to training, you must pre-initialize all weights to small random values. This assures that the network doesn't get lost in a never-ending loop.

Computing the Output Layer Change

The change for any output layer cell, *j*, is computed by the formula:

```
for j = 0 to number_of_output_layer_cells
        output_delta[j] = y[j] * (1-y[j]) * (t[j] - y[j])
        total_error = total_error + abs(output_delta(j))
next j
```

Figure 2 - Computing the output layer error

This routine says that the amount of modification, *output_delta*, for output layer cell, *j*, is equal to the actual output times one minus the actual output times the difference between the desired output and the actual output. Additionally, we accumulate the amount of change for each cell in the *total_error* variable.

Computing the Output Layer Weight Change

Now that we know the amount of change that is required for each output cell, we go ahead and modify the weights for each cell accordingly. The formula in figure 3 shows how this is done.

What this says is that for each output layer cell, we modify the weight between that cell, j , and the hidden layer cell, i. We do that by multiplying the output of the hidden layer times the amount of change needed for the output layer, output_delta(j), and add that to the current value of the weight, OutputWeight[i,j]. The value Nu is a learning rate constant and determines how quickly the algorithm learns; you will see it again in modifying the hidden layer weights. It is usually a value like 0.8 in a range of 0 to 1 indicating that weights should be modified at 80 percent of the error.

```
for i = 0 to number_of_hidden_layer_cells
        for j = 0 to number_of_output_layer_cells
                OutputWeight[i, j] = OutputWeight[i, j] +
        Nu * output_delta( j)  * HiddenLayer(i)
        next j
next I
```

Figure 3 - Changing the Output layer weights

Computing the Hidden Layer Change

Computing the weight change for the hidden layer is a little more difficult since we first have to figure out how a given hidden layer cell effects the output layer and only modify it by that much. The value, *sum1*, is used to figure out the amount of modification for a given hidden layer cell by determining its "efficacy" or amount of effect on the output layer.

```
For j = 0 To number_of_hidden_layer_cells
  sum1 = 0
  For m = 0 To number_of_output_layer_cells
    sum1 = sum1 + output_delta(m) *  OutputWeight(j, m)
  Next m
  hidden_delta(j) = HiddenLayer(j) * (1 - HiddenLayer(j))  * sum1
  total_error = total_error + abs(hidden_delta(j))
Next j
```

Figure 4 - Determining the Hidden Layer Change

As figure 4 shows, we go through each of the hidden layer cells and figure out its efficacy on each cell of the output layer. We then use that to determine the amount of change, *hidden_delta(j)*, for each of the hidden layer cells. Again, we sum up the amount of change for each cell in the value, *total_error*, to determine when we are close to a solution.

Computing the Hidden Layer Weight Change

The last step in the algorithm is to modify the hidden layer weights. This is done identically to the method used for the output layer weights. Figure 5 shows how this is done.

(see Neural Net on page 6)

```
For i = 0 To number_of_input_layer_cells
  For j = 0 To number_of_hidden_layer_cells
    HiddenWeight[i, j] = HiddenWeight[i, j]
             + Nu * hidden_delta[j] * InputLayer[I]
  Next j
Next i
```

Figure 5 - Modifying the hidden layer weights

Again, we modify each weight to be what it previously was plus the learning constant, *Nu*, times the amount of change required for the hidden layer cell times the input layer value.

An implementation of the algorithm

The author used the above algorithm in a small neural net which was able to learn the binary value of the images of the numerals 0 through 9. Each numeral was input into a 35 cell input layer organized in a 5 x 7 matrix. The input layer mapped into 10 hidden layer cells and then the hidden layer cells mapped to 4 output layer cells. By inputting a given numeric character into the 5 x 7 input matrix, the network produced the four-bit binary representation at the output cells.

The network was trained on all 10 numerals, 0 - 9. It took about 1 minute for the error to settle below the threshold for any training session. After training, the network reliably recognized characters even in the presence of upwards of 15 percent noise. In some cases, the input numeral was unidentifiable by the author but the network correctly identified it. Figure 6 illustrates the structure of the network in this application and figure 7 shows the complete learning algorithm used.

```
‘ Backpropagation Learning algorithm
Nu = 0.8
Threshold = 0.1

Repeat
  total_error = 0

  for sample = 0 to number_of_training_samples
    ‘ generate the correct answer for this sample
     t = generate_t(sample)

    ‘ Determine actual answer of network with current weights
     y = generate_y(sample)

    ‘ Determine output layer change
    for j = 0 to number_of_output_layer_cells
      output_delta[j] = y[j] * (1-y[j]) * (t[j] - y[j])
      total_error = total_error + abs(output_delta(j))
    next j

    ‘ Modify output layer weights
    for i = 0 to number_of_hidden_layer_cells
      for j = 0 to number_of_output_layer_cells
        OutputWeight[i, j] = OutputWeight[i, j] + Nu * output_delta( j)  *
             HiddenLayer(i)
      next j
    next I

    ‘ Determine Hidden layer change
    For j = 0 To number_of_hidden_layer_cells
      sum1 = 0
      For  m = 0 To number_of_output_layer_cells
        sum1 = sum1 + output_delta(m) *  OutputWeight(j, m)
      Next m
      hidden_delta(j) = HiddenLayer(j) * (1 - HiddenLayer(j))  * sum1
      total_error = total_error + abs(hidden_delta(j))
    Next j

    ‘ Modify hidden layer weights
     For i = 0 To number_of_input_layer_cells
      For j = 0 To number_of_hidden_layer_cells
        HiddenWeight[i, j] = HiddenWeight[i, j]
             + Nu * hidden_delta[j] * InputLayer[I]
      Next j
    Next I
  Next sample
while total_error > threshold
```

Figure 7 - A complete backpropagation training algorithm

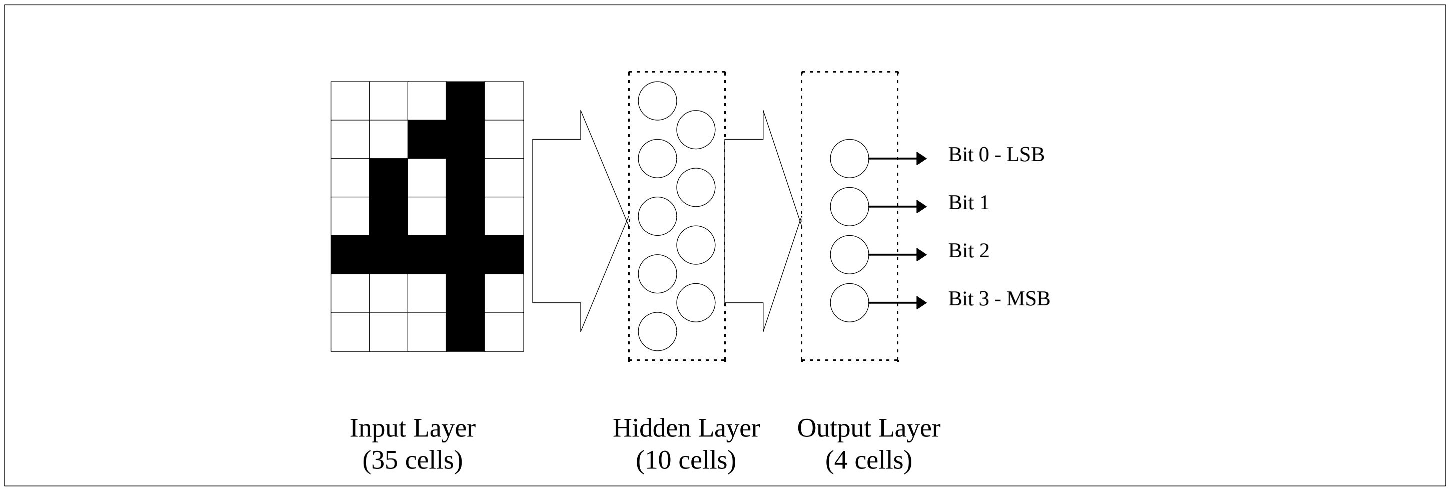

Figure 6 - A character recognizing neural net

Robotics Society of Southern California

The Robot Builder (TRB) is published monthly by the Robotics Society of Southern California. Membership in the Society is $20.00 per annum and includes a subscription to this newsletter.

Membership applications should be directed to:

Robotics Society of Southern California
Post Office Box 26044
Santa Ana, CA 92799-6044

Manuscripts, drawings and other materials submitted for publication that are to be returned must be accompanied by a stamped, self-addressed envelope or container. However, RSSC is not responsible for unsolicited material.

We accept a wide variety of electronic formats but if you are not sure, submit material in ascii or on paper. Electronic copy should be sent to:

apendragn@earthlink.net

Arthur Ed LeBouthillier - editor

The Robotics Society of Southern California was founded in 1989 as a non-profit experimental robotics group. The goal was to establish a cooperative association among related industries, educational institutions, professionals and particularly robot enthusiasts. Membership in the society is open to all with an interest in this exciting field.

The primary goal of the society is to promote public awareness of the field of experimental robotics and encourage the development of personal and home based robots.

We meet the 2nd Saturday of each month at California State University at Fullerton in the electrical engineering building room EE321, from 12:30 until 3:00.

The RSSC publishes this monthly newsletter, The Robot Builder, that discusses various Society activities, robot construction projects, and other information of interest to its members.

Membership/Renewal Application

Name ______________________________

Address ______________________________

City ______________________________

Home Phone ( ) - Work Phone ( ) -

Annual Membership Dues: ($20) Check #
(includes subscription to The Robot Builder)

Return to: RSSC
POB 26044
Santa Ana CA 92799-6044

How did you hear about RSSC? ______________________________

RSSC
POB 26044
Santa Ana CA 92799-6044

Please check your address label to be sure your subscription will not expire!