

chapter twelve

Case Study Summaries

This chapter consists of summaries of case studies. Each summary includes the name of the case study as it appeared in the literature and one or more references that will guide you to more information if desired. We describe the case study and explain why we believe it to be significant. Its significance may be due to contributions in one or more of several areas, which include data representation, preprocessing, algorithm development, system structure, and post-processing. The case study summaries vary considerably in length, primarily according to what (and how many areas) are emphasized.

By reading these case study summaries, you will be able to learn lessons, or “tricks of the trade,” that generally are beyond the scope of the material presented in the first eleven chapters of the book. The studies can also provide a deeper understanding of how to apply the concepts, paradigms, and

implementations in the book to practical problems.

Although the case studies may be left until the end of a course on computational intelligence (they are, we recognize, at the end of the book), we feel it may be better to use them at appropriate times during the course. For example, we often present the EEG case study right after we’ve finished the chapter on neural network implementations.

This chapter includes four case study summaries as the book is going to press. Because the chapter appears on the book’s web site, we will be able to update the existing case studies, and add new ones. We are soliciting additional case study summaries from our readers. If you have a case study summary you would like to have considered for the book’s web site, please contact one of the authors at eberhart@ieee.org or shi@ieee.org. ■

Case Study Preview

The first case study involves the development of a system to detect spike waveforms in an electroencephalogram (EEG). A team of physicians and engineers worked together on many aspects of the problem, including the development of the fitness measures for the system. A key to success in this neural network-based approach was finding the appropriate data preprocessing strategy.

The second case study describes the development of a state-of-charge system for an electric vehicle battery pack. (The state-of-charge measurement for an electric vehicle is like the gas gauge reading in a gasoline-powered automobile.) An unusual variation of neural network training was used, and the key to success was in selecting the appropriate output PE activation function. Minimizing the number of inputs so as to render the solution inexpensive to implement was important, as was post-processing.

The third case study focuses on scheduling. Two examples are used: a facility scheduling system and a transportation resource scheduling system. General issues with scheduling systems are discussed. Methodologies employed include evolutionary algorithms and fuzzy expert systems. Problem representation was a significant issue in this case study, as was the agreement on fitness functions.

The fourth case study focuses on control system design. For our example system, we use the popular cart centering problem. A neural network approach to control system design is applied to the problem first, followed by a fuzzy logic approach that is augmented with evolutionary computation. This case study summary illustrates the fact that it is often advantageous to consider several approaches to solve a problem.

Problem Characterization and Fitness Function Definition

The authors and their colleagues have decades of combined experience in solving real, practical engineering problems. In our experience, two important areas must be addressed in every project: characterization of the problem, and definition of the fitness function. Together, these two areas often account for over one-half of the effort needed to successfully complete a project. They must be addressed whether you choose a computational intelligence methodology or select a more traditional approach. Although it is beyond the scope of this book to go into either area in detail, we would be remiss if we did not alert you to them.

Problem Characterization

Problem characterization involves selection and preprocessing of data so that it is in a form appropriate for the analysis tool (implementation) that you plan to utilize. It is frequently an iterative process. Seldom if ever is raw data in a form ready to be used as input to an analysis tool such as a neural network or evolutionary algorithm.

Furthermore, during the early stages of a project, you usually do not know which tool, or combination of tools, you will eventually choose.

Some of the more obvious things that must be done are to inspect the data for missing elements, noise, and outliers. You and your end user must agree on how to handle these, and other, issues.

Often more difficult are the questions of how to format the data in order to make it usable by a certain tool. In one project, we found it infeasible to characterize the problem data in a way that could be effectively manipulated by the particle swarm optimization algorithm. We just could not figure out how to do it. It was relatively easy, however, to put the data into a format that could be used by a genetic algorithm (GA). It was also relatively easy to format the data as input to a fuzzy expert system. We ended up getting the best results from a GA. (However, we needed to use a very high mutation rate at the end of the run to avoid local optima.)

In summary, we often find that problem characterization is the most difficult part of finding a solution. It seems to us that “If you can characterize the problem, you can solve it.” In our experience, the failure of a project is more often due to improper problem characterization than to the selection of an inappropriate algorithm.

Fitness Function Definition

In every project, we must define how we measure system performance. (We discussed some performance metrics in Chapter 10, but they are only a small subset of those available.) How do we measure how well the system is doing? How do we know when we have achieved success? These questions may be straightforward, but they are anything but trivial. It is up to the system user, your sponsor, to define the performance criteria for the system. It is not up to you, the developer. It is important, however, that you be prepared to guide the process and to help the sponsor think about what is needed.

In our experience, it is not unusual for the sponsor to have only a vague idea of how to measure the effectiveness of a system. One of our experiences involved a logistics system in which the sponsor wanted as many items as possible delivered “on time.” It took six half-day meetings to resolve such seemingly simple issues as

1. What does “on time” mean?
2. What is the penalty function for delivering a shipment early?
3. What is the penalty function for delivering a shipment late?
4. What are the relative priorities for various types of items?
5. Etc., etc.

We believe that it is a good idea to allocate a significant amount of time in the early stages of each project for meetings with the sponsor to discuss, define, and refine the metrics by which the success or failure of the project will be determined.

Only in this way will misunderstandings and finger-pointing be avoided at the project's conclusion.

Case Study 1: Detection of Electroencephalogram Spikes

This case study involves the application of neural network tools to identify spikes (also known as epileptiform discharges, or EDs) in a multichannel electroencephalogram (EEG) signal. It is adapted from Chapter 10 in Eberhart and Dobbins (1990). Additional information can be found in Eberhart et al. (1989) and in Webber et al. (1994).

The problem of EEG spike detection is probably analogous to many interdisciplinary problems that must be solved by teams of engineers, programmers, scientists, physicians, and so on. In approaching such problems it is usually impossible for each member of the team to understand all the details of each aspect or discipline. For example, an engineer or computer scientist cannot understand all of the ramifications of the medical aspects of electroencephalography (the recording of the brain's bioelectric potential patterns). Likewise, it is generally a waste of time for physicians to learn all about implementing computational intelligence tools.

It is important, however, to keep priorities straight. For example, in this case, as in all biomedical applications, it is very important to understand that medicine drives engineering, not the other way around. It is sometimes easy for engineers and computer scientists to forget this! In fact, this guideline can be generalized as follows: The customer is always right. Not paying enough attention to this guideline has been the downfall of many engineering project teams.

The majority of the material in this case study pertains to the first versions of the spike detection system implemented in the early 1990s. It is as relevant now as it was then as an example of an interdisciplinary team working together to arrive at a solution to a challenging problem using computational intelligence. Developments in the spike (ED) detection system since 1995 are summarized in the epilogue to the case study. We sincerely thank W. Robert S. Webber, Ph.D., and Ronald Lesser, M.D., both still at the Department of Neurology of The Johns Hopkins University Hospital, for their collaboration during the project described in this case study and for their contributions to the epilogue.

Goals and Objectives

The presence of EEG waveforms identified as spikes usually indicates some sort of abnormal brain function. The polarity and amplitude pattern of the spikes often provide information on the location and severity of the abnormality, possibly including information such as whether or not seizures are local (focused in one small volume). This information is then used by neurological surgeons when deciding on corrective measures.

In a hospital setting, EEG recordings for a patient can be taken around the clock, from multiple (often up to 64) scalp electrodes per patient. Because of the around-the-clock acquisition of data and the data rate of 200 to 250 samples per second from each channel (electrode), a large quantity of data is being handled. Depending on the number of channels and the data rate, on the order of 10 to 100 Mbytes of data per hour are recorded for each patient.

Accurate interpretation of the data is critical. Many patients in the hospital will have serious corrective measures taken by the neurological surgeons, including removal of a portion of their brain, so the information provided to the medical staff must be complete and accurate. Prior to the development of the automated system described in this case study, interpretation of the data was done manually and painstakingly using costly labor.

The primary goal of the spike detection effort described in this case study was to provide rapid, accurate on-line or off-line analysis of data. A secondary goal was to reduce the amount of data that must be recorded from each patient. For example, one method might be to parameterize the data, recording calculated parameters instead of raw data.

Design Process

As is the case in the development of most systems, the design process was iterative. On one hand, choices must be made relative to the preprocessing and characterization of the raw data. For example, are raw data presented unprocessed to the system, or are waveforms that represent possible signals of interest (in this case EEG spikes) detected and preprocessed prior to presentation? Or is even more preprocessing done and only certain calculated parameters of the possible target signal presented to the system? This choice, ranging over a spectrum from relatively voluminous raw data to relatively few parameters or features, is made in many applications.

On the other hand, choices must be made relative to the system architecture and algorithms. In this case, various neural networks were considered. Choices had to be made between supervised and unsupervised paradigms. Once a back-propagation network was selected, various network architectures were evaluated with respect to design constraints such as on-line analysis requirements, and various combinations of the parameters associated with the learning algorithms were evaluated.

And, of course, to be implemented, the system cost must be as low as possible. In this case, the objective, which was met, was that the total system cost less than \$10,000 U.S. (1990 dollars).

System Specifications

System specification definition is an important step in any system development. In many cases, it is not difficult to specify what results must be obtained for the system to be performing successfully. This was not, however, straightforward for some aspects

of the EEG spike detection system. In fact, it seems that many medical applications of computational intelligence present special challenges to system design.

Some specifications were arrived at relatively easily, for example, the requirement that the system be able to analyze at least eight channels of information in real time. It was also relatively easy to agree that the system could require minimal (up to 1–2 hours) of “training” for each new patient.

Two other specifications were harder to define. First, what constitutes an EEG spike? Are there measures or calculations that can be applied that definitively specify which waveforms are spikes and which are not? Second, once spikes have been defined, what constitutes successful performance? What are acceptable levels of false positives and false negatives?

The most practical answer to the question of what constitutes a spike is that spikes are defined by qualified neurologists. Even after this system had been in use for more than a decade, however, no precise *mathematical* definition was known to the authors. The six neurologists who participated in the original study reported in Eberhart and Dobbins (1990) all marked the records used for this study differently. Of all the “events” marked by one or more of the neurologists as spikes, about 60 percent of them were marked by four or more. After several meetings with the hospital staff, it was decided to define those events marked by at least four of the six neurologists as *spikes*.

The question of what constitutes successful system performance was even more difficult to resolve. After several additional discussions with hospital staff, it was decided that, at least initially, the system would be considered successful if it detected at least 80 percent of the spikes as just defined. In addition, success required that no more than 20 percent of the events identified by the system as spikes be false positives, or non-spikes (marked by none of the neurologists). The metrics of recall and precision, discussed in the performance metrics chapter, were defined as measures of success. However, these metrics were not created equal, at least in the eyes of the users. It was considered much more important to minimize false negatives than false positives; that is, recall was deemed to be more important than precision.

As part of the development of software for viewing EEG data on a personal computer, Johns Hopkins Hospital staff, led by Bob Webber, Ph.D., had developed what was called the Spike Viewer, which calculated various parameters for each waveform identified as a potential spike. Included were parameters such as amplitudes, widths, and sharpnesses (Webber 1988). Prior to the effort described here, hospital staff had written software using various combinations and weights of these parameters in an attempt to detect spikes. Although the version available at the time this project was started yielded unacceptably high numbers of false positives and false negatives to be used as a stand-alone spike detector, a simplified version played an important role as a preprocessor in the system that was developed, as described later.

Data Preprocessing and Categorization

As this project developed, it became evident that the system design effort consisted of two main areas. The first was the preprocessing and categorization of the raw data required prior to its presentation to the neural network for training or testing. The second was the development and implementation of the neural network analysis tools and the associated data manipulation. In retrospect, dividing the design effort into these two areas has proven useful for a number of subsequent projects involving computational intelligence systems.

A complete description of the design appears in Eberhart and Dobbins (1990) and is not repeated here. The discussion of the results reflects these categories, however. What is emphasized are the lessons learned that may be helpful to you in solving application-related problems in a variety of environments.

Turning our attention first to preprocessing, three main preprocessing possibilities were initially considered for performing spike detection using a neural network implementation as part of the analysis system. These three possibilities have arisen in subsequent projects and are probably relatively generic for signal processing classification applications. In this case, each was evaluated in light of the goal of real-time multichannel analysis within time and budget constraints.

The first possibility examined was the analysis of raw data using a sliding window (a window of a fixed time width, sliding with time). For example, given that an epileptiform spike is at most about 200 milliseconds (msec) wide, the sliding window approach using a 250-msec window would require about 20 iterations per second of the neural network to ensure that the spike waveform is entirely inside the window at some time.

Twenty iterations per second of the neural network is feasible, but it results in a significantly higher computational load than other approaches. In addition, the training of the network is more difficult than with other approaches because relatively narrow (50–100 msec) spikes must be accurately detected no matter where they are located within the window. This dictates a large training set and a lengthy training process. Because of these drawbacks, the development of this approach was discontinued. (It was, however, reexamined in an analysis published by Webber and colleagues (1994).)

The second possibility examined was to preprocess the raw data so that candidate spikes were identified and presented, centered in the time window, to the neural network for analysis (classification). This approach usually results in a lower overall system computational load (depending on the preprocessing).

A version of the Johns Hopkins Hospital software designed for spike viewing, described previously, was used to identify candidate spikes. This software wasn't sufficiently accurate to act as a stand-alone spike detector. If the software's adjustable parameters were set too far in one direction, too many real spikes were missed, even

though the number of false positives was acceptable. Adjusting the parameters too far in the other direction resulted in an unacceptably high number of false positives, although all but about 4 to 5 percent of the spikes were detected.

To use the system to do our preprocessing, the software parameters were set for the second of these two situations, so that the number of false negatives (the number of spikes identified by at least four of the physicians but not by the software) was as low as possible. Simultaneously, some effort was made to minimize the ratio of false positives (candidate spikes that are not real spikes) to spikes.

The ratio of false positives to spikes was generally a little less than 3:1. Because the occurrence of spikes in the analyzed records averaged about 1 per second, the computational load on the neural network tool was about 3 or 4 computational iterations per second for real-time analysis.

A common data acquisition rate for EEG data is 200 samples per second. A 240-msec window then results in 48 raw data points per channel for each candidate spike.

The third possibility exploited the fact that in addition to identifying the time of the waveform (candidate spike) center, the Johns Hopkins Hospital software used to identify candidate spikes calculated 9 parameters for each candidate spike waveform. These nine parameters (for each channel) could then be presented to a neural network instead of the raw data.

This increased the computational load on the preprocessing stage but significantly reduced the computational load on the neural network because only 9 input PEs per channel were required instead of 48. This third approach, with some modifications, was eventually implemented. The major lesson here is that existing approaches, even if not successful as system solutions by themselves, should be considered to play a role in the new system design.

Let's now look at how we performed input data scaling. For the back-propagation architecture, all pattern files were constrained to values between 0 and 1. All pattern data, whether raw or parameterized, were thus scaled before being placed into the pattern files. The way scaling was done, particularly in the case of the parameter pattern files, had a significant effect on the trainability and testability of the neural networks.

In the case of raw data, where a number of channels of voltage waveforms are sampled at a specified rate (200 per second in this case), scaling is generally done across all channels uniformly. If data exist that range from a maximum value X_{max} to a minimum value X_{min} , and X_{min} is a negative number, first add X_{min} to all values, resulting in a range from 0 to $(X_{min} + X_{max})$. To get scaled inputs, now divide each value by $(X_{min} + X_{max})$. It is possible, even probable, that the values of X_{min} and X_{max} will occur in different channels. That is what is meant by scaling across channels.

If, however, the input data to the back-propagation network is in the form of calculated parameters, the situation may be very different. For example, several kinds

of parameters might be used as inputs. In the EEG spike detection case, there was a mixture of voltages, time durations, and waveform sharpness parameters. (In other applications, there may be a mixture of many additional kinds of parameters, including statistical parameters such as standard deviations, correlation coefficients, and chi-square goodness-of-fit parameters.)

In the EEG spike detection case, scaling across all channels (all parameters) resulted in a failure to train the network. Next, each channel was individually scaled; another failure to train was the result. Let's examine why each attempt failed and what was done to achieve success.

Scaling across all channels can result in problems in at least one way. For example, if one or more channels represent a parameter, such as the sharpness of a waveform in this case, that can vary only from, say, -0.1 to 0.1 , and other channels represent waveform amplitudes that vary from -50 to 50 , it's easy to see that the sharpness values will be swamped by the amplitude values after scaling. A variation of 0.1 in each type of channel will result in a variation of 0.001 after scaling. This is probably quite acceptable in the case of an amplitude channel, representing only 0.1 percent of the dynamic range. In the sharpness case, however, the 0.1 variation represents 50 percent of the dynamic range, so it is likely that the precision required to train and test the network's discrimination on the sharpness parameters will be severely hampered, if not destroyed.

Scaling on each channel also resulted in difficulties in training the network. The EEG spike detection waveform was parameterized into nine parameters: two amplitudes, three widths, three sharpnesses, and one product slope (an indication of the steepness of the waveform at a point). The unit of measure of the amplitudes was volts, widths were in seconds, and sharpnesses in radians. When the network was scaled on each channel individually, it had difficulty training. Sometimes it would train, sometimes it wouldn't, and, more important, the system performance on the test set was uneven and unpredictable.

Next, the three width channels were scaled as a group. The same was done for the three sharpness channels, and the same for the two amplitude channels. This was successful. The network trained and tested well and was robust with changes in network parameters such as learning coefficients and momentum factors.

Why did this help? Theoretically, normalizing on each channel individually should have both positive and negative effects, the positive being that each channel gets to reflect its dynamic range over the entire interval between 0 and 1 and the negative being that the relationship between any two channels is lost to the extent of an offset and a multiplicative factor. Perhaps it might take longer to train the network, but it should eventually train. In this case, however, normalizing on each channel seems to have made training the network sufficiently difficult that the network's performance was only marginally acceptable, if that.

When summed, two of the width parameters chosen to represent the EEG candidate spike formed the width of a half-waveform between zero crossings. The third

was the inflection width, the width between the two points at which the second derivative of the half-waveform changed sign. In the raw data, the sum of the first two widths was always larger than the third width, and each of the first two widths was smaller than the third. These and any other relationships that existed between the two amplitudes or among the three sharpnesses were obscured in the individual channel scaling process. The network therefore had to learn these relationships along with everything else.

We therefore suggest, on the basis of this experience and several other similar ones, that related inputs be scaled as a group rather than individually, particularly if difficulty is being experienced in training a network with individually scaled inputs. The lesson here is: *Don't make the network learn more than it must.*

The “Universal Solution” Myth

Previously, we discussed the choice of recall and precision as performance metrics. We mentioned that the neurological surgeon users considered minimizing false negatives more important than minimizing false positives. One of the system developers (RCE) presented an early version of the system design at an IEEE International Conference on Engineering in Medicine and Biology, explaining how the design was based on the above feature. The presenter assumed that neurologists the world over had the same priority.

Following the presentation, when the question and answer period was opened, a researcher from New Zealand politely but firmly pointed out that “down under” neurologists had just the opposite priority: It was more important to *them* to minimize false positives than false negatives. One of the most important bases for the design had been turned on its head.

There are at least two lessons to learn from this experience. First, when presenting or publishing papers, always explicitly state any assumptions and/or constraints that shaped the system design. (That, at least, was done in this case.) Second, do not assume that the solution that meets *your* customers' requirements will be perceived by other potential customers as meeting theirs (or even be acceptable to them). This usually is a harder lesson to learn.

There is another important reason to avoid thinking that you have developed a universal solution. Each neurologist has developed his or her style of reading EEGs. Indeed, laboratories at different medical facilities often seem to have, in their “culture,” developed their own clinical preferences. To be accepted by multiple EEG readers at a given facility, or to be adaptable to different viewpoints held at the laboratory level by different clinics, any system must be adaptable, to a degree. Webber and colleagues (1994) discuss two approaches to make the system adaptable to the individual user and clinical preferences. One is to use a certain suite of training cases to train the system. The other is to allow the user to select output PE threshold levels that provide results that best reflect the user's preferences.

Epilogue

The field of EEG spike detection has seen significant advances since our pioneering work in the early 1990s. Perhaps the most advanced spike detection software available as this book goes to press is Persyst Development Corporation's EEG Suite software package, which uses a neural network with some form of parameterized input to detect spike events. The network has a probabilistic output and can be trained by the end user.

Dr. W. Robert S. Webber at The Johns Hopkins University Hospital reports that spike and seizure detection are used fairly routinely for patients during sleep. During awake recordings, however, the detectors often pick up too many artifact events to be useful. More advanced detectors can be tuned by marking events and correcting errors, but this is an extra burden for the EEGers reading the records, so it does not often take place. As Webber says, "There is a very simple measure of success: Do the EEGers actually use the device, however good it appears in a published report?" The most challenging EEG-related problem reported now by Webber is the detection of novel seizures that EEGers might miss.

Case Study 2: Determining Battery State of Charge

The purpose of the project described in this case study was to develop a system that continuously provides a state of charge estimate for a battery pack of approximately 30 batteries in an electric or hybrid vehicle. Most algorithms to determine the state of charge of a battery or battery string are typically accurate to within only 5 or 10 percent. This case study summary briefly reports on using computational intelligence tools, including neural networks, to develop state of charge diagnostics. For a more detailed description of this case, see Eberhart et al. (1996).

The relatively low energy densities in today's battery technologies make it especially important in many applications to determine the state of charge very accurately in order to prevent serious operational problems. The methodology developed in this case study yields results accurate to within 1 or 2 percent over the entire operating ranges of the batteries tested, including variations in load profile and temperature.

The simplest approach to calculating state of charge is by integrating the amp-hours delivered by the battery string over time. This approach, however, requires knowledge of the battery capacity, and the relationship between current and capacity depends on a variety of parameters, such as temperature and other environmental factors. Therefore, this approach is not adequate.

Another approach is to use a family of capacity-voltage-current curves, which are used to derive capacity from known conditions of voltage and current. However,

as batteries age, the calibration of these curves is not sufficient to provide required capacity accuracy.

System Design

The input parameters available from the problem sponsor for the state of charge (SOC) system design were

- Discharge current of the battery pack
- Total number of amp-hours used
- Average temperature of the battery pack
- Minimum individual battery voltage
- Maximum individual battery voltage
- Average individual battery voltage
- Voltage difference between average individual battery voltage and minimum individual battery voltage
- Minimum individual battery voltage at a previous sampling time

Each input parameter was scaled to values between 0 and 1 to make it easier to use a number of network configurations. The target output was the actual SOC as measured in the testing laboratory.

The actual SOC of a string of batteries depends on the voltage, current, temperature, and amp-hours of each battery. For a string of n batteries, then, there can be $4n$ parameters in the input vector. For relatively large strings of batteries ($n > 10$), this can make real-time implementation difficult, given cost and computing power constraints. One goal of this project was to minimize the input dimensionality.

Datasets were available with a variety of dynamic load profiles and at different ambient temperatures. Unlike many projects, a large amount of data was available (more than 2,000 datasets). It was determined that in order to represent the range of temperatures and load curves, it was necessary to use only about 20 percent of the data. Adding more data to training did not improve system performance. So the first main lesson in this case study is to use a sufficient quantity of data to cover all operational environments, but no more than that.

System Evolution

A number of supervised training algorithms were used with the data, including the back-propagation, particle swarm, and Levenberg–Marquardt algorithms. The Levenberg–Marquardt algorithm is based on a modified Newton’s method for parameter optimization. Because of its performance over the large range of training and testing data, it was selected for implementation, initially with an 8–5–1

network configuration. (Particle swarm optimization was still in its infancy. We had very little experience with it. Had it been a few years later, the method implemented would likely have been different.)

The Levenberg–Marquardt update rule for weights in a neural network appears as equation 12.1, where ΔW is the matrix of weight changes, J is the Jacobian matrix of derivatives of each error with respect to each weight, I is the identity matrix, e is the error vector difference between the actual SOC and the SOC estimated by the network, and α is a search parameter that is varied during network training. The initial value of α is chosen sufficiently large to ensure that the algorithm initially uses steepest descent with a relatively large step such that the error is quickly brought close to the minimum. A gradually decreasing α then activates Newton’s method so that the error is rapidly driven to an acceptable minimum. (The value of α increases only if an increased error occurs after an iteration.)

$$\Delta W = (J^T J + \alpha I)^{-1} J^T e \quad (12.1)$$

Analysis of preliminary results of the 8–5–1 network and experience with batteries indicated that some of the inputs were not playing an important role in determining SOC. Therefore, inputs 1–4 plus the minimum individual battery voltage at the previous sampling time (input 8) were selected, reducing to 5 the number of network inputs. Further, after experimentation and testing, the number of hidden PEs was reduced to 3, resulting in a 5–3–1 feedforward network being selected for implementation.

During system development, we observed that the largest estimated SOC errors were occurring at the beginning and end of the battery discharge cycles. We determined that using an output PE with a sigmoid activation function significantly contributed to these errors, a situation illustrated in an exaggerated fashion by Figure 12.1.

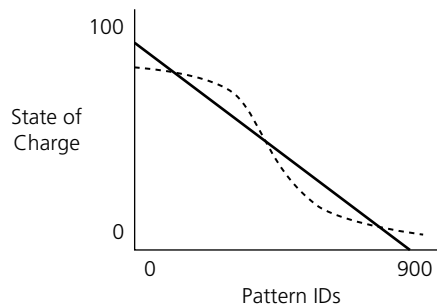


Figure 12.1 Training (solid line) and testing (dotted line) results using a sigmoidal output processing element. There are 900 patterns in each dataset.

The solid line in Figure 12.1 represents a constant current discharge over time, which starts with an SOC of 1 and decreases linearly to 0. A sigmoidal output PE, however, trains to something similar to the dotted line in the figure because of the nonlinearities at the extremes of the sigmoid function. It will, in fact, always exhibit errors at the ends of the discharge cycle, even under other load conditions. The problem is that the most important regions of the SOC prediction for many applications are at the discharge cycle extremes. We want to know accurately when the battery is almost fully charged and when it is almost fully discharged.

When an output PE with a linear activation function was substituted for the sigmoidal output PE, the system training improved significantly. The relatively large errors at the beginning and end of the discharge cycle were eliminated, and the maximum errors of the estimated SOC dropped to around 2 percent, down from about 5 percent using the sigmoidal PE.

The second main lesson from this case study, therefore, is to match network output PE activation functions to the function/data used for training when necessary to achieve desired system performance.

In order to achieve the desired testing accuracy, input parameters used during the testing phase were preprocessed using a second-order Butterworth digital filter with appropriate filter parameters, chosen to smooth the inputs while providing rapid processing. This kind of preprocessing appeared to make no difference during training, but it enhanced performance during testing/running of the system.

The third major lesson thus arises from the filter experience. It is not unusual to preprocess input parameters, but the same preprocessing is usually applied during both training and testing. The learning is that there may be computations or methods that provide significant assistance during *either* the training *or* testing phase that are not necessary (and may even be detrimental) during the other phase.

Conclusion

A 5–3–1 feedforward network was successfully developed for the SOC estimation application for an electric vehicle battery pack. The Levenberg–Marquardt algorithm was chosen for updating weights during training. A second-order Butterworth filter was designed for use when the network is tested or run. It is apparent that the system designed in this case study outperforms all known algorithms in terms of SOC estimate accuracies and trajectory smoothnesses. The U.S. Patent Number 6,064,180 was granted for this state of charge estimation methodology.

Case Study 3: Schedule Optimization

One of the most common applications of evolutionary computation is optimization. And one of the most common optimization applications is optimizing scheduling.

Scheduling optimization belongs to a class of problems, called *NP-complete* (or *NP-hard*), that provides fertile ground for computational intelligence approaches. Being NP-complete means that any deterministic search technique that completely searches the problem domain will probably not find an acceptable answer in an acceptable time. Also, heuristic methods that reduce the search space are not certain to find an acceptable solution, much less the global optimum. Stated more formally, there is no polynomial-time algorithm that can possibly solve them, unless it is proved that $P = NP$. The fact that the problem is NP-complete poses a challenge; in addition, scheduling optimizers are almost always made complex by factors associated with the specific project.

This case study summary uses two scheduling project examples to illustrate issues and approaches to scheduling systems based on computational intelligence. The first is an example of facility scheduling based on work done at a U.S. Navy test station laboratory by Syswerda and Palmucci. The second is an example of transportation resource scheduling based on work done for a sponsor by the Computational Intelligence Laboratory team at the Purdue School of Engineering and Technology in Indianapolis.

We first briefly outline each scheduling project. We then discuss scheduling systems in general and their major components: schedule builders, schedule optimizers, and schedule evaluators. The next issue addressed is representation, one of the most difficult aspects of developing a scheduling system. We then briefly consider the issue of evaluation of scheduling systems. Finally, using the two scheduling project examples, we see how scheduling systems are developed and tested, including considerations such as custom operators often used in evolutionary computation-based systems.

For more information on using evolutionary computation for scheduling, we recommend that you read a survey article such as the one by Dimopoulos and Zalzal (2000).

Facility Scheduling Example

The first example is of facility scheduling based on work at a U.S. Navy test station laboratory, located at Point Magu, California, by Gil Syswerda and Jeff Palmucci, and draws on work published by them. The two main references to their work are Syswerda (1991) and Syswerda and Palmucci (1991). You should read both references to get a complete description of the design and testing of their optimizer.

Since the testing facility was typically open 16 hours per day, five days per week, the basic scheduling optimizer had 80 hours per week with which to work. The scheduler typically scheduled one week at a time. However, an important requirement was that the computerized scheduler be capable of being combined with manual scheduling and be capable of optimizing a limited portion of the week's

schedule. This was important because of the constantly changing task requirements associated with the laboratory.

In this project, there were about 40 categories of equipment *resources*. There were multiple instances of some. Included were F-14 jet fighter frames, flight and radar simulation environment generators, and various kinds of support equipment (Syswerda and Palmucci 1991). There were also personnel resources; certain people operated particular equipment.

One of five levels of time preferences could be assigned for each task, ranging from preemptory (the task must receive the exact time slots requested), through neutral (assign the task to any time slot), to illegal (the task cannot be assigned to that time slot). Each task was also assigned a priority. Then, based on time preferences and other considerations, each task was assigned a fitness value for each hour in the week.

The schedule had to allow for the proper configuration of the equipment for the task. If the equipment was not already in the correct configuration, time for equipment setup had to be inserted into the schedule. Setup time required between one and three hours.

In this case, as in many scheduling problems, we defined two types of constraint: hard (strong) and soft (weak). Hard constraints could not be violated by the scheduler. Period. They could, to some defined extent, be overridden manually. Examples of hard constraints were scheduled maintenance time and equipment setup time. Also, obviously, parts of a single long (>1 hour) task, broken into 1-hour time blocks, could not overlap.

Soft constraints included items such as time preferences and preferences that particular people be available to run equipment. Soft constraints could be violated, with a penalty being assigned to the fitness function each time a soft constraint was violated.

Transportation Resource Scheduling Example

The transportation resource scheduling example is based on a project done for a sponsor by the Computational Intelligence Laboratory team at the Purdue School of Engineering and Technology in Indianapolis, Indiana. Because of the proprietary nature of the project, some details have been altered in this presentation.

The project takes place within a large multilocation corporation that manufactures electro-mechanical systems. The problem involves scheduling the shipping of electro-mechanical subassemblies from a corporate warehouse to three assembly facilities. The geometry of the project, simplified somewhat for this presentation, is represented in Figure 12.2. The warehouse is denoted by **W**, and the final assembly facilities by **A1**, **A2**, and **A3**.

Assume for this case study that facilities **A1**, **A2**, and **A3** are equidistant, separated by 20 kilometers. The warehouse **W** and facility **A1** are 40 kilometers apart, and we assume that **W** and **A2** are separated by about 45 kilometers.

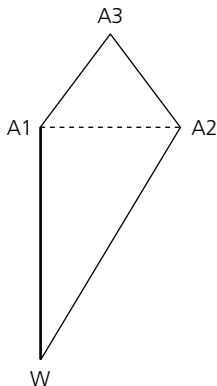


Figure 12.2 Diagram of the transportation resource scheduling example.

Two main kinds of packaging are used for the subassemblies: pallets and cartons. The pallets go only to facility **A1**; cartons are delivered to all three facilities. Special heavy trucks that can carry both pallets and cartons can be scheduled to run *only* between the warehouse and facility **A1**. (The heavy line between **W** and **A1** reminds us that this is our heavy truck route; keep in mind that light trucks can also follow this route.) Light truck transport for cartons only is available to all three facilities and can run any route shown. We assume that all heavy trucks are identical and that all light trucks are identical.

Capacities of both the heavy and light trucks are limited by volume, not weight; to determine when truck capacities are reached, we only have to check on the numbers of cartons for the light trucks and on the combination of numbers of pallets and cartons for the heavy trucks. For the purposes of this case study, assume that there are two categories of pallet, $p1$ and $p2$, and two categories of carton, $c1$ and $c2$. Each category contains a particular type and number of electro-mechanical assemblies needed by a final assembly facility. (In the real world, there would usually be many more than two categories of each shipping package, but we restrict the categories here without any loss of generality.)

A1, **A2**, and **A3** are just-in-time (JIT) assembly facilities and have very little storage capacity on-site. Orders are placed with the warehouse 8 to 12 hours in advance for delivery within a 2-hour time window. Penalties for early delivery are moderate; penalties for late delivery are severe. Each order comprises items only for the facility placing the order. Orders can be split among trucks, and a single truck can carry items for more than one facility on a trip. There can be only one order per facility for any 2-hour time window, but there can be as many as four orders per facility for delivery during an 8-hour shift. Driving times for the routes are estimated when orders are received based on delivery time window time-of-day and weather forecasts.

It is important that rescheduling be redone quickly if the resources or conditions change. For example, if a heavy truck breaks down or an accident blocks traffic on one of the connecting routes, we need to reschedule within a few minutes. Rapid rescheduling capability is very important in this application.

Scheduling System Major Components

Many computational intelligence–based scheduling systems comprise three major components: the schedule builder, the schedule evaluator, and the schedule optimizer. The following descriptions reflect how these components were implemented in our two examples. Other approaches are possible.

The schedule builder generally does not violate any hard constraints. Its main job is to build “legal” schedules, which simply means that no hard constraints are violated. The schedule builder may take into account some of the soft constraints, depending on the system design, however. This may mean that, for example, some of the lower priority tasks may not be scheduled. Using a schedule builder in conjunction with an optimizer is discussed in the scheduling system development section.

The schedule evaluator calculates a fitness value for each schedule put together by the schedule builder. Fitness values are assigned according to a problem-specific algorithm. A variety of approaches are possible for evaluating schedules; a few are reviewed in the section on schedule evaluation.

The schedule optimizer, in our case a computational intelligence algorithm, works in a more or less domain-independent way in our example to construct task order lists. The section on scheduling system development focuses on the optimization function. Before the optimizer can do its work, however, it must be decided how to represent (encode) the problem for the optimizer; this is discussed in the following section.

One of the main design considerations is how much intelligence (complexity) to build into each of the three main components. The approach selected in our example is to make the schedule builder relatively simple-minded, able to build “legal” schedules, and do only relatively minor local optimization. The schedule builder is usually initialized with a “base” schedule that already has time allocated for such activities as scheduled maintenance, equipment setup, and some very high priority tasks.

Representation of the Problem for the Optimizer

In this section, we focus on how to represent a scheduling problem for use by an evolutionary computation algorithm such as a genetic algorithm or particle swarm optimization. (In the section on scheduling system development, we briefly mention representation for fuzzy system approaches; we don’t mention fuzzy systems further in this section.)

Ways to represent a scheduling problem span the spectrum from classic binary coding to direct representation. We touch briefly on only a few methods. For more information, see references such as Dimopoulos and Zalzalá (2000).

The classic binary coding harkens back to the canonical genetic algorithm of Chapter 4. With parameters (machine numbers, task sequences, or whatever) coded as 1s and 0s, this approach is seldom if ever used anymore, primarily because the traditional crossover and mutation operators produce significant disruptions to the feasibility of a schedule that must constantly be repaired to produce feasible schedules.

Direct representation, at the other end of the spectrum, produces a task structure that can be directly used as a schedule; no decoding is needed to assemble a schedule. Each location in the chromosome must completely represent and specify the resource assignment, the process, and the time span. It is obvious that such a representation cannot be initialized randomly; hence, a traditional scheduler is usually used to initialize the population with feasible schedules. Furthermore, during the evolutionary process special operators must be employed to maintain schedule feasibilities.

Most evolutionary computation-based schedulers use an indirect representation method from somewhere in the middle of the spectrum. There are a variety of approaches such as preference lists, dispatching rule representation, and job-based representation. Both of the examples we have selected use preference lists, which aren't direct representations of schedules, but rather indicate the task sequence for each resource. Tasks are scheduled according to the preference list unless a hard constraint is violated. Custom operators are required to convert the list into a usable schedule. See Dimopoulos and Zalzalá (2000) for a complete description of other indirect representation methods.

Schedule Evaluation

Computational intelligence-based schedulers use a variety of evaluation criteria. Historically, the minimization of *makespan* was the most common objective. Makespan is the elapsed time from starting the first task until the completion of the last task. Although makespan was frequently used by researchers, it does not reflect scheduling issues that must be faced in the real world, such as due dates and equipment setup times, and it is not used much anymore.

Seven schedule quality criteria were proposed by Fang and colleagues (1996): maximum tardiness, average tardiness, weighted flow time, weighted lateness, weighted tardiness, weighted number of tardy jobs, and weighted earliness plus weighted tardiness. These criteria (as well as others) can be used and combined in many ways. Determining the weighting functions can require a significant effort by the scheduling system developer, working with the customer to ensure that the evaluation function accurately reflects the customer's objectives.

Let d_i be the desired completion time of an event (the delivery of an order, for example) and c_i be the actual completion time. The *lateness* of a task i is defined as $l_i = c_i - d_i$, which is positive when the task is completed late and negative when it is completed early. A task's *tardiness* is defined as $t_i = \max(l_i, 0)$, which is never negative (Pinedo 1995). In an analogous way, we can define a task's *earliness* in equation 12.2.

$$e_i = \max(d_i - c_i, 0) \quad (12.2)$$

A commonly used fitness function is the sum of squared latenesses. For n events (tasks), equation 12.3 defines the sum of squared latenesses (Bruns 1993). (According to the above definitions, this quantity should more properly be called the sum of squared tardinesses, but we retain the author's terminology.)

$$C(c_1, \dots, c_n) = \sum_{i=1}^n \max(0, c_i - d_i)^2 \quad (12.3)$$

Equation 12.3 is deceptively simple. For example, are the times d_i and c_i measured in seconds, minutes, or hours? Or in some other way? Is an order complete only when all items in the order are delivered? Is partial credit given for delivery of most (say all but one) of the items on time? Many an application has foundered on such questions.

In the next two sections, we see how the two sample applications addressed issues such as representation and evaluation.

Facility Scheduling System Example

As was stated earlier, the basic approach in both of our example schedulers is to make the schedule builder relatively simple-minded, able to build "legal" schedules and to do only minor local optimization. Both examples use preference lists for representation (encoding) of the problem. Let's first consider the facility scheduling example.

For the facility scheduling problem, each task is assigned a fitness (preference) value for each hour in the week, based on time preferences and so on. The tasks are sorted first according to fitness values and then according to the ordering of hours in the week (which results in a sorting by day of the week at given times: If the highest fitness values occur at 8:00 a.m. on Monday, Tuesday, and Friday, these three values will appear in this order in the sorted list). The result is called the *preference vector* for each task. As the schedule builder encounters a task to be scheduled, a number of possible "legal" positions for the task are examined. The maximum number of positions examined is the *depth of search*. The first time slot with the highest fitness (preference) value that can be scheduled legally is selected; if no times are found within the depth of search, the task is not placed on the schedule.

The schedule evaluator establishes the fitness of each schedule in the population by increasing fitness for each task scheduled, decreasing it slightly for soft constraints violated, and penalizing it more significantly for tasks not scheduled at all. In the scheduler being considered here, each task is assigned a priority value, with higher numbers for higher priorities. An initial fitness value for the schedule is assigned that is equal to the sum of all task priority values.

When a task is scheduled, its priority value (typically ranging from 40 to 100) was added to the schedule fitness value. In Syswerda (1991), it was reported that when a task was scheduled but a soft constraint was violated, only one-half of the task's priority was added to the fitness total. When the schedule was completed, the priority value for each task not scheduled was subtracted from the fitness value.

Syswerda and Palmucci (1991) described a slightly different approach in which, for each violation of a soft constraint, 10 was subtracted from the fitness value. Note that the scheduling of a task may result in the violation of one or more soft constraints for one or more tasks.

The schedule optimizer in this example is a GA. Syswerda and Palmucci (1991) state that for this case, “the schedule builder and evaluator have isolated the GA from all problem-specific details; the GA has only to work with permutations of a list of things.” Examining the situation, however, indicates that the GA implementation depends on problem-specific details, both with respect to data structures (representation) and with respect to the chosen crossover and mutation operators.

A simple chromosome was chosen to represent the problem: Each individual is a permutation of the list of tasks to be completed. A simple permutation of tasks doesn't constitute a schedule, however. For one thing, resource conflicts may force some tasks to be deleted from a schedule. For another, soft constraint violations could have a significant effect on relative fitnesses. Finally, just having the task order list doesn't specify the starting time of or the time required to complete (as many as possible of) the tasks.

Once the representation was selected, operators had to be chosen that produce children with new task orders. A variety of crossover and mutation operations are used in scheduling problems. Both operators selected in this case might be appropriate for other order-based GA representations, such as the traveling salesman problem. The mutation operator is called *order-based* mutation in Syswerda and Palmucci (1991) and *swap mutation* in Syswerda (1991). It comprises selecting two tasks in an individual at random and then exchanging their positions. The crossover operator is called *position-based* crossover. For this crossover, two parents are selected, then a set of positions is randomly selected. (The selection of the positions to undergo crossover is conceptually similar to uniform crossover in a traditional GA; see Chapter 3.) The positions of the tasks in one parent are instantiated as the same positions in the second parent; likewise, the tasks at those positions in the second are instantiated in the same positions in the first. In each parent, then, the tasks not imposed from the other parent are used to fill in the remaining

positions in the parent, in the same task order as existed prior to crossover. A simple example, in which each task is represented by a letter and the crossover probability is 0.4, follows (after Syswerda and Palmucci 1991):

```

Parent 1:      A B C D E F G H I J
Parent 2:      E I B D F A J G C H
Positions selected:  * *      *      *
Child 1:      A I B C F D E G H J
Child 2:      I B C D E F A H J G

```

In the case being described, the initial crossover probability was typically set to 0.8 and mutation to 0.2. The probabilities at the end of the run were reversed: crossover probability 0.2, mutation probability 0.8. Linear interpolation was carried out between the initial and final values during the run. It is assumed that the mutation probability is expressed as a single probability for the entire chromosome. In other words, at the beginning of a run, there was a probability of 20 percent that mutation would occur once somewhere in the chromosome; by the end of the run, the probability was 80 percent that it would occur once in the chromosome.

Let's summarize the approaches used in this facility scheduling example. A simple chromosome representation, a task permutation called a preference list, is used that is appropriate for the problem. A relatively simple schedule builder is used that builds legal schedules and does limited local optimization based on the specified depth of search. The fitness of a schedule is higher as more tasks and high-priority tasks are scheduled, with penalties imposed for violating soft constraints and for not scheduling tasks. Special operators are used, order-based mutation and position-based crossover, selected for their effectiveness in this scheduling system.

Transportation Resource Scheduling System Example

The transportation resource scheduling system was implemented in Java 2, using version JDK 1.3. Two Java packages were implemented, one for each computational intelligence paradigm: genetic algorithm and fuzzy logic. The following paragraphs summarize the approach taken for each paradigm.

Genetic Algorithm Approach

We first consider the genetic algorithm approach. As stated earlier, classic binary encoding is not an efficient way to represent real-world problems. We previously mentioned several methods, both direct and indirect, for the representation of a scheduling problem, including job-based representations, dispatching rule representations, preference-list representations, and alternative representations.

Preference-list representation was chosen in this case. A preference list is not an actual schedule, but a preferable sequence of operations. Here the operation is to deliver a particular item from the warehouse to one of the fabrication facility locations.

Each chromosome represents a sequence for delivering all cargo items. That is, we list all items from all the assembly facilities' orders in one chromosome. We schedule them one by one from the sequence. Then we calculate how many of them can be successfully delivered and, using that information with the constraint violations, calculate a fitness value for each chromosome.

For example, we assign each cargo item (a pallet or a carton) a unique ID number. The numbers are consecutive—if we have 50 items to deliver, the ID numbers are 0 to 49.

Each chromosome is a sequence for the delivery of these 50 items as follows:

5	24	9	18	34	46	24	41	36	7
---	----	---	----	----	----	----	----	----	---	-------

The preference list can be used to generate a schedule in a variety of ways. A very simple method is to put the first item into the first light truck if it is a carton and into the first heavy truck if it is a pallet. Put the next item into the first light truck if it is a carton and into the first heavy truck if it is a pallet. Continue doing this until either the first light truck is full or the first heavy truck is full. Send the full truck out, and start loading the next available truck of the same size (light or heavy). If the only truck available is a heavy truck, put both pallets and cartons going to location A1 in it until it is full, then schedule it to depart. Many other ways can be used to build the schedule, as long as no hard constraints (such as scheduled maintenance for a truck) are violated.

The fitness of the schedule developed from a chromosome is closely tied to the JIT nature of the assembly facilities. A severe penalty is imposed for lateness, and a moderate penalty is imposed for earliness. Thus, a basic version of the fitness function is the sum of the priorities of items delivered, reduced by penalties for earliness and sum-squared latenesses as defined in equations 12.2 and 12.3, respectively.

In a practical application of these equations, the quantities $(d_i - c_i)$ and $(c_i - d_i)$ are each multiplied by a scaling constant according to the time units used and the weighting assigned to earliness and tardiness. Thus, if each item has a priority of either 50 or 100 and the time units are minutes, then an earliness weighting factor of 1 would subtract 30 from the fitness function for the delivery of an item 30 minutes early. Because of the squaring of the tardiness figure, a weighting factor of less than 1 might be appropriate for the $(c_i - d_i)$ quantity. A weight of 0.1 would subtract 90 from the fitness function for the delivery of an item 30 minutes late, and subtract 360 for an item one hour late. (A penalty can be limited to the priority value of the item and/or be made proportional to an item's priority as well.)

A standard roulette wheel Selection with elitist strategy was used for reproduction of chromosomes for the next-generation population. This ensures that the chromosome whose fitness is highest in the population is selected at least once for the next

generation. (See Chapter 3 for a discussion of roulette wheel selection and the elitist strategy.)

Two types of crossover were implemented for this application: position-based crossover and two-point crossover. Position-based crossover is discussed above in the facility scheduling example. Two-point crossover was implemented according to the following procedure:

1. Select two chromosomes randomly.
2. Select two crossover points in the chromosome pair randomly.
3. Exchange the item numbers between these two points of the two chromosomes.
4. Modify the two chromosomes in order so that each cargo item number can appear in the chromosome only one time.

For example:

2	4	0	5	1	3
3	1	2	4	0	5
	^		^		
	Point 1		Point 2		

Exchange the parts between point 1 and point 2:

2	1	2	4	1	3
3	4	0	5	0	5

Some numbers are repeated in the two chromosomes and some numbers are missing. So some “repair” work must be done. In the first chromosome, the numbers 0 and 5 disappear and the numbers 2 and 1 appear twice. So we use the missing numbers to replace the numbers that are repeated outside of the portion of the chromosome between point 1 and point 2. We keep these replacement numbers in the same order they were in the original chromosome. In this case, 0 was before 5, so they are in that order in the new chromosome.

0	1	2	4	5	3
3	4	0	5	1	2

Our choice for mutation is to change two items' sequence randomly. This is called the swap mutation operator. Two other methods for mutation that were considered are position-based mutation and scramble mutation (Davis 1991). Note that the swap mutation operator was called order-based mutation in Syswerda (1991).

We implemented mutation with a probability that increased over the run according to equation 12.4, where p_m is the probability of mutation on an item-by-item basis in the chromosome, g is the total number of generations, and n is the index number of the current generation.

$$p_m = 0.001 + \frac{(0.200 - 0.001)n}{g} \quad (12.4)$$

For each cell (item) selected for mutation according to the probability, another cell is selected randomly, and the item numbers in the two cells are exchanged. As you can see, the probability of mutation increases significantly during the run, similar to the approach used by Syswerda and Palmucci (1991).

Fuzzy Logic Approach

Following is a description of the fuzzy logic model implemented for the transportation resource scheduling example. It is a simplified version of the system actually implemented, but enough detail is provided so that you should be able to design a similar system.

We define two input linguistic variables for our fuzzy scheduler. One is the loading time with respect to the desired delivery time window for each item. The other is an importance parameter for loading each item. The parameter is defined as *(number of loaded items for an order)/(total number of items in the order)*. For example, if by loading one item we can finish loading all the items in a particular order for an assembly facility, the importance parameter is 1.0, which is as high as possible.

The definitions of the two linguistic inputs are as follows (all times are in minutes).

Time input for heavy truck:

<i>Linguistic Range</i>	<i>Lower Limit</i>	<i>Upper Limit</i>
Very early	$s - 450$	$s - 300$
Early	$s - 350$	s
On-time	$s - 150$	$e - 150$
Late	$e - 200$	e
Very late	$e - 50$	$e + 100$

Note: s = starting time of time window, e = ending time of window

Time input for light truck:

<i>Linguistic Range</i>	<i>Lower Limit</i>	<i>Upper Limit</i>
Very early	$s - 180$	$s - 120$
Early	$s - 150$	s
On-time	$s - 80$	$e - 60$
Late	$e - 100$	e
Very late	$e - 20$	$e + 30$

Note: s = starting time of time window, e = ending time of window

The definition of the importance parameter is as follows:

<i>Linguistic Range</i>	<i>Lower Limit</i>	<i>Upper Limit</i>
Very minor	0	15
Minor	10	45
Medium	40	70
Large	65	85
Very large	80	100

Note: Percentage of the loaded items divided by the total items ordered

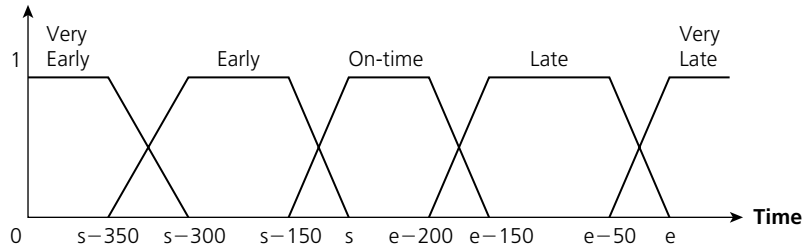
The output variable for the fuzzy scheduler is a decimal number representing the importance of the loading and delivery of each item at the present time.

<i>Linguistic Range</i>	<i>Lower Limit</i>	<i>Upper Limit</i>
Very unimportant	0.00	0.30
Unimportant	0.25	0.55
Fairly important	0.50	0.75
Important	0.70	0.90
Very important	0.85	1.00

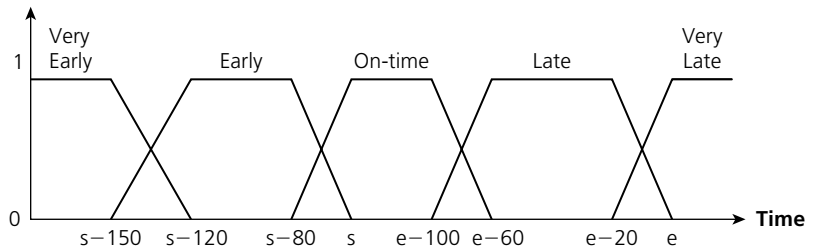
Note: Importance of loading and delivery of each item.

Next, we define the fuzzy membership functions for each input variable. The time scale differences for light and heavy trucks are due to the slower speed of heavy trucks in city traffic and the necessity for the heavy truck to stop at a weight station once in each direction on the route to and from the fabrication facility.

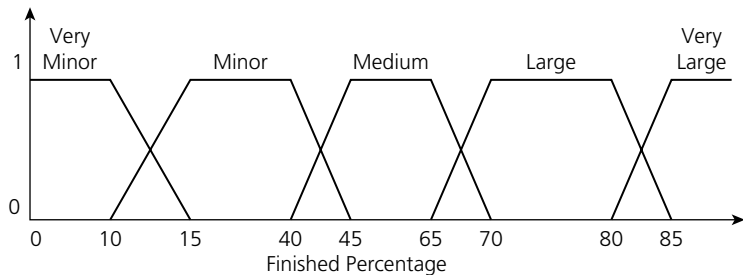
Time input for heavy trucks:



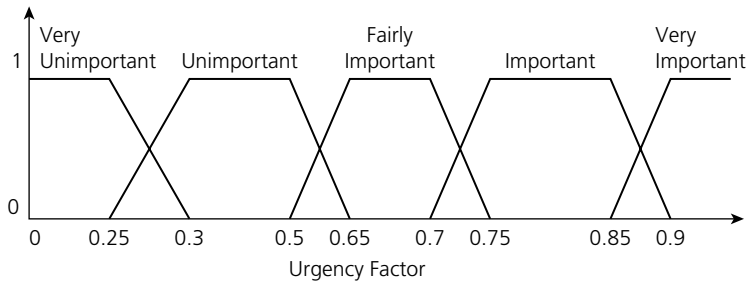
Time input for light trucks:



Importance parameter:



We next define the fuzzy membership function for each output variable, the urgency.



We next define rules for the fuzzy scheduling system. As discussed in Chapter 7, we fill out a rule matrix. For the purposes of illustration, we assume that the same rule matrix applies to light and heavy trucks.

Urgency \ Time	Very Early	Early	On-time	Late	Very Late
Very minor	Very unimportant	Unimportant	Unimportant	Very unimportant	Very unimportant
Minor	Very unimportant	Unimportant	Fairly important	Unimportant	Very unimportant
Medium	Unimportant	Fairly important	Important	Fairly important	Very unimportant
Large	Fairly important	Important	Very important	Important	Unimportant
Very large	Fairly important	Important	Very important	Important	Unimportant

We use a centroid defuzzification method, as discussed in Chapter 7, for the output calculation, as follows:

$$\text{Output} = \frac{\sum x_i \mu_a(x_i)}{\sum \mu_a(x_i)}$$

We initialize the order list by requested time of delivery for each item. We implement the fuzzy logic by calculating the priority parameter for each item not yet delivered each minute of time. We determine items with the highest priority for the

heavy and light trucks using the fuzzy rules defined above. Items thus determined are designated to be loaded starting at that minute. The times included in the schedule include the times needed to load and unload items: 1 minute for each carton and 3 minutes for each pallet.

Other Scheduling Systems

The facilities for which the scheduling systems just described were developed represent only two kinds of activity for which schedules are required. And the types of scheduling system developed, including the structures and the operators, represent but two approaches to schedule system design. This section briefly describes a significantly different type of scheduling requirement and other approaches to scheduling system design.

A major area of scheduling system application is *job shop scheduling*. The output of a job shop is products or parts that are manufactured according to specified sequences of machine operations. These operations require time and resources. The resources typically are specialized machines. Each operation is thus defined as requiring a certain amount of time on a certain machine. These machine operations must be done in sequence.

The main objective for many job shop systems is to ship product orders on time with a minimum amount of unused (idle) machine and labor resources. Another objective common to many job shop scheduling systems is to minimize the amount of materials and partially completed product on hand. These objectives conflict. Finishing jobs more quickly can be done by increasing the number of machines; increasing the number of machines increases idle time for machines (and the human resources who run them).

Each order for product specifies a product and a quantity. There may be more than one process operation sequence that can produce a given product. One of the scheduler's jobs is to select which production process option to implement for a given order for a product.

In the previous scheduling examples, simple permutations of the task list were used as the EA representations. It was noted that this representation is useful for the traveling salesman problem (TSP). A job shop scheduling problem, however, is significantly different from the traveling salesman problem, and more difficult. For example, if an optimal solution for a five-city problem is 12345, then an equally optimum solution is 34512, as is 32154, and so on. Furthermore, the important data in the TSP is in the node connections (edges), which comprise the intercity distances.

The choice of a simple permutation representation, then, may not be the best choice, even for the previous example. Referring to the Navy facility case study presented earlier, Michalewicz and Schoenauer (1996) state:

It seems, however, that the choice of a simple representation was not the best. Judging from other (unrelated) experiments, . . . we feel that the chromosome representation

used should be much closer to the scheduling problem. It is true that in such cases a significant effort must be placed in designing problem-specific “genetic” operators; however, this effort would pay off in increased speed and improved performance of the system.

In the job shop example, each order specifies a quantity of a given product or part, each product or part can be made by one or more processes, and each process can be completed by one or more machine operation sequences. The scheduler may schedule by machine rather than by time slot as in the previous example.

One way to approach this is to build a “preference list” for each machine that lists specific parts or products in order of preference. These parts and products are linked, of course, to specific orders. When scheduling a machine, then, the first “legal” item from its preference list is chosen, if possible. If no choices are available, the machine goes into a waiting status. For machine m_i , the preference list might look something like: {part_3, part_6, part_1, part_2, wait}. Special operators then must be developed to handle the representation. One mutation operator, named the *scramble* operator (Michalewicz 1996), mixes up, or scrambles, the entries of the preference list. One way to do crossover is to exchange preference lists for selected machines. As with standard GAs, the probabilities used with these operators can vary from the beginning to the end of the run.

What is being suggested here, in summary, is that more direct representations of the schedule itself be used as chromosomes. The cost of doing this is the effort required to develop problem-specific crossover and mutation operators. The benefit is faster and more effective system operation.

Case Study 4: Control System Design

This chapter concludes with two case study summaries combined under one heading. The general topic examined is control system design. The simple system used as the subject of the case study summary is a single-axis cart positioning system using a “bidirectional” controller. This cart centering problem has been discussed numerous times in the literature, for example, Bryson and Ho (1975) and Koza and Keane (1990).

As is the case in many, if not most, control design tasks, more than one approach can be used. A neural network approach to control system design is applied to the cart positioning problem first, followed by a fuzzy logic approach that is augmented with evolutionary computation.

The Cart Positioning Problem

The cart positioning problem features a simple one-dimensional system comprising a cart on a frictionless track with a bidirectional control system, illustrated in

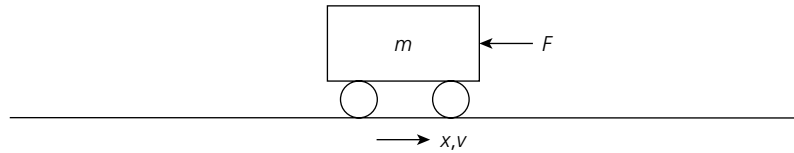


Figure 12.3 One-dimensional cart positioning system.

Figure 12.3. The initial position $x(0)$ and velocity $v(0)$ of the cart are arbitrary within limits. The mass of the cart is m and the force, which can be applied to the cart's center of mass from either the right or the left, is F , which is positive when applied from the right, negative when applied from the left. The objective is to move the cart to a target location x_t on the track in minimum time. The objective is achieved when the velocity drops below a specified value v_{\min} at the same time the cart is within a specified distance Δx of the target location. This problem is a straightforward extension of the cart centering problem analyzed by Bryson and Ho (1975) and others.

In the case of this relatively simple system, the system equations are known and are given by equations 12.5(a) and (b). The parameter Δt is the discretized time increment. It is very important to choose a value for Δt that is appropriate for the problem being solved. A value of 0.02 second is selected in accordance with Bryson and Ho (1975), Koza and Keane (1990), and others who used the value for solving the cart centering problem, although the specific value chosen doesn't enter into our case study approaches to solving the problem. Another system parameter is the maximum number of time steps allowed before the system is considered to have "timed out" without reaching a solution.

Koza and Keane (1990) specified 450; Thrift (1991) used 500 (each for cart centering). Either value seems reasonable, although again the specific value doesn't directly affect our discussion of approaches to solving the example problem. The mass of the cart is set at 2 kg.

$$\begin{aligned} x(t + \Delta t) &= x(t) + \Delta t v(t) & \text{(a)} \\ v(t + \Delta t) &= v(t) + \Delta t \frac{F(t)}{m} & \text{(b)} \end{aligned} \quad (12.5)$$

An "optimal" solution for the cart centering problem using a bidirectional force F was obtained by Bryson and Ho (1975). Force F is applied from the direction shown in Figure 12.3 if equation 12.6 holds; otherwise, F is applied from the other direction. Note that the magnitude of F used by Bryson and Ho is constant. This result is readily extended to the example cart positioning problem by replacing x with $x - x_t$.

$$\frac{v^2 \operatorname{sgn}(v)}{2|F/m|} > -x \quad (12.6)$$

Note that it is frequently not possible to write the system equations as has been done in equations 12.5(a) and (b). It is even more unusual to have an algorithmic solution available as is given in equation 12.6. We would probably implement the algorithmic solution in other situations; here, we are interested in learning *how* to design control systems.

The Neural Network Approach

Neural networks can be powerful tools in control system design. Two main applications exist. The first is in modeling the system to be controlled, as well as modeling the overall system with the controller in place. The second is in the design of the controller.

We first consider the system modeling procedures. Although there are other ways to proceed, often the first step is to model the nonlinear system (without the controller) by training a neural network to predict the system state at the next time increment given its state at the present increment and its inputs. The output of the controller is of course included as system input. In our example case, the controller output is the force applied to the cart. The cart positioning system model is illustrated in Figure 12.4.

In general, if the system equations are known, they can be used to obtain system input/output data. In this case, equations 12.5(a) and (b) are the system equations. Equation 12.6 can be used to calculate a force for each time increment, but the equation is not included in the system model.

If the system equations are not known, input/output data can be obtained by direct measurement. In many cases, the equations will not be known, or they will be very difficult to solve sufficiently well. Relatively inexpensive data acquisition systems can then be used to collect the data needed to train a neural network system model.

Regardless of whether system equations or direct measurements are used, the data should be taken under typical operational conditions. For example, if the control system must catch the cart “on the fly” in many different initial configurations of $x(0)$ and $v(0)$, then data should be taken with a number (perhaps 25 to 50) of such initial conditions, perhaps randomly chosen. It may be that the cart always starts out with $v(0) = 0$ and the goal is to reposition it to some target location x_t . Again, a variety of datasets should be acquired using a number of initial and final positions of the cart.

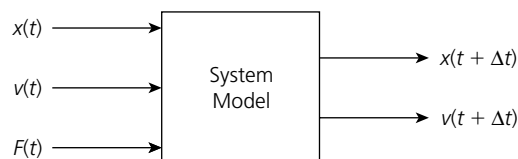


Figure 12.4 Cart positioning system model (without controller).

However these data are taken, they are used to train a neural network that becomes a system model. The datasets are used for training by using the force, position, and velocity values at each time increment as inputs to the network and using the position and velocity at time $t + \Delta t$ as the target values for the network. Back-propagation and radial basis function networks are two candidates for use in the system model. Weights for a feedforward neural network could also be evolved using a genetic algorithm or particle swarm optimization. Once a system model is obtained, the design of the controller can proceed.

To initiate the discussion of the controller design, consider the simplified block diagram of the controller for the cart positioning problem shown in Figure 12.5. Inputs comprise velocity v , position x , and target position x_t . The output of the controller is the force $F(t)$.

The first main step in the controller design is to develop a neural network model of the *overall system with controller*. It is then possible to generate training pattern pairs using this overall model, shown schematically in Figure 12.6. There are several ways this can be done. In our case, we can use equations 12.5 and 12.6 to model the system. (Remember to replace x with $x - x_t$ in equation 12.6.) As before, it is important to model the system over expected operating conditions.

It is also possible in many cases that overall system specifications will exist in the form of system performance curves. A neural network can be trained as a function approximator in these cases. Finally, it is possible that a trained operator may be able to operate the system in the desired fashion, allowing the data for the overall system model to be obtained with a data acquisition system. Whatever method is used, a sufficiently complete series of training patterns is obtained.

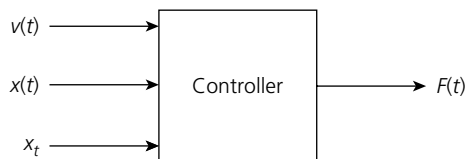


Figure 12.5 Cart positioning system controller.

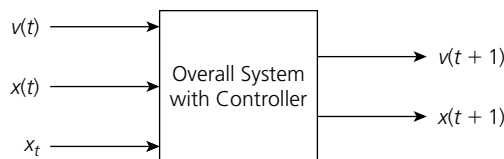


Figure 12.6 Overall cart positioning system with controller.

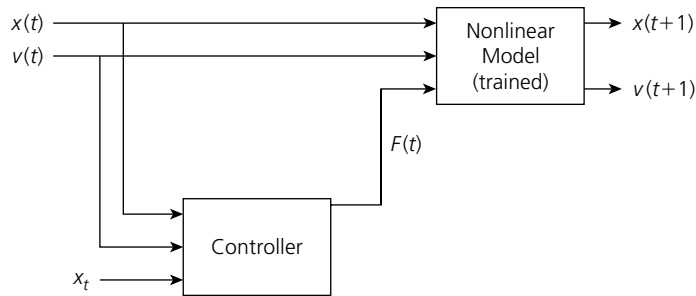


Figure 12.7 Nonlinear model (with weights frozen) with controller.

The second main step in the controller design is to combine the nonlinear model developed above (with weights frozen) with the untrained controller. Recall that the nonlinear model is a trained neural network. The configuration is illustrated in Figure 12.7.

Now the series of training patterns obtained from the overall system model (with controller) is used to develop the controller. With the weights frozen in the nonlinear model, the patterns are applied to the combined system, and the weights of the controller neural network are trained. The output PE of the controller can be one of the input PEs of the trained nonlinear model. Once the controller weights are trained, the nonlinear model can be replaced with the actual system; in other words, the controller can be placed in service controlling the system.

Neural networks have thus been used to develop a nonlinear model of the system to be controlled, to develop a model of the overall system with controller in place and generate appropriate training patterns, and to develop the controller. And it has all been done without solving a single differential equation!

The Fuzzy Logic Approach

A different approach is now described for solving the same problem. We use fuzzy logic, augmented with evolutionary computation, to design the control system. The concept of using a genetic algorithm to evolve fuzzy rules in a control system is described in Karr (1991a, 1991b). Also described is the use of evolutionary algorithms to evolve the locations (peak point and end points) of the membership functions. An approach for evolving fuzzy rules is described in Thrift (1991). We also discuss evolution of fuzzy expert systems in Chapters 7 and 8 of this book.

We assume that the basic groundwork described in the section on the cart positioning problem has been done, as in the case of the neural network approach. The mass of the cart is assumed to be 2 kg, and the time increment Δt is 0.02 sec. The first step is to partition the domains of the input variables (x and v) and the output

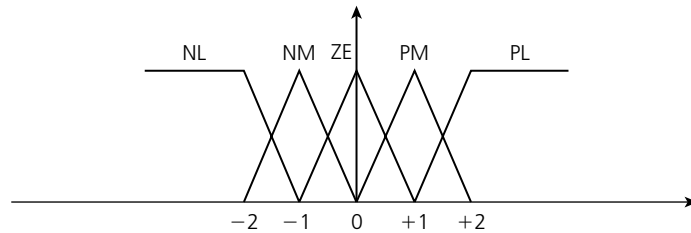


Figure 12.8 Fuzzy membership functions for position error x (meters), velocity (meters/second), and force (Newtons).

variable (F) into fuzzy sets, as illustrated in Figure 12.8. Since the cart can be directed to a specified location x_t , the value x appearing in the fuzzy set definitions, the fuzzy rule set, and so on, is actually the difference between the current location of the cart and x_t . Remember that the objective is to move the cart to the specified location in minimum time.

Note that the definition of the fuzzy sets is made to appear far too straightforward in Figure 12.8. The process has been simplified significantly for the purposes of this example. Defining fuzzy membership functions is typically one of the most time-consuming activities associated with fuzzy logic system design. It is likely, for example, that one or more of the variables could require more (or less) than five fuzzy sets, and it is strictly for convenience, and not having to print three separate figures, that all variables have the same limits and peak points for their membership functions in this example.

Since there are two input variables with five membership functions each, there is a 5×5 matrix representing the 25 possible fuzzy rules for the system. The empty matrix is illustrated in Figure 12.9. Each position in the matrix corresponds to the consequent for the rule whose antecedents are at the heads of the row and column intersecting in the matrix position.

		Position (error)				
		NL	NM	ZE	PM	PL
Velocity	NL					
	NM					
	ZE					
	PM					
	PL					

Figure 12.9 Empty matrix for fuzzy rules for cart centering example.

The situation illustrated in Figure 12.9 is straightforward enough that any competent engineer could probably fill in most of the matrix locations with reasonable guesses. However, we will treat the problem as somewhat more complicated (perhaps of higher dimensionality) so that the method described is more generally applicable. In many practical situations, however, there will be matrix locations that can be filled in. In this case, it should be fairly obvious that the force value of ZE should appear at velocity = ZE and position = ZE. It is a good guess to place a force value of NL at the intersection of the PL column and row and a force value of PL at the NL/NL intersection. The point is not to make the system evolve any more than required. Let's say, for this example, that these are the only three locations we feel sure enough about to fill in.

We are then left with 22 matrix locations. An evolutionary algorithm with 22 variables in each individual can thus be used to evolve an acceptable fuzzy rule set. One coding that would be possible is to let each variable range over $\{0,1,2,3,4,5\}$ corresponding to fuzzy membership functions NL, NM, ZE, PM, PL, and *, respectively, where the symbol * indicates that there is no entry at that position.

If we use a genetic algorithm, one- or two-point crossover could be applied, and mutation could be defined as incrementing or decrementing a variable by 1, where 0 and 5 are contiguous. It would also be possible to let * mutate to any other value (Thrift 1991). This case would also be appropriate for the application of evolutionary programming, particle swarm optimization, and other techniques.

Fitness can be calculated by selecting the maximum number of time steps acceptable for a solution, say 500, and then measuring the fitness of the fuzzy rule configuration for each of a number of initial conditions as $500 - t_i$, where t_i is the number of time steps required for initial condition configuration i . The fitness resulting from perhaps 20 to 30 initial conditions should be averaged to arrive at the system fitness.

Thrift (1991) used a genetic algorithm approach, with a mutation rate of 0.01, a crossover rate of 0.70, and the elitist strategy. (See Chapter 3 for more information about genetic algorithms.) Using a population of 31 individuals over 100 generations, Thrift found a fuzzy rule set that resulted in a control strategy that compared well with the "optimal" bang-bang strategy described in equation 12.6. Results were reported for final "acceptable" values of position and velocity of less than 0.5 each and for acceptable values of less than 0.2 each.

Once the fuzzy rule set is evolved to an initially acceptable level of performance, another evolutionary algorithm can be used to evolve better locations for the peak points and end points of the fuzzy membership functions and, if desirable, the type of membership functions used at each position (triangular or nonlinear, for example). The same fitness function as before can be used (average number of time steps), as well as the same set of initial conditions.

Evolving the fuzzy rule set prior to evolving the shape and location of membership functions can lead to an iterative process. Once the membership functions have been evolved, it may be desirable to "touch up" the fuzzy rule set by running the same

evolutionary algorithm as before with the adjusted membership functions. It may *then* be a good idea to touch up the membership functions again. And so on.

Other Approaches

The use of genetic programming has also been described for the cart centering application (Koza and Keane 1990). In this case, computer programs were evolved using populations of individuals comprising LISP S-expressions. The set of functions used was multiplication (*), the absolute value function (ABS), and a greater-than function (GT). By the fifth generation, an individual program had evolved that represented a control strategy equivalent to the optimal solution of equation 12.6.

It might also be possible to use an approach similar to that in Chapter 8 for classification of the Iris dataset with a fuzzy expert system. In this case, system simulation equations or actual operating data could be used to obtain a number (at least several hundred) of system state vectors. LVQ or another clustering algorithm would then cluster those vectors, and fuzzy expert system rules would be formulated using the cluster centroids. The fuzzy membership functions could then be fine-tuned with an evolutionary algorithm.

Summary

In this chapter, we provide examples of applying computational intelligence to practical problems. For a couple of them, we describe more than one approach to solve the problem. One message we'd like to leave you with is that there are usually several reasonably good ways to design and implement a solution to a problem in engineering or computer science. We hope that in addition to providing you with some new computational intelligence tools for your toolbox, we've given you a new way to look at problem solving.

Exercises

1. Propose an approach to implement an EEG spike detection system that utilizes fuzzy logic. State the advantages and disadvantages of your approach.
2. Describe the data preprocessing for the fuzzy system of Exercise 1.
3. Repeat Exercise 1 using a computational intelligence approach (use more than one CI methodology).
4. Find an application of CI in a refereed technical journal, such as one of the *IEEE Transactions*. Provide a citation for the reference. Summarize the problem and the approach described. Propose another approach to solve the same problem, and state the advantages and disadvantages of your proposed approach.

