Creating Artificial Neural Network Modules For Use In Rapid Application Development

Garrett S. Harris [a], Bruce E. Segee [a], Vincent M. Allen [b]
[a] University of Maine at Orono / [b] Modicon Corporation

Abstract

Neural networks and fuzzy logic have emerged as useful tools for the calibration of arrays of thin film gas sensors. Properly choosing network parameters is essential to achieve acceptable network performance. Often, choosing said parameters involves a time consuming search of many possible candidate networks. When the neural network code is incorporated with in an application with other code, such as for data capture, presentation, and hardware control, interdependencies often form between code segments. Enhancements, modifications, and fixes to the code lead to an extensive and time-consuming rewrite of many parts of the software. Thus, the need arises for neural network software modules that can be easily incorporated in application software but whose interface is well defined and whose implementation is entirely separate from the functionality it provides. By providing debugged and proven software modules encapsulating neural network functionality that can be simply inserted into any application, the entire software system can be modularized. These modules can be reused easily, and changing the neural network operating parameters no longer involves a complete software rewrite or even a recompile. By following the guidelines of rapid application development techniques, and using emerging technologies such as ActiveX and OLE, these modules can be easily developed.

Introduction

Artificial neural networks (ANNs) have emerged as useful tools in a number of areas including but not limited to gas sensor array calibration [Bajaria, 1996]. Artificial neural networks are so named because they employ a large number of simple processing elements and are able to "learn" appropriate behavior based on training data. The training data usually consists of data gathered under known conditions. The training process consists of adjusting network parameters to reduce the difference between the actual network output and the desired output for that data. In a gas sensing application, arrays of thin film gas sensing platforms are normally used. These devices usually have responses to a wide variety of gases, target as well as interferents. While the sensors are responsive to the majority of the gases presented to them, individual sensors respond differently to different gases. By combining the sensors into an array configuration, combinations of sensors can reliably detect specific gases. Artificial neural networks can learn to approximate the actual gas concentrations based on training from data gathered under laboratory conditions for known concentrations of mixtures of gases.

An unfortunate side effect of using neural networks is the complexity the associated code adds to the overall application. Neural networks are complex tools that require complicated mathematical routines. Often the neural network code makes up the majority of the application, and consumes most of the programmer's time both writing and debugging. Simply changing network parameters can require a time consuming rewrite of the code.

It would be much more useful and worthwhile to create a modularized control that effectively "wraps up" the neural network functionality and provides a software "drop in" object. Thus, the neural network code could be written and tested once, and used anytime a programmer needed the provided functionality. This approach would save programmers time and effectively money, which would otherwise be spent "reinventing the wheel" [Harris, 1999].

Thin film chemiresistive gas sensors

Recent advances in thin film chemiresistive semiconductors have fueled the development of gas sensors for the detection of many different target gases [Bajaria, 1996]. A rendition of a thin film chemiresistive sensor is shown in Figure 1. These sensors are capable of detecting gasses at very low concentrations, quickly return to a baseline after exposure, and have a broad range of concentrations to which they are sensitive. The nature of the sensors makes them very sensitive, but non-selective. By combining several different sensors into an array, the overall array response can be made unique for any mixture of gases, even though all of the individual sensor elements may respond to all of the gases in the mixture, and indeed, individual sensors may respond the same to different gases. Since each individual sensor reacts to a gas differently, the aggregate response is therefore different for each gas. The mathematical modeling of these arrays is very complex, due to their decidedly nonlinear nature; also analyzing the data requires arbitrary multidimensional nonlinear mappings between the sensor response and the actual gas concentration. Thus, determining the relationship between sensor response and gas concentration would be difficult if not impossible. One of the most promising methods of dealing with these shortcomings is Artificial Neural Networks.

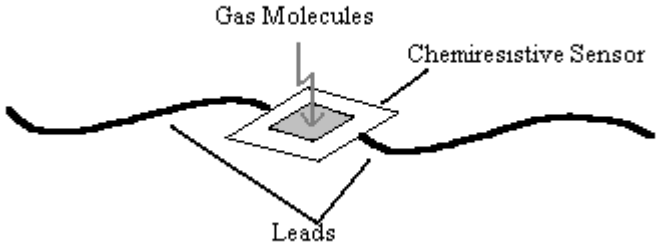

Figure 1. Thin Film Chemiresistive Sensor

Artificial Neural Networks

Artificial Neural Networks (ANNs) are systems capable of generating a non-linear mapping from one large set of variables (e.g., sensor response) to another (e.g., gas concentration). ANNs are adaptive, in that their mappings are effectively "learned" from a set of training data. The

conceptual idea behind these mappings is depicted in Figure 2. The neurons mathematically map one or more inputs to one or more outputs. To develop an ANN for detecting a target gas, one would use the training data with known concentrations through the network to adjust the network parameters, and then test the system using a set of data collected, but not used for training. ANNs are also capable of generating an output for an unknown or unencountered input by generalizing the response from the training data.

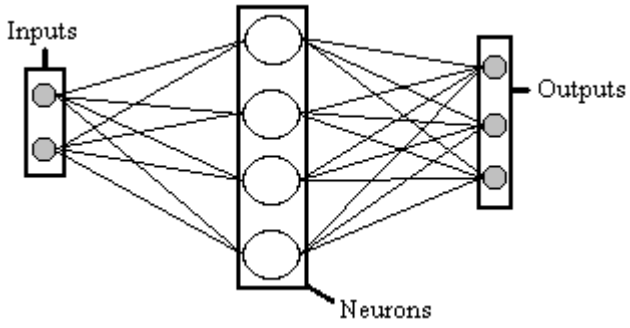

Figure 2. Conceptual Design of an ANN

ANN functionality has been and can be applied to a wide variety of system control tasks. While it is possible to efficiently implement ANN functionality in software, the effort required is still an issue. Thus, if a software module were designed to allow an ANN to simply be added to another project without having to write or even understand the complex code associated with it, considerable time and money would be saved.

Rapid application development techniques

Perhaps the greatest advance of this decade in software engineering has been the introduction of rapid application development techniques, most notably being the idea of software modules that can be easily reused and combined to make whole, fully functional programs. While the idea of modularity isn't new, it has only recently been realized with the introduction of Microsoft Windows and ActiveX. This allows developers to create the aforementioned modules, and simply drop them into any application. Because the modules are entirely self-contained, they can be reused with a variety of applications. This allows a complex piece of code, such as a neural network, to be written, debugged, and proven once and then reused multiple times with ease. When developers wish to write applications, they simply need to write the code that "glues" the modules together. This has the net effect of considerably reducing development time as well as reducing the number of errors encountered.

Overview of software design

Computers have evolved tremendously since the days of punch cards and vacuum tubes. Today's PCs can outperform mainframes of decades ago with many clock cycles to spare. Software engineering has kept pace with these advancements. From the early days of assembly language programming to today's custom controls, as computers get faster software engineering techniques become more and more advanced.

The early years

At the start of the computer era, the only tool for programming was assembly language. Essentially one step up from machine code, assembly is a platform specific language that is "assembled" into byte code and is subsequently executed on the processor. In the early years, every time a new application was desired, a programmer would sit down and code an entire application in assembly. Applications had to be written specifically for the computer they were run on, and if different applications with some similar features were desired, then those features would essentially need to be written twice. Assembly also provided the least amount of abstraction of the system that was being programmed. While this allowed programmers greater control of the underlying system, it meant that each and every operation had to be encoded instruction by instruction.

While assembly language may have seemed a much better alternative to programming actual byte code in the past, today's tools have far surpassed assembly's limited feature set. As computers became more powerful, more robust development environments replaced assembly language.

C is for reusable libraries

In the early 70's, the programming language known as C was developed at Bell Labs [Pheatt, 1996]. Considered to be one of the most important software development tools of the time, C brought a host of new features to programmers, making their job much easier. One of the greatest advantages of the language from a software engineering standpoint was the idea of reusable libraries. These allowed programmers to code functions once and subsequently include them as libraries wherever they were needed. This was the first step down the long road to the idea of reusable code modules that we know today. C also had a robust set of built in control structures, data types, and built in functions. This considerably reduced the amount of overhead programming that was required, allowing software engineers to concentrate on the applications that they were writing.

C turned out to be so popular that it was later formalized as ANSI Standard C, and is supported across multiple platforms. However, even with all of the niceties that C brought, there were still quite a few difficulties. While it was possible to create reusable libraries, all too often when other programmers went to use library subroutines, they often found them either too specific for their current application, or so general that they were difficult to understand. There was still the need for a software development tool that could provide the modularity Rapid Application Development (RAD) called for.

C++: like C, only better

Perhaps the next software engineering advance of note is the introduction of C++. In the early 80's, C++ was introduced as an extension to the C language that provided some additional features [Stroustrup, 1994]. One of the most important new features was the idea of classes. Classes were designed with Object Oriented Programming (OOP) in mind. Essentially a library

of functions and the data types that said functions supported, classes could simply be included into any project that their functionality was desired. Classes took the idea of reusable libraries one step further by adding the data types into the class definition. With the advent of classes, the idea of reusable software modules that were already coded and could simply be added into a project was becoming more and more of a reality. C++ also introduced the idea of constructors and destructors. These were routines in the class that described how to create an instance of it. They allowed classes to be dynamically allocated and destroyed, as needed.

C++ was specifically designed to provide a basic form of software reusability. The idea of classes that could be used, reused, and shared allowed programmers to work more efficiently in teams. Likewise, one programmer didn't need to know how another programmer's code worked, as long as he had a basic idea of the classes interface. Unfortunately, this was the Achilles heel of classes. Because of the non-standard interface, classes written by one programmer were often harder for another programmer to use. Getting a basic idea of the classes' interface could actually take another programmer longer than writing their own class. This led to programmers writing their own classes specifically for their own application, nullifying the effectiveness of the idea behind classes. Also, while C/C++ was more widely used than other software engineering tools, a few other environments, such as Pascal, were becoming popular. Classes were written in and specifically for C++, but it wasn't possible to import a class into Pascal, or even ANSI C.

Classes were a step in the right direction, but by being unable to provide a standard interface programmers were still faced with the need to figure out how someone else's class functioned before they could use it. As classes got very complex, programmers spent more time figuring out the interface to the class than it would have taken them to write their own class. Additionally, it would have been beneficial if classes could be reused outside of C++. Thus, a tool was needed to modularize software components, but also needed to provide a standard interface and be used across multiple development tools.

The coming of OLE

In 1991, Microsoft made available to developers a new technology that it called OLE, or Object Linking and Embedding [Brockschmidt, 1996]. OLE was initially designed to facilitate the easier creation of compound documents. An example of a compound document is a Word document that contains some graphics and an Excel spreadsheet. The idea of OLE extended beyond just Microsoft products, allowing any application conforming to the OLE specification to share objects with other OLE applications. The document in which an object was placed retained the information about the format and the native data used to create the object. If the object needed to be edited, a simple double click placed the data in the original editor.

The designers of OLE soon realized that they were on to something far greater than compound documents. They had actually designed a specific instance of reusable modules. So, in 1993 Microsoft released a completely redesigned OLE, known as OLE 2.0 [Brockschmidt, 1996]. OLE 2.0 retained and improved on the idea of compound documents, but boasted a completely redesigned infrastructure to support component software. This infrastructure is known as the Component Object Model, or simply COM. COM provides the resolutions to the difficulties encountered with other solutions to the lack of reusable objects. The scaffolding to support

reusable software components is provided by COM, and therefore OLE, which moved from a specific technology to an extensible systems object technology with an architecture that accommodates new and existing designs.

OLE and COM also provide the means for a standard interface into components and objects. These objects are entirely self contained, and can be used and reused numerous times. Borrowing from the dynamic creation of classes in C++, OLE and COM objects can be dynamically created and destroyed as software needs them and their associated functionality. OLE calls for object to have properties and methods. Properties are attributes of the object that can be changed or set. Methods are the functions that the object provides. Even programs such as Microsoft Word and Excel expose properties and methods that can be used in other applications through OLE. Using Microsoft Excel as an example, the name of the current sheet would be a property that could be set, and selecting a column of cells would be a method that could be called.

ActiveX Controls and "glue"

ActiveX controls are objects based on COM yet provide services substantially different than OLE. While OLE uses COM to provide high-level application services, ActiveX provides a slimmed down infrastructure that is optimized for size and speed. Additionally, ActiveX allows controls to be embedded in web pages, thus their need to be small and fast. ActiveX controls have a fifty to seventy percent reduction in size, and support several Internet innovations [Microsoft, 1996]. Despite these differences, ActiveX controls evolved from OLE controls, and are components that can be added to a project to reuse packaged functionality that someone else designed. ActiveX controls can be used in many programming languages, including all Microsoft programming and database languages, such as Visual Basic, Visual C++, FoxPro, and even Fortran.

ActiveX controls have many features that make them very suitable for the design of reusable software objects. The objects are said to be functionality oriented, meaning that the concern lies more with the function of the object than how to implement it or interface with it. Since all ActiveX controls have a standard interface, there is no need for the programmer to know the details of the control’s implementation. By dropping it into the desired project, the programmer has access to all of the methods and properties the control provides. Thus, the programmer need only be concerned with the functionality of the module.

ActiveX modules also allow multiple interfaces. This allows the module to be accessible in many different environments. For example, a neural network control could be "dropped" into a Visual Basic project, or it could be used in conjunction with Microsoft Excel. This is due in part to the multiple interfaces the programmer has given the module. Providing multiple interfaces to an object allows that object to be used in a much wider variety of situations.

With the advent of ActiveX, the tools are in place for the design of reusable software modules. These modules all conform to a standard interface, work across multiple development environments, and can easily be shared amongst programmers. Thousands of ActiveX controls are available today, ranging from a timer control to a full-featured word processor. To take

advantage of these controls, one simply needs to write the "glue" that holds the application together. This is most easily accomplished in Visual Basic. ActiveX controls are simply "dropped" onto a form. These controls include the aforementioned, as well standard buttons, text boxes, etc. The programmer then simply writes a small portion of code that ties or "glues" the whole project together. This includes tasks such as taking data from one control and passing it to another, and presenting said data. Thus, in a relatively short amount of time complex applications can be developed.

Development of an Artificial Neural Network Control

Using the aforementioned techniques, it was desired to design a neural network control that could simply be dropped in to any project where its functionality was desired. By choosing an ActiveX implementation, the control would be easily integrated into Visual Basic code, cutting software design time considerably. Implementing multiple interfaces allows the control to be used both in a programming environment, as well as Microsoft Excel, which is the native format of the gas sensor data that is received.

The control was designed and debugged in Visual C++, by a programmer with extensive experience in both C++ and neural networks. By choosing to implement the ANN as a control however, programmers with little or no experience with ANNs can use their functionality in the software that they develop. This was the driving force behind designing the ANN as a control. Not only that, but the code is entirely reusable in a variety of situations. This is due to the lack of front end on the control. The ANN control as designed has no facilities for data presentation. It merely returns the raw data and relies on the programmer to "glue" the control together with some manner of a data presentation control or program. This was done purposefully, allowing the control maximum effectiveness in a variety of programming tasks. By allowing individual programmers to determine the front end, the control is fully universal and not specific to one application.

As previously mentioned, ActiveX controls interface to other objects with properties and methods. The ANN control has numerous properties for things such as learning rate, generalization, and other operating parameters for the ANN. After setting the network operating parameters to the desired values, one can simply call methods like TrainNetwork, TestNetwork, GetTestError, and the like to have the ANN operate on the given data. Excel is available as an object to VB programmers, and its functionality was incorporated in the ANN control to provide data storage functionality in an environment that is easily viewable and portable. The data is stored in Microsoft Excel format, allowing the data to be easily viewed afterwards outside of the application.

The development of software utilizing the control is very simple to develop and deploy. This is not only due to the reusable, modular nature of the control, but when coupled with the RAD features of Visual Basic (VB), development times are slashed. Once one has determined the function of the software, one creates a form or forms in VB. Forms are essentially "containers" that hold controls and other objects. After the creation of the forms, the next step is to usually drop all of the desired controls onto the form. Examples of these are buttons, text boxes, simple graphics, serial communication, etc. After all of the controls are dropped onto the form, the

programmer can proceed to write the code that ties the controls together. This is furthered hastened by VB presenting the programmer with comprehensive lists of all the properties and methods that can be used for each control. Finally, after these few short steps, the debug process can begin. After a normally short debug period, the code can be compiled into an executable, and then packaged and deployed. Packaging and deploying involves generating a series of cabinet, or archived files containing the executable and those controls associated with it. Also generated is an installation program that not only installs the main program but also any needed controls onto the user's computer.

Example VB Form

Shown in Figure 3 is a sample Visual Basic form utilizing the ANN control. This particular application was designed for training the neural network weights. The form is essentially "blank" to start with, and all of the objects seen above were placed on the form by the developer. All of the buttons and text boxes are themselves ActiveX controls, while the form serves as a container for them. The ANN has already been placed on the form, as seen above the arrow. All that is required to bring the ANN functionality to this program was merely placing the icon representing the control onto this form. After placing the control, all of its associated properties and methods are available to the software developer. For example, the text box "Network File" has a property called text, which refers to the text it displays. Clicking the "Load Network File" invokes the click method of the button. This in turn gets the text property from the "Network File" box, and calls Excel to load the file. This is all accomplished in a few simple lines of code, demonstrating the ease of use of programming with components.

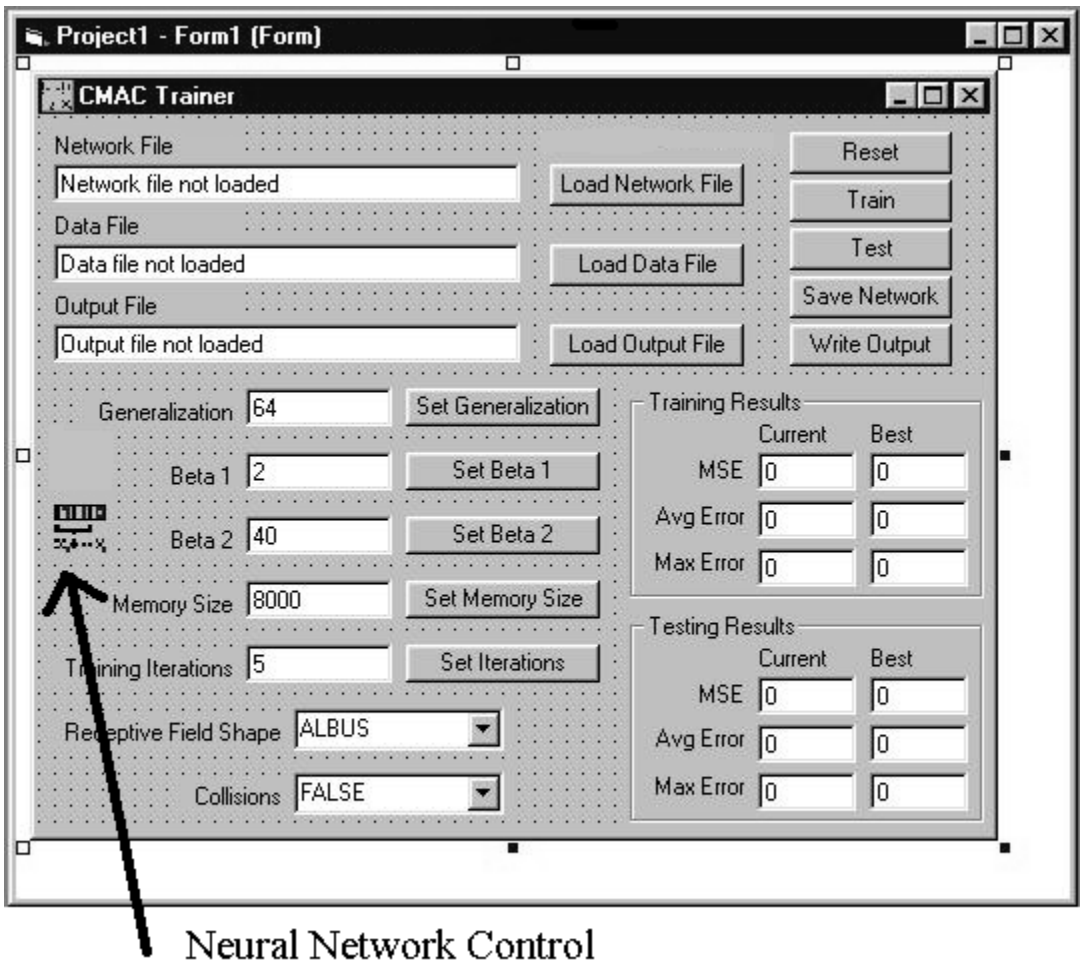

Figure 3. Example Visual Basic Form

To further examine the simplicity that the control provides, Figure 4 below shows the code required to initialize and use a Radial Basis Function (RBF) neural network. The first step is to drop the control onto the form, as in Figure 3 above. Then the object RBF1 is available to the programmer. The first line loads an Excel file containing the training and testing data. The next four lines set parameters required by the network for training and operation. The second to last line is the actual training and the last line invokes the method of the control that tests the data. The output is written to a standard Excel spreadsheet file, which can then be opened and manipulated in anyway the user sees fit.

```
RBF1.DataFilename = "C:\example.xls"
RBF1.Epochs = 1000
RBF1.LearningRate = .3
RBF1.MSETarget = .1
RBF1.Variance=2
RBF1.TrainNetwork
RBF1.TestNetwork
```

Figure 4. Visual Basic Code Fragment

Conclusion

Software engineering

Since the dawn of the computer era, application development techniques have evolved to keep pace with the cutting edge technology. The trend in software engineering techniques is akin to the trend of house building techniques in modern America. In the early days, when it was desired to build a house, one would go in the forest and cut down trees and then saw and plane them into the desired length and thickness. Next, one would forge nails to hold the boards together. Perhaps one would then go and select each stone for the chimney. This is not unlike the first days of software engineering when programmers coded in assembly. As time progressed, sawmills popped up that would cut the trees to standard lengths and plane them to standard thicknesses. Also, standard size nails were forged. Now, when it was desired to build a house, one could buy standard size boards and cut them to desired length and buy nails to put them together with. This is similar to the process of code design with C/C++. Programmers had reusable libraries that could be used in their code, but like the boards had to be tailored to their specific application. In modern times, it is possible to buy sections of houses already built, and join them together in any manner desired. A whole house can be made literally room by room, with a minimal amount of construction required. All the homebuilder needs to do is join the ready-made modules together. This is completely reflected in the design nature of ActiveX controls. Previously written components are joined together with minimal amounts of code.

The progression of software engineering to reusable components has significantly reduced the amount of time for developers to go from the initial conception of an idea to a finished product. Developers have been given the tools they need to quickly produce powerful applications. Not only has ActiveX technology cut development times, but by providing a standard interface has allowed software engineers to tightly integrate their modules with those created by others. This standard interface has played a major part in the success of ActiveX. The standard interface allows the implementation of the object to be abstracted away from the user. This allows complex controls to be created, and by adhering to a standard interface, makes the usable by anyone familiar with the standard.

Development of an ANN control

By implementing a neural network as a control rather than as an entire program, programmers can easily integrate the code into their project whenever the ANN functionality is desired. Developing an ANN control in Visual C++ allowed the power of the C language to be used to develop the control and facilitated the code's design as a control. While not the only language in which to write ActiveX controls, C++ provides a host of templates and wizards to speed the design process. After the control was compiled, it could easily be dropped into any desired project. The aforementioned ANN control is usually combined with other controls in the VB development environment to provide presentation of the data. This allows robust applications to be created in less time, as the ANN control is already debugged and proven. In addition, the programmer responsible for coding the application need not understand the implementation of the ANN, as the interface abstracts the implementation away from the user. By allowing multiple interfaces, the control can be used in variety of situations. In most circumstances, the control is

dropped onto a VB form as mentioned previously and integrated with other controls that provide data presentation. However, the control can also be integrated into a Microsoft Excel spreadsheet. This allows the methods and properties of the control to be accessed from Excel, essentially providing the functionality of the control right inside the spreadsheet.

Applications for Chemical Sensors

The applications for chemical sensors for this software are virtually limitless. ANNs can be used in a variety of situations involving calibrating and determining the output of the sensors. Since the majority of sensors have nonlinear multidimensional mappings from the input to the outputs, neural networks are perfect candidates for data analysis. Again, by providing a drop-in module capable of neural network processing, the desired functionality can be easily incorporated in a variety of applications. Since chemical sensors and neural networks are so well suited for each other, it only makes sense to have a module that can be easily deployed available for use. By deploying the ANN software on a PC platform, the processing power, storage space, and presentation architecture can be fully utilized. PCs provide more than sufficient processing power to handle the advanced computation required by neural network code. In addition, the PC platform is also best suited for the presentation and display of the data. That coupled with the price performance curve of today's PCs makes them an ideal choice from a monetary standpoint.

In the future, one would expect more of a trend toward the use of Artificial Neural Networks due in part to their unique processing ability. Within the PC platform, ANNs have found a good mix of processing power, presentation capability, and price performance. A reusable, modular, plug-in software object conforming to the Microsoft Component Object Model specification will only help to further facilitate the use of Artificial Neural Networks in a chemical sensor software package.

References

[Bajaria, 1996]	P. Bajaria, "Calibration of Solid State Gas Sensors Using Cerebellar Model Arithmetic Computer Artificial NeuralNetworks", UMaine Masters Thesis 1996
[Brockschmidt, 1996]	K. Brockschmidt, "What OLE is Really About", http://msdn.microsoft.com/library/techart/msdn_aboutole.htm, 1996
[Harris, 1999]	G. Harris "Neural network programming using rapid application development techniques", SPIE Conference Proceedings, 1999
[Pheatt, 1997]	C. Pheatt, "A Brief History of C", http://academic.emporia.edu/pheattch/cs207s96/hand2.htm, 1997
[Microsoft, 1996]	Microsoft Corp., "ActiveX FAQ", http://msdn.microsoft.com/library/backgrnd/html/msdn_faq.htm, 1996
[Stroustrup, 1994]	B. Stroustrup, "The Design and Evolution of C++", Addison Wesley, Reading, MA. 1994