Artificial Intelligence Voice Recognition and Beyond

by Nathan Papke

nathan.papke@juno.com

For years, programmers have been trying to make computers think for themselves. Science fiction writers show us how the world could be if machines had consciousness. Unfortunately, making this so seems to belong in the realm of the gods.

To know how artificial intelligence works, it is important to know what intelligence is. Most dictionaries and encyclopedias would say that there are three things that make up intelligence. They are consciousness, the ability to learn, and the ability to use learned knowledge.

Consciousness being too advanced, I will start with the second attribute of intelligence. The ability to learn is probably the easiest thing for a computer to do. Anything presented to the computer that it does not recognize can be put into a knowledge base. A knowledge base is pretty much a database with relational links between objects, but I'll get into that later.

The third attribute, the ability to use learned knowledge, is slightly more complex than the ability to learn. It requires an algorithm to search through the knowledge base and relate that knowledge with what the AI is presented with. Such an algorithm would be fairly complex due to the possibly infinite paths that it could take in the knowledge base.

Of course, the biggest problem for a computer to collect and use its knowledge is the difficulty in recognizing data presented to it from the real world. This may not be such a problem in a virtual world where everything is already clearly defined, but real world data requires preprocessing. Fortunately, computer science has given us the neural net to deal with this problem.

And finally, to deal with the problem of storing the knowledge, we can use a knowledge base. Like I said, knowledge bases are basically databases with relational links between objects. For example, a bird has wings, so the knowledge base entry for bird would contain a link indicating that wings are part of the bird. Before you know it, a knowledge base could look like a complex web of relational links.

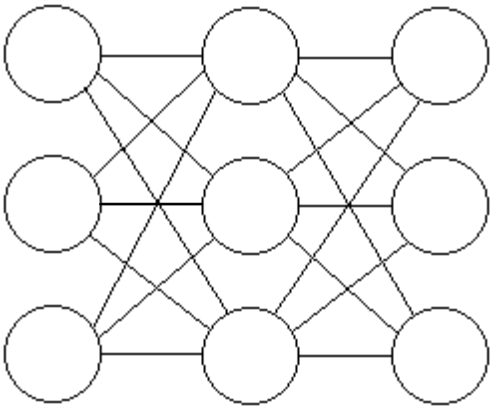
Pattern Recognition

Neural network technology developed as a way to model what a group of neurons do in an organism's nervous system. Over time, programmers have used this technology to develop complex systems of neurons, which can recognize patterns in much the same way that you or I do. Images, sounds, and even the AI's actions can be processed with neural networks.

One of the most popular neural net systems is the back-propagated neural network. It was independently discovered by one team and two individuals. They were the team of D. E. Rumelhart, G. E. Hinton, and R. J. Williams, and individuals Y. Le Cun and D. Parker. This type of neural net contains an input layer, an output layer, and any number of hidden layers. Each layer can have any number of neurons, but the input and output layers should have enough neurons for the data that they calculate. The back-propagated neural network learns by back propagating, or giving feedback to the layers in reverse order and adjusting a set of weights.

How would the structure of a back-propagated neural net look? Take a look on the next page at Figure 1. It shows a neural net consisting of three layers: an input layer, a hidden layer, and an output layer. You can see that each neuron of one layer has a connection to each neuron of an adjacent layer. Each connection has a value from 0 to 1 associated with it known as its weight. The weight is analogous to how close the synaptic connection is with 0 being too distant to affect the next neuron and 1 being so close that the synapses touch each other.

Figure 1



With that in mind, how do we write code to mimic this behavior? Check out Listing 1. In it, I have created a series of **for** loops. Also, I have included a structure with all of the neural net's data so that we don't have to pass everything individually. Each neuron in a layer has its own value. For every neuron in the following layer, the preceding layer's neuron values are multiplied by their corresponding weights and added together.

Listing 1

```
struct NEURALNET
{
  float Input[512];              // the input data
  float Hidden[64];              // the hidden layer neurons
  float Output[8];               // the output data
  float Weight1[512][64];    // the first set of weights
  float Weight2[64][8];       // the second weight set
  float Target[8];

  float dk[8];
  float dj[64];
};
```

```
void Net_Recognize(NEURALNET Net)
{
  int i, j, k;
```

```
  for(j=0; j<64; j++)
  {
        float sum=0;
    for(i=0; i<512; i++)
    {
        sum=+(Net.Input[i])*(Net.Weight1[i][j]);
    }
    Net.Hidden[j]=1/(1+exp(-sum));
  }
  for(k=0; k<8; k++)
  {
        float sum=0;
    for(j=0; j<64; j++)
    {
        sum=+(Net.Hidden[j])*(Net.Weight2[j][k]);
    }
    Net.Output[k]=1/(1+exp(-sum));
  }
}
```

Now that we know enough to make the neural net recognize something, how can we get it to learn so that it will be of use to us? That's where the back-propagation comes in. If you look at Listing 2, you will see that we will need a target input and a target output to be associated with that input. First, the neural net tries to recognize the target input. If the output is different from the target output, then the set of weights between each layer are adjusted. It may take many times for the neural net to eventually correctly recognize the target input, so you have to be patient.

Listing 2

```
void Net_Learn(NEURALNET Net)
{
  int i, j, k;
        Net_Recognize(Net);
        for(k=0; k<8; k++)
  {
        Net.dk[k]=(Net.Target[k]-Net.Output[k])*(Net.Output[k])*(1-Net.Output[k]);
  }
  for(j=0; j<64; j++)
  {
        for(k=0; k<8; k++)
    {
        Net.Weight2[j][k]=+(Net.dk[k])*(Net.Hidden[k]);
    }
  }
  for(j=0; j<64; j++)
  {
        float sum=0;
    for(k=0; k<8; k++)
```

```
    {
        sum=+(Net.dk[k])*(Net.Weight2[j][k]);
    }
    Net.dj[j]=(Net.Hidden[j])*(1-Net.Hidden[j])*sum;
  }
  for(i=0; i<512; i++)
  {
        for(j=0; j<64; j++)
    {
        Net.Weight1[i][j]=+(Net.dj[j])*(Net.Input[i]);
    }
  }
}
```

A common use for a neural net is to give it a complex set of data and have it give us a simple code. One example would be to give it a hand-written capitol letter A and have it give us the 8-bit ASCII code for it. It is also possible to give it sound from a human voice and have it give us a code for each phoneme so that we can do voice recognition in our programs. I will go over voice recognition later on in this article.

We can take a complex input and have the neural net recognize it as some simple code, but can we give a neural net a simple code and have it give us more complex output? Why not? For example, we can give a robot a code that tells it which way to walk, some gyro information, and pressure on each foot and it will give us some fairly complex data on how its servos should move to get to where it's going. This is very helpful if part of the robot's leg is damaged and it needs to figure out how to walk in that condition. It would simply use the learning algorithm while it is trying to walk.

You've seen how a neural network works. It can recognize input and even learn new data. With that in mind, lets move on to putting that knowledge to use.

Voice Recognition

Voice recognition has been a problem for software developers for years. For one thing, everybody's voice is different. You may think that's obvious, but if you've ever had a chance to plug a microphone into an oscilloscope, then you know what I mean. The other problem is computer speed.

With the increase in computer speed over the past few years, we don't have to worry about that, but the difference in people's voices is a problem. Neural nets are a

simulation of a biological process, so lets see if we can figure out how to model the way an ear hears. An ear contains many little hair-like structures that have various resonant frequencies. When a phoneme is broken up into its spectrum, it may be easier to recognize.

Breaking up a wave form such as a sound can be done with a Fourier (pronounced "foray") transform. A Fourier transform takes twice as many samples as the rate of the frequency of the sound. In return it gives you half as many harmonics as the number of samples it is given.

Doing a Fourier analysis is fine if you have all of the time in the world, but we need to optimize it for speed so we can run it in real time. Fortunately, there is what is known as a fast Fourier transform. The constraint with using a fast Fourier transform is that the number of harmonics must be an integral power of 2.

As a general rule of thumb, you could say that human hearing has a range from about 32Hz to 16,384Hz. Actually, it's a little greater than that, but this simplifies the math. The number of harmonics is 512, an integral power of 2. Not only that, but 1/32nd of a second would fit nicely into a MIDI file if it uses 1 beat per second.

A fast Fourier transform, or FFT, even looks simpler than a regular Fourier transform. Listing 3 shows a **for** loop to change harmonic and a **for** loop in that to add the data together. If you look at the code, you will notice that we can perform some pre-calculations to speed up the function. Since we already know that we want k, the number of harmonics, to be 512 and n, the number of samples, to be 1024, then we can also unroll the loops for even more speed. Doing this will cut three multiplies and one exponent from the function 524,288 times, but it would take almost forever to write.

Listing 3

```
void FFT(float *X, float *Y)
{
  int k, n;                                   // integers for the loops
  complex mult;                               // a complex number
  float prod;                                 // a floating point number for the real
component
  complex i=complex(0, 1);                    // the square root of -1

  for(k=0; k<512; k++)                        // loops through each sub harmonic
```

```
  {
    for(n=0; n<1024; n++)                          // loops through each sample
    {
      mult=exp((-2*M_PI*i*k*(n+1))/1024);          // calculates the equation
      prod=real(mult);                             // gets the real component
      Y[k]=+X[n]*prod;                             // multiplies the real component
    }
  }
}
```

Maybe it would be possible to write a program that will write source code for us. Well, it turns out that it is possible. Just write a console application that writes to a file. I'll let you figure this out, but it's a good way to be able to tell your friends that you did 500,000 lines of code in less than an hour. If you give up, you can check out the "unroller" programs that I have included with this article. The only problem is that you'll need about 1 GB of memory to compile the source code that's generated!

Similarly, we can unroll the loop for the neural network, but first we might want to figure out how many neurons we want in the hidden layer and if we want more than one hidden layer or not. We can do this with a genetic algorithm.

Genetic Algorithms

A genetic algorithm, as its name implies, models genetic evolution. In the case of artificial intelligence, it models the evolution of an idea. Doing this gives the computer power to figure out many different problems. As long as the algorithm runs, it gets closer and closer to its perfect solution.

First, a genetic algorithm generates sets of random numbers. Each set of numbers is tested and rated based on how close it gets to the target solution. After that, numbers from sets are mixed and randomly mutated to create the next generation. Eventually, a set of numbers from one of the generations gets as close as possible to the target solution.

Creating a genetic algorithm for a neural network would lead to having an almost perfect network configuration. Such a neural net could recognize a phoneme with a 100% accuracy in as little time as possible. Doing so could even find a new paradigm that would be best suited for voice recognition. Anyway, the biggest problem is that both genetic algorithms and neural nets take allot of processor cycles and the effort of training thousands of neural nets to find which one has the optimal configuration would take a

long time. As a result, I will simply use a back-propagated neural net with 512 input neurons, 64 hidden neurons, and 8 output neurons in the voice recognition program for this article.

I also mentioned that using the knowledge in a knowledge base would take a complex algorithm due to the possibly infinite relational links. It turns out that a genetic algorithm would be ideal for such a purpose. Imagine when you've tried to figure something out and your thoughts stray as one leads to another. If one idea doesn't work out, you usually return back to the thought of, "How do I…" That's the same concept behind a genetic algorithm.

Knowledge Bases

Okay, now that you have an idea of the complex workings of how an idea forms, lets move on to what a knowledge base is and how it affects the AI. Like I said before, a knowledge base is basically a database with relational links between entries. But how do we add these relational links to entries in a database? After all, not every entry will have every type of relational link. And not every entry will have a uniform number of each type of link.

Some various relational links will be used for different types of words. For example, objects could have links indicating other objects that are part of that object or links indicating actions that are performed by or can be performed with that object. Actions could have links for objects used to perform that action. Any way that one word relates to another word can be indicated by relational links in a knowledge base.

It may be possible to create a knowledge base with an object oriented database. Unfortunately, object oriented databases are still in a relatively primitive stage. Since our only need for a knowledge base right now is voice recognition, we only need fields for the word, pronunciation, and the part of speech. For now, it would probably be best to wait for someone to release an object oriented database program on the market before worrying to much about creating a knowledge base.

Personalities

Personality is what makes each of us unique and it can be applied to artificial intelligence characters as well. The idea behind a personality is not too complex; a data structure would contain information about a character's mood, pointers to memories, and a list of goals. Using a personality structure, you would simply pass a pointer to it to an AI engine.

Uses for a personality would range from games to social user interfaces. Characters in games would be more interactive because instead of simply figuring out how to beat you, they could carry on a conversation. I'm sure you've seen movies where the good guy talks his or her way out of a situation. Even a social user interface would have some personality. Do you remember HAL? Okay, bad example, but the point is that any AI must have personality.

Now that I've shown you some of the highlights of artificial intelligence, maybe you'll put them to use in the next best selling computer game. Maybe your next computer will have a user interface like one on the Enterprise. Heck, maybe we can even get robots to mine the asteroid belt for us. Whatever happens in the world of artificial intelligence though, it will be part of a new and exciting frontier.