The Design and Implementation

of an Artificial Neural Network in ANSI C

David Plotz

11/26/14

B. Hecker

Introduction

To create artificial intelligence, a natural first step would be to imitate the biological organ from which intelligence springs. The human brain is made up of around one hundred billion neurons, with each neuron connected to thousands of other neurons, creating a vast network of almost unimaginable complexity [1]. In an Artificial Neural Network (ANN) we mimic the biology by first creating artificial neurons and then chaining these together into networks of various shapes and sizes.

Over the last thirty years there has been an incredible explosion of applications of ANNs. From computer vision and pattern recognition, to natural language processing; from stock market prediction to non-linear system modeling, ANNs have proven again and again how useful and powerful they are[2] . For my project, I was not be as ambitious as in the previous examples, instead I contented myself with simply creating an ANN from scratch. The ANN I created, while simple, still possesses many of the powerful features of industrial ANNs, most importantly, I have given my ANN the ability to learn.

In this paper I will provide a step-by-step walk-through of both how I implemented my ANN and also why I chose to make the design decisions that I did. Along the way, I will discuss working with individual bits, perceptrons, topology, three-dimensional matrices, catastrophic forgetting and backpropagation.

Language Choice

The first design decision that I had to make was what language I would implement my program in. For me, the only two possibilities were C or Python. I have been programming with Python for quite some time, but have only recently picked up C. I ended up choosing C both because I wanted to get some more practice with C and because I wanted to implement a really efficient system and one can usually get fast performance using a minimum of space using C. In retrospect this was probably a mistake. Given the complexity of the task, Python would have been a better choice for at least two reasons. One, Python has a way of making the complex seem, if not trivial, at least tackle-able. Two, because of the aforementioned complexity, I was not able to implement the program in a clean and precise way and was therefore unable to utilize the speed and efficiency of programming in C.

The nice thing about C is how close it lets one get to the machine. Instead of operating on bytes, or even larger data structures, one can actually operate at the individual level of bits. With this said, it is still not the easiest thing to do, because the C language's smallest “word” is the character, which contains one byte of information. In order to manipulate the individual bits, one has to use 'masks'. Here is a little procedure to print the individual bits in a byte:

#define BYTE 8

void print\_byte(char a){

int i;

for(i = 0; i < BYTE; i++) {

printf("%d", !!((a << i) & 128));

}

printf("\n");

}

Obviously all the magic is contained in that seemingly simple printf statement. The easiest way to understand the statement is to work backwards. 128 in binary is 10000000, we then take the bit-level & operator and do an and operation. As I hope should be clear, 128 & any byte will either equal 0 or equal 128. On the left-hand side of &, we see a<<i, << is the symbol for bit shift to the left, thus with each incrementation of i we are bit shifting over one more. Thus when i is zero, we are going to pick off only the left-most bit of the character. When i is one, we pick-off the second left-most bit, and so on. Finally, we come to the double !!, as we just mentioned, the &128 gives only two possible answers: 128 and 0. If the answer is zero then the first ! turns this into one and the second ! turns this back into zero. If, however, the answer is 128, the ! turns this into zero, and the second ! turns it to 1 as we require. Now that we know how to operate on the individual bits in a byte, it is time to turn our attention to building up a model of a neuron.

Neurons

When modeling the human brain, our first step is to model a neuron. In the brain, each neuron is connected to thousands of other neurons by synapses. One can think of synapses as the wiring between the neurons that actually do the calculations. In theory, a neuron will fire a signal when certain stimuli from the various input synapses reach a certain threshold. Our computer model will try to copy this behavior.

In modeling the behavior of a neuron, one of the most import decisions we can make is deciding when a signal is sent. There are two obvious choices: One, we can have a hard limit or threshold – when the input reaches the threshold a signal is fired, otherwise it is not. Two, we can have a soft threshold, where the intensity of the signal is controlled by an appropriate “activation” function. When we use a hard threshold, we generally call the neuron model a perceptron[3], however, I will use the term perceptron for both soft and hard models.

In explaining exactly how a perceptron works, it will be beneficial to limit our attention to simple boolean functions like AND and OR. Later on, we will allow the various weights of the synapses or connections to vary according to their importance, however, here we want to hard-wire the correct behavior to elucidate how a perceptron behaves.

Let's take a close look at how an AND perceptron functions assuming a hard threshold. There are two binary inputs into the perceptron, plus there is an ever present bias unit. This bias unit is sort of like a door or a gate, and only when the added weight of all the inputs bypass the weight of the door, does the perceptron fire. The bias unit is always set to one, and the weight attached to it will set the threshold level. Here is the math:

if ∑ inputs∗weights ≥ 0 then FIRE!

With two inputs and one bias unit, we need to have three weights. These are the hard-wired weights that will make an AND perceptron work: (2 , 2, -3). Then, if both inputs are zero, only the bias unit will contribute, and will contribute 0∗2+0∗2+1∗(−3)=−3 and therefore the perceptron will not fire. If the first input is a one, but the second input is zero, then the total contribution is 1∗2+0∗2+1∗(−3)=−1 and again the perceptron will not fire. If the first input is zero and the second input is one, the situation is almost exactly the same. Finally if both inputs are one we have 1∗2+1∗2+1∗(−3)=1 and finally the perceptron will fire, matching the exact performance of an AND gate. If you have never done so, I recommend trying to figure out the weights for an OR perceptron yourself.

For single perceptrons, a hard threshold is an acceptable activation function, however, when using the backpropagation algorithm, which we will discuss later, an activation function with a soft threshold is necessary (continuous derivatives are a necessity). There are a few standard models of soft activation functions, but the most common choice is the sigmoid function also called the logistic function. I have actually had considerable dealings with this function in the past[4], as it crops up in more than just this one situation, and I encourage the interested reader to take a look, but I will not burden the reader with that here and now. The sigmoid function is defined as

Also, the analytic derivative of the sigmoid function has the extremely simple form: which, as I just mentioned, will come into play when we discuss backpropagation.

Turning our attention to my implementation of an AND perceptron necessitates an explanation of what I was originally planning to do with my ANN. At this point in the project, I was hoping to be able to train my network to do floating point arithmetic, and therefore I needed to train my network to accept input much larger than just two bits plus a bias unit. At the time I had hoped that the final architecture of my network was going to be flexible, but what happened in reality is that at every step I made decisions that had implications for the life of the project.

At the time, I decided that each neuron would accept a byte for each of two inputs and would also accept a final control byte. The control byte would operate as my bias unit, but could be used in the future for some extra flexibility (I have not been able to take advantage of this.). I also decided to have only one byte for the output (I now regret this, and would have much preferred having two bytes of output). This means that my perceptron would accept 24 bits of information and would output 8 bits. This makes for a total of 192 weights internally in each neuron or perceptron.

In order to calculate the output of the neuron we must sum over all the inputs multiplied by their weights. Two Bytes of input plus one byte for the control bit, gives us 24 bits to multiply together. We then run this through the sigmoid function that I detailed earlier. I created a simple function for both of these two procedures:

#define SIZE 3\*BYTE

double sigmoid(double z){

if(z>4) return 1;

if(z<-4) return 0;

double denom = 1 + exp(-z);

return 1.0/denom;

}

double dot(double \*a, double \*b){

double out = 0;

int i;

for(i = 0; i < SIZE; i++){

out += a[i]\*b[i];

}

return out;

}

After that, setting up the AND neuron was fairly straightforward. The only crucial piece of technology was that I wanted to ensure that the control bit did indeed have a leading one, otherwise it couldn't be used as a bias. I did this with a simple mask:

a = a|128

128 being the byte that has a leading 1 with the rest of the digits zero. I then initialized a matrix:

double theta[BYTE][SIZE] = {};

and initialized it in the obvious way i.e., exactly as above for single bits, but now utilizing a whole byte. Then, I combined the control bytes and the two input bytes into one array:

double input[SIZE];

int i;

for(i = 0; i < BYTE; i++){

input[i] = !!((a << i) & 128);

input[i+BYTE] = !!((b << i) & 128);

input[i+2\*BYTE] = !!((c << i) & 128);

}

Finally, I was able to run this through the sigmoid function:

char out = 0;

for (i = 0 ; i< BYTE; i++){

double temp;

temp = sigmoid(dot((double \*)input, theta[i]));

if(temp>0.5){

out = out|(1<<(BYTE-i-1));

}

}

return out ;

Obviously hard-wiring the weights is not learning, the next step was to have a neuron that could accept arbitrary weights as inputs along with the normal inputs and control bits. This is what the definition of the procedure looked like:

char neuron(char a, char b, char c, double \*\*theta);

Where the internals are basically the same as the hard-wired neuron, but rather than having the weights predetermined, we now accept a two dimensional matrix, which we call theta. Keep in mind that this two dimensional matrix has dimensions eight by twenty-four.

Now that I had my neurons ready, it was time to start chaining them together to form a net. This was also where I started to have some intense problems with my design and my limited abilities to code in C. In trying to chain together a net, two problems became apparent. I resolved these problems separately over the course of weeks, so I will talk about these problems separately, but I should remind you that I had both problems at the same time and fundamentally, these problems are intertwined.

The Network

Before getting on to my implementation of the network, we should probably be asking ourselves why we need a network in the first place. We have just seen that a single perceptron is quite capable of some impressive tasks all by itself, is there really a good reason to add the extra complications of an entire network? And the answer is a definite YES! While a single perceptron can distinguish between linearly separable data, you can convince yourself that it cannot distinguish data that is linearly inseparable, for example XOR. XOR is an important ingredient when computing arithmetic, and we therefore will need to have more than one perceptron chained together. In fact, it has been proven[5], that to do simple addition requires a network with at least one hidden layer (a hidden layer is a layer of perceptrons in between the input layer of perceptrons and the output layer of perceptrons) and to do multiplication requires at least two hidden layers. Thus a network is important both because it makes for a good model of the brain and because it actually allows us to calculate things that are otherwise incalculable.

Moving on to the implementation, I started off with the following code:

typedef struct net{

int number;

int length;

int \*layers;

double \*theta;

} net;

net \*make\_net(int number, int length, int \*layers){

double \*theta = malloc(SIZE\*BYTE\*sizeof(double));

net \*out = malloc(sizeof(net));

out->number = number

out->length = length;

out->layers = layers;

out->theta = theta;

return out;

}

void free\_net(net \*x){

//I never had the chance to implement this

}

With a typical invocation looking like

int layers[3] = {3,2,1};

net \*my\_net = make\_net(6, 3, layers);

free\_net(my\_net);

I added in some redundancy to the program. At the time I was bouncing between two representations. One used the length of the network, in this example, the length is three. The other representation was interested in the number of neurons in the net, in this example there are 6 (actually 3, because the first layer is only concerned with how many inputs there are, but I dealt with this later). Obviously, these are dependent on each other, and with the help of the array “layers”, you can figure one quantity from the other. Truthfully length (or size) is redundant, but it just seemed easier to pass this extra information. In a future version, I will suppress one in the favor of the other, but for now, the extra information is welcome.

The two problems I was having should become apparent while looking at my implementation.

Topology

The first problem is seen by looking at how I invoked the layers. My topology is a 3X2X1 network. In theory this should be fine, but if you actually think about what my neurons are capable of, i.e., each neuron accepts 2 inputs and outputs one output, that this is topology is not acceptable at the present moment. It needs either fuller specification of exactly how the wiring between the nodes will be accomplished, or the individual neurons themselves will have to be made more flexible. It took me several weeks of playing around before I finally came upon the solution that I would stick with. Because of the shortness of time to complete the project, I decided I needed to keep the problem as simple as possible, while still keeping all the main ingredients. What this meant was that I had to limit what my net was going to be able to accomplish. The way I decided to limit my net, was by making the architecture or topology quite restricted. What I decided was that the only acceptable topology was to be of the form 2X2X2X...X2X1.

This topology is obviously severely limiting, and I hope that in a future version of this program, I will be able to relax this requirement. However, even with this topology, we should still be able to do some fun experiments with the net, however floating point calculations (which is what I originally intended to do) are not going to be possible, because each floating point number requires, at a minimum, about 2 bytes of information, and in order to add or multiply, you would therefore need to be able to input about 4 bytes into the net. My net can only accommodate 2 bytes of information, and was thus unsuitable for floating point arithmetic.

Three Dimensional Matrices

However, at the time the final topology was not even the most pressing problem. I don't know if you can see it, but there is a bug in my make\_net procedure. The problem is in this line:

double \*theta = malloc(SIZE\*BYTE\*sizeof(double));

The problem is that each neuron accepts a two dimensional theta matrix in order to calculate its output. Therefore, in a net with, say, 10 neurons, you will need a three dimensional matrix with dimensions 10X32X8. The programming language C does not have an efficient way to make three dimensional matrices, and we must program them by hand. Now, there is a good way to do this, a bad way to do this, and the way I did it. Let me detail each way.

First is the good way to do it. What one should really should do is lay all the values in a single array and then use a small algorithm or ordering principle to gain access to each element in the array. A typical implementation of this might look like:

int index(int x, int y, int z) {

return x + (y\*xSize) + (z\*ySize\*xSize);

}

double value = array[index(a, b, c)];

Not only is this the smart way to do this, but one can also imagine that performing operations on this would be fast and efficient. And, equally valuable would be the fact that all the memory would be allocated in one big chunk of memory.

The bad way to do this would be to actually implement a three dimensional matrix in C, in much the same way that you would do a two dimensional matrix. This would necessitate three separate calls to malloc, and therefore your data would be spread out all over the RAM, and you would expect slower performance. Also because the implementation is more complicated, it would be much easier to introduce subtle bugs into your program.

Finally is the way that I implemented it, which is purely an ugly hack that barely gets the job done. First, what I did was make a new structure to wrap the individual theta matrices:

typedef struct thet{

double \*\*t;

} thet;

This also meant that I had to change the definition of my net ever so slightly:

typedef struct net{

int length;

int size;

int \*layers;

thet \*theta;

} net;

Then I made a procedure to initialize a new two dimensional theta:

double \*\*make\_theta(){

double \*\*out = malloc(BYTE\*sizeof(double \*));

int i,j;

for(i = 0; i < BYTE; i++){

out[i] = malloc(SIZE\*sizeof(double));

for(j = 0; j < SIZE; j++){

out[i][j] = rand()/((double)RAND\_MAX) -0.5;

}

}

return out;

}

Let me call attention to the fact that each weight in the theta matrix is a random number between negative one-half and positive one-half. This turns out to be important in practice, otherwise after training, the weights can often move in tandem rather than move independently.

Finally, I made a new make\_net initializer, using my extremely awkward theta[i].t construction.

net \*make\_net(int number, int length, int \*layers){

int i;

thet \*theta = malloc((number-layers[0])\*sizeof(thet \*));

for( i = 0 ; i< (number-layers[0]); i++){

theta[i].t = make\_theta();

}

net \*out = malloc(sizeof(net));

out->number = number;

out->layers = layers;

out->theta = theta;

out->length = length;

return out;

}

One thing that I would like to point out is that the reason we are subtracting layers[0] from the loop is because 2X2X1 architecture is actually composed of two inputs, two hidden neurons, and one neuron and therefore only needs three neurons.

Obviously, this was an ugly hack. On the other hand, it does get the job done and therefore I was satisfied with it. Now with the topology of the network laid out and the representation of the weights of the network established, it was time to turn to the central question of learning in an ANN.

Learning

Now we are almost ready to discuss the central algorithm that makes an artificial neural network actually function. To understand how the algorithm works, it is best to go back to a single perceptron and analyze in detail how learning works in that simple case.

Let us examine how we would change the weights for an OR perceptron with a soft sigmoid threshold. Let us assume that in the beginning all of the theta components are set to zero, , i.e., omega = (0,0,0). We will also set a learning rate of three on errors to help make the example really clear. Reiterating, we only make changes to the weights when the error is equal or more than one-half. (Keep in mind that this is a toy example, on the real algorithm, I change the weights no matter how small the error). First, we put in the combination (0,0) expecting zero to come out. However with all the weights set to zero, i.e., omega = (0,0,0), the sigmoid function returns one-half. The only possible weight that can be changed is the weight leading from the bias unit, and we must lower the connection, because the learning rate is set at three, the new omega will now be (-3,0,0). If we were to run the activation again, now the sigmoid function would receive a value of negative one and therefore would output a value closer to zero as we expected. Now imagine we give it the combination (1,0) expecting a one to come out. Unfortunately the sigmoid function would give the same result as previously, with an answer close to zero. Thus we would change the weights in the positive direction, again using the learning rate of three would yield omega = (0,3,0), and now it would output a value above one half, much as we require. Then we give the perceptron the combination (0,1), it again fires a value at one-half, re-shuffling the weights we would expect to get something like (3,3,3). Finally, we check the last combination (1,1) and it outputs close to one, so we accept this and don't change the weights. Finally we come back around to the beginning, with omega equal to (3,3,3) we get an error, and therefore change omega to (0, 3, 3). All the other combinations pass. Doing a further round, we would get (0, 6, 3). One more round would give (0, 6, 6) and finally one last round would give (-3, 6, 6). Which is the stable result from this algorithm. If this was difficult to understand, I recommend trying it out with pencil and paper and working out an example by hand.

The previous toy example is a good demonstration of how we would train a single perceptron. There are two things that I would like to point out. First, we saw that every time we had an error, we changed the weights by a factor of three. This factor is obviously set by the programmer and is called the learning rate. A small learning rate means that the algorithm will take a long time to get to suitable weights, however it will get there eventually. A large learning rate means that sometimes you are not even able to converge upon a solution, however, if you are able to converge upon a solution, then it will happen rapidly. I have found that adjusting the learning rate can alter the algorithm dramatically.

Another thing is that we did what is called batch learning. What we could have done, and which is what I tried to do in the first place, was to get one example, say (0,0) and learn it really, really well before moving on to the next example. If you were to use this type of learning instead you will quickly find that the neuron becomes extremely forgetful and will usually forget what it learned previously. This is called “catastrophic forgetting” and is a constant source of frustration for machine learning algorithms[6]. If however, you use the method above, and feed it all examples at the same time, and change the weights only slightly between different examples, the results will be better, and the perceptron should in theory be able to learn all linear functions.

Now that we know how to train a single neuron, you might think that it would be easy to train a whole network. But if you spend anytime thinking about it at all, you quickly realize, that while it is easy to train the neurons that are at the output level, where you actually know what the error is, it is far from obvious what the error is going to be like on the hidden layers. Basically, the question is how do we assign blame to the hidden layers? This question is so hard that it took several decades before a conclusion was discovered.

Backpropagation

Finally, in the mid 1970s, Paul J. Werbos hit upon the idea of using directed partial derivatives to assign blame[7]. While the derivation of the mathematics is outside the scope of this paper, the result is fairly simple, especially if we first think of a good toy example. The toy example that I think illuminates the algorithm the best is to think of a long chain of neurons. So instead of having a network topology like 2X2X...2X1, let us for a moment only consider topologies of the form 1X1X1X...X1. With a long chain like this, it becomes fairly transparent what we must do. For the output layers we assign blame much as we would expect:

where t is the target and a is the output. The slope of the function was already calculated above when we discussed the sigmoid function and is simply equal to a\*(1-a). The addition of the slope means that weights that are undecided, i.e. have there weights close to zero, will have the biggest errors. And weights that are either really positive or really negative will give activations that are either close to zero or close to one and will therefore not have much error attached to it. This means that weights that are undecided change most rapidly with this scheme, and that weights that are already either positive or negative will be difficult to move in either direction.

Moving to the next layer in our chain, let us call this the layer i, we assign blame by simply asking how much was the error at the previous layer, let us call it layer j, and then multiply that by the weight of the connection:

where, again, the slope is the derivative of the sigmoid function. We then continue this process until we reach the end of the chain.

After we know exactly who contributed what error, it is straightforward to change the weights according to the usual procedure:

where is the learning rate.

Moving from a chain to a whole network is actually not that difficult, instead of only taking the error from the one neuron in the previous layer, we must include all the errors from the previous layer, so that now the error at each n perceptron in the layer i will be:

Changing the weights is done the same way as before.

Moving on to the implementation of this algorithm I want to point out a difference between the math and the way I programmed the algorithm in practice. In my implementation what I did was that as soon as I discovered an error, I would then go ahead and then change the weights. This seemed appropriate from a programming perspective, because it means that I did not need to have two loops. (one to discover the errors, and another to correct the weights). However, I think mathematically this is a little fishy, but in practice does not seem to cause much harm.

Before showing off the code that implemented this backpropagation algorithm, I want to briefly mention that my design decision to limit the topology to simple chains of the form 2X2X...X2X1 really paid off in this algorithm because I only had three cases to consider. The absolute last output layer, the layer that came right before the output layer, and all the other layers. This made my life much simpler. Without further ado:

void train(char a, char b, char c, char out, net \*x, int epochs){

double RATE = 0.5;

double storage[x->size - 2][BYTE]; //activation storage, for later calculation

double input\_storage[x->size -2][SIZE]; //input storage, from layer to layer

a = a|128; //usual bit masking

double \*input = make\_input(a,b,c);

//here I call an outside function, that turns the three characters

//into a vector of 24 numbers

//first we feed forward

int i,n;

for(i = 0 ; i < x->length - 2;i++){

char b\_new = activate\_neuron(a,b,c,x->theta[2\*i].t);

c = activate\_neuron(a,b,c,x->theta[2\*i+1].t);

b = b\_new;

for(n=0; n<BYTE; n++){

storage[2\*i][n] = sigmoid(dot(input,x->theta[2\*i].t[n]));;

storage[2\*i+1][n] = sigmoid(dot(input, x->theta[2\*i+1].t[n]));

}

input = make\_input(a,b,c);

for(n =0 ; n<SIZE; n++){

input\_storage[i][n] = input[n];

}

}

//our final output layer

for(n=0; n<BYTE; n++){

storage[x->size-3][n] = sigmoid(dot(input, x->theta[x->size -3].t[n]));

}

//Now we go backwards

for(n=0; n< epochs; n++){

//we set up two temporary error arrays

double error1[BYTE];

double error2[BYTE];

for(i=0; i<BYTE; i++){

//first the output layer

error1[i] = (!!((out << i) & 128) - storage[x->size-3][i])\*(storage[x->size-3][i])\*(1-storage[x->size-3][i]);

error2[i] = 0;

int j\_here;

for(j\_here = 0 ; j\_here <SIZE ; j\_here++){

//now we change the weights

x->theta[x->size -3].t[i][j\_here] += RATE\*error1[i]\*input\_storage[x->length - 3][j\_here];

}

}

int j;

for(j = x->length - 2; j>0; j--){

if(x->layers[j+1] == 1){

//for the layer right before the output layer

int m;

for(m = 0; m<BYTE; m++){

double \*activation\_top = storage[2\*j];

double \*activation\_bottom = storage[2\*j + 1];

int j\_p;

for (j\_p = 0;j\_p< BYTE; j\_p ++){

error1[m] = activation\_top[m]\*(1 - activation\_top[m])\*error1[j\_p]\*(x->theta[x->size -3].t[m][j\_p+8]);

error2[m] = activation\_bottom[m]\*(1 - activation\_bottom[m])\*error1[j\_p]\*(x->theta[x->size -3].t[m][j\_p+16]);

}

for(j\_p = 0 ; j\_p <SIZE ; j\_p++){

x->theta[x->size - 5].t[m][j\_p] += RATE\*error1[m]\*input\_storage[x->length - 5][j\_p];

x->theta[x->size - 4].t[m][j\_p] += RATE\*error2[m]\*input\_storage[x->length - 4][j\_p];

}

}

} else if (x->layers[j+1] == 2){

//for all other layers

int m;

for(m = 0; m<BYTE; m++){

double \*activation\_top = storage[2\*j];

double \*activation\_bottom = storage[2\*j + 1];

int j\_p;

for (j\_p = 0;j\_p< BYTE; j\_p ++){

double sum1 = error1[j\_p]\*(x->theta[2\*(j)].t[m][j\_p+8]) + error2[j\_p]\*(x->theta[2\*(j)].t[m][j\_p+8]);

double sum2 = error1[j\_p]\*(x->theta[2\*(j)+1].t[m][j\_p+16]) + error2[j\_p]\*(x->theta[2\*(j)+1].t[m][j\_p+16]);

error1[m] = activation\_top[m]\*(1 - activation\_top[m])\*sum1;

error2[m] = activation\_bottom[m]\*(1 - activation\_bottom[m])\*sum2;

}

for(j\_p = 0 ; j\_p <SIZE ; j\_p++){

x->theta[2\*(j-1)].t[m][j\_p] += RATE\*error1[m]\*input\_storage[2\*j][j\_p];

x->theta[2\*(j-1) + 1].t[m][j\_p] += RATE\*error2[m]\*input\_storage[2\*j + 1][j\_p];

}

}

}

}

}

}

Couple of things to note in this algorithm. One is that I hard-wired the learning rate to be 0.5. In a future version of this program I will definitely want to be able to adjust this parameter, so I will have to take this out of the function, and use it as a parameter. Another thing to note is that one of the parameters that one can adjust is the number of epochs. As I was saying earlier, generally we will want to have a small learning rate and then repeat the learning over several examples. The number of repeated learning is usually called the number of epochs in the artificial neural network community. I do have to say, however, that my implementation is completely messed up right here, because we are not reiterating over many examples, but are instead reiterating over only one single example, which completely defeats the purpose of a small learning rate reiterated over many epochs.

Batch Learning

With the backpropagation algorithm written, I immediately trained it on the XOR problem. As I mentioned in the previous paragraph, I never actually ended up using the epochs parameter because that was causing my algorithm to really only learn one example at a time. Then, when I trained a new example, the network forgot what I had trained previously. This is a common problem in machine learning and has a few remedies. The most common remedy is to train many examples at the same time. With that in mind, here is what the code for my main function looks like:

int main(){

srand(time(NULL)); //random initialization

int layers[3] = {2,2,1};

net \*XOR = make\_net(5,3,layers);

int i;

//batch training

for (i = 0; i<100; i++){

train(0,0,0,0^0,XOR,1);

train(0,0,1,0^1,XOR,1);

train(0,1,0,1^0,XOR,1);

train(0,1,1,1^1,XOR,1);

}

print\_byte(activate\_net(0,0,0,XOR));

print\_byte(activate\_net(0,0,1,XOR));

print\_byte(activate\_net(0,1,0,XOR));

print\_byte(activate\_net(0,1,1,XOR));

return 0;

}

Printing to the screen exactly what we hoped.

Conclusion

Putting together an Artificial Neural Network from scratch in C is a daunting task. However, once it is completed the network is capable of so much. While I have not had time to experiment with it yet, I am excited about all of the new things I can try next, from giving it a short term memory, to allowing a more flexible topology; from training it on some real-world examples, to finally fixing up that three-dimensional matrix problem. Most importantly it is exciting to see my computer learn, and the way an ANN learns is in many ways exactly the same as the way anyone learns: practice, practice, practice.

References

[1] Voytek, Bradley. “Are There Really as Many Neurons in the Human Brain as Stars in the Milky Way?” Nature Publishing Group, May 20, 2013.

[2] Hassoun, Mohamad. “What is a neural network and how does its operation differ from that of a digital computer? (In other words, is the brain like a computer?)” Scientific American, May 14, 2007.

[3] Russel, Stuart & Norvig, Peter. “Artificial Intelligence: A Modern Approach, Third Edition.” Pearson Education, Inc., 2011.

[4] Plotz, David. “The Logistic Equation & Euler's Method.” August 14, 2014. <http://nbviewer.ipython.org/github/Chuphay/chuphay.github.io/blob/master/blog/blog_8_14_14.ipynb>

[5] Franco, Leonardo & Cannas, Sergio. “Solving Arithmetic Problems Using Feed-Forward Neural Networks.” Neurocomputing 18 (1998) 61 – 79.

[6] Robins, Anthony. “Consolidation in Neural Networks and in the Sleeping Brain.” University of Otago, 1996.

[7] Werbos, Paul J. “The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting.” John Wiley & Sons, Inc. 1994.