

A Simple Neural Network Example
A. Mattox Beckman, Jr.
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616 USA

1 Introduction

Artificial Neural Networks are an exciting tool for machine learning. I decided that I wanted to understand them: meaning that I can both implement it and explain it. The implementation result was a simple neural network written in Python that learns the boolean operations of and and or for two inputs.

My goal for this amb is to explain very briefly what an Artificial Neural Network (ANN) is, and show how to write one. Because there are many good explanations of the formulas involved, I've been somewhat brief with the mathematics. The contribution of this paper is to show a working version of the program and how it implements these formulas. An interested reader should be able to follow this and get a working ANN in about an hour, provided the reader knows Python.

2 Neural Networks

Artificial Neural Networks are a simplified model of biological neural networks. Our brains are made up of an amazing number of nerve cells. These cells are connected to each other by a long filament called an *axon*. Each cell originates an axon, which in turn terminates at one or more other nerve cells. When a nerve cell activates, it sends an electro-chemical pulse down the axon, into the target cells. The target cells in turn collect these impulses. Once the impulse level crosses a certain threshold, the target cells also fire, affecting still more cells.

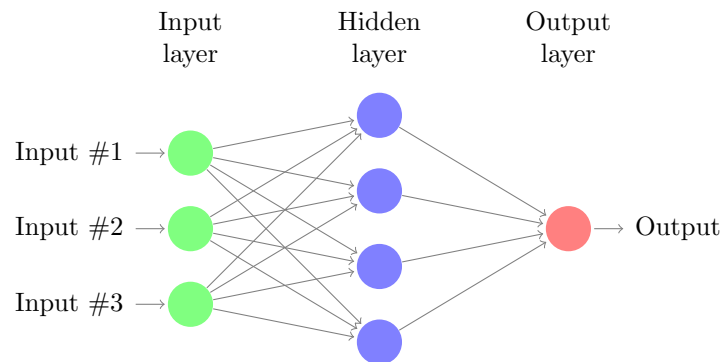


Figure 1: A typical neural network

2.1 Geometry of the Network

An ANN has a similar, but greatly simplified structure. Perhaps the most common arrangement is the three-layer *feed-forward* network, an example is illustrated in figure 1. The first layer is called the *input layer*. This layer collects inputs to the network and relays them to the nodes of the second layer, which is known as the *hidden layer*.

Typically, each node of the input layer has connections to each of the nodes of the hidden layer, forming a complete bipartite graph. These connections have *weights* that attenuate or amplify the signal from the input node. Each node also has an “extra” input called a *bias* that does not originate from any other node. The result is that each node in the hidden layer receives as its input the weighted sum of the inputs plus the bias.

Each node contains an *activation function* that takes the node’s input and determines the node’s output. A very common activation function is the *sigmoid*, i.e. $f(x) = \frac{1}{1+e^{-x}}$. It is often called the *squashing function*, because it compresses the input into the range $(0, 1)$. See figure 2 for a graph of this function.

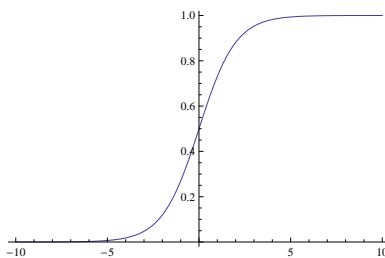


Figure 2: The Sigmoid Function

Similarly, the hidden layer connects to the third layer, the *output layer*. These outputs form the output of the entire network.

2.2 Notation

To describe the use of the network adequately we need some notation, which I will borrow from [1].

Let the input from node i to node j be denoted x_{ji} . Let the weight of that connection be denoted w_{ji} . Let the bias of a node i be denoted b_i . The total input $x_i = b_i + \sum x_{ji}w_{ji}$, where j ranges over all nodes that have input connections to i .

The activation function for node i will be denoted f_i , and the output of node i is simply $o_i = f_i(x_i)$.

Let there be three sets of nodes, one for each layer. The input layer will be denoted \vec{i} , the hidden layer \vec{h} , the output layer \vec{o} .

To train the network, we will need a set of input vectors \vec{x} along with their expected outputs \vec{t} . The error between output node i and the associated output t will be called δ_i .

2.3 Backpropagation

To train the network, you first put the inputs into the input nodes, and feed the inputs forward until you get the network outputs.

At this point, you need to calculate the error terms.

For the output layer, there are two choices. If node i 's activation function is the selection function, then $\delta_i = t_i - o_i$. At least, that's what I used. If the activation function is the sigmoid, then use $\delta_i = o_i(1 - o_i)(t_i - o_i)$. Most textbooks will tell you that $o_i(1 - o_i)$ is the derivative of the sigmoid of o_i . What they usually don't tell you is that it's the derivative of the *differential equation* for which the sigmoid is the solution. It is *not* the normal calculus derivative.

These error terms will be used to adjust the weights of the connections coming into the output nodes.

The hidden layer uses a more complex formula, since the error must propagate backwards from multiple connections. The formula is $\delta_i = o_i(1 - o_i)\sum_{j \in \text{output}} w_{ji}\delta_j$. These terms will be used to adjust the weighted connections from the input layer to the hidden layer.

The adjustment formula is $w_{ji} = w_{ji} + \eta\delta_j x_{ji}$, where x_{ji} is the input to j from i . (In the program, it will simply be denoted `output[i]`, since for any i , x_{ji} is the same for all j .) We use η as a learning factor: it dampens the changes to prevent instabilities.

3 The Program

Here is a Python program that implements a simple three-layer feed-forward network. I wanted something simple but not trivial, so I decided to train the computer to learn the “and” and “or” operations on two bits. There are six nodes in the network, two each of input, hidden, and output.

The first part imports the math library, defines the training set, and the selection functions. The training set is defined as a hash table: the key is a tuple of the input, the value is a tuple containing the boolean conjunction and disjunction of the inputs.

```
import math

# This is the and/or function
training = { (1,1):(1,1), (1,0):(0,1), (0,1):(0,1), (0,0):(0,0) }

def sum(w,i):
    s = 0.0
    for (a,b) in zip(w,i):
        s = s + a * b
    return s

def squash(x):
    return 1.0 / (1.0 + math.exp(-x))

def select(x):
    if x>0:
        return 1
    else:
        return 0

def repeat(x):
    return x
```

With these out of the way, I defined a `main` function to do all the work, so that I could load this in an interactive Python shell if I needed to. First, I initialize the weight variable `w` and bias variable `bias`. Next, I set an array that delimits the layers, and provide a small utility to make selecting a layer for a loop easy. The idea was to be general so I could add multiple hidden layers in the future.

```
def main():
    w = {}
    bias = {}

    layers = [0,2,4,6]
    def lrange(x):
        return range(layers[x],layers[x+1])

    print "Input Layer: ",lrange(0)
    print "Hidden Layer: ",lrange(1)
    print "Output Layer: ",lrange(2)
```

At this point, it's time to start configuring the network. I set η , the learning rate, the number of epochs (training sessions), the activation functions for each layer, and the biases. I initialize the weights of each connection to 0.5. Usually we use a small random number here, but I wanted to be able to reproduce my results easily.

```
eta = 0.1
epochs = 20

# Set activation functions and biases
f = {}
for i in lrange(0):
    f[i] = repeat

for i in lrange(1):
    f[i] = squash
    bias[i] = 0.5

for i in lrange(2):
    f[i] = select
    bias[i] = 0.5

# Set weights
for l1 in [1,2]:
    for j in lrange(l1):
        w[j] = {}
        for i in lrange(l1-1):
            w[j][i] = 0.5
```

At this point, we are ready to start training. I start my loop, and have it print out some status information.

```

for r in range(0,epochs):
    print "Epoch ",r
    print "Weights:"
    for j in w.keys():
        for i in w[j].keys():
            print "w",j,",",i," = ",w[j][i],"\t"
    print "\n"
    print "Biases: "
    keys = bias.keys()
    keys.sort
    for k in keys:
        print "b",k,' ',bias[k],"\t"
    print "\n"

    print "Eta: ",eta

```

Now I start an inner loop. During each epoch, we loop over all the training inputs and adjust the network. The output of the input layer is simply the input.

```

# Run training sets
for x in training.keys():
    print '----- '
    print 'Training set ',x

    # Correct answers
    t = training[x]

    # Outputs of input layer
    output = {}
    input = {}
    err = {}
    delta = {}
    for i in xrange(0):
        output[i] = x[i]

```

For the remaining layers, I take the weighted sum of the inputs, plus the bias. So input $x_i = b_i + \sum x_{ji}w_{ji}$, and output is $f_i(x_i)$.

```

for l1 in [0,1]:
    for j in xrange(l1+1):
        input[j] = bias[j]
        for i in xrange(l1):
            input[j] = input[j] + w[j][i] * output[i]
        print "input ",j," = ",input[j]
        output[j] = f[j](input[j])
        print "output ",j," = ",output[j]

```

Now we calculate the error of the output layer by comparing it to the expected output.

```

for i in xrange(2):
    print "i is ", i
    err[i] = t[i-4] - output[i]
    print "Err ",i,": ",err[i]

```

Then we calculate the error of the hidden layer. In this program δ_i is represented by `err[i]`.

```

for i in xrange(1):
    err[i] = 0
    for j in xrange(2):
        err[i] = err[i] + w[j][i] * err[j]
    err[i] = err[i] * output[i] * (1-output[i])
    print "Err ",i,": ",err[i]

```

Finally, we are ready to update the weights and biases. I use a temporary variable `adjust` to calculate the adjustment factor. In some books this is called Δw_{ji} . We also adjust the biases according to the error.

```

for l in [1,2]:
    for j in xrange(1):
        for i in xrange(1-1):
            adjust = eta * err[j] * output[i]
            print "Adjust ",j,"," ,i,": ",adjust
            w[j][i] += adjust

for i in range(2,6):
    bias[i] += err[i] * eta

```

That's it. If you run `main`, you will get convergence in 17 epochs.

Epoch	$w_{2,0}$	$w_{2,1}$	$w_{3,0}$	$w_{3,1}$	$w_{4,2}$	$w_{4,3}$	$w_{5,2}$	$w_{5,3}$
0	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000
1	0.4916	0.4902	0.4916	0.4902	0.2922	0.2922	0.4382	0.4382
2	0.4871	0.4843	0.4871	0.4843	0.0869	0.0869	0.3771	0.3771
3	0.4874	0.4828	0.4874	0.4828	0.0955	0.0955	0.3164	0.3164
4	0.4878	0.4812	0.4878	0.4812	0.1042	0.1042	0.2559	0.2559
5	0.4883	0.4796	0.4883	0.4796	0.1130	0.1130	0.1957	0.1957
6	0.4890	0.4779	0.4890	0.4779	0.1218	0.1218	0.1356	0.1356
7	0.4910	0.4775	0.4910	0.4775	0.1307	0.1307	0.1553	0.1553
8	0.4936	0.4773	0.4936	0.4773	0.1396	0.1396	0.1751	0.1751
9	0.4966	0.4774	0.4966	0.4774	0.1486	0.1486	0.1949	0.1949
10	0.4978	0.4756	0.4978	0.4756	0.1576	0.1576	0.1353	0.1353
11	0.4993	0.4765	0.4993	0.4765	0.1668	0.1668	0.1461	0.1461
12	0.5009	0.4746	0.5009	0.4746	0.1760	0.1760	0.1461	0.1461
13	0.5026	0.4727	0.5026	0.4727	0.1852	0.1852	0.1461	0.1461
14	0.5045	0.4707	0.5045	0.4707	0.1945	0.1945	0.1461	0.1461
15	0.5065	0.4686	0.5065	0.4686	0.2038	0.2038	0.1461	0.1461
16	0.5065	0.4644	0.5065	0.4644	0.1339	0.1339	0.1461	0.1461
17	0.5065	0.4644	0.5065	0.4644	0.1339	0.1339	0.1461	0.1461

4 Colophon

The document was typeset in \LaTeX , and the neural network diagrams were produced using the PGF/TIKZ package. The template for the neural network diagram was taken from <http://www.fauskes.net/pgftikzexample/network/>, part of a large collection of `pgf/tikz` code examples. The graph of the sigmoid function was produced using Mathematica.

References

- [1] Tom M Mitchell. *Machine Learning*. McGraw-Hill, 1997.