An introduction to neural networks

Pattern learning with the back-propagation algorithm

Level: Intermediate

Andrew Blais (onlymice@gnosis.cx), Author, Gnosis Software, Inc.
David Mertz (mertz@gnosis.cx), Developer, Gnosis Software, Inc.

01 Jul 2001

Neural nets may be the future of computing. A good way to understand them is with a puzzle that neural nets can be used to solve. Suppose that you are given 500 characters of code that you know to be C, C++, Java, or Python. Now, construct a program that identifies the code's language. One solution is to construct a neural net that learns to identify these languages. This article discusses the basic features of neural nets and approaches to constructing them so you can apply them in your own coding.

More dW content related to: **neural networks introduction**

According to a simplified account, the human brain consists of about ten billion neurons -- and a neuron is, on average, connected to several thousand other neurons. By way of these connections, neurons both send and receive varying quantities of energy. One very important feature of neurons is that they don't react immediately to the reception of energy. Instead, they sum their received energies, and they send their own quantities of energy to other neurons only when this sum has reached a certain critical threshold. The brain learns by adjusting the number and strength of these connections. Even though this picture is a simplification of the biological facts, it is sufficiently powerful to serve as a model for the neural net.

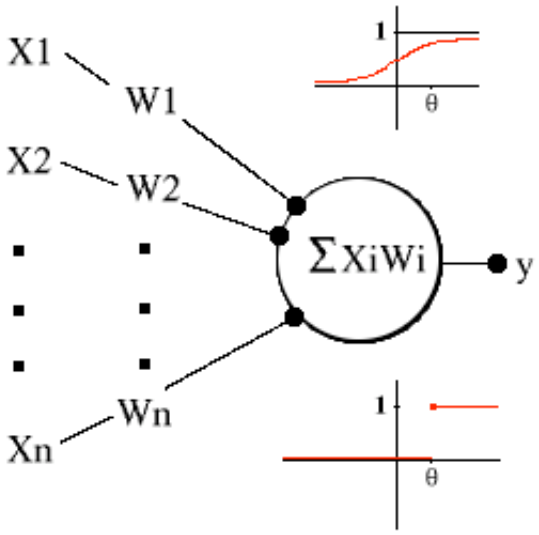
Threshold logic units (TLUs)

The first step toward understanding neural nets is to abstract from the biological neuron, and to focus on its character as a *threshold logic unit (TLU)*. A TLU is an object that inputs an array of weighted quantities, sums them, and if this sum meets or surpasses some threshold, outputs a quantity. Let's label these features. First, there are the inputs and their weights: X_1,X_2, ..., X_n and W_1, W_2, ...,W_n. Then, there are the X_i*W_i that are summed, which yields the activation level a, in other words:

a = (X1 * W1)+(X2 * W2)+...+(Xi * Wi)+...+ (Xn * Wn)

The threshold is called theta. Lastly, there is the output: y. When a >= theta, y=1, else y=0. Notice that the output doesn't need to be discontinuous, since it could also be determined by a squashing function, s (or sigma), whose argument is a, and whose value is between 0 and 1. Then, y=s(a).

Figure 1. Threshold logic unit, with sigma function (top) and cutoff function (bottom)



A TLU can classify. Imagine a TLU that has two inputs, whose weights equal 1, and whose theta equals 1.5. When this TLU inputs <0,0>, <0,1>, <1,0>, and <1,1>, it outputs 0, 0, 0, and 1 respectively. This TLU classifies these inputs into two groups: the 1 group and the 0 group. Insofar as a human brain that knows about logical conjunction (Boolean AND) would similarly classify logically conjoined sentences, this TLU knows something like logical conjunction.

This TLU has a geometric interpretation that clarifies what is happening. Its four possible inputs correspond to four points on a Cartesian graph. From $X_1*W_1+ X_2*W_2$ = theta, in other words, the point at which the TLU switches its classificatory behavior, it follows that $X_2 = -X_1 + 1.5$. The graph of this equation cuts the four possible inputs into two spaces that correspond to the TLU's classifications. This is an instance of a more general principle about TLUs. In the case of a TLU with an arbitrary number of inputs, N, the set of possible inputs corresponds to a set of points in N-dimensional space. If these points can be cut by a hyperplane -- in other words, an N-dimensional geometric figure corresponding to the line in the above example -- then there is a set of weights and a threshold that define a TLU whose classifications match this cut.

How a TLU learns

Since TLUs can classify, they know stuff. Neural nets are also supposed to learn. Their learning mechanism is modeled on the brain's adjustments of its neural connections. A TLU learns by changing its weights and threshold. Actually, the weight-threshold distinction is somewhat arbitrary from a mathematical point of view. Recall that the critical point at which a TLU outputs 1 instead of 0 is when the SUM(Xi * Wi) >= theta. This is equivalent to saying that the critical point is when the SUM(X_i * W_i)+ (-1 * theta) >= 0. So, it is possible to treat -1 as a constant input whose weight, theta, is adjusted in learning, or, to use the technical term, *training*. In this case, y=1 when SUM(X_i * W_i)+ (-1 * theta) >= 0, else y=0.

During training, a neural net inputs:

1. A series of examples of the items to be classified
2. Their proper classifications or targets

Such input can be viewed as a vector: <X_1, X_2, ...,X_n, theta, t>, where t is the target or true classification. The neural net uses these to modify its weights, and it aims to match its classifications with the targets in the training set. More precisely, this is supervised training, as opposed to unsupervised training. The former is based on examples accompanied by targets, whereas the latter is based on statistical analysis (see Resources later in this

article). Weight modification follows a learning rule. One idealized learning algorithm goes something like this:

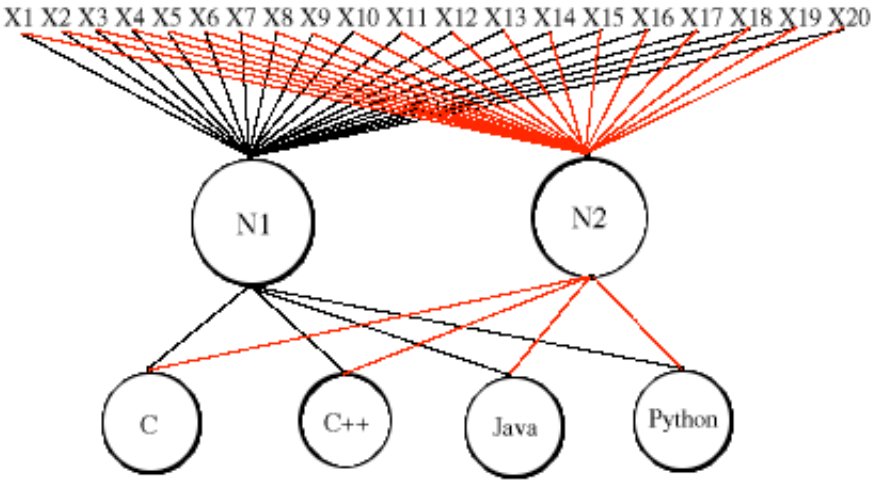
Listing 1. Idealized learning algorithm

```
fully_trained = FALSE
DO UNTIL (fully_trained):
    fully_trained = TRUE
    FOR EACH training_vector = <X1, X2, ..., Xn, theta, target>::
                        # Weights compared to theta
        a = (X1 * W1)+(X2 * W2)+...+(Xn * Wn) - theta
        y = sigma(a)
        IF y != target:
            fully_trained = FALSE
        FOR EACH Wi:
        MODIFY_WEIGHT(Wi)               # According to the training rule
    IF (fully_trained):
        BREAK
```

You're probably wondering, "What training rule?" There are many, but one plausible rule is based on the idea that weight and threshold modification should be determined by a fraction of (t - y). This is accomplished by introducing alpha (0 < alpha < 1), which is called the *learning rate*. The change in W_i equals (alpha * (t - y) * Xi). When alpha is close to 0, the neural net will engage in more conservative weight modifications, and when it is close to 1, it will make more radical weight modifications. A neural net that uses this rule is known as a *perceptron*, and this rule is called the *perceptron learning rule*. One result about perceptrons, due to Rosenblatt, 1962 (see Resources), is that if a set of points in N-space is cut by a hyperplane, then the application of the perceptron training algorithm will eventually result in a weight distribution that defines a TLU whose hyperplane makes the wanted cut. Of course, to recall Keynes, eventually, we're all dead, but more than computational time is at stake, since we may want our neural net to make more than one cut in the space of possible inputs.

Our initial puzzle illustrates this. Suppose that you were given N characters of code that you knew to be either C, C++, Java, or Python. The puzzle is to construct a program that identifies the code's language. A TLU that could do this would need to make more than one cut in the space of possible inputs. It would need to cut this space into four regions, one for each language. This would be accomplished by training a neural net to make two cuts. One cut would divide the C/C++ from the Java/Python, and the other cut would divide the C/Java from the C++/Python. A net that could make these cuts could also identify the language of a source code sample. However, this requires our net to have a different structure. Before we describe this difference, let's briefly turn to practical considerations.

Figure 2. Preliminary (and insufficient) Perceptron Learning Model



Considerations of computational time rule out taking N-character chunks of code, counting the frequency of the visible ASCII characters, 32 through 127, and training our neural net on the basis of these counts and target information about the code's language. Our way around this was to limit our character counts to the twenty most frequently occurring non-alphanumeric characters in a pool of C, C++, Java, and Python code. For reasons that concern the implementation of floating point arithmetic, we decided to train our net with these twenty counts divided by a normalizing factor. Clearly, one structural difference is that our net has twenty input nodes, but this should not be surprising, since our description has already suggested this possibility. A more interesting difference is a pair of intermediary nodes, N1 and N2, and an increase in the number of output nodes from two to four, O1-O4.

N1 would be trained so that upon seeing C or C++, it would set y1=1, and upon seeing Java or Python, it would set y1=0. N2 would be similarly trained. Upon seeing C or Java, N2 would set y2=1, and upon seeing C++ or Python, it would set y2=0. Furthermore, N1 and N2 would output 1 or 0 to the Oi. Now, if N1 sees C or C++, and N2 sees C or Java, then the code in question is C. Moreover, if N1 sees C or C++, and N2 sees C++ or Python, the code is C++. The pattern is obvious. So, suppose that the Oi were trained to output 1 or 0 on the basis of the following table:

Intermediate nodes mapped to output (as Boolean functions)

N1	N2	O1 (C)	O2 (C++)	O3 (Java)	O4 (Python)
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

If this worked, our net could identify the language of a code sample. It is a pretty idea, and its practical ramifications seem to be incredibly far reaching. However, this solution presupposes that the C/C++ and the Java/Python inputs are cut by one hyperplane, and that the C/Java and the C++/Python inputs are cut by the other. There is an approach to net training that bypasses this kind of assumption about the input space.

The delta rule

Another training rule is the delta rule. The perceptron training rule is based on the idea that weight modification is best determined by some fraction of the difference between target and output. The *delta rule* is based on the idea of gradient descent. This difficult mathematical concept has a prosaic illustration. From some given point, a Southward path may be steeper than an Eastward path. Walking East may take you off a cliff, while walking South may only take you along its gently sloping edge. West would take you up a steep hill, and North leads to level ground. All you want is a leisurely walk, so you seek ground where the overall steepness of your options is minimized. Similarly, in weight modification, a neural net can seek a weight distribution that minimizes error.

Limiting ourselves to nets with no hidden nodes, but possibly having more than one output node, let p be an element in a training set, and t(p,n) be the corresponding target of output node n. However, let y(p,n) be determined by the squashing function, s, mentioned above, where a(p,n) is n's activation relative to p, y(p,n) = s(a(p,n)) or the squashed activation of node n relative to p. Setting the weights (each Wi) for a net also sets the difference between t(p,n) and y(p,n) for every p and n, and this means setting the net's overall error for every p. Therefore, for any set of weights, there is an average error. However, the delta rule rests on refinements in the notions of average and error. Instead of discussing the minutiae, we shall just say that the error relative to some p and n is: ½ * square(t(p,n) - y(p,n)) (see Resources). Now, for each Wi, there is an average error defined as:

Listing 2: Finding the average error

```
sum = 0
FOR p = 1 TO M:          # M is number of training vectors
    FOR n = 1 TO N:      # N is number of output nodes
        sum = sum + (1/2 * (t(p,n)-y(p,n))^2)
average = 1/M * sum
```

The delta rule is defined in terms of this definition of error. Because error is explained in terms of all the training vectors, the delta rule is an algorithm for taking a particular set of weights and a particular vector, and yielding weight changes that would take the neural net on the path to minimal error. We won't discuss the calculus underpinning this algorithm. Suffice it to say that the change to any Wi is:

alpha * s'(a(p,n)) * (t(p,n) - y(p,n)) * X(p,i,n).

X(p,i,n) is the i^{th} element in p that is input into node n, and alpha is the already noted learning rate. Finally, s'(a(p,n)) is the rate of change (derivative) of the squashing function at the activation of the n^{th} node relative to p. This is the delta rule, and Widrow and Stearns (see Resources) showed that when alpha is sufficiently small, the weight vector approaches a vector that minimizes error. A delta rule-based algorithm for weight modification looks like this.

Downward slope (follow until error is suitably small)

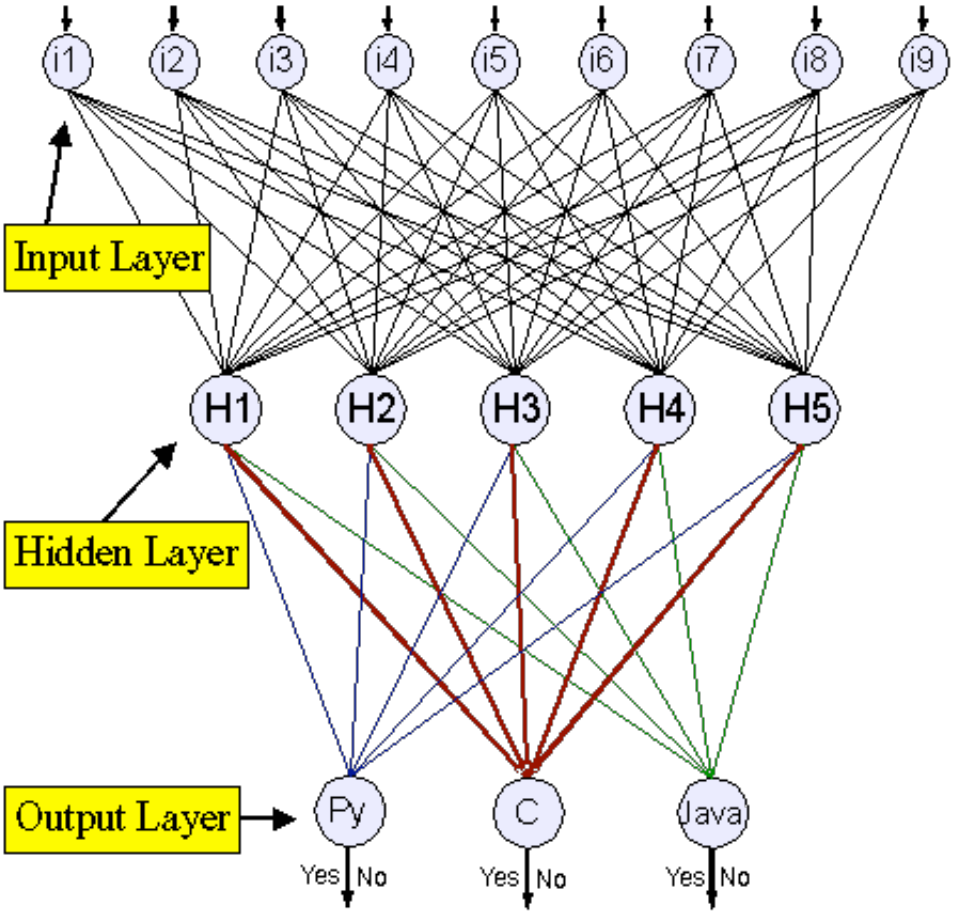
```
step 1: for each training vector, p, find a(p)
step 2: for each i, change Wi by:
            alpha * s'(a(p,n)) * (t(p,n)-y(p,n)) * X(p,i,n)
```

There are important differences from the perceptron algorithm. Clearly, there are quite different analyses underlying the weight change formulae. The delta rule algorithm always makes a change in weights, and it is based on activation as opposed to output. Lastly, it isn't clear how this applies to nets with hidden nodes.

Back-propagation

Back-propagation is an algorithm that extends the analysis that underpins the delta rule to neural nets with hidden nodes. To see the problem, imagine that Bob tells Alice a story, and then Alice tells Ted. Ted checks the facts, and finds that the story is erroneous. Now, Ted needs to find out how much of the error is due to Bob and how much to Alice. When output nodes take their inputs from hidden nodes, and the net finds that it is in error, its weight adjustments require an algorithm that will pick out how much the various nodes contributed to its overall error. The net needs to ask, "Who led me astray? By how much? And, how do I fix this?" What's a net to do?

Figure 3. "Code Recognizer" back-propagation neural network



The back-propagation algorithm also rests on the idea of gradient descent, and so the only change in the analysis of weight modification concerns the difference between t(p,n) and y(p,n). Generally, the change to W_i is:

alpha * s'(a(p,n)) * d(n) * X(p,i,n)

where d(n) for a hidden node n, turns on (1) how much n influences any given output node; and (2) how much that output node itself influences the overall error of the net. On the one hand, the more n influences an output node, the more n affects the net's overall error. On the other hand, if the output node influences the overall error less, then n's influence correspondingly diminishes. Where d(j) is output node j's contribution to the net's overall error, and W(n,j) is the influence that n has on j, d(j) * W(n,j) is the combination of these two influences.

However, n almost always influences more than one output node, and it may influence every output node. So, d(n) is:

SUM(d(j)*W(n,j)), for all j

where j is an output node that takes input from n. Putting this together gives us a training rule. First part: the weight change between hidden and output nodes, n and j, is:

alpha * s'(a(p,n))*(t(p,n) - y(p,n)) * X(p,n,j)

Second part: the weight change between input and output nodes, i and n, is:

alpha * s'(a(p,n)) * sum(d(j) * W(n,j)) * X(p,i,n)

where j varies over all the output nodes that receive input from n. Moreover, the basic outline of a back-propagation algorithm runs like this.

Initialize Wi to small random values.

Steps to follow until error is suitably small

Step 1: Input training vector.
Step 2: Hidden nodes calculate their outputs.
Step 3: Output nodes calculate their outputs on the basis of Step 2.
Step 4: Calculate the differences between the results of Step 3 and targets.
Step 5: Apply the first part of the training rule using the results of Step 4.
Step 6: For each hidden node, n, calculate d(n).
Step 7: Apply the second part of the training rule using the results of Step 6.

Steps 1 through 3 are often called the *forward pass*, and steps 4 through 7 are often called the *backward pass*. Hence, the name: back-propagation.

Recognizing success

With the back-propagation algorithm in hand, we can turn to our puzzle of identifying the language of source code samples. To do this, our solution extends Neil Schemenauer's Python module *bpnn* (see Resources). Using his module is amazingly simple. We customized the class `NN` in our class `NN2`, but our changes only modify the presentation and output of the process, nothing algorithmic. The basic code looks like:

Listing 3: Setting up a neural network with bpnn.py

```
# Create the network (number of input, hidden, and training nodes)
net = NN2(INPUTS, HIDDEN, OUTPUTS)

# create the training and testing data
trainpat = []
testpat = []
for n in xrange(TRAINSIZE+TESTSIZE):
                        #... add vectors to each set

# train it with some patterns
net.train(trainpat, iterations=ITERATIONS, N=LEARNRATE, M=MOMENTUM)
```

```
# test it
net.test(testpat)

# report trained weights
net.weights()
```

Of course, we need input data. Our utility code2data.py provides this. Its interface is straightforward: just put a bunch of source files with various extensions into a subdirectory called ./code , and then run the utility listing these extensions as options, for example:

python code2data.py py c java

What you get is a bunch of vectors on STDOUT , which you can pipe to another process or redirect to a file. This output looks something like:

Listing 4: Code2Data output vectors

```
0.15 0.01 0.01 0.04 0.07 0.00 0.00 0.03 0.01 0.00 0.00 0.00 0.05 0.00 > 1 0 0
0.14 0.00 0.00 0.05 0.13 0.00 0.00 0.00 0.02 0.00 0.00 0.00 0.13 0.00 > 1 0 0
[...]
```

Recall that the input values are normalized numbers of occurrences of various special characters. The target values (after the greater than sign) are YES/NO representing the type of source code file containing these characters. But there is nothing very obvious about what is what. That's the great thing, the data could by *anything* that you can generate input and targets for.

The next step is to run the actual code_recognizer.py program. This takes (on STDIN) a collection of vectors like those above. The program has a wrapper that deduces how many input nodes (count and target) are needed, based on the actual input file. Choosing the number of hidden nodes is trickier. For source code recognition, 6 to 8 hidden nodes seem to work very well. You can override all the parameters on the command line, if you want to experiment with the net to discover how it does with various options -- each run might take a while, though. It is worth noting that code_recognizer.py sends its (large) test result file to STDOUT , but puts some friendly messages on STDERR . So most of the time, you will want to direct STDOUT to a file for safe keeping, and watch STDERR for progress and result summaries.

Listing 5: Running code_recognizer.py

```
> code2data.py py c java | code_recognizer.py > test_results.txt
Total bytes of py-source: 457729
Total bytes of c-source: 245197
Total bytes of java-source: 709858
Input set: ) ( _ . = ; " , ' * / { } : - 0 + 1 [ ]
HIDDEN = 8
LEARNRATE = 0.5
ITERATIONS = 1000
TRAINSIZE = 500
OUTPUTS = 3
```

```
MOMENTUM = 0.1
ERROR_CUTOFF = 0.01
TESTSIZE = 500
INPUTS = 20
error -> 95.519... 23.696... 19.727... 14.012... 11.058... 9.652...
8.858... 8.236... 7.637... 7.065... 6.398... 5.413... 4.508...
3.860... 3.523... 3.258... 3.026... 2.818... 2.631... 2.463...
2.313... 2.180... 2.065... 1.965... 1.877... 1.798... 1.725...
[...]
0.113... 0.110... 0.108... 0.106... 0.104... 0.102... 0.100...
0.098... 0.096... 0.094... 0.093... 0.091... 0.089... 0.088...
0.086... 0.085... 0.084...
Success rate against test data: 92.60%
```

The decreasing error is a nice reminder, and acts as a sort of progress meter during long runs. But, the final result is what is impressive. The net does a quite respectable job, in our opinion, of recognizing code -- we would love to hear how it does on your data vectors.

Summary

We have explained the basics of the neural net in a way that should allow you to begin to apply them in your own coding. We encourage you to take what you have learned here and attempt to write your own solution to our puzzle. (See Resources for our solution.)

Resources

- Our Code Recognizer program is based on Neil Schemenauer's back propagation module.
- For the distinction between supervised and unsupervised training, and neural nets in general, see Machine Learning, Neural and Statistical Classification, edited by D. Michie, D.J. Spiegelhalter, and C.C. Taylor. Specifically, see Chapter 6.
- For Rosenblatt's result about perceptrons, see his *Principles of Neurodynamics*, 1962, New York: Spartan Books.
- For some of the minutiae of the delta rule, see Kevin Gurney's excellent *An Introduction to Neural Networks*, 1997, London: Routledge. Also see Neural Nets for an early on-line version.
- For the proof about the delta rule, see: B. Widrow and S.D. Stearns, *Adaptive Signal Processing*, 1985, New Jersey: Prentice-Hall.
- For an implementation of the perceptron with a GUI, see The Perceptron by Omri Weisman and Ziv Pollack.
- What is a subject without its FAQ? See the ever useful Neural Net FAQ.
- For a wide-ranging collection of links, see The Backpropagator's Review.

- University courses are a rich source of information for the beginner in any subject. Consult this list of courses.
- The Neural Network Package is a LGPL package written in Java. It permits experimentation with all the algorithms discussed here.
- See Neural Networks at your Fingertips for a set of C packages that illustrate Adaline networks, back-propagation, the Hopfield model, and others. A particularly interesting item is The Back-propagation Network, a C package that illustrates a net that analyzes sunspot data.
- At Neural Networking Software, you will find neural net code with graphical interfaces, and it's both DOS and Linux friendly.
- NEURObjects provides C++ libraries for neural network development, and this has the advantage of being object oriented.
- Stuttgart Neural Network Simulator (SNNS) is what the name says. It is written in C with a GUI. It has an extensive manual, and is also Linux friendly.
- Browse more Linux resources on *developerWorks*.
- Browse more Open source resources on *developerWorks*.

About the authors

Andrew Blais divides his time between home schooling his son, writing for Gnosis, and teaching philosophy and religion at Anna Maria College. He can be reached at onlymice@gnosis.cx.

David Mertz thinks that if there are any uninteresting Natural numbers, there must be a least such uninteresting number. You can reach David at mertz@gnosis.cx; you can investigate all aspects of his life at his personal Web page.
