On Determining the Optimal Network Structure in Neural Network Pattern Recognition

By Charl Linssen
<charl@turingbirds.com>
Radboud University Nijmegen

5th of August, 2007

In current day, the value of artificial neural networks (ANNs) to perform pattern recognition tasks is well-established. However, despite an abundancy of literature describing the basic structure and functioning of ANNs, few hints are given as to the details, including choice of parameters and network structure. The difficulty is partly because these choices depend on the pattern recognition task at hand. In this paper, we empirically explore the optimum number of hidden units in a 3-layer backpropagation network. This optimum is determined by analyzing the generalization capability of the network after being trained to small error on a training set.

§1 Application of ANNs to pattern recognition tasks

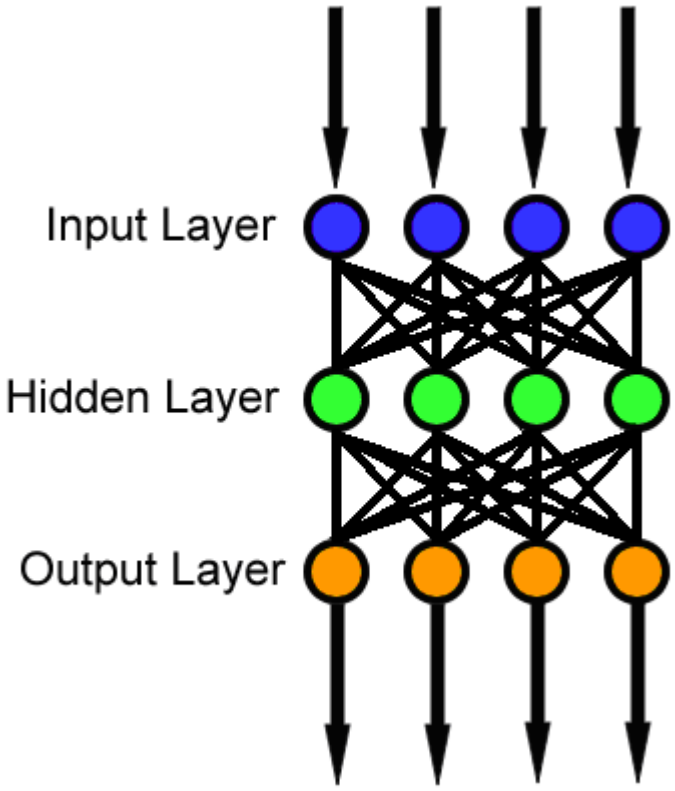
ANNs have been a subject of research since the early part of the twentieth century. It wasn't until the 1990s however that some major theoretical breakthroughs were made, establishing ANNs as a viable and practical method for problems.
There are many application areas for ANNs, including data filtering, clustering, robotic motion control, time series prediction, but probably most notably: pattern classification.
Although inspired by the complex brains that exist in nature, ANNs are straightforward. Consisting of a number of *units*, also called *neurons*, each of these units can only "see" other units that connect to it. A single neuron is a very simple processing unit, performing some mathematical functions on its inputs. In itself, a neuron usually has a single output, but may connect to multiple other units, or even back to itself. In this way, there are many variations in architecture.
Each connection has a *weight* associated with it. It is in these values that the network stores information. The network "learns" by adjusting these weights.

There are a few different learning paradigms. The most commonly used method is "learning by example", which is called *supervised*

learning. Before putting the ANN to use on actual data, there exists a *training* phase, in which samples of the data are presented to the ANN. When the network output vector deviates from its desired value, the connections are adjusted according to some learning rule. This process is repeated for as many input patterns as can be reasonably obtained.

§2 ANNs for pattern recognition

In this paper, I shall take my pick of all the available different methods in all areas of ANN design, and settle on a type that is often used for pattern classification. More concretely, this is a supervised training approach, used on a network that is organized in layers. Each layer consists of a number of neurons, each of which is connected to each neuron in the adjacent layer. This is graphically depicted as follows:

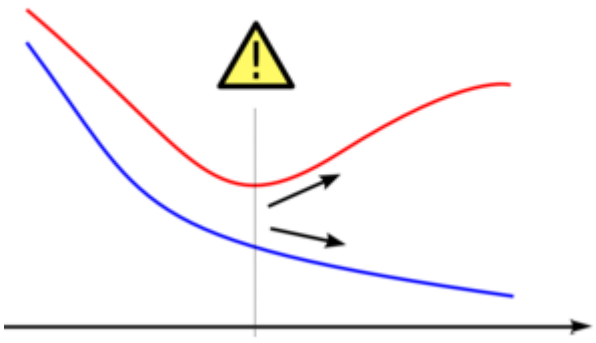


Indeed, I use three layers. There are still a large number of parameters that have yet to be chosen: many variables relating to optimizing the learning process, but also something as rudimentary as the number of units to use in each layer.

§3 Architectural considerations

Despite having chosen the "fully connected", layer-based architecture, we have not yet decided how many units go in each layer.
The input layer is straightforward: we need as many inputs as required by our data. For example, trying to classify patterns that are 10x10 pixels would require 100 input units.
The number of output units is also a simple matter: we use as many output units as there are target classes. If we are aiming to classify the uppercase letters A-Z, we would use 26 output units. Again, note that these choices are just that – choices. We could use a different approach, but this is the most straightfoward and least prone to errors.
Finally, we have to decide how many units to place in the hidden layer. This is a rather more difficult task. After all, there is nothing by which you can easily derive the number of units best fit for the task at hand. This question is the focus of this paper. Is there a certain optimum – that is, a single value at which performance is best? What does this value depend upon?
Ideally, we would establish a relationship between any or all of the known input parameters (for instance, number of different classes, degree of noise) and the number of hidden units that gives the best result.

What are the dangers of having the wrong number of units?
Having too few units will cause the network to be unable to learn the training data well, resulting in a large training error. The test set will have the same or even greater error. Having too few units is a problem that will usually be caught in an early stage, because of training difficulties.



Having too many units will result in a more subtle error: that of overfitting. Overfitting is a classical problem with ANNs. The problem is that the network ends up accomodating the training data so closely that it fails to account for small variations. This results in a very low error on the training set, but a relatively large error on the test set, depicted in the diagram below. The red line is the error on the test set, and the blue line is the error on the training set.

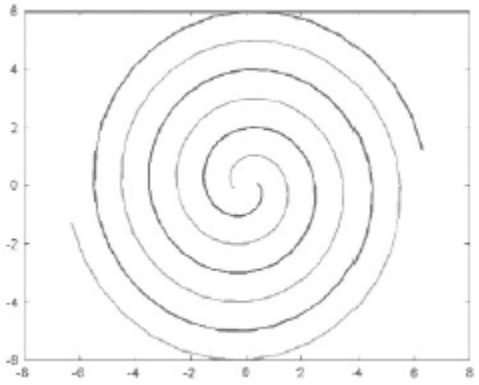
Although overfitting is usually attributed to "training for too long", i.e. training for too low an error, it can also be caused by having too many hidden units.
Beside overfitting, another downside of having too many hidden units is computational complexity. Because the network is fully connected – that is, each neuron connects to all others in adjacent layers – the amount of connections per hidden neuron is high. Although technically speaking, the operations are quick (addition and multiplication), for large networks and computationally limited platforms, this may be a hurdle.

§4 Earlier work

Several "rules of thumb" have been given by many academics in the past. Some examples are:

- "A rule of thumb is for the size of this [hidden] layer to be somewhere between the input layer size ... and the output layer size ..."
- "To calculate the number of hidden nodes we use a general rule of: (Number of inputs + outputs) * (2/3)"
- "You will never require more than twice the number of hidden units as you have inputs" (in an MLP with one hidden layer)
- "Typically, we specify as many hidden nodes as dimensions needed to capture 70-90% of the variance of the input data set."

However, these "rules" not only fail to capture the complexity of the problem, they also fail to provide an accurate outcome. We can easily teach a small network a sine function, if the activation function is a (domain-limited) sine. However, if we use a tanh or sigmoid activation function, we will need upwards of 20 hidden units.
The classic two-spirals problem, where an ANN is required to distinguish two interlocking spirals as depicted to the right, can be solved adequately by at least 50 tanh units. However, how do we define "adequate"? Perhaps a very low error is required, increasing this number above 100. Also notice that we can solve the problem using a single hidden unit if we design the network right!



In conclusion, we can say that "rules of thumb" as shown above are not nearly sufficient for practical use.

§5 Test-lab program

In order to investigate the effects of hidden layer size variations, an empirical approach was chosen. A program was set up to automatically generate ANNs of varying size, then train and test them. The number of patterns classified incorrectly by these ANNs was then counted and used as a performance measure.
Some experimentation was done in determining the learning rate, but this value was not changed for different network sizes.
In other aspects, the design was intended to be as "standard" as possible, using the common and well-established sigmoid activation function.
The program was written in ANSI C and compiled with gcc on a Linux platform. On a 733MHz Pentium III, even the large networks (with up to 600 hidden units) were simulated completely within approximately 20 minutes.
In the program, attention has been dedicated to ensure that the code is "human-readable". Where iterators are used, they are named *i*, *h* and *o* for respectively input, hidden and output units:

```
for (o=0; o < nr_output; ++o)
  for (h=0; h < nr_hidden; ++h)
```

Global variable names were also held as simple as possible: *nr_hidden* is the number of hidden units in the current network, and weights were stored as a two-dimensional array, to be indexed by the origin of the connection followed by the destination of the connection:

```
weights_hidden_output[h][o]
```

signifies the weight going from hidden neuron *h* to output neuron *o*.
Finally, patterns were stored as a 3-dimensional array, to be indexed by an *x*,*y*-coordinate, followed by the index of the pattern:

```
patterns[x][y][pattern_index]
```

The total amount of code, excluding the printing of status/debug information, is 400 lines. The actual training functions take up 120 lines.

When training networks of very large size, namely those over 400 neurons, it was found that they often stopped prematurely. Although normally a *momentum term* should be applied to solve this problem, a "quick fix" was established by disallowing the network to stop training

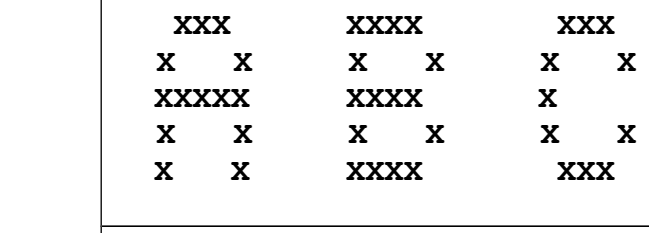
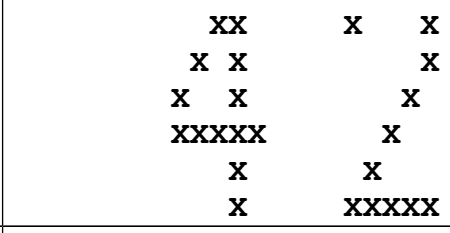
if a certain epoch count was not yet achieved. This resulted in succesful training. Of course, this is not acceptable for live networks, but it does not interfere with the test results, because the effect is the same as a momentum term.

§6 Data set

The data set was crafted by hand. It consists of two seperate sets: the characters 0-9, and the characters A-Z. Both sets were divided into training and test-sets at random. Adding noise to the original data was a simple matter: a random x and a random y coordinate were generated using the standard pseudo-random number generator of the gcc package, seeded by the system time.
After dividing them up and adding various degrees of noise, we end up with the following data sets:

- Numbers 0 through 9 (5x7 pixels, binary) [200 train, 1000 test]
  - 2 noise points (5.7%)
  - 7 noise points (20%)
- Letters A through Z (5x5 pixels, binary) [390 train, 2002 test]
  - 2 noise points (8%)
  - 5 noise points (25%)

Sets with a higher degree of noise than 25% were made, but the network was unable to achieve high enough recognition rates on these sets to extract usable data.

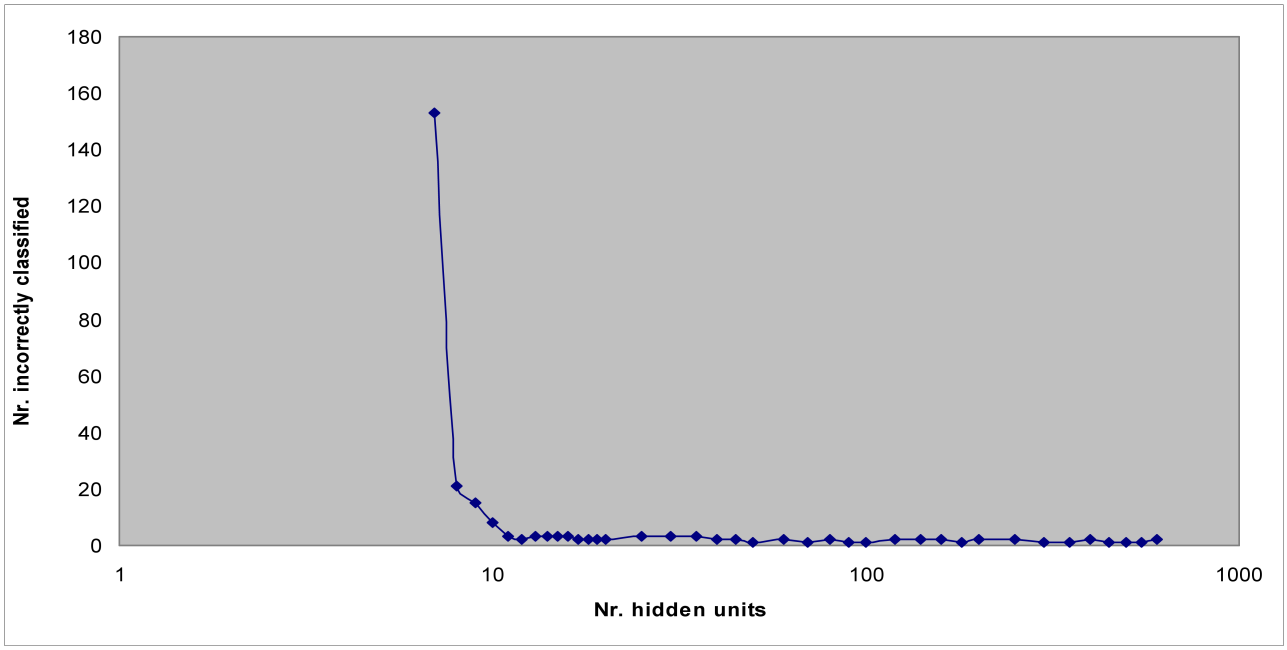
	
Three examples of the noiseless letter set.	Two examples of the noiseless numbers set.

§7 Results

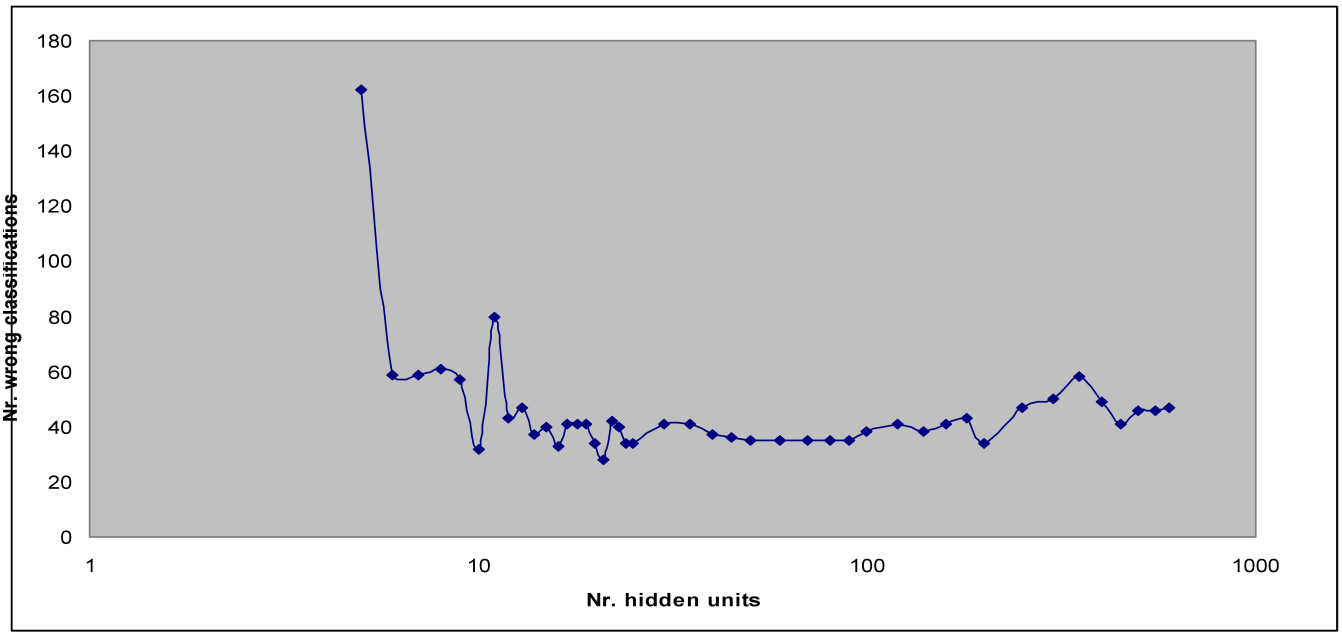
Results on test sets.

- Learning rate = 1
- Training error ≤ 0.0001

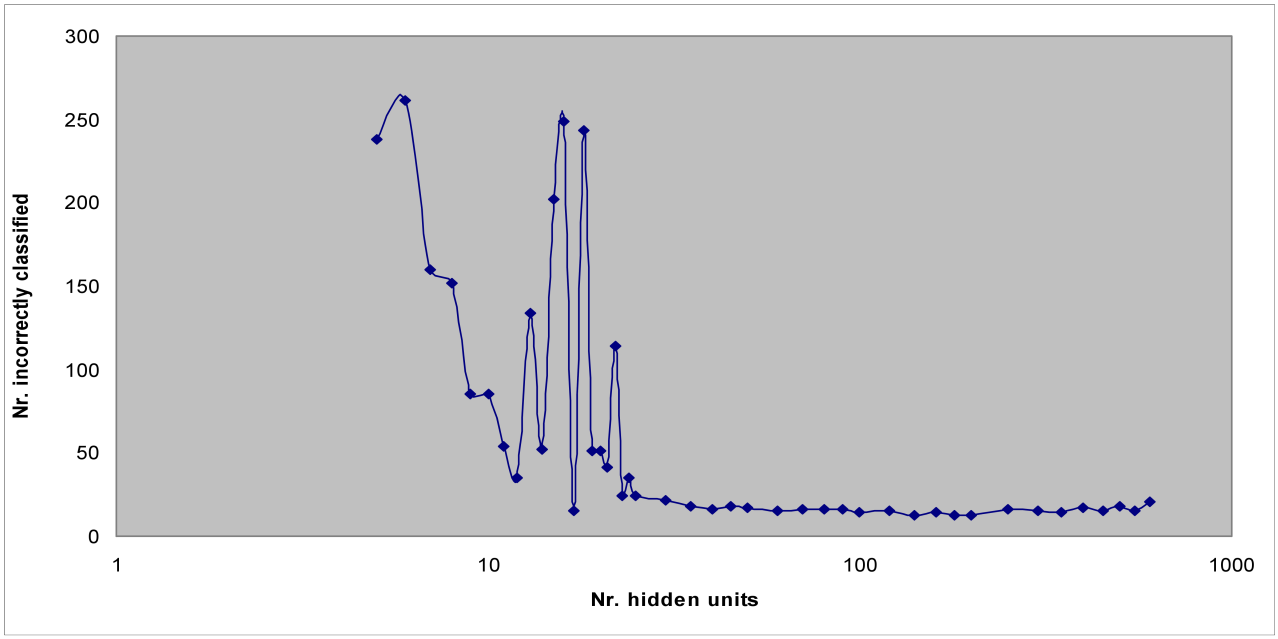
Numbers 0-9; 5.7% noise; 1000 test patterns:



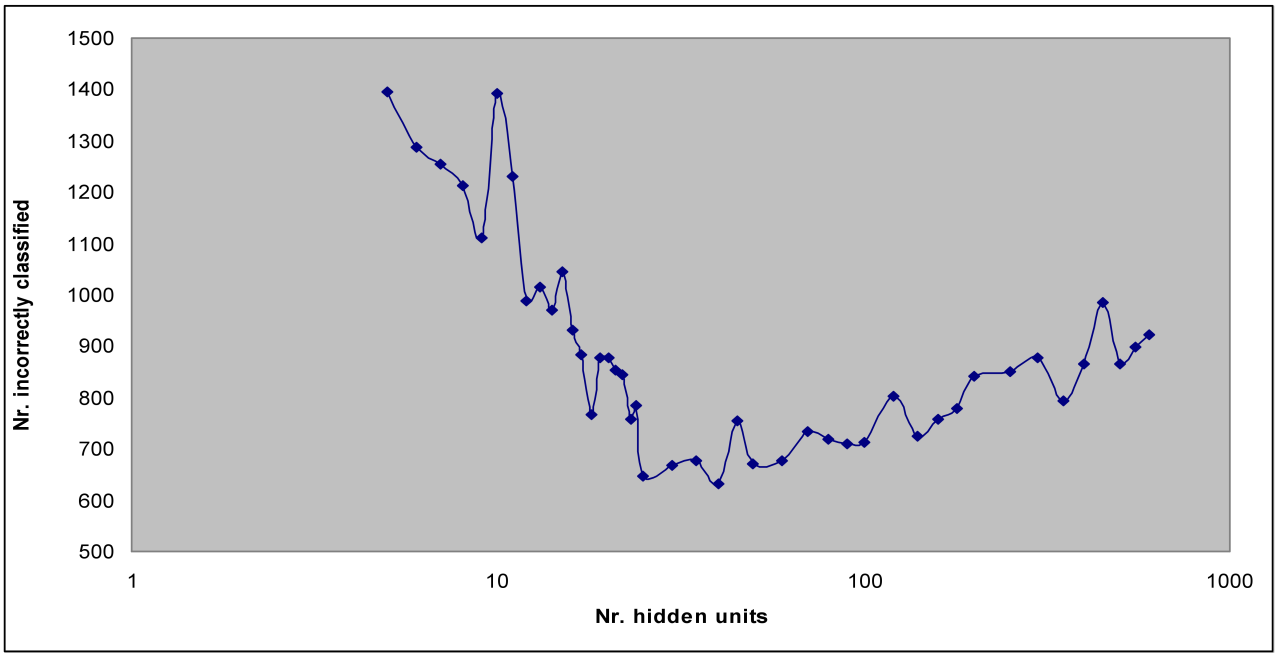
Numbers 0-9; 20% noise; 1000 test patterns:



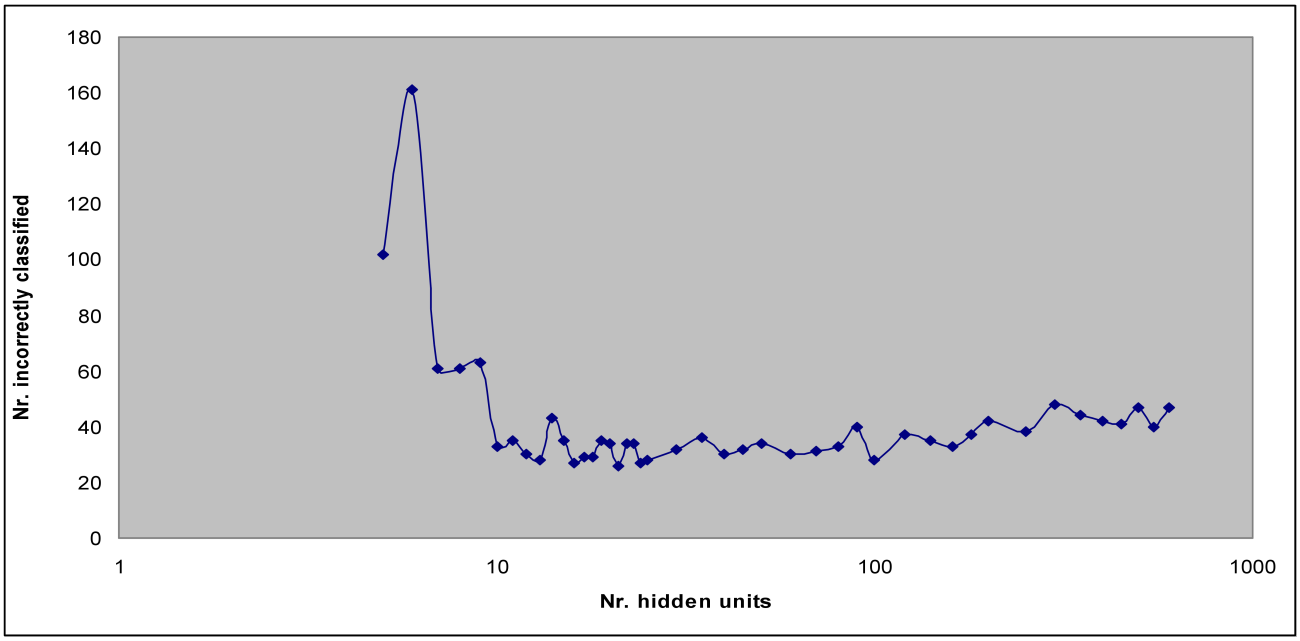
Letters A-Z; 8% noise; 2002 test patterns:



Letters A-Z; 25% noise; 2002 test patterns:



Additional result: Trained with numbers 0-9; 0% noise; 10 training patterns; tested with 20% noise, 1000 test patterns (learning rate = 0.25)



Before getting to the conclusion, let me add some remarks about the diagrams above that do not directly relate to the research subject.

The *Letters, noise=2* graph shows some oscillations between n = 10 and n = 23. These oscillations prepend the steadier area to the right, and are not believed to have any influence on that area. It is my expectation that these oscillations can be resolved by a more careful choice of parameters, mostly the learning rate, but perhaps also by careful design of a momentum term.

The diagrams have different starting points. They all start around n = 5. The reason is simply that the error for n < 5 is so high, that it is clear that the network cannot do anything useful with the data. Moreover, these values lie outside the area of interest, and so they have been cropped to improve the readability of the diagrams.

The diagram for *letters; noise=5* shows a rather high average error rate, roughly between 30% and 50%. The reason for this becomes clear when observing the generated patterns:

```
 X X
X X X
XXX
X
X   X
```

Not even I can tell this pattern to be an "A". If I had to guess I would call it a "P", and the ANN probably does the same. These kind of unrecognizable patterns are prevalent throughout the set. The reason I decided not to scrap the set and use a lower noise rate was because of the useful data that could still be recovered.

Finally, the last diagram is marked as "additional", and has been made only after the other data was already collected and analyzed. It was believed that by using these sets, data could be obtained that was of importance to the researched matter at hand.

§8 Conclusion

Before I draw my conclusion, let me restate the hypothesis: the use of too many hidden units leads to overfitting, and thus poor performance on the test set.
The goal of my research was to establish the conditions for which this was the case. Looking at the simulation results, we can now say that although the correct choice of the number of hidden units matters somewhat, it is all but critical.
It is clear to see that the flat region of both *2-noise* diagrams are just that – flat. There is no indication whatsoever of overfitting occuring.
Turning our attention to the *5-noise* diagrams, there is some evidence of overfitting. In the *numbers* diagram, we see an increase in error of 24 (from the lowest point) around n = 350. The same behaviour is displayed in the *letters* curve.
After reviewing these diagrams, it seemed that I should be able to get the same results for the *2-noise* series. Indeed, using the *numbers* set, I was able to achieve the same behaviour by training the network poorly: by only using the original, noise-less patterns. These results are displayed in the *additional* diagram. An overfitting ramp can be observed starting from n = 180.

The result of these ramps on the right side is the creation of a valley in the middle: a location where the error is at an optimal, low level. However, taking a closer look reveals that the *overall* optimum lies outside the flattest area for all curves. This poses a practical difficulty: to get the overall optimum, we can no longer rely on using a ballpark figure to try and get somewhere in the flat part.
In fact, in my simulations, I have demonstrated that the error curve can be very unpredictable; when an optimal result is required, the specific ANN should be *simulated* for a varying number of hidden units, *continuously evaluated*, and afterwards *selected for the optimum.* Because the network's performance is established during training, the user can generate test sets up ahead, that is, before deployment of the ANN "in the field" (something that should be done regardless).
I have demonstrated that it is a simple matter to set up an automated testing and evaluation system for these networks. In current day, raw computer power is by no means a limiting factor, and a testbench program as described can be set up to run overnight, yielding an accurate diagram like the ones I have generated. This can then be quickly assesed as to which number of hidden units gives the optimum results. If the actual inputs are expected to vary from the test set, it is adviced to position oneself in the flat region of the curve. If the test

set is representative, the more adventurous programmer might select a global minimum.

In conclusion, I can say that ANNs used for pattern recognition are more stable than I had thought, not requiring a very specific architecture. However, in the "real world", people are always striving for the optimum. In order to achieve this optimum, a method has been described. It is left to the ANN designer whether this extra effort is worth the increase in performance.

§9 References

- Blum, A. (1992), Neural Networks in C++
- Tan, Steinbach, Kumar, Introduction to Data Mining
- Claudio Moraga, Design of Neural Networks
- Rich Caruana, Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping
- Alex J. Champandard, AI Game Development