Bio-Inspired Adaptive Machines

# Mini-Project

*Neural Networks – Character Recognition*


Students:

Nada Amin
Louis Villedieu

Assistant:

Claudio Mattiussi


April 2004

# Table of Contents

# 1. Task of the Neural Network

The task of the network is to recognize the letters of the alphabet.

Each input unit represents a pixel. There are 5x7 binary input units, forming an image that represents a character. For example, the character `A` is represented by:

```
0 1 1 1 0
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
```

You can see that the 1's "draw" an `A`. There are 26 output units, each of which stands for a letter, thus the representation is local.

So the task of the network is to activate the output unit that corresponds to the letter drawn in the grid of input units.

# 2. Varying the Learning Rate

## *Some theory*[1]

At every cycle, during training, the synapses weights are changed according to the back-propagation of error formula, $\Delta w_{ij} = \eta \, \delta_i \, x_j$, where:

- $\Delta w_{ij}$ stands for the weight alteration applied to the considered synapse,
- $x_j$ is the output of the presynaptic neuron,
- $\delta_i = (t_i - y_i) \quad \dot{\Phi} \quad (A_i)$
  where:
  - $(t_i - y_i)$ is the error on the presynaptic neurone.
  - $\dot{\Phi} \quad (A_i)$ is the derivate of the output function at the current activation as illustrated in **Illustration 1**.

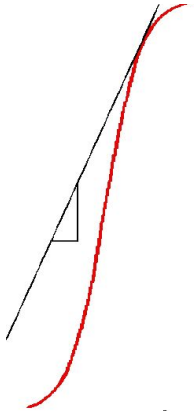


**Illustration 1** *Output function derivate*

This back-propagation of error allows the neural network to learn.

1 Inspired by the Professor Dario Floreano's courses notes "*Bio-inspired Adaptive Machines: Neural System"*

# *Observations*

We have trained the network with various learning rates. **Table 1** lists our observations.

| **Learning rate η** | **Approximate number of cycles to obtain an error < 0.1** | **Comments** |
|---|---|---|
| 8.0 | not effective | |
| 6.0 | either 250 or not effective | fluctuation |
| 5.0 | 200-300+ | fluctuation |
| 4.5 | 250-350 | fluctuation |
| 4.0 | 250-350 | fluctuation |
| 3.0 | 300 | little fluctuation |
| 2.5 | 350-450 | |
| 2.0 | 500 | |
| 1.0 | 1000 | |
| 0.5 | 2000 | |
| 0.25 | 4000 | |
| x < 2 | 1000/x | |

**Table 1** *number of cycles to train the network as a function of the learning rate*

So, at first sight, it seems that the learning rate determines the speed with which the error converges to a minimum. Thus the number of cycles should be inversely proportional to the learning rate. This is verified for small learning rates ($0 < \eta < 2$).

However, with higher learning rates, we may jump over a local minimum and even increase the error during a step.

The figures 1 and 2 show the same gradient descent with two different learning rates. The vertical axis represents the error and the horizontal axis the weight. In **Figure 1**, the learning rate is small, and the weight is slightly changed in the direction of the local minimum. In **Figure 2**, the learning rate is high, and the weight is changed so much that the local minimum is jumped over.

The principle illustrated by these figures explains why we observe fluctuations with higher learning rates.

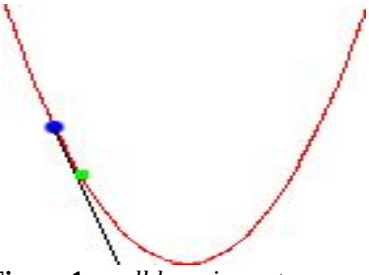


**Figure 1** *small learning rate*

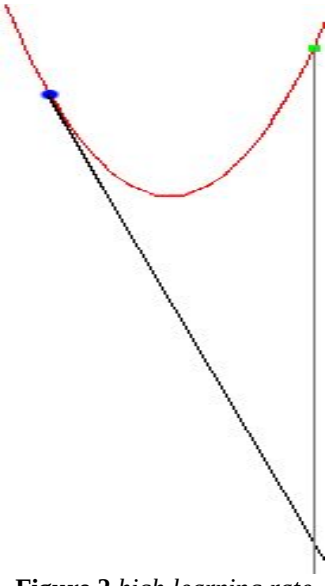


**Figure 2** *high learning rate*

# 3. Varying the Patterns

In order to recognize the figures from `0` to `9`, the input units don't need to change since we can represent the image of the digits with the same grid. However, for the 10 digits, we would need an additional 10 output units to represent them locally. As for the hidden layer, our observations of the section *"Varying the Number of Hidden Units"* are helpful here. We realize that with 10 hidden units, we can "code" for $2^{10} = 1024$ values[2], so 10 hidden units are largely sufficient in this respect. We guess that we actually need only 6 units, since $2^6 = 64 > 26+10$.

We have implemented the changes for the digits `0` and `1`. We added two output units to the network and changed the pattern set as follows:

In the header section, we changed the number of patterns and the number of output units:

```
No. of patterns : 28
No. of output units : 28
```

To each output pattern, we appended two 0's to account for the two new output units.

Finally, we added two new patterns to recognize the figure `0` and the figure `1`.

```
# Input pattern 27:
1 1 1 1 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1
# Output pattern 27:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
# Input pattern 28:
0 0 1 0 0
0 1 1 0 0
1 0 1 0 0
0 0 1 0 0
0 0 1 0 0
0 0 1 0 0
0 1 1 1 0
# Output pattern 28:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Obviously, we couldn't use the same figure for the digit `0` and the letter `O`. So we decided to add 1's in the four corners.

---

2 This formula is not accurate:
- Since the outputs of the hidden units are continuous, the minimal number of hidden units may be lower (see the remark "On the Continuity of Outputs" at the end of the section "Varying the Number of Hidden Units").
- Since the minimal binary encoding might not be linearly separable, the minimal number of hidden units may be higher.

But it does give a good indication.

If we had actually used the same figure for both the digit 0 and the letter O, then we would ask the network to give two different outputs for the same input. This implies a permanent error, because if the network learns one of the inputs, then it will get the other input wrong, and so on. As shown in **Figure 3**, we observe a noisy stabilization above 1.

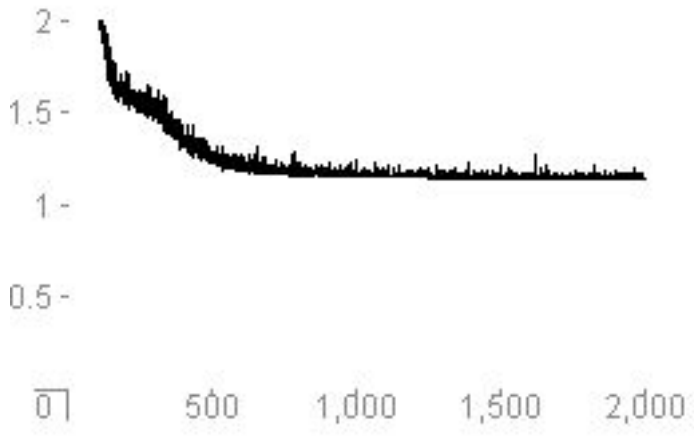



**Figure 3** *error graph when 2 patterns conflict*

If we use just slightly different figures for the digit 0 and the letter O, as we did, then it is possible for the network to distinguish between them. Perhaps, the network might need more cycles to be trained. In our case, the network was able to learn all the patterns with no change to the hidden layer in about 500 cycles with a learning rate of 2.0. We even tested a case where the digit 0 and the letter O only differ by one pixel. The network does not seem to need much more cycles to learn the patterns. Our conclusion is that it seems the network is pretty good at learning all the patterns when these can be distinguished from one another, even if only slightly.

# 4. Varying the Number of Hidden Units

We delete one by one the nodes of the hidden layer, trying to re-train the network after each deletion.

If we think about the problem before hand, if we make the hypothesis that the hidden units can be either active (=1) or inactive (=0) (step activation function, which is not the case), then, for the network to be trainable, there must be enough hidden units to "code" for all the different characters we want to recognize. In our case, we have 26 characters, so we need at least ceil($\log_2(26)$) = 5 bits, that is units in the hidden layer. However, in our case, the activation functions are not discrete, and therefore, there might be other ways to code for the 26 characters with less units in the hidden layer (see the remark *"On the Continuity of Outputs"* at the end of this section). Our results are in **Table 2**.

| Number of hidden units | Approximate number of cycles to obtain an error < 0.1 |
|---|---|
| 10 | 500 |
| 9 | 650 |
| 8 | 700-850 |
| 7 | 1250 |
| 6 | 1600 |
| 5 | 3000 |

**Table 2** *number of cycles as a function of the number of hidden units (learning rate=2.0)*

The network is not able to learn the patterns with less than 5 hidden units. For 4 hidden units, the error fluctuates around[3] $10 = 26 - 2^4 = nb\_chars - 2^{nb\_hidden\_units}$ as shown in **Figure 4**.

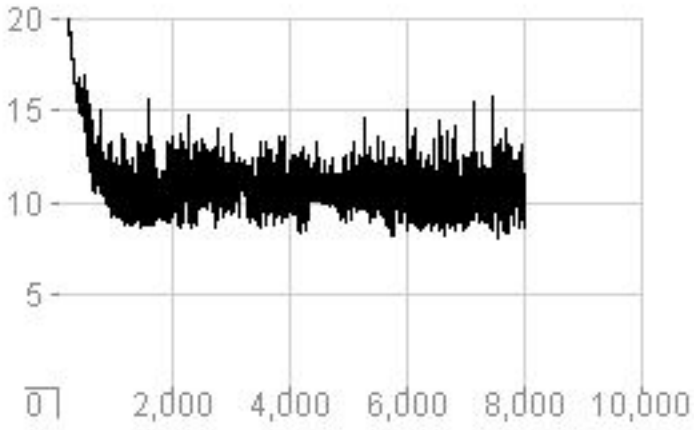



**Figure 4** *error graph for network with 4 hidden units*

So, indeed, this confirms that the network is trainable when the number of hidden units is greater than or equal to 5.

We observe that the number of cycles needed to train the network increases as the number of hidden units decreases. Simply, there are more good ways to encode the input information in the hidden layer when there are more hidden units, and therefore, it is easier to find one of these good ways. Talking about speed and efficiency, adding hidden units may not always be a good idea as it takes more memory to store the network structure and above all, more resources and time to correct the weights of the synapses.

Regarding the way the input information is coded in the hidden layer, here is what we predicted beforehand:

3 This formula may be a coincidence but it seems correct

There are two extreme ways in which the input information could be encoded in the hidden layer:

1. as in a local representation, that is, each hidden input represents one pattern, (for an absolute local encoding, the problem should be linearly separable)
2. as in a binary representation, that is, each activation combination in the hidden layer represents a pattern.

Our intuition is that as the number of hidden units increases, the encoding will tend to be closer to way 1 (local representation), and, inversely, as the number of hidden units decreases, the encoding will tend to be closer to way 2 (binary representation).

These trends were hard to verify in the character-recognition network, because of its irregularity and complexity. So we decided to analyze a smaller network to see if we could confirm these trends.

To do this analysis, we used a simple network that simply recognizes binary numbers up to 7. There are three inputs representing the bits of the number, and there are 8 outputs representing the numbers from 0 to 7 locally.

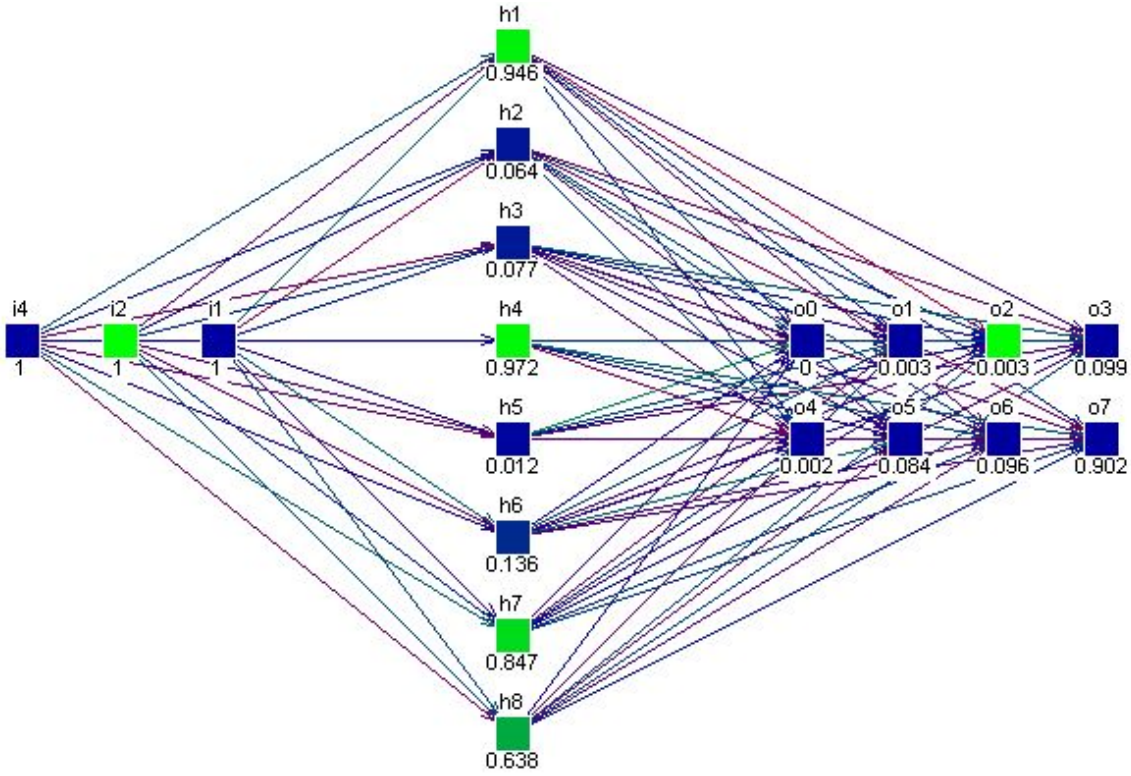



**Figure 5** *simple network that recognizes binary numbers from 0 to 7*

We trained the simple network with 8 and 3 hidden units, and looked at the encoding in the hidden units. Our observations are summarized in **Table 3**.

| | 8 hidden units | | | | | | | | | 3 hidden units | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pattern | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **number of X's** | 1 | 2 | 3 | **number of X's** |
| 0 | | X | | | X | | | | **2** | | | | **0** |
| 1 | X | | | | X | X | | X | **4** | | | | **0** |
| 2 | | X | | X | | | | X | **3** | | X | | **1** |
| 3 | | | X | X | | | | X | **3** | | X | | **1** |
| 4 | X | X | | | | | X | | **3** | | | X | **1** |
| 5 | X | | | | | X | | | **2** | X | | | **1** |
| 6 | | X | | X | | | X | | **3** | | X | X | **2** |
| 7 | X | | | X | | | X | | **3** | X | X | | **2** |

**Table 3** *comparison of encoding in trained simple networks with 8 and 3 hidden units*

The rows represent the patterns (number 0 to 7) and the columns the hidden units.
An X indicates an output of the hidden unit >0.75. For the smaller network.

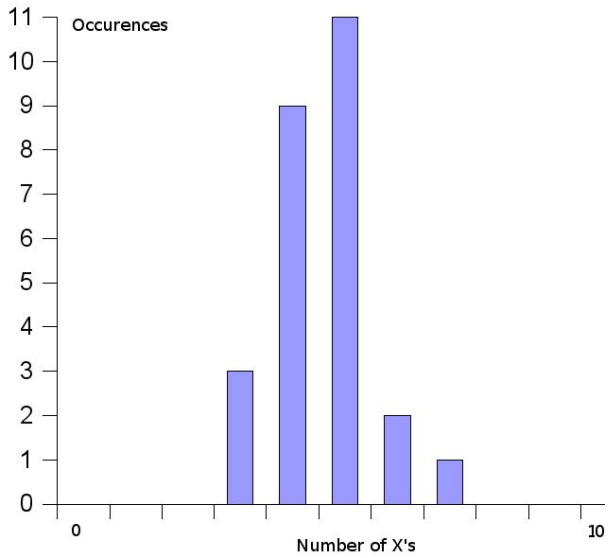
We see that our intuition is hard to verify because of the continuity of the output functions of the hidden units. For example, in the case with 3 units, if we look only at the X's, we would not be able to distinguish between pattern 0 and pattern 1. Yet the network can. So the continuity plays a role that we have neglected.

We notice something however. When we count the number of X's for every pattern, the range of possible values is not exploited when the number of hidden units is large. Here, with 3 hidden units, the number of X's varies between 0 and 2, while with 8 hidden units, it varies between 2 and 4. This somewhat matches our intuition, because a binary representation would have a larger range than a local representation.

To confirm our observations, we looked again at our character-recognition network, with 10 hidden units and 5 hidden units. We counted the number of X's (hidden output > 0.75) for every pattern. Our results are displayed in **Table 4**.

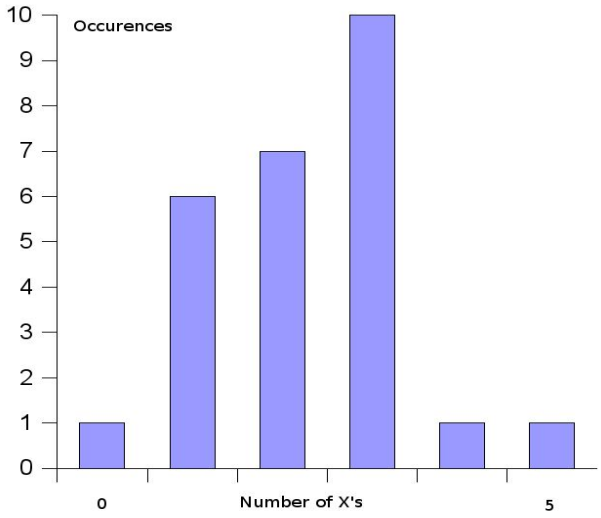
| Pattern # | Number of X's with 10 hidden units | Number of X's with 5 hidden units |
|---|---|---|
| 1 | 5 | 2 |
| 2 | 5 | 1 |
| 3 | 5 | 2 |
| 4 | 4 | 1 |
| 5 | 4 | 1 |
| 6 | 4 | 3 |
| 7 | 5 | 3 |
| 8 | 4 | 2 |
| 9 | 5 | 3 |
| 10 | 4 | 2 |
| 11 | 3 | 3 |
| 12 | 3 | 3 |
| 13 | 5 | 0 |
| 14 | 5 | 2 |
| 15 | 5 | 2 |
| 16 | 6 | 3 |
| 17 | 4 | 3 |
| 18 | 4 | 5 |
| 19 | 6 | 3 |
| 20 | 5 | 4 |
| 21 | 4 | 1 |
| 22 | 7 | 2 |
| 23 | 3 | 3 |
| 24 | 4 | 1 |
| 25 | 5 | 3 |
| 26 | 5 | 1 |
| min | 3 | 0 |
| max | 7 | 5 |

**Table 4** *comparison of #X's in trained networks with 10 and 5 hidden units*




**Histogram 1** *Occurrences of X's counts with 10 hidden units*

*Remark: we notice that choosing to place X's where the activation is greater than 0.75 instead of the normal value of 0.5 implies a bias that should not be a problem as it appears in both histograms.*




**Histogram 2** *Occurrences of X's counts with 5 hidden units*

The observations concur. With 5 hidden units, we see that the full range 0-5 is represented, while with 10 hidden units, only the range 3-7 is represented. We observe that the range for more hidden units is even smaller. With the histograms we can see that X's counts have a better repartition in the 5 hidden units case. All this allows us to affirm that more hidden units leads to a more local representation and fewer hidden units leads to a more binary representation.

## *On the Continuity of Outputs*

Indeed, the simple network shows that it is possible, because of the continuous nature of the output of the hidden units, to "code" for all patterns with less than the minimum number of units required for a binary representation of all these patterns. We tried to train the simple network with only two hidden units and we succeeded after 6000 cycles (with a learning rate of 2.0), though the error fluctuates a lot in the range 2'000 - 4'000 cycles as shown on **Figure 6**.

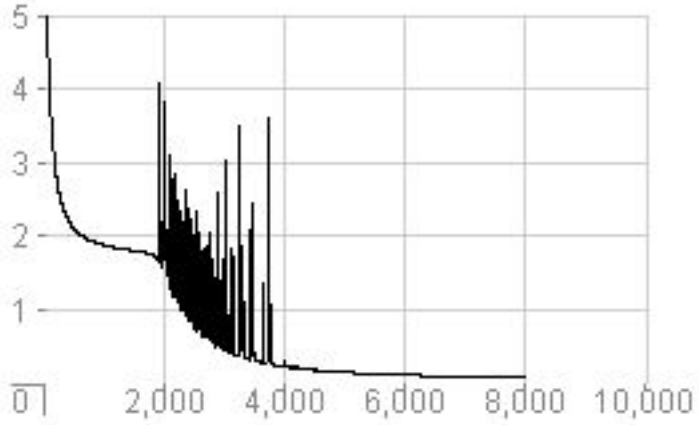



**Figure 6** *error graph of simple network with only 2 hidden units*

# 5. Varying which input units are linked

We wrote a little python program, listed in **Appendix A**, to find the smallest sets of input units needed to recognize all the characters.

In the best imaginable case, we would need only 5 (=ceil(log2(26))) input units. However, given the figures to recognize, we found that the smallest number of input units needed is 8. There are a few such small sets. Here is the first one of them (see **Appendix B** for a complete listing):

```
11000000000000101100001000000011000
```

That is we keep only the units 1, 2, 15, 17, 18, 23, 31, 32. We generated a network with only these input units linked to the hidden units. It is shown in **Figure 7**.

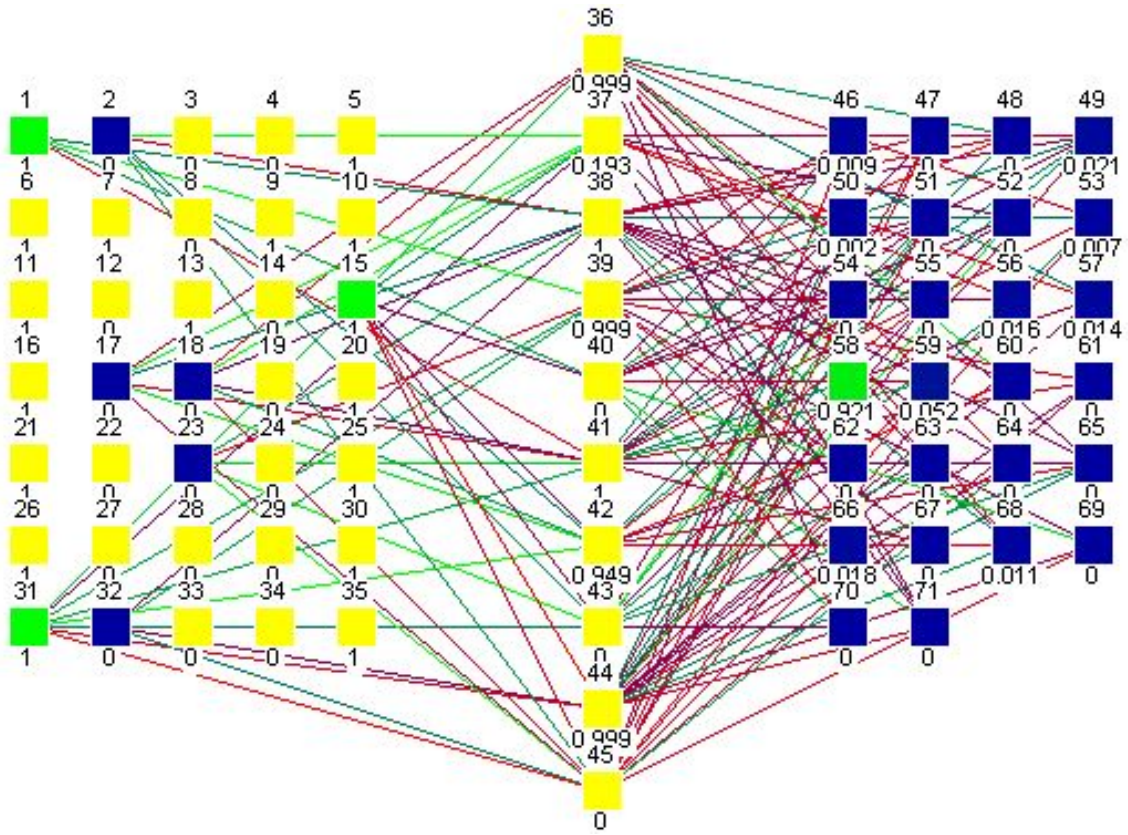


**Figure 7** *Neural network with only eight, well-chosen linked input units*

This network takes ~600 cycles to train with a learning rate of 2.0. It is interesting to note that the network does not require that many more cycles, though the task seems harder because there are less "clues" for the network to work with.

Now, if we remove the links of input 1 from this network, we get an ambiguous network. Using our python program (see the function `lettersamb` in **Appendix A**), we find that pattern 10 (figure `J`) and pattern 15 (figure `O`) look the same, as do pattern 1 (figure `A`) and pattern 16 (figure `P`). Since the network is ambiguous, it will not be possible to train it. We tried to train the network. The error stabilizes around 2. As before, refer to sections *"Varying the Patterns"* and *"Varying the Number of Hidden Units"*, it seems that the error stabilizes around the number of patterns that the network fails to learn. In this case, it is 2, because the network, at best, can only learn one element of each list of indistinguishable figures. So here, since we have two pairs of ambiguous patterns, it can, at best, only learn 2 of them, and thus will fail to learn the 2 others.

# 6. Conclusion

Two compromises have to be found with back-propagation neural networks:

1) If we choose a high learning rate, we may get faster to a good configuration of the network but we also have a higher risk of having to wait longer. This dispersion in the number of accomplishment cycles can be explained by the length of the "jumps" in the weights space that are longer with higher learning rates.
2) The number of neurones in the hidden layer strongly affects the speed of the whole process and the robustness of the solution. A particular attention should be taken considering the choice of the architecture of the network.

The character recognition is a problem that will never be fully solved as there will always be at least one font or one way to write something that will not be recognized. But, the neural networks approach was very interesting for us to apply to this problem.

In spite of the theoretical omnipotence of the neural networks (any kind of algorithm could be synthesized) we cannot expect to get a solution to every problem as the choices of the parameters and the architecture are critical and there are always the problems of the local minimum and the specialization. But, we can spare ourselves a lot of understanding and modeling of the phenomenons included in the problem by letting the neural network just learn.

# Appendix A (ambiguous.py)

This is ambiguous.py, a Python program to run with the command:

```
# python ambiguous.py
```

See the output given in **Appendix B**.

One interesting trick used here is the iterator of genMask which is a “Simple Generator”. As there is a huge quantity of possible masks, it was a necessity to use the masks one by one.

Further explanations on “Simple Generators” can be found at http://www.python.org/peps/pep-0255.html .

```
"""Masks, patterns and ambiguities.

When run as a program, this module finds all the smallest perfect
masks (this may take a while!) for 26 patterns representing the
letters of the alphabet. A perfect mask is one which still allow all
the pattern to be distinguished from one another.

Authors:      Nada Amin <nada.amin@epfl.ch> (mostly)
              Louis Villedieu <louis.villedieu@epfl.ch>
Version: 2004-04-15
"""

def maskPattern(pattern, mask):
    """Returns the masked pattern.

    Examples:

    >>> maskPattern('110110', '000000')
    []
    >>> maskPattern('110110', '001100')
    ['0', '1']
    """
    newpattern = []
    for i in range(0,len(mask)):
        if mask[i] == '1':
            newpattern.append(pattern[i])
    return newpattern

def genMasks(ones, size):
    """Returns an iterator of all masks of the given size containing
    the given number of 1's.

    Examples:

    >>> gen = genMasks(2, 3)
    >>> gen.next()
    ['1', '1', '0']
    >>> gen.next()
    ['1', '0', '1']
    >>> gen.next()
    ['0', '1', '1']
    >>> gen.next()
    Traceback (most recent call last):
    StopIteration
    """
```

```
    if ones > size:
        print 'ones (%d) is bigger than size (%d)' % (ones, size)
        return

    if size == 0:
        yield []
        return

    if ones > 0:
        gen = genMasks(ones-1,size-1)
        while 1:
            try:
                next = gen.next()
                next.insert(0,'1')
                yield next
            except StopIteration:
                break

    if size > ones:
        gen = genMasks(ones, size-1)
        while 1:
            try:
                next = gen.next()
                next.insert(0,'0')
                yield next
            except StopIteration:
                break

    return

def isamb(mask, patterns):
    """Given the mask, returns True if at least two patterns look the
    same when masked, false otherwise.

    Examples:

    >>> isamb('100', ['100', '110'])
    True
    >>> isamb('010', ['100', '110'])
    False
    """
    masked = []
    for p in patterns:
        m = maskPattern(p, mask)
        if m in masked:
            return True
        masked.append(m)
    return False

def genPerfectMasks(ones, size, patterns):
    """Returns an iterator of all masks of the given size containing
    the given number of 1's that still allow the given patterns to be
    distinguished unambiguously.

    Examples:

    >>> gen = genPerfectMasks(2, 3, ['111', '101'])
    >>> gen.next()
    ['1', '1', '0']
    >>> gen.next()
```

```
    ['0', '1', '1']
    >>> gen.next()
    Traceback (most recent call last):
    StopIteration
    """
    masks = genMasks(ones, size)
    while 1:
        try:
            mask = masks.next()
            if not isamb(mask, patterns):
                yield mask
        except StopIteration:
            break
    return

def findamb(mask, patterns):
    """Given the mask, finds which patterns look the same. Returns a
    list of (pattern, list of indices).

    Examples:

    >>> findamb('110', ['111', '110', '011', '010'])
    ([['0', '1'], [2, 3]), (['1', '1'], [0, 1])]
    >>> findamb('011', ['111', '110'])
    ([['1', '0'], [1]), (['1', '1'], [0])]
    """
    lst = [(maskPattern(patterns[i],mask), i) for i in range(0,len(patterns))]
    lst.sort()

    # find the duplicate masked patterns
    retlst = []

    # we start with the first pattern
    (prev_pattern,num) = lst[0]
    numlst = [num]

    for i in range(1,len(lst)):
        (p,num) = lst[i]
        if prev_pattern == p:
            # add the index to the numlst for this pattern
            numlst.append(num)
        else:
            # add the old list to retlst
            retlst.append((prev_pattern,numlst))
            # and start a new pattern list
            prev_pattern = p
            numlst = [num]

    # add the last found pattern to retlst
    retlst.append((prev_pattern,numlst))

    return retlst

def keepamb(mask, patterns):
    """Returns a list of lists, each list listing the index of the
    elements that have the same masked patterns. Only returns the
    lists with > 2 elements.

    >>> keepamb('110', ['111', '110', '011', '010', '000'])
    [[2, 3], [0, 1]]
```

```
    >>> keepamb('011', ['111', '110'])
    []
    """
    lst = findamb(mask, patterns)
    redlst = []
    for (p,els) in lst:
        if len(els) >= 2:
            redlst.append(els)
    return redlst

# this is the input patterns for the characters of the 26 letters
letters = [
    '01110100011000111111100011000110001',
    '11110100011000111110100011000111110',
    '01110100011000010000100001000101110',
    '11110100011000110001100011000111110',
    '11111100001000011110100001000011111',
    '11111100001000011110100001000010000',
    '01110100011000010111100011000101110',
    '10001100011000111111100011000110001',
    '01110001000010000100001000010001110',
    '11111000010000100001000011000101110',
    '10001100101010011000101001001010001',
    '10000100001000010000100001000011111',
    '10001110111010110001100011000110001',
    '10001100011100110101100111000110001',
    '01110100011000110001100011000101110',
    '11110100011000111110100001000010000',
    '01110100011000110001101011001101111',
    '11110100011000111110101001001010001',
    '01110100011000001110000011000101110',
    '11111001000010000100001000010000100',
    '10001100011000110001100011000101110',
    '10001100011000110001100010101000100',
    '10001100011000110001101011010101010',
    '10001100010101000100010101000110001',
    '10001100010101000100001000010000100',
    '11111000010001000100010001000011111'
    ]

def lettersamb(mask):
    """Returns a list of lists of undistinguisable letters under the
    given mask.

    Example:

    >>> lettersamb('01000000000000101100001000000011000')
    [['j', 'o'], ['a', 'p']]
    """
    alpha = map(chr, range(ord('a'), ord('a')+27))
    return map(lambda x: map(lambda i: alpha[i], x), keepamb(mask, letters))

def _main():
    print 'Find a smallest perfect mask'
    print '============================'
    start = 5
    stop = 20
    mask = None
    for i in range(start, stop):
        print 'Trying with %d ones' % i
```

```
        gen = genPerfectMasks(i, 35, letters)
        try:
            mask = gen.next()
            break
        except StopIteration:
            pass
    if not mask:
        print 'No perfect mask found.'
    else:
        print 'Found mask %s.' % "".join(mask)
        print
        print 'Find all smallest perfect masks'
        print '==============================='
        print 'Found mask %s.' % "".join(mask)
        while 1:
            try:
                print 'Found mask %s.' % "".join(gen.next())
            except StopIteration:
                break

def _test():
    import doctest, ambiguous
    return doctest.testmod(ambiguous)

if __name__ == '__main__':
    _test()
    _main()
```

# Appendix B (`python ambiguous.py`)

This is the output of ambiguous.py (**Appendix A**). These are all the sets of 8 input units needed to differentiate the alphabet (the perfect masks). It took a lot of time to get this output. As there are many perfect masks with eight 1's, it would have been faster to find one using a genetic algorithm. However, it would have been hard to confirm – as we did – that there were no perfect masks with less than eight 1's.

```
Find a smallest perfect mask
============================
Trying with 5 ones
Trying with 6 ones
Trying with 7 ones
Trying with 8 ones
Found mask 11000000000000101100001000000011000.

Find all smallest perfect masks
===============================
Found mask 11000000000000101100001000000011000.
Found mask 11000000000000101100001000000010010.
Found mask 10100000000000101100001000000011000.
Found mask 10100000000000101100001000000010010.
Found mask 10010000000000101100001000000011000.
Found mask 10010000000000101100001000000010010.
Found mask 01001010000000000011001000000011000.
Found mask 01001010000000000011001000000010010.
Found mask 01001000100000000011001000000011000.
Found mask 01001000100000000011001000000010010.
Found mask 01001000010000000011001000000011000.
Found mask 01001000010000000011001000000010010.
Found mask 01001000001000000011001000000010100.
Found mask 01001000000000001101001000000011000.
Found mask 01001000000000001101001000000010010.
Found mask 01001000000000001101000000001010100.
Found mask 01001000000000001100001010000011000.
Found mask 01001000000000001100001010000010010.
Found mask 01001000000000001100001000010010100.
Found mask 01001000000000001110001000000011000.
Found mask 01001000000000001110001000000010010.
Found mask 01001000000000001110000000001010100.
Found mask 01001000000000001100110000000011000.
Found mask 01001000000000001100110000000010010.
Found mask 00101010000000000011001000000011000.
Found mask 00101010000000000011001000000010010.
Found mask 00101000100000000011001000000011000.
Found mask 00101000100000000011001000000010010.
Found mask 00101000010000000011001000000011000.
Found mask 00101000010000000011001000000010010.
Found mask 00101000001000000011001000000010100.
Found mask 00101000000000001101001000000011000.
Found mask 00101000000000001101001000000010010.
Found mask 00101000000000001101000000001010100.
Found mask 00101000000000001100001010000011000.
Found mask 00101000000000001100001010000010010.
Found mask 00101000000000001100000010001010100.
Found mask 00101000000000001110001000000011000.
Found mask 00101000000000001110001000000010010.
Found mask 00101000000000001110000000001010100.
Found mask 00101000000000001100110000000011000.
Found mask 00101000000000001100110000000010010.
Found mask 00011010000000000011001000000011000.
Found mask 00011010000000000011001000000010010.
Found mask 00011000100000000011001000000011000.
Found mask 00011000100000000011001000000010010.
Found mask 00011000010000000011001000000011000.
Found mask 00011000010000000011001000000010010.
Found mask 00011000001000000011001000000010100.
Found mask 00011000000000001101001000000011000.
Found mask 00011000000000001101001000000010010.
Found mask 00011000000000001101000000001010100.
Found mask 00011000000000001100001010000011000.
Found mask 00011000000000001100001010000010010.
Found mask 00011000000000001100000010001010100.
Found mask 00011000000000001110001000000011000.
Found mask 00011000000000001110001000000010010.
Found mask 00011000000000001110000000001010100.
Found mask 00011000000000001100110000000011000.
Found mask 00011000000000001100110000000010010.
Found mask 00010000011000001010010000000011000.
Found mask 00010000011000001010010000000010010.
Found mask 00010000011000001010000000001010100.
Found mask 00010000011000001100100000000011000.
Found mask 00010000011000001100100000000010010.
Found mask 00010000011000001100000000001010100.
Found mask 00010000010100001100100000000011000.
Found mask 00010000010100001100100000000010010.
Found mask 00010000010100001100000000001010100.
Found mask 00010000010001010100100000000011000.
Found mask 00010000010001010100100000000010010.
Found mask 00010000010001010000101000000011000.
Found mask 00010000010001010000101000000010010.
Found mask 00010000010001010000010001010100.
Found mask 00010000010001001100100000000011000.
Found mask 00010000010001001100100000000010010.
Found mask 00010000010001001100000000001010100.
Found mask 00010000010001001000101000000011000.
Found mask 00010000010001001000101000000010010.
Found mask 00010000010001001000010001010100.
Found mask 00010000010000010110100000000011000.
Found mask 00010000010000010110100000000010010.
Found mask 00010000010000010010101000000011000.
Found mask 00010000010000010010101000000010010.
Found mask 00010000010000010010001000101010100.
Found mask 00010000010000001110100000000011000.
Found mask 00010000010000001110100000000010010.
Found mask 00010000010000001101010000000011000.
Found mask 00010000010000001101010000000010010.
Found mask 00010000010000001101000100010101010.
Found mask 00011000010000001100100000000011000.
Found mask 00011000010000001100100000000010010.
Found mask 00011000001000001100100000000011000.
Found mask 00011000001000001100100000000010010.
Found mask 00011000000100001100100000000011000.
Found mask 00011000000100001100100000000010010.
Found mask 00011000000010001100100000000011000.
Found mask 00011000000010001100100000000010010.
Found mask 00011000000001001100100000000011000.
Found mask 00011000000001001100100000000010010.
Found mask 00011000000000101100100000000011000.
Found mask 00011000000000101100100000000010010.
Found mask 00011000000000011100100000000011000.
Found mask 00011000000000011100100000000010010.
Found mask 00011000000000001110100000000011000.
Found mask 00011000000000001110100000000010010.
Found mask 00011000000000001101010000000011000.
Found mask 00011000000000001101010000000010010.
Found mask 00011000000000001101000100010101010.
```