Some Experimental Results on Feed-Forward Networks for Text-Classification

Richard Bergmair

August 15, 2004

Abstract

One well-known design principle guiding the construction of neural networks is that a feed-forward network will be able to filter noise when few hidden units are used. Another design principle states that a network will be able to solve more complex problems when many hidden units are used. The experimental setup presented herein aims at analyzing this tradeoff in the situation where *both* the noise in the input increases *and* the problem grows more complex, at least in the problem-domain of text-classification by unigram-frequencies.

Contents

1 Introduction 2

2 Problem Representation 3

3 Problems and their Difficulty 4

4 Experimental Setup 9

5 Interpretation of Results 9

6 Further Experiments 11

7 Conclusions 12

Project Proposal

Type of project Original Research

Research strategy Experimentation

Resource requirements Nothing will be needed beyond my own personal computing equipment at home.

Literature, and Engineering-tools required The Stuttgart Neural Network Simulator (SNNS) is readily available and has been confirmed to work in my computing environment. Useful literature included Zell (1994) and Zell et al. (n.d.).

Background and Related Research The study of feed-forward networks and the influence of hidden layer neurons to network performance for different problem-classes has been an important topic studied in the ANNE-module as taught at UDA in the summer-semester of 2004. In the course of the final year project I handed in to HTL-Leonding last year and the COL-module taught at UDA this year, I studied topics from Computational Linguistics in detail, and in the course of the final year project I handed in to UDA this year I studied cryptography and steganography in detail. The research-material presented herein relies on these disciplines as well.

Aims and Objectives In accordance with this year's coursework specification, my main objective was to learn how to design neural networks and to implement them (in my case using SNNS).

1 Introduction

As neural networks are successfully applied to ever more problems, we gain experience about network-topologies and their implications on the performance of neural networks in specific problem domains. In the experiment described in this report, neural networks have been successfully applied to problems of text-classification. It aims at pointing out some relations between a network's topology (in particular: about the number of hidden units) and a network's accuracy in classifying a given natural language text.

One well-known design principle used in the construction of neural networks concerns the complexity of classification tasks: The more complex a problem grows in terms of linear separability of output classes of points in the input-space, the more hidden layer neurons are required for a network to successfully distinguish between these classes. Another principle concerns noisy input: The more noise is expected in a network's input, the fewer hidden layer neurons should be used, so the noise does not distract the classification.

This makes the situation straightforward for the designer of a network: If the problem is complex, use many hidden layer neurons, if the input is noisy use few hidden layer neurons. However there are situations where this decision is not that straightforward. What if *both* the problem is complex and the input is noisy? Which design principle should be given precendence in this case? Text-classification is one problem-domain, where this dilemma arises. As text-classification tasks grow more difficult, both the noise in the input and the problem-complexity increase. Evidence for this will be provided later in this report.

The aim of the experiment described herein is to resolve that conflict, and to give advice on a suitable number of hidden layer neurons for text-classification problems. Therefore, a number of text-classification tasks of varying difficulty have been defined and a number of feed-forward-networks with different numbers of hidden-layer neurons have been built. The overall accuracy of all the networks, performing any of the classification tasks was measured, thereby collecting evidence for a possible correlation between problem-difficulty and optimal number of hidden-layer neurons.

2 Problem Representation

The overall problem domain studied herein is text-classification by unigram frequencies (see Jurafsky & Martin (2000) or Charniak (1996) for good descriptions about corpus linguistics with n-grams). The text we will always operate on is taken from the King James version of the Bible. In order to keep down processing times during experimentation, the Torah (only the first five books of the bible) was chosen as the corpus of interest. A python-script was implemented, that converts plaintext from to a format that can be handeled by SNNS. It breaks up the whole text into verses, so that each verse can be used for one pattern. Given an ASCII-string representing one verse with letters converted to upper case, the python script counts how often each of the 26 English alphabetic characters occurs.

Thereby the following bible-verse:

```
033:002
And he said, The LORD came from Sinai, and rose up from Seir
unto them; he shined forth from mount Paran, and he came with
ten thousands of saints: from his right hand went a fiery law
for them.
```

would first be filtered to give

```
AND HE SAID THE LORD CAME FROM SINAI AND ROSE UP FROM SEIR
UNTO THEM HE SHINED FORTH FROM MOUNT PARAN AND HE CAME WITH
TEN THOUSANDS OF SAINTS FROM HIS RIGHT HAND WENT A FIERY LAW
FOR THEM
```

and then the letters counted to give the frequencies which serve as input patterns. The following would then end up in the pattern-file:

```
0.079545 0.000000 0.011364 0.022727 0.085227 0.045455 0.005682
```

```
0.085227 0.039773 0.000000 0.000000 0.011364 0.051136 0.051136
0.079545 0.005682 0.000000 0.062500 0.056818 0.056818 0.011364
0.000000 0.017045 0.000000 0.017045 0.000000
```

Note that input patterns consist of 26 numeric values between 0 and 1. The patterns are given in alphabetic order. If, for example, the letter `A` occurs a times, and there are l letters in this verse, the probability that a randomly chosen letter X is `A` is given by $P(X = \mathtt{A}) = \frac{a}{l}$, in this case `0.079545`. Since `A` is the first letter in the alphabet, this is the activation for the first input neuron, and appears first in the input pattern for the network.

Since the problem always amounts to distinguishing text from the King James version of the Bible from other text, the output-layer of the network always contains two neurons, one to encode for King James text, one to encode for other text. Therefore the output patterns are always encoded as `1 0` (King James) respectively `0 1` (other text). Actually one output-neuron would have been sufficient, to encode for the problem, instead of these pairs of antivalent outputs, but this way it was easier to compare the results to those of networks with more than two output-classes (which were only used in early phases of the prototyping).

3 Problems and their Difficulty

All tasks were simple binary classification-tasks, where inputs from one referece-class (the King James Bible) had to be distinguished from another class, in particular:

1. **ROT**: ROT-5 encrypted Torah from the King James Translation
2. **Lut**: Torah from Luther's Translation
3. **BNC**: Random sample from the British National Corpus
4. **WEB**: Torah from the World English Bible

Throughout the rest of the report we will refer to the different classification-tasks that were studied by these abbreviations.

These test-setups could be seen as representative for the following kinds of text classifications:

1. **ROT**: distinguish human language from machine language.
2. **Lut**: distinguish between English and German language text.
3. **BNC**: distinguish between old-fashioned biblical English and modern everyday English.
4. **WEB**: distinguish between the rather old-fashioned English found in the King-James Bible and the rather modern English found in the World English Bible.

The above lists are ordered by linguistic difficulty of the classification task. Whereas it is easily possible for anyone to tell plaintext from ciphertext, the distinction of "King-James-style" and "World-English-style" would sometimes pose difficulties even to linguists. The performance of the networks also reflects this ordering of difficulty.

In order to gain a deeper understanding of the problems being investigated, I wrote a python-script to count letter-frequencies in the used corpora. Figures 1 and 2 show the results.

Note that the ROT frequencies are the same as the KJV frequencies, when shifted five units to the right.[1] This is why it is possible, observing the frequency of any single letter to tell ciphertext from plaintext. For example, observing the letter `E` one can easily conclude that the present text is plaintext, if it occurs most often, and that the text is ciphertext if it hardly ever occurs (because it originates from a plaintext `Z` which occurs seldomly in English). The problem is more difficult in the case of distinguishing English from German. Most of the letters do not offer good evidence for such a distinction any more. However, by selectively picking the right letters, one can still classify texts quite well. For example `Y` occurs much more frequently in English than in German. The same applies to the British National Corpus. The letters `H`, `I`, and `S` provide some evidence, but the other frequencies hardly deviate. The situation is, of course, even worse for the World English Bible.

The issue of problem complexity could be related to combinatorial considerations. While the ROT-task can be solved by observing (almost) any one letter in the input, the Lut.-task must be solved by picking the right letter to observe. As the letter-frequencies get more similar, for the BNC and WEB tasks, the network has to observe, not only the right letter, but the right combinations of letters.

While the complexity of the task increases, the noise increases as well. Noise in this input channel is due to sparse data. Recall that an input to the network always consists of the letter-frequencies observed in a single Bible-verse. Since Bible-verses are rather short, they do not always accurately reflect the letter frequencies of the whole corpus. This is not yet a significant problem for simple tasks. For example if the verse has 30 letters, then we can observe 30 repetitions of the "choose a letter experiment" which has 26 possible outcomes. However, if we need to classify the text by combinations of three letters, then the 30 letters will provide for $\binom{30}{3} = 4060$ repetitions of the "choose three letters experiment" which has $26^3 = 17576$ outcomes.

This is why, as the tasks grow more difficult, both the noise and the problem-complexity increase, which confronts us with the dilemma explained before: If few hidden layer neurons are used (to filter noise from the input) the problem cannot be represented accurately, and if many hidden layer neurons are used (to reflect the problem complexity) the noise in the input will distract the classification-system.

[1] It is interesting to note that this is how monoalphabetic substitution ciphers have traditionally been broken in military cryptography.

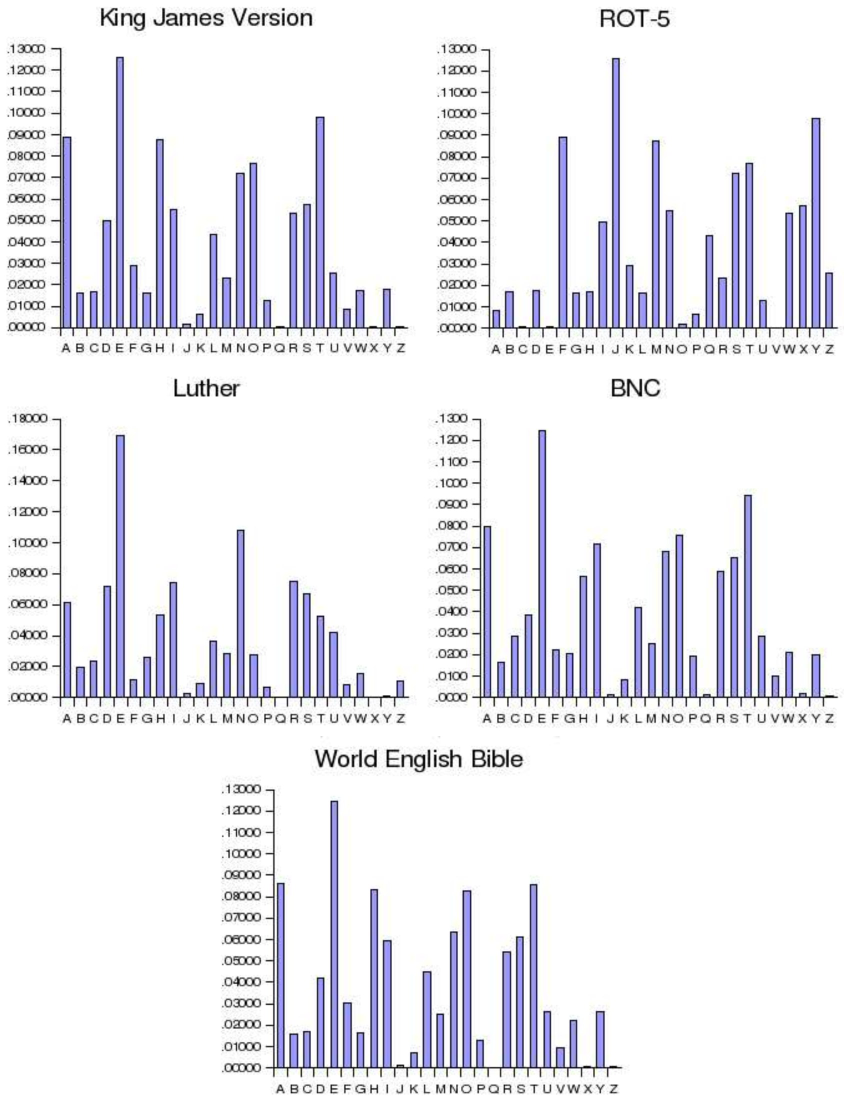


Figure 1: Letter-Frequencies in the used corpora (chart).

	KJV	WEB	BNC	Lut.	ROT
A	.0889	.0859	.0800	.0617	.0085
B	.0163	.0161	.0162	.0192	.0172
C	.0170	.0168	.0287	.0234	.0005
D	.0496	.0423	.0387	.0717	.0178
E	.1259	.1246	.1245	.1693	.0006
F	.0289	.0301	.0220	.0115	.0889
G	.0163	.0166	.0205	.0262	.0163
H	.0875	.0833	.0565	.0533	.0170
I	.0549	.0596	.0715	.0739	.0496
J	.0016	.0015	.0015	.0025	.1259
K	.0063	.0068	.0080	.0092	.0289
L	.0434	.0450	.0420	.0368	.0163
M	.0233	.0249	.0251	.0281	.0875
N	.0721	.0637	.0679	.1081	.0549
O	.0766	.0827	.0759	.0275	.0016
P	.0129	.0132	.0194	.0067	.0063
Q	.0002	.0002	.0011	.0000	.0434
R	.0534	.0544	.0586	.0747	.0233
S	.0572	.0612	.0654	.0668	.0721
T	.0979	.0856	.0942	.0526	.0766
U	.0254	.0261	.0284	.0424	.0129
V	.0085	.0095	.0100	.0082	.0002
W	.0172	.0222	.0211	.0153	.0534
X	.0005	.0005	.0021	.0000	.0572
Y	.0178	.0265	.0201	.0007	.0979
Z	.0006	.0006	.0006	.0103	.0254

Figure 2: Letter-Frequencies in the used corpora (table).

4 Experimental Setup

In order to provide evidence to resolve this conflict, a number of patterns has been prepared, by processing the respective corpora with the program described in the section about problem representation. Each of these patternsets was then divided into a training- and a testing-set by randomly sampling one tenth out of the pattern for testing. The result was a training-set of about 12600 patterns and a testing-set of around 1400 patterns for each of the four tasks.

Simple feed-forward networks were built, differing only in the number of hidden layer neurons. Networks of 3, 4, 8, 16, 32, 64, 128, and 256 neurons were built. The smallest network had 3 neurons, because a network of only 2 hidden layer neurons, could not be trained on the problems any more. Since $26 * \log_2(12600) = 354$ hidden units could simply learn the training-set "by heart" it did not make sense to use larger numbers of hidden layer neurons. The numbers in between were chosen as powers of two, to reflect linearly increasing information-content. All the networks used the logistic activation function and unity as output-function.

Each of these 32 combinations of networks and classification tasks was then examined, by

1. initializing the network with random weights in $[-1 : +1]$

2. training a network using the training-pattern compiled for the task using backpropagation with momentum for a learning-constant $\eta = 0.3$, a momentum term $\mu = 0.2$, a flat spot elimination value $c = 0.1$ and a maximum difference between teaching value and output value $d_{max} = 0.1$ in topological order for 1000 cycles.

3. testing the network using the testing-pattern compiled for the task creating a result-file

4. analyzing the results for correct and incorrect classifications. A classification is considered correct if the activation for the right class has the highest activation. For example if the class is `0 1`, and the test-values are `0.1 0.9`, then the classification is considered correct. If the test-values are `0.55 0.45`, then the classification is considered incorrect.

The result of each of these analyzing-steps is given in Figure 3. The table on the top shows the number of incorrect classifications and the total number of patterns in the testing-set. The table on the bottom shows the error-rate as a percentage of incorrect classifications. Results minimizing the error for each task were highlighted.

5 Interpretation of Results

Figure 4 shows a possible interpretation of these results. The table on the top highlights the global minima. With increasing noise and complexity the number of hidden-layer neurons resulting in the best accuracy is increasing as

	WEB	BNC	Lut.	ROT
3	526/1352	257/1413	17/1386	$\boxed{\mathbf{0/1357}}$
4	525/1352	258/1413	16/1386	0/1357
8	518/1352	176/1413	$\boxed{\mathbf{12/1386}}$	0/1357
16	522/1352	261/1413	13/1386	0/1357
32	519/1352	$\boxed{\mathbf{163/1413}}$	14/1386	0/1357
64	521/1352	246/1413	12/1386	0/1357
128	516/1352	255/1413	14/1386	0/1357
256	$\boxed{\mathbf{502/1352}}$	170/1413	15/1386	0/1357

	WEB	BNC	Lut.	ROT
3	38.905%	18.188%	1.227%	$\boxed{\mathbf{0.000\%}}$
4	38.831%	18.259%	1.154%	0.000%
8	38.314%	12.456%	$\boxed{\mathbf{0.866\%}}$	0.000%
16	38.609%	18.471%	0.938%	0.000%
32	38.388%	$\boxed{\mathbf{11.536\%}}$	1.010%	0.000%
64	38.536%	17.410%	0.866%	0.000%
128	38.166%	18.047%	1.010%	0.000%
256	$\boxed{\mathbf{37.130\%}}$	12.031%	1.082%	0.000%

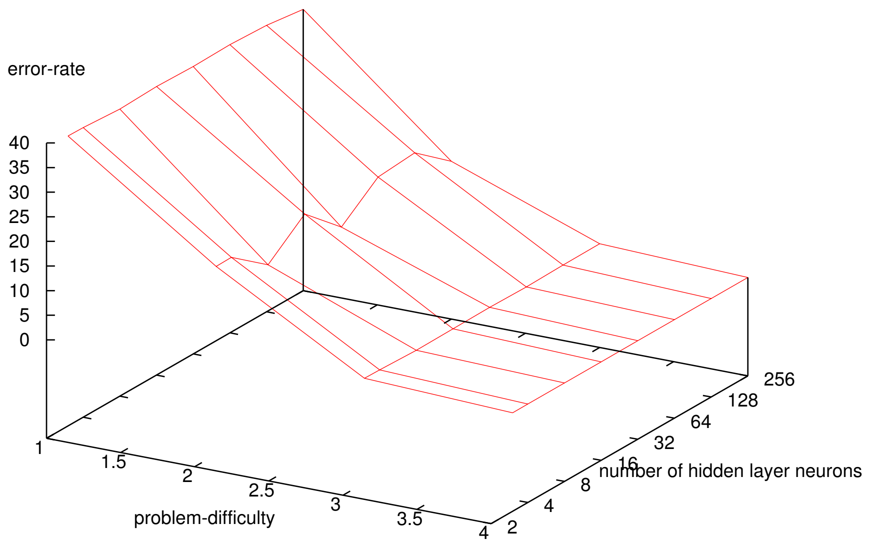
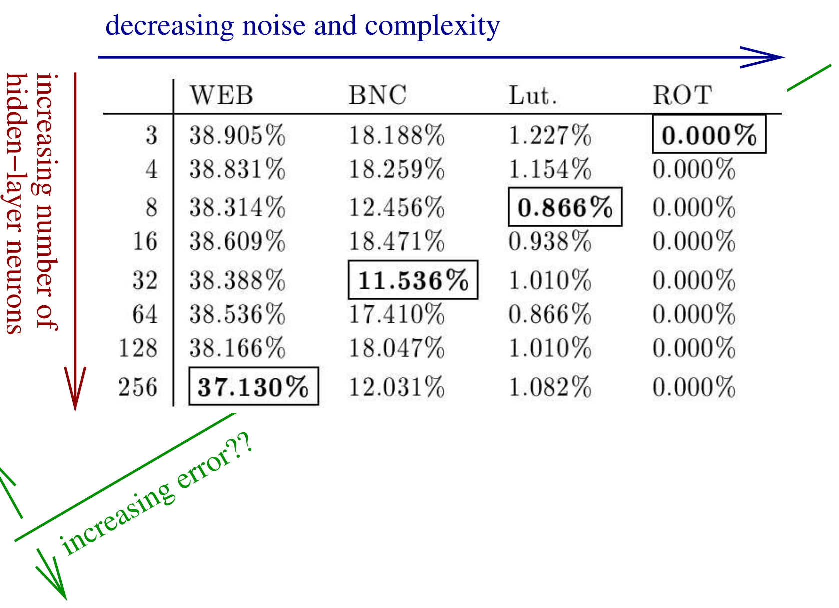
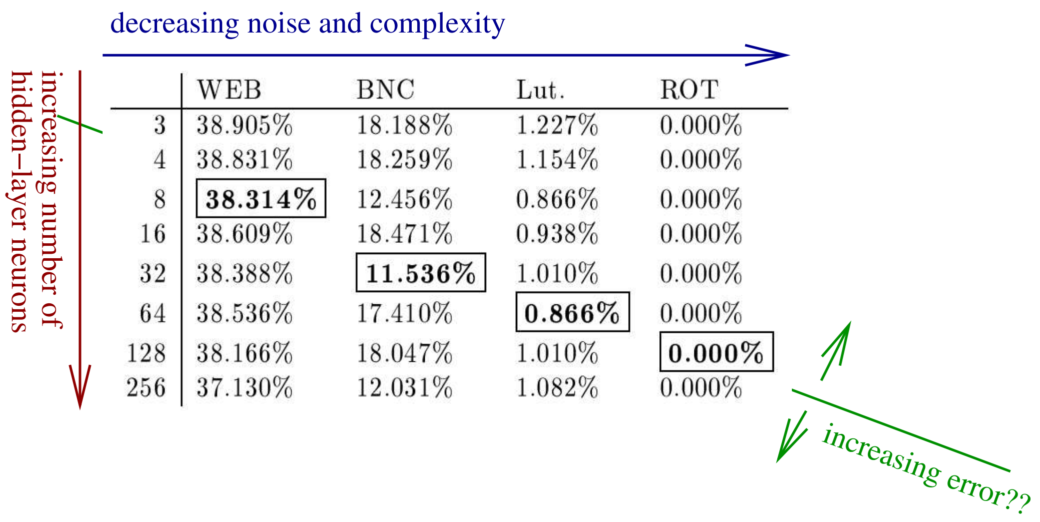


Figure 3: Error of the networks for the different Tasks.


decreasing noise and complexity
increasing number of hidden–layer neurons
increasing error??

	WEB	BNC	Lut.	ROT
3	38.905%	18.188%	1.227%	**0.000%**
4	38.831%	18.259%	1.154%	0.000%
8	38.314%	12.456%	**0.866%**	0.000%
16	38.609%	18.471%	0.938%	0.000%
32	38.388%	**11.536%**	1.010%	0.000%
64	38.536%	17.410%	0.866%	0.000%
128	38.166%	18.047%	1.010%	0.000%
256	**37.130%**	12.031%	1.082%	0.000%


decreasing noise and complexity
increasing number of hidden–layer neurons
increasing error??

	WEB	BNC	Lut.	ROT
3	38.905%	18.188%	1.227%	0.000%
4	38.831%	18.259%	1.154%	0.000%
8	**38.314%**	12.456%	0.866%	0.000%
16	38.609%	18.471%	0.938%	0.000%
32	38.388%	**11.536%**	1.010%	0.000%
64	38.536%	17.410%	**0.866%**	0.000%
128	38.166%	18.047%	1.010%	**0.000%**
256	37.130%	12.031%	1.082%	0.000%

Figure 4: Possible correlations.

well, which reflects the design principle, that more hidden-layer neurons are needed to represent more complex problems. This is opposed to the design principle that states that fewer hidden layer neurons must be used, in order to compensate for increasing noise.

This could lead to the possible conclusion that, as opposed to noise, problem complexity is the predominant factor guiding the design of feed-forward networks for text-classification. Unfortunately the results do not support this claim at a very high confidence. For example for the WEB-task the error actually increases from 8 to 16 and from 32 to 64 hidden layer neurons. Errors for the BNC task increase from 3 to 4, and from 8 to 16 neurons and decrease from 128 to 256. For the Lut task, error decreases from 32 to 64 neurons. All of these are, evidence against the above hypothesis. The ROT task does unfortunately not provide any evidence at all.

The table on the bottom shows an alternative interpretation. Here some local minima have been highlighted that reflect the design principle about noise-filtering, i.e. minima where the number of hidden layer neurons decreases with increasing noise.

Therefore the results of the present experiment demonstrate quite well the design principles about noise and problem complexity, but unfortunately they do not provide significant evidence to support claims about which of these principles should be predominant for design-considerations about text-classification networks.

6 Further Experiments

A lot of effort went into finding good classification-tasks, and carrying out the experiment explained in the previous section. First I had anticipated to classify the text by bigram and trigram frequencies, feeding coded characters to a recurrent network keeping track of characters read before. These letters would then be fed to a hidden layer. I had expected activations, that are logically equivalent to bigram and trigram frequencies to emerge on that layer, so they could be combined on another hidden layer to classify the texts. Unfortunately this did not work out. After a lot of unsuccessful experimentation with recurrent networks I gave up on feeding variable length sequences of coded characters to the network and performed the letter-counting in the python-script preparing the input and feeding unigram frequencies to feed-forward networks.

However it was not yet clear which kind of text-classification could offer for a representative account of problem-difficulty. My first idea was to use different kinds of substitution ciphers. First I experimented with different rotation-ciphers. I included into one pattern-file plaintext as well as ciphertext originating from ROT-7, ROT-9, ROT-13, and ROT-17 ciphers. The network had to classify the patterns into one of these five categories, thereby breaking a cryptosystem with a keyspace of size five. Since the network never made a single error, I turned to a more complex cipher, using a random-number generator to generate random permutations of the alphabet. The neural network still broke all the codes I tried.

For me it was very intersting to observe that simple feed-forward-networks can be applied that successfully to codebreaking, but unfortunately if the network never made a single mistake, it was difficult to make statements correlating network-topology and problem difficulty.

Then I turned back to computational linguistics, examining unigram-frequencies in natural language texts. I used the Bible because it was the only multilingual corpus I could think of, that is available for free from the internet. Of course the texts contained some formatting, which I had to filter out in the python-script preparing the input-patterns. I then compared the performance of networks with 3, 40 and 500 hidden layer neurons, which turned out not be a good choice, since 500 hidden layer neurons are more than enough to learn the input-patterns by heart, and the network with 40 hidden layer neurons alone does not yet provide for any evidence about a correlation. In this first experiment I compared the performance of the networks for WEB, LUT and ROT-5. Since the results about ROT-5 are not representative for any correlation I had to find a third task.

This is where the BNC came in. Unfortunately, preparing patterns from the BNC turned out to be rather difficult, since this corpus is SGML-annotated. I had to write another python-script to pull a random sample from the BNC-texts, and preprocess the text to take out all the SGML-annotation. Since everyday written text is not grouped in verses, I used a random-number generator to pick 2-5 sentences and include them into one "verse". Another difficulty was the fact that the BNC also contained spoken-language text, so I had to run the program several times. Finally I was left with the BNC-patterns and could train the networks. This time I chose the numbers of hidden-layer neurons more wisely.

In my previous experiments I had always used the XGUI to SNNS. Training and testing networks nine times using the graphical interface already turned out a long and tedious task. This is why I used shell-scripts with SNNS' command-line tools and the SNNS batch-interpreter to automate the tasks this time, avoiding the use of the graphical interface to do the essentially same thing 32 times. The results were presented above.

7 Conclusions

An experiment was described to collect evidence about the tradeoff between a network's capability to represent complex problems and to filter out noise. The setup aimed at relating these to the number of hidden layer units needed to give a minimal error in classification tasks of varying difficulty. The results turned out to reflect these design-principles quite well but unfortunately they do not provide significant evidence to support claims about which of the above design principles should be considered predominant in the construction of networks for text-classification.

Appendix A: Programs

Preparing Bible-Text as Input Patterns

```
#!/usr/bin/python

import sys;
import string;
import pickle;
import random;
import copy;



inchars=string.letters+' ';
outchars=string.ascii_uppercase;



def getRandCode():
  code={};
  plain=[];
  for k in outchars:
    plain.append(k);
  cipher=copy.copy(plain);
  random.shuffle(cipher);
  for i in range(0,len(plain)):
    code[plain[i]]=cipher[i];
  code[' ']=' ';
  return code;

def getRotCode(rot):
  code={};
  for c in outchars:
    i=string.find(outchars,c)+rot;
    if i>=len(outchars):
      i-=len(outchars);
    code[c]=outchars[i];
  code[' ']=' ';
  return code;



def readPattern(filename,mark,code):
  f=open(filename);

  i=1;
  a=[];

  line=f.readline();
  while line!='':
    block='';
    xline='';
    while xline!=' ':
      tmpline='';
      if string.find(line,'{')!=-1:
        i=string.find(line,'}');
        tmpline=line[i+1:];
        line=line[:i];

      xline='';
      for c in line:
        if string.find(string.letters+' ',c)!=-1:
          xline+=string.upper(c);
      xline=' '+string.strip(xline);
      block+=xline;

      if tmpline!='':
        line=tmpline;
        tmpline='';
        break;
      else:
        line=f.readline();

    block=string.strip(block);
    if block!='':
      tmp='';
      for c in block:
        tmp+=code[c];
      a.append((tmp,mark));

  f.close();

  return a;




def printPattern(filename,pats):
  out=open(filename,"w");

  maxl=0;
  for pat in pats:
    l = len(pat[0]);
    if l > maxl:
      maxl = l;

  out.write("""SNNS pattern definition file V3.2
generated at Aug  3 00:00:44 1999

No. of patterns : %d
No. of input units : %d
No. of output units : 2


""" % (len(pats),len(outchars)));

  i=1;
  x=0.0;
  cdict={};

  for pat in pats:
    out.write("#\n");
    out.write("# " + pat[0] + "\n");

    out.write("# Input pattern %d\n" % (i));

    for c in outchars:
      cdict[c]=0.0;

    for c in pat[0]:
      if string.find(outchars,c)!=-1:
        cdict[c]=cdict[c]+1;

    for c in outchars:
      x = cdict[c] / len(pat[0]);
      out.write("%f " % (x));
    out.write("\n");

    out.write("# Output pattern %d\n" % (i));
    out.write(pat[1]+"\n");
    out.write("#\n\n");

    i=i+1;

  out.close();

#
# pats=[];
#
# # Random Codes
# pats+=copy.copy(readbible("kjv/test.txt", "1 0 0 0 0", getRotCode(0)));
# pats+=copy.copy(readbible("kjv/test.txt", "0 1 0 0 0", getRandCode()));
# pats+=copy.copy(readbible("kjv/test.txt", "0 0 1 0 0", getRandCode()));
# pats+=copy.copy(readbible("kjv/test.txt", "0 0 0 1 0", getRandCode()));
# pats+=copy.copy(readbible("kjv/test.txt", "0 0 0 0 1", getRandCode()));
#
# # Rotation Codes
# pats+=copy.copy(readbible("kjv/test.txt", "1 0 0 0 0", getRotCode(0)));
# pats+=copy.copy(readbible("kjv/test.txt", "0 1 0 0 0", getRotCode(7)));
# pats+=copy.copy(readbible("kjv/test.txt", "0 0 1 0 0", getRotCode(11)));
# pats+=copy.copy(readbible("kjv/test.txt", "0 0 0 1 0", getRotCode(13)));
# pats+=copy.copy(readbible("kjv/test.txt", "0 0 0 0 1", getRotCode(17)));
#



all=[];
all.append(copy.copy(readPattern("kingjames-torah.txt", "1 0", getRotCode(0))));
# # # all.append(copy.copy(readPattern("luther-torah.txt", "0 1", getRotCode(0))));
# # all.append(copy.copy(readPattern("web-torah.txt", "0 1", getRotCode(0))));
# all.append(copy.copy(readPattern("kingjames-torah.txt", "0 1", getRotCode(5))));
all.append(copy.copy(readPattern("bnc.txt", "0 1", getRotCode(0))));


test=[];
train=[];
for bible in all:
  tst=random.sample(bible,len(bible)/10);
  for x in bible:
    if x in tst:
      test.append(x)
    else:
      train.append(x);

random.shuffle(test);
random.shuffle(train);

# # # printPattern("tsk01/test.pat",test);
# # # printPattern("tsk01/train.pat",train);
# # printPattern("tsk02/test.pat",test);
# # printPattern("tsk02/train.pat",train);
# printPattern("tsk03/test.pat",test);
# printPattern("tsk03/train.pat",train);
printPattern("tsk04/test.pat",test);
printPattern("tsk04/train.pat",train);
```

BNC Preprocessing

```
#!/usr/bin/python

import sys;
import os;
import string;
import copy;
import random;
from sgmllib import SGMLParser;

class bncCollector(SGMLParser):
  handler=0;

  def handle_charref(self,ref):
    # self.handle_data('&#'+ref+';');
    None;

  def handle_entityref(self,ref):
    # self.handle_data('&'+ref+';');
    None;

  textact=0;
  sents=[];
  dta='';
  x=2+random.random()*4;

  def appen(self):
    print string.upper(string.replace(self.dta,"\n",""));
    if self.x<1:
      print "";
      self.x=2+random.random()*4;
    self.x-=1;

    self.dta="";

  def start_s(self, attributes):
    if self.textact==1:
      self.appen();
    self.textact=1;

  def end_s(self):
    self.textact=0;
    self.appen();

  def handle_data(self,data):
    if self.textact==1:
      self.dta+=data;

  def printresults(self):
    x=2+random.random()*4;
    for sent in self.sents:
      if x<1:
        print "";
        x=2+random.random()*4;
        print sent;
        x-=1;



BNC_HOME='/massdata/bnc/'
TEXTS_HOME=BNC_HOME+'Texts/'

f=open("corpus.txt","r");
texts=[];
for line in f:
  texts.append(string.replace(line,"\n",""));
f.close();

sample=random.sample(texts,16);

for stri in sample:
  coll=bncCollector();
  coll.feed((open(TEXTS_HOME+stri)).read());
  coll.printresults();
```

Counting Letters in Corpora

```
#!/usr/bin/python

import sys;
import string;
import pickle;
import random;
import copy;



inchars=string.letters+' ';
outchars=string.ascii_uppercase;

count={};
for ch in outchars:
  count[ch]=0.0;
```

```
overall=0.0;

f=open(sys.argv[1]);

for line in f:
  for ch in string.upper(line):
    if string.find(outchars,ch)!=-1:
      count[ch]+=1;
      overall+=1;

f.close();

for ch in outchars:
  print "%s        %f" % (ch,count[ch]/overall);
```

Creating the Networks

```
#!/bin/bash

BIGNET=snns/ff_bignet

for VAL in 3 4 8 16 32 64 128 256
do
$BIGNET -p 1 26 Act_Logistic Out_Identity input \
        -p 1 $VAL Act_Logistic Out_Identity hidden \
        -p 1 2 Act_Logistic Out_Identity output \
        -l 1 + 2 + \
        -l 2 + 3 + \
        nets/ff-26-$VAL-2.net
done
```

Creating an SNNS-batch for training and testing

```
#!/usr/bin/python

for task in ['tsk01', 'tsk02', 'tsk03', 'tsk04']:
  for netw in [3, 4, 8, 16, 32, 64, 128, 256]:
    print """loadNet("nets/ff-26-%(net)s-2.net");
loadPattern("%(tsk)s/test.pat");
loadPattern("%(tsk)s/train.pat");

setInitFunc("Randomize_Weights",1.0,-1.0)
initNet()

setLearnFunc("BackpropMomentum", 0.3, 0.2, 0.1, 0.1)
while CYCLES < 1000 and SIGNAL == 0 do
  if CYCLES mod 10 == 0 then
```

```
    print ("cycles = ", CYCLES, " MSE = ", MSE)
  endif
  trainNet()
endwhile

setPattern("%(tsk)s/test.pat")
testNet ()
saveResult("%(tsk)s/result-%(net)s.res",1,PAT,FALSE,TRUE)

delPattern("%(tsk)s/test.pat");
delPattern("%(tsk)s/train.pat");


""" % {"tsk":task,"net":netw};
```

Analyzing a Result-File

```
#!/usr/bin/python

import sys;
import string;
import pickle;
import random;
import copy;

f=open(sys.argv[1]);

wcorrect=0;
wincorrect=0;

ncorrect=0;
nincorrect=0;

soll0=0.0;
soll1=0.0;
ist0=0.0;
ist1=0.0;

l1=f.readline();
l2=f.readline();
while l1!='' and l2!='':
  soll=string.split(l1,' ');
  ist=string.split(l2,' ');

  soll0=float(soll[0]);
  soll1=float(soll[1]);
  ist0=float(ist[0]);
  ist1=float(ist[1]);

  # print "soll: %f %f / ist: %f %f" % (soll0, soll1, ist0, ist1);

  if (((soll0>soll1) and (ist0>ist1)) or ((soll0<soll1) and (ist0<ist1))):
    wcorrect+=1;
  else:
    wincorrect+=1;

  if ((soll0>0.5) and (ist0>0.5)) or ((soll0<0.5) and (ist0<0.5)):
    ncorrect+=1;
  else:
    nincorrect+=1;


  l1=f.readline();
  l2=f.readline();

print "by winner:"
print wcorrect;
print wincorrect;

print "by number:"
print ncorrect;
print nincorrect;

print "TeX:"
print "%d/%d (%2.3f\%%)" % (wincorrect,wcorrect+wincorrect,(float(wincorrect)/(float(w
```

Controlling the Analysis of all Results

```
#!/bin/bash

for VAL in `find -name '*.res'`
do
  cat $VAL | grep -e '^[01]' > "${VAL}_"
  rm $VAL
  mv "${VAL}_" $VAL

  echo
  echo "-------- $VAL ---------- "
  echo

  ./resanalyze.py $VAL | tee $VAL.analyze
done
```

References

Charniak, E. (1996), *Statistical Language Learning*, MIT Press.

Jurafsky, D. & Martin, J. H. (2000), *Speech and Language Processing*, Prentice Hall.

Zell, A. (1994), *Simulation neuronaler Netze*, R. Oldenbourg Verlag München Wien.

Zell, A., Mamier, G., Vogt, M., Mache, N., Hübner, R., Döring, S., Herrmann, K.-U., Soyez, T., Schmalzl, M., Sommer, T., Hatzigeorgiou, A., Posselt, D., Schreiner, T., Kett, B., Clemente, G., Wieland, J. & Gatter, J. (n.d.), *SNNS User Manual*. Version 4.2.