情報工学実験IIレポート（探索アルゴリズム2）

月曜日　7グループ

2010年12月20日

グループメンバ

- 095701B 青木 史林: 担当 Level1.1, 1.2, 3.1, 3.4
- 095703J 岩瀬 翔　: 担当 Level3.1,3 .2, 3.3
- 095707B 大城 佳明: Level2, 3.1

1 Level1: 線形分離可能な OR 問題への適用

1.1 課題説明

2 入力 1 出力で構成される単純パーセプトロン（ニューラルネットワーク）を用いて、4 つの教師信号を用意した OR 問題へ適用し、重みが適切に学習可能であることを確認する。また、学習が収束する様子をグラフとして示す。

1.2 Level1.1: OR 問題が解けることの確認

1.2.1 学習が収束する回数

指定されたシード値を用いた際の、学習が終了した回数を表す。1 に示す。

表 1: OR 問題の学習に要した回数

シード値	収束した回数
100	99
200	125
300	139
400	114
500	94
600	97
700	93
800	110
900	125
1000	96

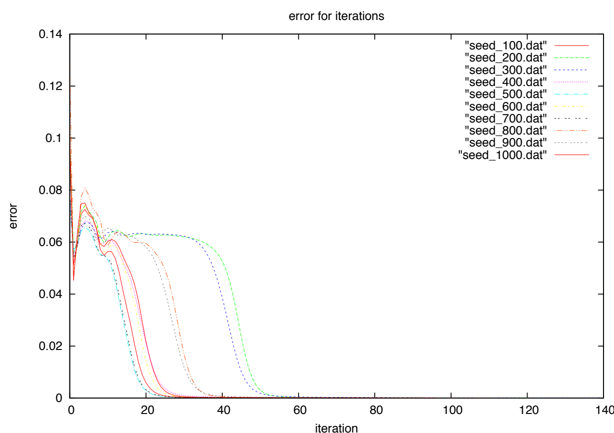


図 1: seed 値 100〜1000 まで実行したときの結果

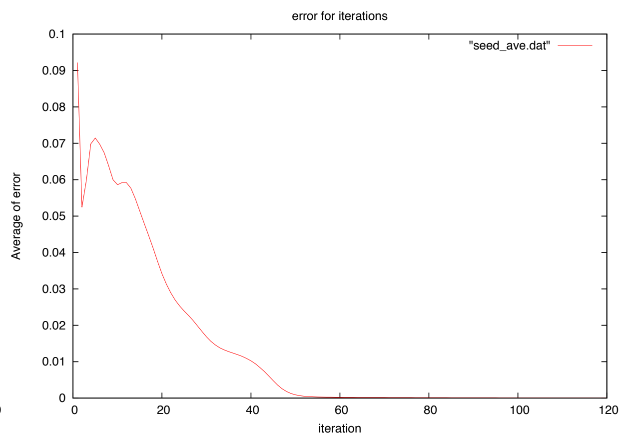


図 2: seed 値 100〜1000 まで実行したときの平均

1.2.2 考察

上記の 2 つの図は x 軸が収束回数に、y 軸が誤差に対応する。また、図 1 は 10 パターンの収束回数との誤差の関係、図 2 は 10 パターンの結果の平均の誤差を表したものである。上記の 2 つの図から、収束回数が 60 回を超えた時点で誤差が限りなく 0 に近づいていることがわかる。また、作成したソースの説明はソースのコメントとして表記してある。

bp_mo.c

```
 /* 数字だけ抽出をするために「 , 」の位置を少しずらした*/
  fprintf(stdout,"iteration = %4d , error = %1.5f\n",ite,err);
```

gnuplot.sh

```
#!/bin/sh

#----------------------------------------------------------------#
#-------------------------bp_mo.eps を作成-------------------------#
#----------------------------------------------------------------#



sum=0
#------------$random_seed を 100〜1000 までを実行する----------#
for a in 100 200 300 400 500 600 700 800 900 1000
do
    #--------実行結果の左から 3 番目と 7 番目の数値を抽出--------#
    num=‘./bp_mo $a  grep iteration — tr -s ” ” ” ” — cut -f3 -f7 -d ” ”‘—

    #--------抽出した数字を seed_(番号).dat に記録---------#
    echo "$num" >> seed_$a.dat
done

#------------出力画像の形式を設定．ここでは eps で出力する-----------#
echo ’set terminal postscript eps color’ > bp_mo.gnuplot

#------------出力ファイル名を決定------------#
echo ’set output "bp_mo.eps"’ >> bp_mo.gnuplot

#------------グラフのタイトルを設定-----------#
echo ’set title "error for iterations"’ >> bp_mo.gnuplot

#------------x 軸のラベルを設定-----------#
echo ’set xlabel "iteration"’ >> bp_mo.gnuplot

#------------y 軸のラベルを設定----------#
echo ’set ylabel "error"’ >> bp_mo.gnuplot
```

```
#------------y 軸に 2 カラム目（月）, x 軸に 1 カラム目（気温）をプロット----------#
echo 'plot "seed_100.dat" with lines ,"seed_200.dat" with lines,
"seed_300.dat" with lines,"seed_400.dat" with lines,"seed_500.dat" with lines,
"seed_600.dat" with lines,"seed_700.dat" with lines,"seed_800.dat" with lines,
"seed_900.dat" with lines,"seed_1000.dat" with lines'>> bp_mo.gnuplot
gnuplot < bp_mo.gnuplot
#------end for gnuplot
#-----------------------------------------------------------------#
#-----------------------bp_mo_ave.eps を作成----------------------#
#-----------------------------------------------------------------#


j=1
while :
do
    ave=0

    #--------それぞれの seed_(番号).dat のデータの平均を取る--------#
    for a in 100 200 300 400 500 600 700 800 900 1000
    do
 #----seed_(番号).dat のファイルから j 行目の左から 2 番目の数値を抽出する--#
 num=‘cat seed_$a.dat   tr -s " " " " — cut -f2 -d " " — sed -n "j"p‘|
   #----抽出した数値が存在するかどうか判断-----#
 if [ "$num" != "" ] ; then
     #存在するならば ave に値を加算する (bc は小数表示)—-—
     ave=‘echo "$ave + $num" bc‘—
 fi

    done

    #--------seed を 100〜1000 のデータを平均する-------#
    ave=‘echo "scale=5;$ave / 10" bc‘—

    #--------j 行目の平均値を seed_ave.dat に記録する-----#
    echo "$ave"
    echo "$j  $ave" >> seed_ave.dat

    #--------120 行目ならば処理は終了される--------#
    j=‘expr $j + 1‘
    if [ $j -gt 120 ] ; then
        break
    fi

done
```

```
#------------出力画像の形式を設定．ここでは eps で出力する-----------#
echo 'set terminal postscript eps color' > bp_mo_ave.gnuplot

#------------出力ファイル名を決定------------#
echo 'set output "bp_mo_ave.eps"' >> bp_mo_ave.gnuplot

#------------グラフのタイトルを設定-----------#
echo 'set title "error for iterations"' >> bp_mo_ave.gnuplot

#------------x 軸のラベルを設定-----------#
echo 'set xlabel "iteration"' >> bp_mo_ave.gnuplot

#------------y 軸のラベルを設定----------#
echo 'set ylabel "Average of error"' >> bp_mo_ave.gnuplot


#------------y 軸に 2 カラム目（月），x 軸に 1 カラム目（気温）をプロット----------#
echo 'plot "seed_ave.dat" with lines'>> bp_mo_ave.gnuplot

gnuplot < bp_mo_ave.gnuplot
#------end for gnuplot


#------------最後にいらなくなったデータファイルを削除する---------#
rm *.dat
```

2 Level2: 線形分離不可能な ExOR 問題への適用

2.1 課題説明

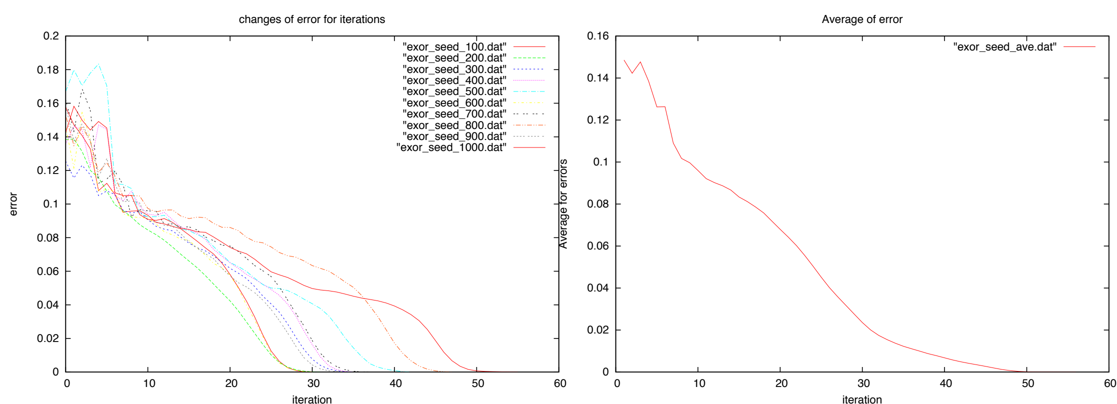
階層型ニューラルネットワークを ExOR 問題へ適用し、線形分離できない問題においても学習可能であることを確認する。特に Level2 では、この問題を解決するために中間層を導入することで拡張した階層型ニューラルネットワークにより学習可能であることを確認する。

2.2 階層型 NN による学習

2.2.1 実行結果

表 2: 階層型 NN による ExOR 問題の学習に要した回数

シード値	収束した回数
100	30
200	31
300	35
400	35
500	42
600	30
700	37
800	47
900	35
1000	55
10 試行の平均値	37.7



2.2.2 考察

詳しい説明はソースにて記載。

```
int main(argc,argv)
    int argc;
    char **argv;
{

  int i,j,ite,ctg;
  double err,sum;
  int seed;

  /* マクロを main 関数の中の変数に変更した。*/

  double ETA;   /* 学習係数 (0 以上の実数) */
  double ALPHA;   /* 慣性項 (0〜1 の実数) */
  int HIDDEN;/* 中間層のニューロン数 */
  /* 最も早く収束する組み合わせを格納する変数 */
  double min_a=0;
  int min_h=0;
  double min_e=0;
  int min_c=1000;

  /* HIDDEN の値を変えて繰り返す */
  for(HIDDEN=1;HIDDEN<30;HIDDEN+=1){

    /*各 HIDDEN の最小の時の組み合わせを格納する*/
    double min_a_a=0;
    int min_h_a=0;
    double min_e_a=0;
    int min_c_a=1000;

    /* ETA の値を変えて繰り返す */
    for(ETA=1.9;ETA<2;ETA+=0.01){

      /* ALPHA の値を変えて繰り返す */
      for(ALPHA=0.1;ALPHA<1;ALPHA+=0.1){
    int ave=0; //10 試行の平均を求めるために、ave に合計を格納する

    /* seed の値を 100〜1000 までを調べる */
    for(seed=100;seed<=1000;seed+=100){

      double  i_lay[CTG][INPUT+1],
        h_lay[HIDDEN+1],
        o_lay[OUTPUT],
        teach[CTG][OUTPUT];
```

```
    double ih_w[INPUT+1][HIDDEN],
      ho_w[HIDDEN+1][OUTPUT];

    double h_del[HIDDEN+1],
      o_del[OUTPUT];

    double ih_dw[INPUT+1][HIDDEN],
      ho_dw[HIDDEN+1][OUTPUT];

    /* seed 値をもらい乱数を求める */
    srandom((int)seed);
    ---省略 (break するまで同じ)---
     /* 10 試行の平均を格納する */
   int m_ave = ave/10;
   printf("HIDDEN:%d ETA:%lf ALPHA:%lf count:%d\n",HIDDEN,ETA,ALPHA,m_ave);
  /* 最も平均試行回数が少なかった組み合わせを格納する (HIDDEN) */
   if(m_ave < min_c_a){
    min_h_a=HIDDEN;
    min_e_a=ETA;
    min_a_a=ALPHA;
    min_c_a=m_ave;
  }
  /* 最も平均試行回数が少なかった組み合わせを格納する */
   if(m_ave < min_c){
    min_h=HIDDEN;
    min_e=ETA;
    min_a=ALPHA;
    min_c=m_ave;
  }
     }
   }
   /* HIDDEN , ETA , ALPHA , count  の組み合わせを出力する (グラフ用)*/
   //printf("%d %lf %lf %d\n",min_h_a,min_e_a,min_a_a,min_c_a);
 }
 printf("min -> HIDDEN:%d ETA:%lf ALPHA:%lf count:%d\n",min_h,min_e,min_a,min_c);

 return 0;
}
---省略 (以下初期状態と同じなため)---
```

上のソースを使い、HIDDEN と ETA と ALPHA の値の最適値を求めた。

1.HIDDEN の値を 2〜30、ETA を 1.0〜1.9、ALPHA を 0〜0.99 までの最適値を探した。

実行結果

min -> HIDDEN:15 ETA:1.900000 ALPHA:0.890000 count:61

2.HIDDEN の値を 13〜17、ETA を 1.80〜1.99、ALPHA を 0.500〜0.999 までの最適値を探した。

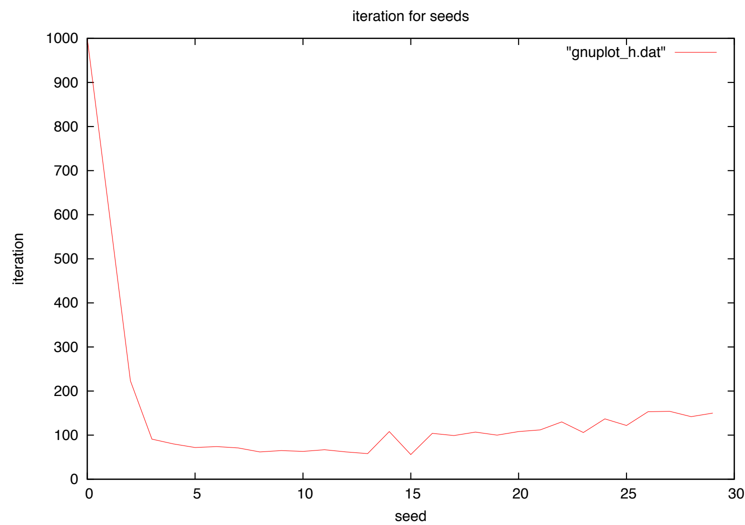


図 3: 実行結果のグラフ

実行結果

min -> HIDDEN:13 ETA:1.920000 ALPHA:0.949000 count:37

3.HIDDEN の値を 13、ETA を 1.9〜1.99、ALPHA を 0.8000〜0.9999 までの最適値を探した

実行結果

HIDDEN:13 ETA:1.920000 ALPHA:0.948700 count:37

(省略)

HIDDEN:13 ETA:1.920000 ALPHA:0.949000 count:37

(省略)

min -> HIDDEN:13 ETA:1.920000 ALPHA:0.948700 count:37

実行結果より count が 37 が最小であることがわかる。ALPHA の値をどんなに小さくしても count が 37 より小さくなることはなかった。以上のことより、HIDDEN=13、ETA=1.92、ALPHA=0.9487〜0.9490 の時に最適値を取るということになる。

3　Level3: 応用事例：文字認識問題への適用

3.1　課題説明

階層型 NN を文字認識に適用し、考察する。特に、用意された教師データと認識のしやすさに関する関係性や、学習最適化のためのパラメータのチューニングおよび、より柔軟性の高い認識方法に関する検討を行う。

3.2　Level3.1: パラメータのチューニング

3.2.1　探索方法

`level3.1` では、データ集計を効率良く行うためのプログラムの修正と、グラフ作成を行うシェルスクリプトの作成を行った。以下に修正したプログラム及びシェルスクリプトを載せる。(変更点だけを載せたいので、`main` 関数のみを記載したいと思う。)

表 3: num.c

```
int main(int argc, char **argv)
{
  int i,j;
  int final_flag;
  int seed;
  int ave=2000;
  double ETA;
  double ALPHA;
  double eta=0,alpha=0;
  for(ETA=1.99;ETA<2;ETA+=0.01){
    for(ALPHA=0.1; ALPHA<=0.9; ALPHA+=0.1){
      int count=0;
      for(seed=100; seed<=1000; seed += 100){


 srandom((int)seed);
 random();


 /* read & set test-case */
 set_problem();

 /* initialize */
 h_lay[HIDDEN]=ON;
 for(j=0;j<HIDDEN;j++)
   for(i=0;i<=INPUT;i++){
     ih_w[i][j]=WD*drand();
     ih_dw[i][j]=0.;
   }
```

```
 for(j=0;j<OUTPUT;j++)
   for(i=0;i<=HIDDEN;i++){
     ho_w[i][j]=WD*drand();
     ho_dw[i][j]=0.;
   }
 usage2();

 ite = 1;
 final_flag = 0;

 count += nn_learn(ETA,ALPHA);
      }
      printf("ETA=%lf , ALPHA=%lf, count=%d\n",ETA,ALPHA,count/10);
      if(count/10 < ave){
 eta = ETA;
 alpha = ALPHA;
 ave = count/10;
      }
    }
  }
  printf("min: ETA=%lf , ALPHA=%lf , COUNT=%d",eta,alpha,ave);
  return 0;
}
```

```
j=1
while :
do
    ave=0

    #--------それぞれの seed_(番号).dat のデータの平均を取る--------#
    for a in 100 200 300 400 500 600 700 800 900 1000
    do

    #----seed_(番号).dat のファイルから j 行目の左から 2 番目の数値を抽出する--#
 num=‘cat result-seed$a.data  cut -f2 -d ” ” — sed -n ”j”p‘|

 #----抽出した数値が存在するかどうか判断-----#
 if [ "$num" != "" ] ; then

 #存在するならば ave に値を加算する (bc は小数表示)—-—
     ave=‘echo "$ave + $num" bc‘—
 fi
   done

    #--------seed を 100〜1000 のデータを平均する-------#
    ave=‘echo "scale=5;$ave / 10" bc‘—

    #--------j 行目の平均値を seed_ave.dat に記録する-----#
    echo "$ave"
    echo "$j  $ave" >> result-seed-ave.data

    #--------120 行目ならば処理は終了される--------#
    j=‘expr $j + 1‘
    if [ $j -gt 553 ] ; then
        break
    fi

done
```

表 4: gnuplot.sh

```
#!/bin/sh
#--seed 値毎の実行結果を gnuplot 用に変換する--#
for i in 100 200 300 400 500 600 700 800 900 1000
do
   num=‘./nn_char $i  tr -s ” ” ” ” — cut -f3 -f6 -d ” ”‘—
   echo "$num" > result-seed$i.data
done

#------------以下の情報を result-seed.gnuplot へ入力していく------------#
echo ’set terminal postscript eps color’ > result-seed.gnuplot

#------------出力の形式を設定．ここでは eps で出力する------------#
echo ’set output "result-seed.eps"’ >> result-seed.gnuplot

#------------グラフのタイトルを設定-----------#
echo ’set title "Average for seeds"’ >> exor_bp_mo.gnuplot

#------------x 軸のラベルを設定-----------#
echo ’set xlabel "Frequency"’ >> result-seed.gnuplot

#------------y 軸のラベルを設定----------#
echo ’set ylabel "Error"’ >> result-seed.gnuplot


#------------y 軸に平均値，x 軸に seed 値をプロット----------#
echo ’plot "result-seed100.data" with lines,"result-seed200.data" with lines,"result-
seed300.data" with lines,"result-seed400.data" with lines,"result-seed500.data" with
lines,"result-seed600.data" with lines,"result-seed700.data" with lines,"result-seed800.data"
with lines,"result-seed900.data" with lines,"result-seed1000.data" with lines’>> result-
seed.gnuplot

gnuplot < result-seed.gnuplot
#------end for gnuplot
```

```
echo 'set terminal postscript eps color' > result-seed-ave.gnuplot

#------------出力ファイル名を決定------------#
echo 'set output "result-seed-ave.eps"' >> result-seed-ave.gnuplot

#------------グラフのタイトルを設定-----------#
echo 'set title "Average for seeds"' >> exor_bp_mo_ave.gnuplot

#------------x 軸のラベルを設定-----------#
echo 'set xlabel "Frequency"' >> result-seed-ave.gnuplot

#------------y 軸のラベルを設定----------#
echo 'set ylabel "Error"' >> result-seed-ave.gnuplot


#------------y 軸に平均値, x 軸に seed 値をプロット----------#
echo 'plot "result-seed-ave.data" with lines'>> result-seed-ave.gnuplot

gnuplot < result-seed-ave.gnuplot
#------end for gnuplot

rm *.data
```

以下に実行結果のグラフを示す。

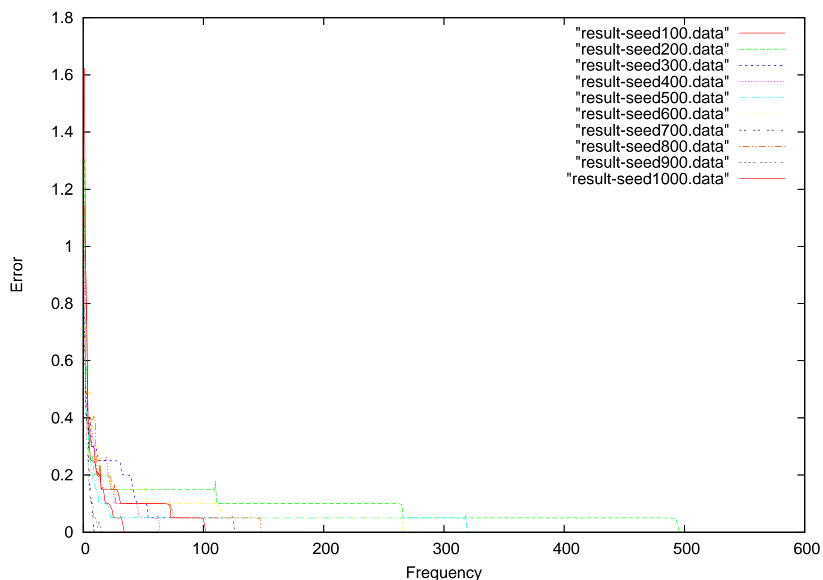


図 4: seed 値 100〜1000 までの結果

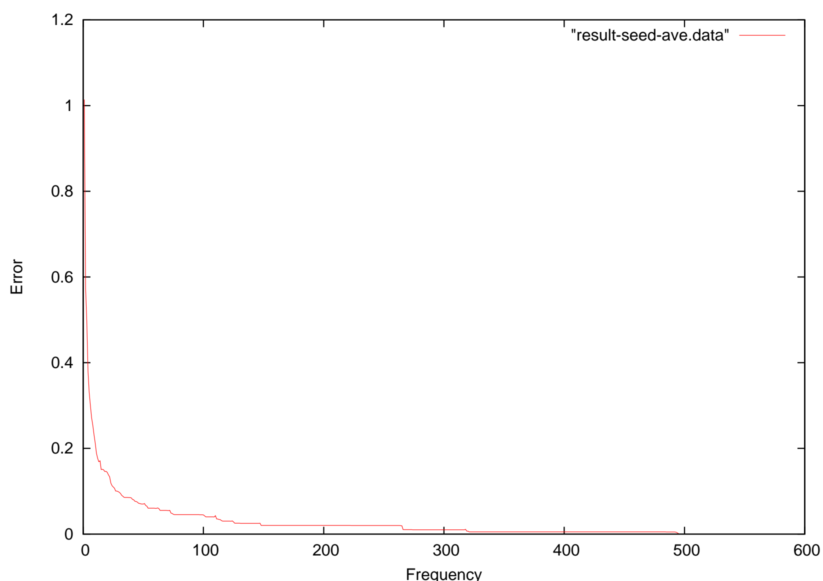


図 5: seed 値 100〜1000 までの平均

元々のプログラムを ETA と Alpha の値を parameter.h に依存しないで、for 文で指定した数だけインクリメントしながら変わるように修正した。レポートに記載されている内容は、ETA の値を 1.99 のまま変えず、Alpha の値を 0.1 から 0.9 までインクリメントさせながらループさせている。実際は、for 文の条件をその都度変更させながらプログラムを実行する。例えば、ETA が 1.90 から 1.99、Alpha の値が 0.11 から 0.20 までのそれぞれの収束値を調べたいときは、

「for(ETA=1.9; ETA<2; ETA+=0.01)」
「for(ALPHA=0.11; ALPHA<=0.2; ALPHA+=0.01)」

と値を変更し、実行する。このプログラムを用いて、班員 3 人で分担して、20〜29、30〜39、40〜49 の範囲で学習後の誤差が最短で収束する組み合わせを探した。parameter.h の HIDDEN の値を変えていき、それを ETA1.99、ALPHA0.1

〜0.9 の間で最も早く収束する組み合わせを探し、さらにそれを細かく範囲を狭めて行く。HIDDEN の値 20~49 の間では以下のような結果になった。

min: ETA=1.990000 , ALPHA=0.301647 , COUNT=389 (HIDDEN 48)

本来ならばこの時点で終わっていたはずだが、金曜日組の学生から情報提供を受け、HIDDEN60 代にも早く収束する値があることがわかった。我々はただちに HIDDEN60 から 69 の間で最も早く収束する組み合わせを捜索した。すると、以下のような結果が出た。

min: ETA=1.990000 , ALPHA=0.200739 , COUNT=315 (HIDDEN 62)

よって、HIDDEN が 62、ETA が 1.99、ALPHA が 0.200739 の時に 315 回で収束することがわかった。この結果を我々は最速だと判断する。また、以下にシード値を 100〜1000 までの 10 パターンで試した際の収束に要した学習回数と平均回数、グラフを示す。グラフは横軸がシード値、縦軸が収束した回数となっている。ちなみに、図 4 、図 5 で示したグラフの HIDDEN、ETA、ALPHA の値はこの時の値を用いた。

表 5: 階層型 NN による文字認識問題の学習に要した回数

シード値	収束した回数
100	212
200	552
300	322
400	300
500	376
600	392
700	249
800	324
900	210
1000	222
10 試行の平均値	315

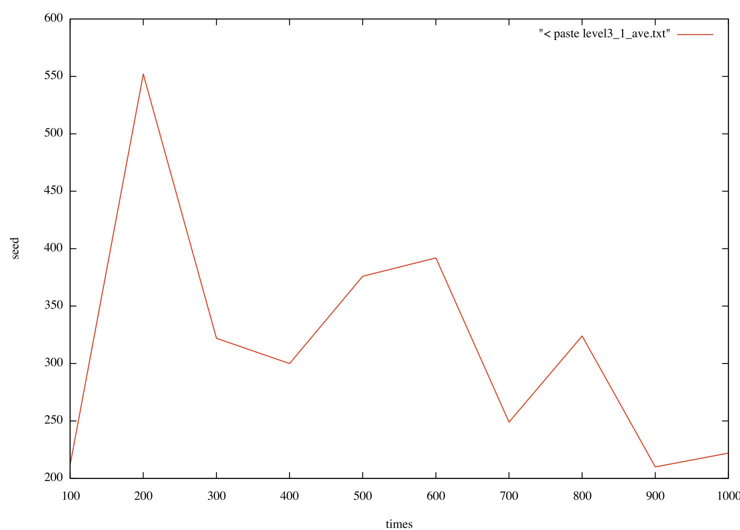


図 6: 重みを更新する様子（平均値）

3.3 Level3.2: パラメータと収束能力の関連性について

3.3.1 関係性を確認するためのアプローチ

3 つのパラメータがどのような関係にあるかを検証するため、ケース 1,2,,, を設定し、学習曲線からその関係性について考察する。

3.3.2 結果

level3.2
以下、HIDDEN の値を 20、30、40 にしたときのそれぞれの結果を載せる。

```
HIDDEN 20
ETA=1.990000 , ALPHA=0.100000 , count=1315
ETA=1.990000 , ALPHA=0.200000 , count=1171
ETA=1.990000 , ALPHA=0.300000 , count=1031
ETA=1.990000 , ALPHA=0.400000 , count=888
ETA=1.990000 , ALPHA=0.500000 , count=757
ETA=1.990000 , ALPHA=0.600000 , count=788
ETA=1.990000 , ALPHA=0.700000 , count=1227
ETA=1.990000 , ALPHA=0.800000 , count=2000
ETA=1.990000 , ALPHA=0.900000 , count=2000
min: EAT=1.990000 , ALPHA=0.500000 , COUNT=757
```

```
HIDDEN 30
ETA=1.990000 , ALPHA=0.100000 , count=896
ETA=1.990000 , ALPHA=0.200000 , count=821
ETA=1.990000 , ALPHA=0.300000 , count=784
ETA=1.990000 , ALPHA=0.400000 , count=689
ETA=1.990000 , ALPHA=0.500000 , count=608
ETA=1.990000 , ALPHA=0.600000 , count=884
ETA=1.990000 , ALPHA=0.700000 , count=1694
ETA=1.990000 , ALPHA=0.800000 , count=1876
ETA=1.990000 , ALPHA=0.900000 , count=2000
min: EAT=1.990000 , ALPHA=0.500000 , COUNT=608
```

```
HIDDEN 40
ETA=1.990000 , ALPHA=0.100000 , count=655
ETA=1.990000 , ALPHA=0.200000 , count=585
ETA=1.990000 , ALPHA=0.300000 , count=571
ETA=1.990000 , ALPHA=0.400000 , count=456
ETA=1.990000 , ALPHA=0.500000 , count=748
ETA=1.990000 , ALPHA=0.600000 , count=947
ETA=1.990000 , ALPHA=0.700000 , count=1620
ETA=1.990000 , ALPHA=0.800000 , count=1934
ETA=1.990000 , ALPHA=0.900000 , count=2000
min: EAT=1.990000 , ALPHA=0.400000 , COUNT=456
```

また、以下に HIDDEN の値を 20、25、30、35、40、45、49 にし、それぞれ ALPHA を 0.1 から 0.9 まで変えて実行したとき、最も早く誤差を収束する ALPHA の値をグラフに表した。横軸が HIDDEN の値、縦軸が ALPHA の値になっている。

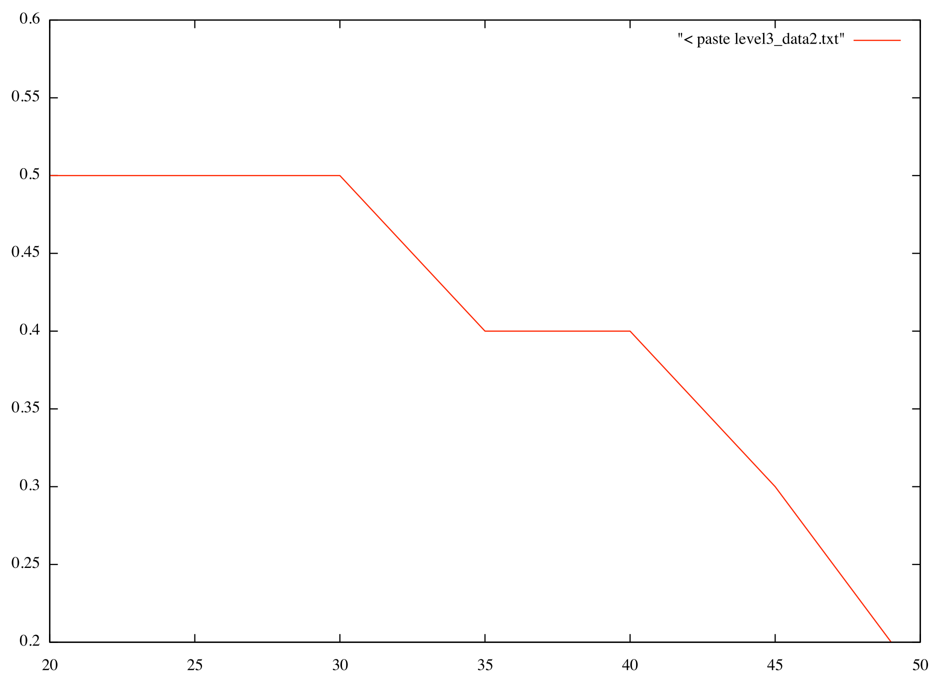


図 7: HIDDEN20〜49

- 以上の結果から、HIDDEN の値が大きいほど、ALPHA の値を小さくした方が比較的速く収束することがわかった。
- ETA の値は、範囲内ならば比較的値が大きいほうが、速く収束することがわかった。
- 但し、なるべく 1.99 から大きくしないことが望ましい。

3.4 Level3.3: 任意の評価用データを用いた評価

評価用データは「い」をモデルとしたデータにした。書き順で 1bit ずつ 0 に変更し、その度に認識率を計った。以下がその結果である。縦軸が変更させた箇所の数、横軸が認識率である。

表 6: 認識率

0	0.99259
1	0.99105
2	0.98439
3	0.96972
4	0.97245
5	0.94737
6	0.92674
7	0.81308
8	0.85892
9	0.74486
10	0.66770
11	0.63317
12	0.51748
13	0.39029
14	0.13272
15	0.14465
16	0.04594
17	0.01258
18	0.00561

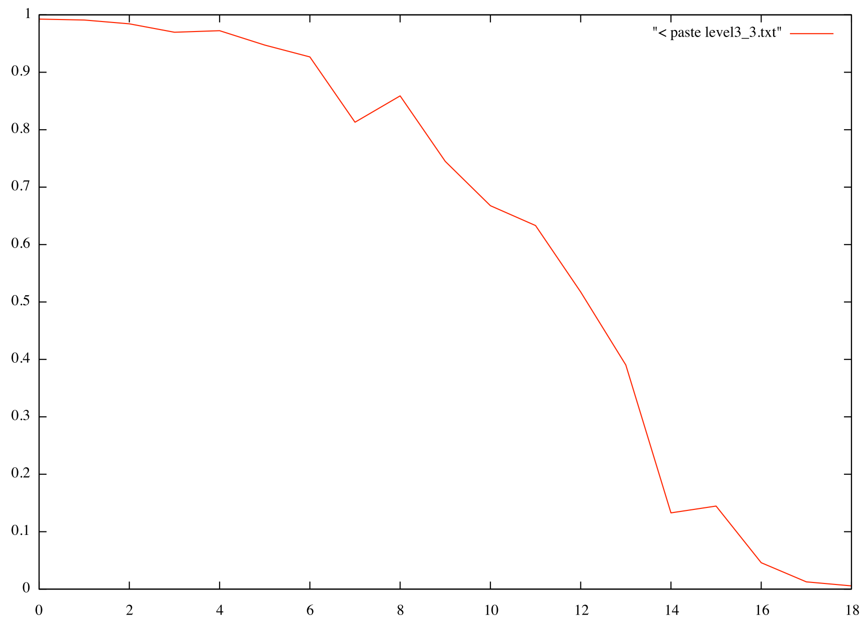


図 8: 認識率の変動

グラフより、1bit ずつ 0 に変更する度に、認識率が減少していくのがわかる。以上の結果から、学習時のデータとの違いが少ないほど認識率が高いことを示すことが出来た。

また、様々な評価用のデータを用いてそれぞれの認識率を求めた。以下に実行結果を記載する。

```
(全部 1 にしたときの認識率)
EVA o[0] = 0.00255, correct[0] = 0.0
EVA o[1] = 0.00827, correct[1] = 1.0
EVA o[2] = 0.00001, correct[2] = 0.0
EVA o[3] = 0.01146, correct[3] = 0.0
EVA o[4] = 0.00094, correct[4] = 0.0
EVA o[5] = 0.01258, correct[5] = 0.0
EVA o[6] = 0.00729, correct[6] = 0.0
EVA o[7] = 0.00049, correct[7] = 0.0
EVA o[8] = 0.00275, correct[8] = 0.0
EVA o[9] = 0.00697, correct[9] = 0.0
EVA sum_error = 1.03676

(上半分を 1 にしたときの認識率)
EVA o[0] = 0.00054, correct[0] = 0.0
EVA o[1] = 0.00002, correct[1] = 1.0
EVA o[2] = 0.00181, correct[2] = 0.0
EVA o[3] = 0.00330, correct[3] = 0.0
EVA o[4] = 0.00042, correct[4] = 0.0
```

```
EVA o[5] = 0.04811, correct[5] = 0.0
EVA o[6] = 0.01698, correct[6] = 0.0
EVA o[7] = 0.00153, correct[7] = 0.0
EVA o[8] = 0.01435, correct[8] = 0.0
EVA o[9] = 0.00611, correct[9] = 0.0
EVA sum_error = 1.09312

(下半分を 1 にしたときの認識率)
EVA o[0] = 0.00987, correct[0] = 0.0
EVA o[1] = 0.13059, correct[1] = 1.0
EVA o[2] = 0.00004, correct[2] = 0.0
EVA o[3] = 0.02911, correct[3] = 0.0
EVA o[4] = 0.00461, correct[4] = 0.0
EVA o[5] = 0.18996, correct[5] = 0.0
EVA o[6] = 0.00056, correct[6] = 0.0
EVA o[7] = 0.01782, correct[7] = 0.0
EVA o[8] = 0.00080, correct[8] = 0.0
EVA o[9] = 0.01963, correct[9] = 0.0
EVA sum_error = 1.14182

(左半分を 1 にしたときの認識率)
EVA o[0] = 0.00529, correct[0] = 0.0
EVA o[1] = 0.07808, correct[1] = 1.0
EVA o[2] = 0.00037, correct[2] = 0.0
EVA o[3] = 0.00264, correct[3] = 0.0
EVA o[4] = 0.00008, correct[4] = 0.0
EVA o[5] = 0.03204, correct[5] = 0.0
EVA o[6] = 0.00873, correct[6] = 0.0
EVA o[7] = 0.00445, correct[7] = 0.0
EVA o[8] = 0.00592, correct[8] = 0.0
EVA o[9] = 0.01576, correct[9] = 0.0
EVA sum_error = 0.99720

(右半分を 1 にしたときの認識率)
EVA o[0] = 0.00365, correct[0] = 0.0
EVA o[1] = 0.00009, correct[1] = 1.0
EVA o[2] = 0.00184, correct[2] = 0.0
EVA o[3] = 0.00508, correct[3] = 0.0
EVA o[4] = 0.00878, correct[4] = 0.0
EVA o[5] = 0.02136, correct[5] = 0.0
EVA o[6] = 0.00203, correct[6] = 0.0
EVA o[7] = 0.00555, correct[7] = 0.0
EVA o[8] = 0.00197, correct[8] = 0.0
EVA o[9] = 0.00191, correct[9] = 0.0
EVA sum_error = 1.05206
```

```
(全部を 0 にしたときの認識率)
EVA o[0] = 0.01058, correct[0] = 0.0
EVA o[1] = 0.00561, correct[1] = 1.0
EVA o[2] = 0.01383, correct[2] = 0.0
EVA o[3] = 0.00441, correct[3] = 0.0
EVA o[4] = 0.00203, correct[4] = 0.0
EVA o[5] = 0.02256, correct[5] = 0.0
EVA o[6] = 0.00417, correct[6] = 0.0
EVA o[7] = 0.00568, correct[7] = 0.0
EVA o[8] = 0.02570, correct[8] = 0.0
EVA o[9] = 0.01554, correct[9] = 0.0
EVA sum_error = 1.09890
```

上記の結果を以下にグラフで表す。

1	全部を 1 にしたときの認識率	0.00827
2	上半分を 1 にしたときの認識率	0.00002
3	下半分を 1 にしたときの認識率	0.13059
4	左半分を 1 にしたときの認識率	0.07808
5	右半分を 1 にしたときの認識率	0.00009
6	全部を 0 にしたときの認識率	0.00561

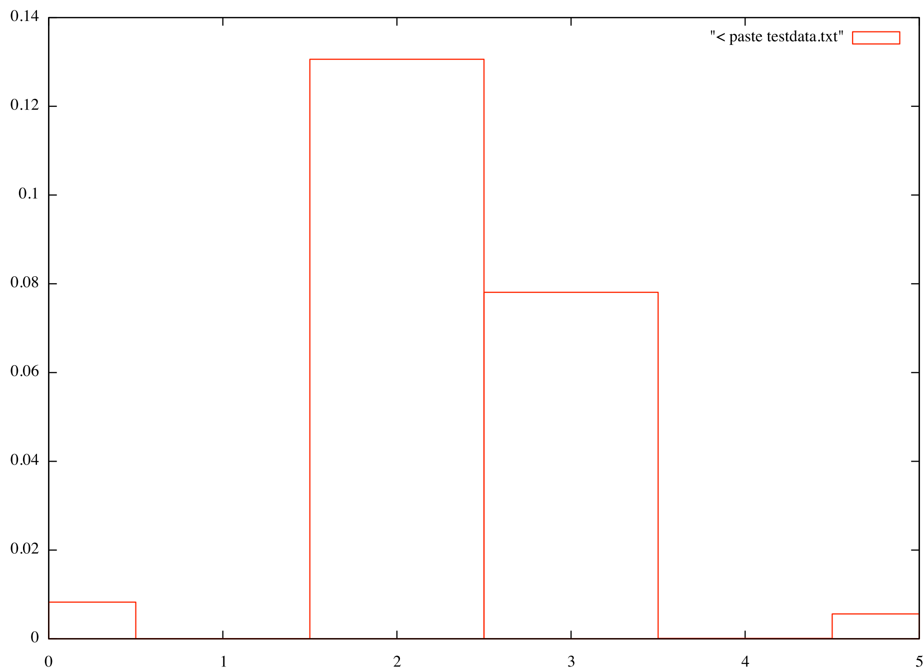


図 9: 評価用のデータの認識率

3.5 Level3.4: 認識率を高める工夫

3.5.1 対象とする問題点

- データが規定よりも大きい場合
- データが規定よりも位置がずれた場合
- 規定と比較してデータに欠落が見られた場合

3.5.2 改善方法の提案

- 誤差をカウントする

たとえば”T”を認識するとき、1 の順列をカウントし、”￣”を描く 1 の羅列が 5 列だったとき、”|”は 5 列ある。
これが理想のパターンであるとし、ある実例で”￣”を描く 1 の羅列が 9 列であったとき、
”|”も 9 列でなければならない。
このとき点数として”0”を与える。
このように各文字についての誤差をカウントして一番誤差の少ない対象を抽出し、結果を出力する。
もしも最小の誤差の結果が 2 以上の複数パターンだった場合はランダムに 1 つ選択して表示する。

- 直線・曲線を見分ける
  直線の場合、一直線に隣接する数値が並ぶことになるが、
  曲線の場合は数値の直線的に並ぶ数を 3 までと限定するなどして曲線と判定する。
  この判別によって曲線を持つ”あ、う、お”であったり、直線を持つ”く、よ”など限定されるので認識できる可能性が向上する。

- 直線・曲線で囲まれた空間を認識する
  直線や曲線によって囲まれた空間が形成されることを認識させる。
  たとえば直線や曲線にそれぞれに順番に番号を付け、連続していく線の番号が今まで名付けていた番号と隣接するとき、囲まれた空間が存在すると認識できる。
  この認識により”あ、お、の、ゆ、る”など特定の線に囲まれた空間を持つ字に限定できる。

- 直線・曲線の本数を認識する
  認識対象の文字がすべて一直線で表現できるのかを認識させる。
  方法は、認識した 1 または 0 と同じ値が現在の場所に必ず隣接するかを判断させる。
  もし線が 1 本のみであるならば”く、し、そ、つ、て、る”などの一画数で表現できる文字にのみ限定できる。

4 その他: 実験の内容・進め方に関するコメント等

(1) 計算機実験を実施するにあたっての考え方、特に、実験計画・実験・結果収集・結果解析・レポート作成までの一連の作業をするにあたって考慮すべき点の発見と、対処方法の検討。

この実験の内容とはとても楽しかったです。データを収集し、そこから新しい発見や見つけることができました。でも仮説がほとんど間違っていたのはとてもやる気がなくなりました。でもいい経験ができたと思います。

(2)(1) の実験を効率良く実施する為に検討すべき項目の調査と、それを解決する手段に関する検討。

とくに悪いところは見当たらなかったので、よかったと思います。しいて言うならば、グループ作業でうまく効率的に実験が行えなかったことです。自分たちのグループの進め方がうまくなかったので、今後はもっと全員が均等に効率よく働けるようなタスク分けができたら、よかったと思いました。

(3) パラメータ・チューニングの必要性やそれらを効率的に実現する為の前処理・後処理のためのテキスト処理や自動化に関する考え方。

自動化の考え方はとても楽しい作業でした。ソースを解析し、可能な限り自動化することができました。探索アルゴリズム２の Level3 の HIDDEN の値が自動化できなかったのがとても残念でした。方法はなくはなかったのですが、処理がものすごく遅くなるか、ソースをかなり書き換えないといけなかったので、諦めました。でも自動化の考え方はとても良かったと思います。

**************進め方************
(1)「解説→グループ討論/実施→全体討論」という形式を取り、他の人/グループが同じテーマを与えられた時にどのような事を考え、アイデアを整理し、どのように発表するのかについて、学生全員が実験時間中に把握出来るように心がけた。

実験中の討論はなかなか新鮮でした。全体討論では自分たちでは考えつかないような思考が聞けて良かったです。授業もあっという間に進んでいき、退屈のしない良い授業だったと思います。

(2)「共同作業（グループ制）」とする事で、個人レベルでの理解度の底上げに努めた。これは、解説の段階で理解度の早い学生は他の人へ教示する事でそのスキルとより理解度が深まる上に、理解度が不十分であった学生は同環境にいる学生の視点からの教示により理解しやすくなる事を期待して実施した。

共同作業はとても大変でした。いつもの数倍の負担だったと思います。でも、がんばった分いいレポートができたのではないかと思います。みんなのレポートの書き方や考察の仕方のを触れることで、今までになかったような実験レポートがかけたと思います。とてもよかったです。平均的に理解度が上がったのではないかと思います。ただ、私たちのグループはとてもよかったですが、他のグループはわかりあません。実験を頑張ってやる人とやらない人とでは明らかに負担の量が違います。そこでいろいろと問題があるのではないかと思いました。でも、私たちのグループはとても満足なレポートを作成できたのですばらいい実験になったのではないかと思います

(3)(2) の間接的な効果として、自分の意見を第三者へ伝えるコミュニケーション能力・レポート作成技術の向上が挙げられます。

コミニュケーションの向上はあまりなかったと思います。なぜならば、私たちのグループはいつも実験は一緒なのでいつもどおりでした。ただ、みんなで頑張って１つのレポートを完成させたという点では団結力が向上したのではないかと思います。

**********今後、実施を検討している以下の項目に関する、賛成/反対等の意見********
(1) レポートの開示。取りあえず、今回の分については採点後、評価の高いレポートについて了承を得てから開示の有無を決定するつもりです。

評価の高いレポートを見せるのは賛成です。評価の高いレポートを見ることにより、自分たちの足りなかった点などを学ぶことができるからです。自分たちとは違った、実験の結果や考察などを見るのもいろいろと勉強になるので良いことだと思います。

(2) 最急降下法、NN、(GA) 以外のアルゴリズムを用いた実験 (「アルゴリズムとデータ構造」等、他の講義で出てくるアルゴリズムを利用した実験等)。

最急降下法は単純なアルゴリズムでしたが、いろいろと面白かったです。NN の実験もデータ収集し、考察を考えるのが楽しかったと思います。しかし、NN のソースの中身を理解できなかったのが残念でした。ソースの解析などを行いたかったですが、全く時間がありませんでした。なので、このソースの意味をどこかのページにアップしてくれたらより楽しく実験ができたのではないかと思います。

***********やって欲しかった内容、その他に関する意見************
先ほどもありましたが、NN のソースの意味が知りたかったです。ソースの仕組みを考えるとまた違った考察ができたのではないかと思います。
全体を通して、この実験は全体的にとても良い実験だったと思います。またこのような楽しい実験があればいいと思います。

参考文献

[1] 情報工学実験 2: 探索アルゴリズムその 1（當間）
http://www.eva.ie.u-ryukyu.ac.jp/~tnal/2010/info2/search2/