Neural Network 1

Project 1 report

Team name: MicVanVan

Members: m5261108 Kazuki Fujita
m5261110 Mizuki Goto
m5261132 Haruki Maeda
m5261147 Kaito Ogino

Table

1. Part 1: AND gate

1.1. Problem

Write a computer program to realize the perceptron learning rule and the delta learning rule. Train a neuron using your program to realize the AND gate. The input pattern and their teacher signals are given as follows:

- Data: (0,0,-1); (0,1,-1); (1,0,-1); (1,1,-1)
- Teacher signals: -1, -1, -1, 1
- Program outputs: Weights of the neuron and Neuron output for each input pattern

1.2. Our solution methods

We remade C-program that Mr. Zhao produced for delta-learning rule and resolved AND gate problem. Two programs for each rule are as follows:

1.2.1. Perceptron learning rule version 1_1_per.c

Our perceptron learning rule program is named 1_1_per.c. To use perceptron learning rule, we changed functions and variable in his program and added a function to determine desire outputs from outputs. Our program was uploaded to GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/1_1_per.c We detail our program in the following chapters.

1.2.1.1. Values of variables

First, we added “time.h” because we want to use rand(). Second, this problem gives us 4 data, which includes two-dimensional value and dummies, and 4 teacher signals, so we changed n_sample = 4, I = 3, contents of x [][], and contents of d[]. These changes is seen in Picture 1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <time.h>

#define I             3
#define n_sample      4
#define eta           0.5
#define lambda        1.0
#define desired_error 0.1
#define sigmoid(x)    (2.0/(1.0+exp(-lambda*x))-1.0)
#define frand()       (rand()%10000/10001.0)
#define randomize()   srand((unsigned int)time(NULL))

double x[n_sample][I]={
 {0,0,-1},
  {0,1,-1},
  {1,0,-1},
  {1,1,-1}
};

double w[I];
double d[n_sample]={-1,-1,-1,1};
double o;

void Initialization(void);
void FindOutput(int);
void PrintResult(void);
void PrintDesiredOutput(void);

```

Picture 1: Global values and prototype declaration in 1_1_per.c

1.2.1.2. main()

We changed delta learning rule to perceptron learning rule in the main() of Mr. Zhao's program. Resume lecture 2-18 shows how we put the rule in a program, so we read it and built main() in 1_1_per.c. This time, a solution don't need error values, thus we commented out its printf().

```
41  int main(){
42      int    i,p,q=0;
43      double learningSignal,Error=DBL_MAX;
44
45      Initialization();
46      while(Error>desired_error){
47          q++;
48          Error=0;
49          for(p=0; p<n_sample; p++){
50
51              FindOutput(p);
52              Error+=0.5*pow(d[p]-o,2.0);
53              learningSignal=eta*(d[p]-o);
54
55              for(i=0;i<I;i++){
56                  w[i]+=learningSignal*x[p][i];
57              }
58              //printf("Error in the %d-th learning cycle=%f\n",q,Error);
59          }
60      }
61      PrintResult();
62  }
```

Picture 2: main() in 1_1_per.c

1.2.1.3. FindOutput()

In perceptron learning rule, *f(u)* is the follows:

$$f(u)\begin{cases} 1 & \text{if } u > 0 \\ -1 & \text{if } u < 0 \end{cases}$$

We let *u as* temp and *f(u)* as o in 1_1_per.c.

```
74  /*************************************************************/
75  /* Find the actual outputs of the network                    */
76  /*************************************************************/
77  void FindOutput(int p){
78      int    i;
79      double temp=0;
80
81      for(i=0;i<I;i++) temp += w[i]*x[p][i];
82      if(temp>0){
83          o=1.0;
84      }else{
85          o=-1.0;
86      }
87  }
```

Picture 3: FindOutput() in 1_1_per.c

1.2.1.4. PrintResult() and PrintDesiredOutput()

We added printing neuron outputs for inputs in PrintResult() and new function PrintDesiredOutput() that print desired outputs for inputs. This program uses o as neuron outputs, so PrintDesiredOutput() compares if o is less than or greater than 0. If o is greater than 0, the function prints 1. In contrast, if o is less than 0, the function prints -1.

```
89  /*************************************************************/
90  /* Print out the final result                                */
91  /*************************************************************/
92  void PrintResult(void){
93    int i,j;
94    double u;
95
96    printf("\n\n");
97    printf("The connection weights of the neurons:\n");
98    for(i=0;i<I;i++) printf("%5f ",w[i]);
99    printf("\n\n");
100
101   for(i=0;i<n_sample;i++){
102     u=0;
103     printf("Input: (");
104     for(j=0;j<I;j++){
105       printf("%1.0f", x[i][j]);
106       if(j!=I-1)printf(",");
107       u+=x[i][j]*w[j];
108     }
109     printf(")   ");
110
111     FindOutput(i);
112     printf("f(u)=%f   ",o);
113
114     PrintDesiredOutput();
115
116   }
117
118 }
119
120 void PrintDesiredOutput(void){
121   printf("The desired output: ");
122   if(o>0) printf("1\n");
123   else printf("-1\n");
124 }
```

Picture 4: PrintResult() and PrintDesiredOutput() in 1_1_per.c

1.2.2. delta learning rule version 1_1_delta.c

Our delta learning rule program is named 1_1_delta.c. We remade Mr. Zhao's program to solve Project 1-1 problem. Values of variables, PrintResult() and PrintDesiredOutput() are the same as 1_1_per.c, so we omit the description. Our program was uploaded to GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/1_1_delta.c We detail our program in the following chapters.

1.2.2.1. main()

His program used delta learning rule, so I used it in 1_1_delta.c as it was. This time, a solution don't need error values, thus we commented out its printf().

```
43  int main(){
44      int    i,p,q=0;
45      double delta,Error=DBL_MAX;
46
47      Initialization();//Initialize the weights at random
48      while(Error>desired_error){
49          q++;
50          Error=0;
51          for(p=0; p<n_sample; p++){
52              FindOutput(p);
53              Error+=0.5*pow(d[p]-o,2.0);
54              for(i=0;i<I;i++){
55                  delta=(d[p]-o)*(1-o*o)/2;
56                  w[i]+=eta*delta*x[p][i];
57              }
58              //printf("Error in the %d-th learning cycle=%f\n",q,Error);
59          }
60      }
61      PrintResult();
62  }
```

Picture 5: main() in 1_1_delta.c

1.2.2.2. FindOutput()

In delta learning rule, *f(u)* is the follows:

If we use the unipolar sigmoid function: $f(u) = \dfrac{1}{1+e^{-\lambda u}}$ ①

If we use the bipolar sigmoid function: $f(u) = \dfrac{2}{1+e^{-\lambda u}} - 1$ ②

Mr. Zhao's program used the bipolar sigmoid function, so we used the same as it. We let *u as* temp and *f(u)* as o in 1_1_delta.c.

```
23  #define sigmoid(x)     (2.0/(1.0+exp(-lambda*x))-1.0)
```

Picture 6: defined sigmoid(x) in 1_1_delta.c

```
74  /*************************************************************/
75  /* Find the actual outputs of the network                    */
76  /*************************************************************/
77  void FindOutput(int p){
78      int    i;
79      double temp=0;
80
81      for(i=0;i<I;i++) temp += w[i]*x[p][i];
82      o = sigmoid(temp);
83  }
```

Picture 7: FindOutput() in 1_1_delta.c

1.3. Results

We shows results of executing each program. Each result outputted to txt files using redirection.

1.3.1. A result using perceptron learning rule

```
3   The connection weights of the neurons:
4   1.513849 1.175482 2.196080
5
6   Input: (0,0,-1)   f(u)=-1.000000   The desired output: -1
7   Input: (0,1,-1)   f(u)=-1.000000   The desired output: -1
8   Input: (1,0,-1)   f(u)=-1.000000   The desired output: -1
9   Input: (1,1,-1)   f(u)=1.000000   The desired output: 1
```

Picture 8: a result of executing 1_1_per.c, f(u) is a neuron output.
GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/ans1_1_per.txt

1.3.2. A result using delta learning rule

```
3   The connection weights of the neurons:
4   6.281142 6.277623 9.502379
5
6   Input: (0,0,-1)   f(u)=-0.999851   The desired output: -1
7   Input: (0,1,-1)   f(u)=-0.923511   The desired output: -1
8   Input: (1,0,-1)   f(u)=-0.923251   The desired output: -1
9   Input: (1,1,-1)   f(u)=0.910115   The desired output: 1
```

Picture 9: a result of executing 1_1_delta.c, f(u) is a neuron output.
GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/ans1_1_delta.txt

1.4. Discussion

In Part 1, we discussed the following points.

- Why was the output correct for the input?
- Why neuron output f(u) was not 1 or -1 using delta learning rule?

1.4.1. Why was the output correct for the input?

The reason is that w[] is calculated until the total error is smaller than desired_error. w[] is initialized by rand() and calculated in main(). The total error is defined as $\frac{1}{2}\sum_{p=0}^{n\_sample-1}(d[p]-o)^2$. If the total error is larger than desired_error, w[] is calculated again. This time, we set desired_error 0.1 in 1_1_per.c and 1_1_delta.c, thus w[] is recalculated until the error becomes smaller than 0.1.

1.4.2. Why neuron output f(u) was not 1 or -1 using delta learning rule?

This reason is that the activation function is a bipolar sigmoid function in delta learning rule. A sigmoid function is continuous and temp is temp is the inner product of weight w[] and data x[] in Picture 7. In addition, if error is smaller than desired_error, the output is the correct judgement. Therefore, neuron output does not have to be 1 or -1.

For neuron output > 0, it would be correct if the desired output is 1. Neuron output is closer to 1, the better learned. For neuron output < 0, it would be correct if the desired output is -1. Neuron output is closer to -1, the better learned.

A Bipolar Sigmoid function is defined as $f(u)=\frac{2}{1+e^{-\lambda u}}-1$. The larger the λ size, the greater the slope of a sigmoid function in graph. In Figure 1, b is used instead of λ.

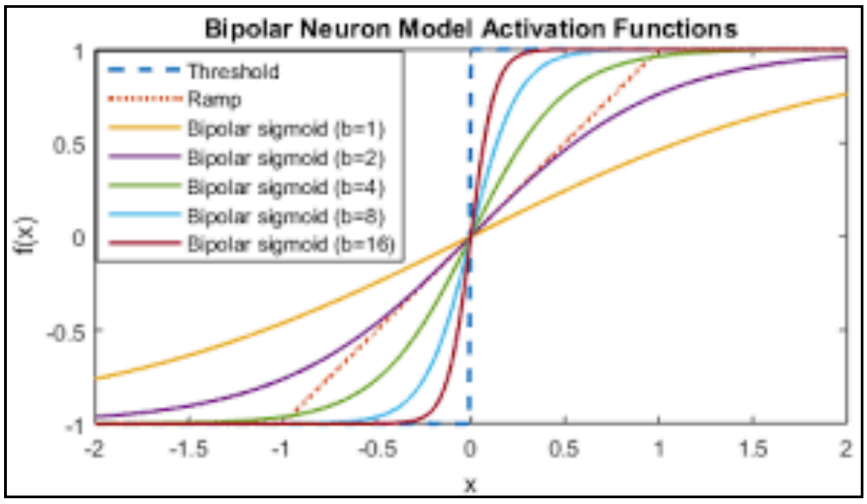

Figure 1: a bipolar sigmoid function [1]

2. Part 2: Extending the program

2.1. Problem

Extend the program written in the first step to learning of single layer neural networks.

The program should be able to design:
- Case 1: A single layer neural network with discrete neurons.
- Case 2: A single layer neural network with continuous neurons.

Test your program using the following data:
- Inputs: (10,2,-1), (2,-5,-1), (-5,5,-1)
- Teacher signals: (1,-1,-1), (-1,1,-1), and (-1,-1,1)

2.2. Our solution methods

We remade C-program that Mr. Zhao produced. Two programs for each rule are as follows:

2.2.1. Case 1 version: 1_2_case1.c

Our case 1 program is named 1_1_case1.c. We used perceptron learning rule to accomplish a single layer neural network with discrete neurons. The program was uploaded to GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/1_2_case1.c We detail our program in the following chapters.

2.2.1.1. Values of variables

Values of variables of Mr. Zhao's program are the same as the data given in Part 2, so we didn't change them. If we must say, we added “time.h” because we want to use rand().

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <time.h>

#define N             3
#define R             3
#define n_sample      3
#define eta           0.5
#define lambda        1.0
#define desired_error 1.0
#define sigmoid(x)    (2.0/(1.0+exp(-lambda*x))-1.0)
#define frand()       (rand()%10000/10001.0)
#define randomize()   srand((unsigned int)time(NULL))

double x[n_sample][N]={
  {10,2,-1},
  {2,-5,-1},
  {-5,5,-1},
};
double d[n_sample][R]={
  {1,-1,-1},
  {-1,1,-1},
  {-1,-1,1},
};
double w[R][N];
double o[R];

void Initialization(void);
void FindOutput(int);
void PrintResult(void);
void PrintDesiredOutput(void);
```

Picture 10: Global values and prototype declaration in 1_2_case1.c

2.2.1.2. main()

We changed delta learning rule to perceptron learning rule in the main() of Mr. Zhao's program. Resume lecture 2-18 shows how we put the rule in a program, so we read it and built main() in 1_2_case1.c. This time, a solution don't need error values, thus we commented out its printf().

```c
int main(){
  int    i,j,p,q=0;
  double Error=DBL_MAX;
  double learningSignal;

  Initialization();
  while(Error>desired_error){
    q++;
    Error=0;
    for(p=0; p<n_sample; p++){
      FindOutput(p);
      for(i=0;i<R;i++){
        Error+=0.5*pow(d[p][i]-o[i],2.0);
      }
      for(i=0;i<R;i++){
        learningSignal=eta*(d[p][i]-o[i]);
        for(j=0;j<N;j++){
          w[i][j]+=learningSignal*x[p][j];
        }
      }
    }
    //printf("Error in the %d-th learning cycle=%f\n",q,Error);
  }
  PrintResult();
}
```

Picture 11: main() in 1_2_case1.c

2.2.1.3. FindOutput()

In perceptron learning rule, *f(u)* is the follows:

$$f(u)\begin{cases} 1 & \text{if } u > 0 \\ -1 & \text{if } u < 0 \end{cases}$$

We let *u as* temp and *f(u)* as o[] in 1_2_case1.c.

```c
/*****************************************************************/
/* Find the actual outputs of the network                        */
/*****************************************************************/
void FindOutput(int p){
  int    i,j;
  double temp;

  for(i=0;i<R;i++){
    temp=0;
    for(j=0;j<N;j++){
      temp+=w[i][j]*x[p][j];
    }
    if(temp>0.0)o[i]=1.00;
    else o[i]=-1.00;
  }
}
```

Picture 12: FindOutput() in 1_2_case1.c

2.2.1.4. PrintResult() and PrintDesiredOutput()

We added printing neuron outputs for inputs in PrintResult() and new function PrintDesiredOutput() that print desired outputs for inputs. This program uses o as neuron outputs, so PrintDesiredOutput() compares if o is less than or greater than 0. If o is greater than 0, the function prints 1. In contrast, if o is less than 0, the function prints -1.

```
103 void PrintResult(void){
104   int i,j;
105 
106   printf("The connection weights are:\n\n");
107   for(i=0;i<R;i++){
108     for(j=0;j<N;j++)
109       printf("%5f ",w[i][j]);
110     printf("\n");
111   }
112   printf("\n\n");
113 
114   printf("Neuron outputs:\n\n");
115   for(i=0;i<n_sample;i++){
116     printf("Input: (");
117     for(j=0;j<R;j++){
118       if(j<R-1)printf("%f,",x[i][j]);
119       else printf("%f)   f(u): ",x[i][j]);
120     }
121 
122     FindOutput(i);
123 
124     printf("(");
125     for(j=0;j<R;j++){
126       if(j<R-1)printf("%f,",o[j]);
127       else printf("%f",o[j]);
128     }
129     printf(")   ");
130 
131     PrintDesiredOutput();
132   }
133 }
134 
135 
136 void PrintDesiredOutput(void){
137   printf("Thus, the desired output is ");
138   printf("(");
139     for(int j=0;j<R;j++){
140       if(o[j]>0)printf("1");
141       else printf("-1");
142       if(j<R-1)printf(",");
143     }
144     printf(")\n");
145 }
```

Picture 13: FindOutput() in 1_2_case1.c

2.2.2. Case 2 version: 1_2_case2.c

Our case 2 program is named 1_2_case2.c. We used delta learning rule to accomplish a single layer neural network with continuous neurons. Values of variables, PrintResult() and PrintDesiredOutput() are the same as 1_2_case2.c, so we omit the description. The program was uploaded to GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/1_2_case2.c We detail our program in the following chapters.

2.2.2.1. main()

His program used delta learning rule, so I used it in 1_2_case2.c as it was. This time, a solution don't need error values, thus we commented out its printf().

```
45  int main(){
46      int    i,j,p,q=0;
47      double Error=DBL_MAX;
48      double delta;
49
50      Initialization();
51      while(Error>desired_error){
52          q++;
53          Error=0;
54          for(p=0; p<n_sample; p++){
55              FindOutput(p);
56              for(i=0;i<R;i++){
57                  Error+=0.5*pow(d[p][i]-o[i],2.0);
58              }
59              for(i=0;i<R;i++){
60                  delta=(d[p][i]-o[i])*(1-o[i]*o[i])/2;
61                  for(j=0;j<N;j++){
62                      w[i][j]+=eta*delta*x[p][j];
63                  }
64              }
65          }
66          //printf("Error in the %d-th learning cycle=%f\n",q,Error);
67      }
68      PrintResult();
69  }
```

Picture 14: main() in 1_2_case2.c

2.2.2.2. FindOutput()

In delta learning rule, *f(u)* is the follows:

If we use the unipolar sigmoid function: $f(u) = \frac{1}{1+e^{-\lambda u}}$

If we use the bipolar sigmoid function: $f(u) = \frac{2}{1+e^{-\lambda u}} - 1$

Mr. Zhao's program used the bipolar sigmoid function, so we used the same as it. We let *u as* temp and *f(u)* as o in 1_2_case2.c.

```
23  #define sigmoid(x)     (2.0/(1.0+exp(-lambda*x))-1.0)
```

Picture 15: defined sigmoid(x) in 1_2_case2.c

```
86  void FindOutput(int p){
87      int    i,j;
88      double temp;
89
90      for(i=0;i<R;i++){
91          temp=0;
92          for(j=0;j<N;j++){
93              temp+=w[i][j]*x[p][j];
94          }
95          o[i]=sigmoid(temp);
96      }
97  }
```

Picture 16: FindOutput() in 1_2_case2.c

2.3. Results

We shows results of executing each program. Each result outputted to txt files using redirection.

2.3.1. Case 1 result

```
1   The connection weights are:
2
3   7.706179 7.037846 -0.484302
4   -2.541996 -11.658284 0.688381
5   -9.907859 -1.717578 0.753675
6
7
8   Neuron outputs:
9
10  Input: (10.000000,2.000000,-1.000000)   f(u): (1.000000,-1.000000,-1.000000)   Thus, the desired output is (1,-1,-1)
11  Input: (2.000000,-5.000000,-1.000000)   f(u): (-1.000000,1.000000,-1.000000)   Thus, the desired output is (-1,1,-1)
12  Input: (-5.000000,5.000000,-1.000000)   f(u): (-1.000000,-1.000000,1.000000)   Thus, the desired output is (-1,-1,1)
13
```

Picture 17: a result of executing 1_2_case1.c, f(u) is a neuron output.
GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/ans1_2_case1.txt

2.3.2. Case 2 result

```
1   The connection weights are:
2
3   1.955117 1.331439 0.162875
4   -1.276752 -2.669913 -0.009606
5   -2.564880 0.236389 0.261903
6
7
8   Neuron outputs:
9
10  Input: (10.000000,2.000000,-1.000000)   f(u): (1.000000,-1.000000,-1.000000)   Thus, the desired output is (1,-1,-1)
11  Input: (2.000000,-5.000000,-1.000000)   f(u): (-0.896661,0.999959,-0.997210)   Thus, the desired output is (-1,1,-1)
12  Input: (-5.000000,5.000000,-1.000000)   f(u): (-0.927561,-0.998096,0.999998)   Thus, the desired output is (-1,-1,1)
```

Picture 18: a result of executing 1_2_case2.c, f(u) is a neuron output.
GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/ans1_2_case2.txt

2.4. Discussion

In Part 2, we discussed the following points.

- What is the difference in output between discrete and continuous neuron?
- What can't be done with a single-layer neural network?

2.4.1. What is the difference in output between discrete and continuous neuron?

The difference in output is f(u) value because discrete neuron used the perceptron learning rule and continuous neuron used the delta learning rule. Case 1 using the perceptron learning rule discriminate o -1 or 1 in FindOutput(), so f(u) outputs -1 or 1 in ans1_2_case1.txt. Case 2 using the delta learning rule calculates **o** using bipolar sigmoid function in FindOutput(), so f(u) outputs numbers close to -1 or 1 in ans1_2_case2.txt.

2.4.2. What can't be done with a single-layer neural network?

A single-layer neural network cannot solve nonlinear issues. For example, this neural network can not solve a XOR gate. We wondered why it could not solve the XOR gate, so we unraveled them one at a time. First, if (x1, x2) is (1,0) or (0,1), XOR gate is true. A single-layer neural network delimit a graph using a single straight line, but XOR gate cannot be separated by a single straight line. Thus, a single gate neural network cannot solve nonlinear issues. The XOR gate can be solved by a multilayer neural network.

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

Table 1: a Truth table for XOR gate

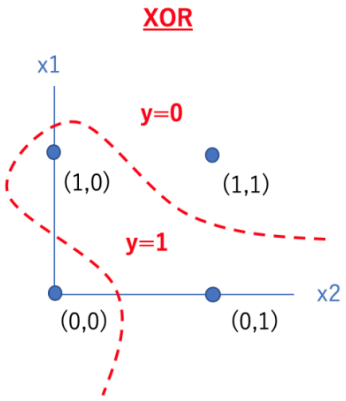

Figure 2: a graph of XOR gate [2]

3. Bonus project: NAND gate using ReLU

3.1. Our new problem

Write a computer program to realize ReLU. Train a neuron using your program to realize the NAND gate. The input pattern and their teacher signals are given as follows:

- Data: (0,0,-1); (0,1,-1); (1,0,-1); (1,1,-1)
- Teacher signals: 1, 1, 1, 0
- Program outputs: Weights of the neuron and Neuron output for each input pattern

3.2. Differences from Project 1

In Project 1, we used the perceptron learning rule and the delta learning rule, but there are various other activation functions in addition to these one. ReLU is one of the activation functions. ReLU gradient is either zero or constant, so it is valid for gradient loss or gradient explosion problems. In addition, ReLU has proven to work well in a variety of situations and is the highest standard at this time [3]. Therefore, we used ReLU to create a single-layer neural network.

3.3.Our solution method

We modified Mr. Zhao's C program and created a single layer neural network using ReLU. The program was uploaded to GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/1_3_ReLU.c
Details of this program are described in the following chapters.

3.3.1. Values of variables

We added a header "time.h" to use rand() and inputted data values and teacher signal values given the problem. We changed desired_error to 1 because ReLU is determined by 0 and a positive constant in this program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <time.h>

#define I             3 //the number of data (include dammy input -1)
#define n_sample      4 //the number of sample
#define eta           0.5 //learning rate
#define lambda        1.0
#define desired_error 1
#define sigmoid(x)    (2.0/(1.0+exp(-lambda*x))-1.0)
#define frand()       (rand()%10000/10001.0)
#define randomize()   srand((unsigned int)time(NULL))

double x[n_sample][I]={
  {0,0,-1},
  {0,1,-1},
  {1,0,-1},
  {1,1,-1}
};

double w[I];
double d[n_sample]={1,1,1,0};
double o;

void Initialization(void);
void FindOutput(int);
void PrintResult(void);
void PrintDesiredOutput(void);
double ReLU(double);
```

Picture 19: Values of variables in 1_3_ReLU.c

3.3.2. main()

Main() has the same contents as the perceptron learning rule in Part 1 because differentiation of f(u) yields 0 or 1. A detailed explanation for this is given in the discussion part. The error printouts were commented out.

```
44  int main(){
45    int    i,p,q=0;
46    double relu,Error=DBL_MAX;
47
48    Initialization();
49    while(Error>desired_error){
50      q++;
51      Error=0;
52      for(p=0; p<n_sample; p++){
53
54        FindOutput(p);
55        Error+=0.5*pow(d[p]-o,2.0);
56
57        relu=(d[p]-o);
58
59        for(i=0;i<I;i++){
60          w[i]+=eta*relu*x[p][i];
61        }
62        //printf("Error in thEe %d-th learning cycle=%f\n",q,Error);
63      }
64    }
65    PrintResult();
66  }
```

Picture 20: main() in 1_3_ReLU.c

3.3.3. FindOutput() and ReLU()

We added a functions to calculate ReLU: ReLU(). The neuron output with ReLU is expressed as follows.

$$f(u)\begin{cases} u & \text{if } u > 0 \\ 0 & \text{if } u < 0 \end{cases}$$

ReLU is a rectified linear, so a node is activated only if the input is above a certain value. When u exceeds the threshold, a linear relationship is created between the dependent variable and u. In FindOutput(), the neuron output **o** is calculated using ReLU().

```
77  /*************************************************************/
78  /* Find the actual outputs of the network                    */
79  /*************************************************************/
80  void FindOutput(int p){
81    int    i;
82    double temp=0;
83
84    for(i=0;i<I;i++) temp += w[i]*x[p][i];
85    o = ReLU(temp);
86  }
87
88  /*************************************************************/
89  /* Calcurate using ReLU function                             */
90  /*************************************************************/
91
92  double ReLU(double u){
93    if(u>0) return u;
94    else return 0;
95  }
96
```

Picture 21: FindOutput() and ReLU() in 1_3_ReLU.c

3.3.4. PrintResult() and PrintDesiredOutput()

PrintResult() and PrintDesiredOutput() are the same as 1_1_per.c and 1_1_delta.c.

```
/************************************************************/
/* Print out the final result                               */
/************************************************************/
void PrintResult(void){
  int i,j;
  double u;

  printf("\n\n");
  printf("The connection weights of the neurons:\n");
  for(i=0;i<I;i++) printf("%5f ",w[i]);
  printf("\n\n");

  for(i=0;i<n_sample;i++){
    u=0;
    printf("Input: (");
    for(j=0;j<I;j++){
      printf("%1.0f", x[i][j]);
      if(j!=I-1)printf(",");
      u+=x[i][j]*w[j];
    }
    printf(")   ");

    FindOutput(i);
    printf("f(u)=%f   ",o);

    PrintDesiredOutput();

  }

}

void PrintDesiredOutput(void){
  printf("The desired output: ");
  if(o>0) printf("1\n");
  else printf("0\n");
}
```

Picture 22: PrintResult() and PrintDesiredOutput() in 1_3_ReLU.c

3.4. Result

We shows a result of executing 1_3_ReLU.c. A result outputted to txt files using redirection.

```
1
2
3   The connection weights of the neurons:
4   -0.930945 -0.620097 -1.171139
5
6   Input: (0,0,-1)   f(u)=1.171139   The desired output: 1
7   Input: (0,1,-1)   f(u)=0.551042   The desired output: 1
8   Input: (1,0,-1)   f(u)=0.240194   The desired output: 1
9   Input: (1,1,-1)   f(u)=0.000000   The desired output: 0
10
```

Picture 23: a result of executing 1_3_ReLU.c, f(u) is a neuron output.
GitHub: https://github.com/K-Fujita-onkyo/neural_network/blob/main/Project1/ans1_3_ReLU.txt

3.5. Discussion

In Part 2, we discussed the following points.

- In 3.3.2. main(), why was main() the same content as the Perceptron learning rule?
- What are the advantages and disadvantages of ReLU?

3.5.1. In 3.3.2. main(), why was main() the same content as the Perceptron learning rule?

The reason is that differentiating f(u) leads to 0 or a constant. Once again f(u) is shown below.

$f(u)\begin{cases} u & \text{if } u > 0 \\ 0 & \text{if } u < 0 \end{cases}$ Differentiating f(u), we obtain $f'(u)\begin{cases} 1 & \text{if } u > 0 \\ 0 & \text{if } u < 0 \end{cases}$

For Lecture 2-17: $W^{k+1} = W^k + \eta(d - f(u))f'(u)x$

At this time, If u > 0, $f'(u) = 1$. Therefore, W^{k+1} is the following as: $W^{k+1} = W^k + \eta(d - f(u))x$

Also, If $u < 0$, The derivative is 0, so the parameter W^{k+1} is not updated. Thus, main() is the same as the Perceptron learning rule in 1_3_ReLU.c.

3.5.2. What are the advantages and disadvantages of ReLU?

ReLU have 2 advantages. First, since the derivative is constant, the amount of computation can be reduced overwhelmingly, and the error back propagation is computationally efficient. When u is 0, f(u) is non-differentiable, but differentiable over the rest of the range. Furthermore, differentiating f(u) results in a constant, so the computation is faster. Second, the constant derivative value prevents the problem of gradient disappearance due to error back propagation. In the case of sigmoid, the gradient disappears when the input is extremely large or small. However, as mentioned earlier, differentiating ReLU results in a constant. So ReLU can prevent the problem of gradient loss due to error back propagation.

ReLU have 2 disadvantages. First, ReLU should only be used within the hidden layer of the neural network model. The output layer should use a Softmax function to compute class probabilities for the Classification problem and simply use a linear function for the Regression problem. Second, the gradient is fragile and can die during training. It can cause a weight update that prevents it from being activated again at any data point, which is just to say that ReLu brings about Dead Neurons.

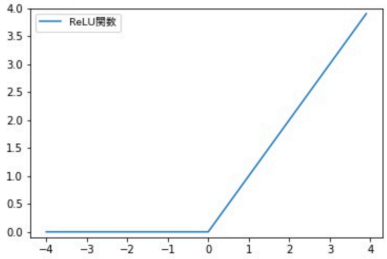

Figure 3: ReLU function

To solve this problem, another fix, Leaky ReLu, was introduced to solve the Dead Neurons issue. A small slope has been introduced to keep the update alive.

4. Conclusion

In Project 1, we built 2 types of a single neural network using perceptron learning rule and delta learning rule.

Perceptron learning rule is a discrete function. This function is expressed as follows:

$$f(u)\begin{cases}1 & \text{if } u > 0\\ -1 & \text{if } u < 0\end{cases}$$

If u is larger than 0, f(u) outputs 1. If u is smaller than 0, f(u) outputs -1.

Delta learning rule is a continuous function. This function is expressed as follows:

If we use the unipolar sigmoid function: $f(u) = \dfrac{1}{1+e^{-\lambda u}}$

If we use the bipolar sigmoid function: $f(u) = \dfrac{2}{1+e^{-\lambda u}} - 1$

The Sigmoid function is continuous with respect to the input and can be differentiated. This ability to differentiate is important for neural networks that involve learning by error back propagation. After solving Project 1, we realized the importance of the gradient method. The Perceptron learning rule is very fast because it determines whether it is a 1 or a -1. Delta learning rule is very accurate because the weights are modified sequentially using the gradient method.

In our bonus issue, we built a single neural network using ReLU. The neuron output with ReLU is expressed as follows.

$$f(u)\begin{cases}u & \text{if } u > 0\\ 0 & \text{if } u < 0\end{cases}$$

ReLU is usually used in the hidden layer, but we used output layer this time. ReLU is very fast because it is constant when differentiated. However, it is non-differentiable at u=0, so that must be taken into account. Also, there is a phenomenon called dying ReLU, where the gradient is broken in the middle of the process, so we have to take this into account.

References

[1] Zhang, Lei. "Implementation of fixed-point neuron models with threshold, ramp and sigmoid activation functions." *IOP Conference Series: Materials Science and Engineering*. Vol. 224. No. 1. IOP Publishing, 2017.

[2] Udemy media. “ニューラルネットワークとは？人工知能の基本を初心者向けに解説！” https://udemy.benesse.co.jp/data-science/ai/neural-network.html, 2021.

[3] Agarap, Abien Fred. "Deep learning using rectified linear units (relu)." *arXiv preprint arXiv:1803.08375* (2018).