Multi-Layer Feed Forward Neural Network with Back Propagation Training Algorithm

Coding Assignment Report

CHETAN SHANTARAM NALAVADE

Roll No: 214103009

Department of Mechanical Engineering,

Indian Institute of Technology Guwahati,

Assam, India-781039

1.Introduction

- In this assignment, the application of artificial neural networks (ANNs) for predicting the oblique wave shock angle with respect to horizontal by providing given upstream Mach no. (M) and corner angle (θ).
- An oblique shock wave is a shock wave that, unlike a normal shock, is inclined with respect to the incident upstream flow direction. It will occur when a supersonic flow encounters a corner that effectively turns the flow into itself and compresses.
- The upstream streamlines are uniformly deflected after the shock wave. The most common way to produce an oblique shock wave is to place a wedge into supersonic, compressible flow. Similar to a normal shock wave, the oblique shock wave consists of a very thin region across which nearly discontinuous changes in the thermodynamic properties of a gas occur.
- The upstream and downstream flow directions are unchanged across a normal shock, they are different for flow across an oblique shock wave.
- For a given Mach number, M1, and corner angle, θ, the oblique shock angle, β, and the downstream Mach number, M2, can be calculated.
- Unlike after a normal shock where M2 must always be less than 1, in oblique shock M2 can be supersonic (weak shock wave) or subsonic (strong shock wave).
- Weak solutions are often observed in flow geometries open to atmosphere (such as on the outside of a flight vehicle). Strong solutions may be observed in confined geometries (such as inside a nozzle intake)
- For our ANN model we are using 2 parameters – upstream Mach number and corner angle to find the output – oblique shock angle(β)
- Values for upstream Mach no. (M) range from 1.5 to 20.5, for corner angle (θ) from 5 to 45 deg and for oblique shock angle(β) from 7.12 to 66.67 deg.
- ANN model is trained with over 150 data-points out of the total of 200 data-points, (75% of the total number) and the validation and testing of the trained ANN were performed with the remaining 50 data-points.

2.The θ-β-M equation

Using the continuity equation and the fact that the tangential velocity component does not change across the shock, trigonometric relations eventually lead to the θ-β-M equation which shows θ as a function of M1 β, and ɣ, where ɣ is the Heat capacity ratio.

$$\tan\theta = 2\cot\beta\frac{M_1^2\sin^2\beta - 1}{M_1^2(\gamma + \cos 2\beta) + 2}$$

It is more intuitive to want to solve for β as a function of M1 and θ, but this approach is more complicated, the results of which are often contained in tables or calculated through a numerical method.

3.Data

Initial data is generated using the θ-β-M equation.

Inputs and output used in the ANN are:

Variable Representation	Variable Name
X1	Upstream Mach no. (M)
X2	Corner angle (θ)
Y1	Oblique shock angle(β)

4.Methodology

For the above parameters, a fully connected feed forward back propagating 3 layer ANN model was chosen to predict the two outputs based on the eight inputs. The key features of the model are:

- 150 patterns were taken as the training data while the remaining 50 patterns were used to verify the accuracy of the trained model.
- Transfer functions in the hidden layer were taken as tan-sigmoid with constant as 1

$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

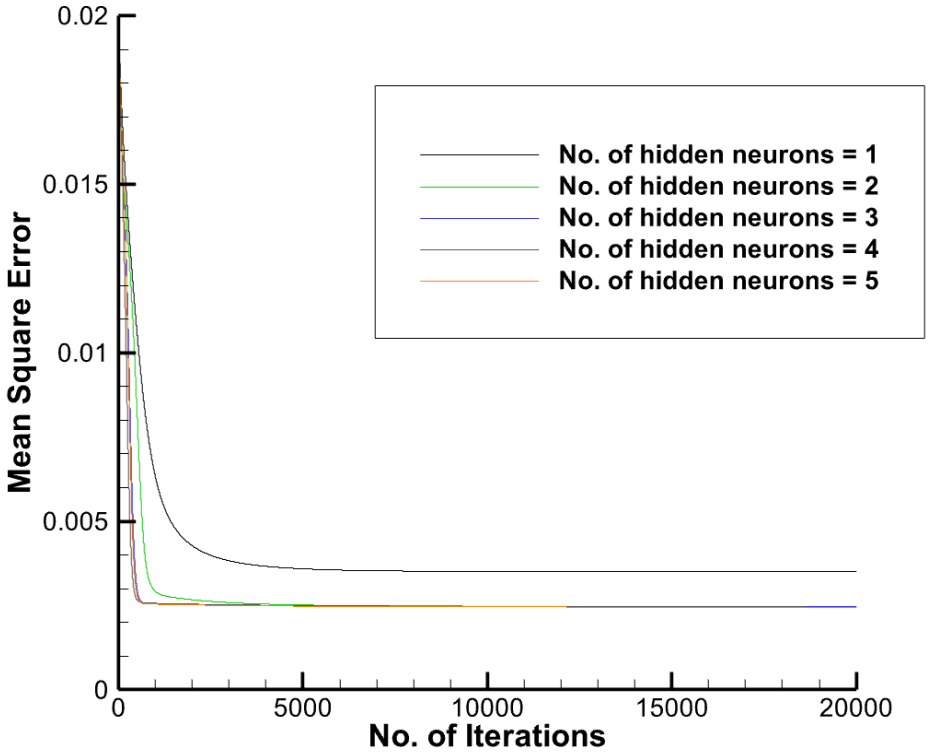
- Transfer functions in the output layer were taken as log-sigmoid with constant as 1

$$f(x) = \frac{1}{1 + e^{-x}}$$

- The input data was normalized between the limits 0.1 to 0.9 in order to accommodate the output within range.
- The initial weight matrices were taken as zero for both input-hidden (Vij) and hidden-output (Wjk) for constant starting point to facilitate ease in comparison during the trial-and-error analysis method.

- The end condition of training was set to be mean square error less than 0.001 or maximum number of iterations less than 1,10,000 as the error curve flattens after reaching a certain point.
- The final number of neurons in the hidden layer was taken as 4 from experimental trial and error analysis method by comparing in the graphs mentioned below for different no. of hidden neurons.

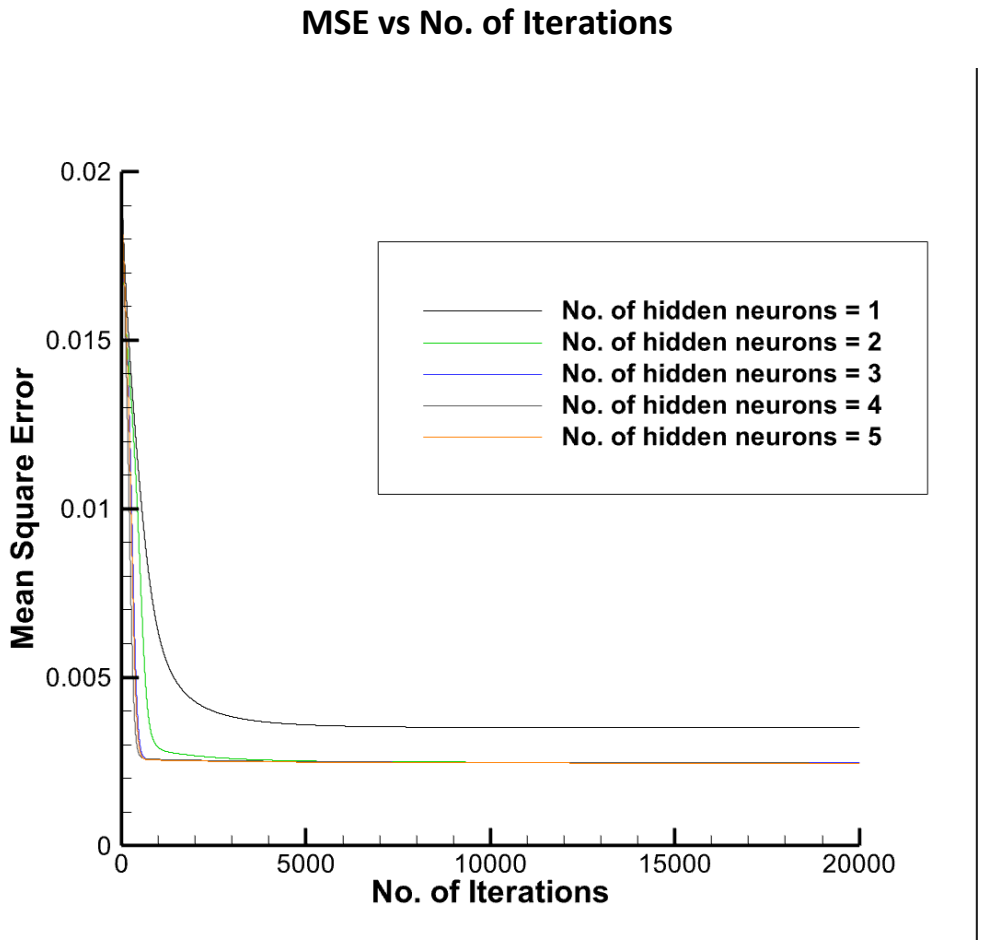
- **MSE vs No. of Iterations**



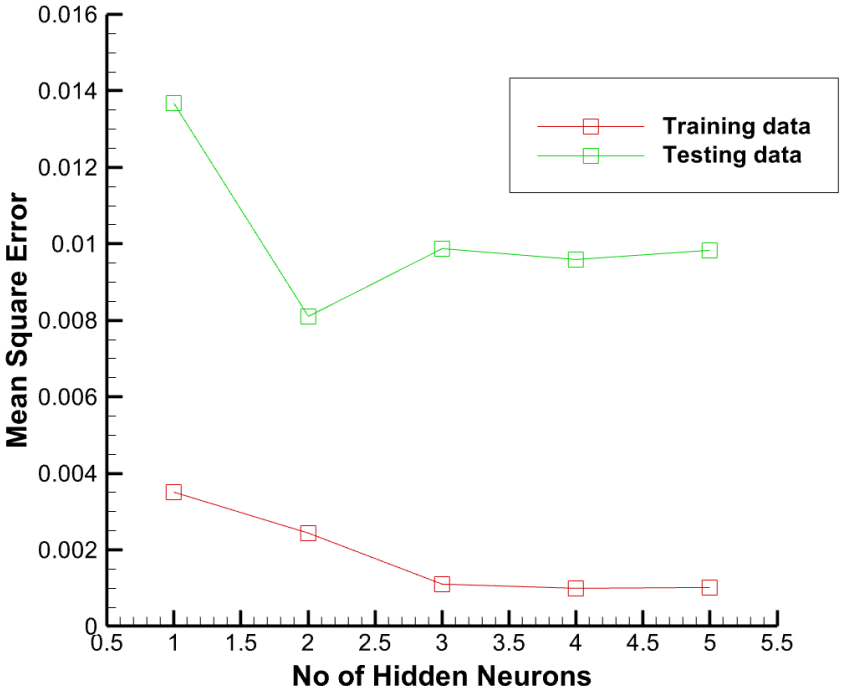
- Final values of learning rate is taken as 0.9 from experimental trial and error analysis method by comparing in the graphs mentioned below for different learning rate value.
- De-normalization of output values is done at the end to compare with the target output.

5.Results

Using the experimental trial and error analysis, the optimum number of hidden neurons and learning rate was found taking the lowest mean square error as the principle criteria.

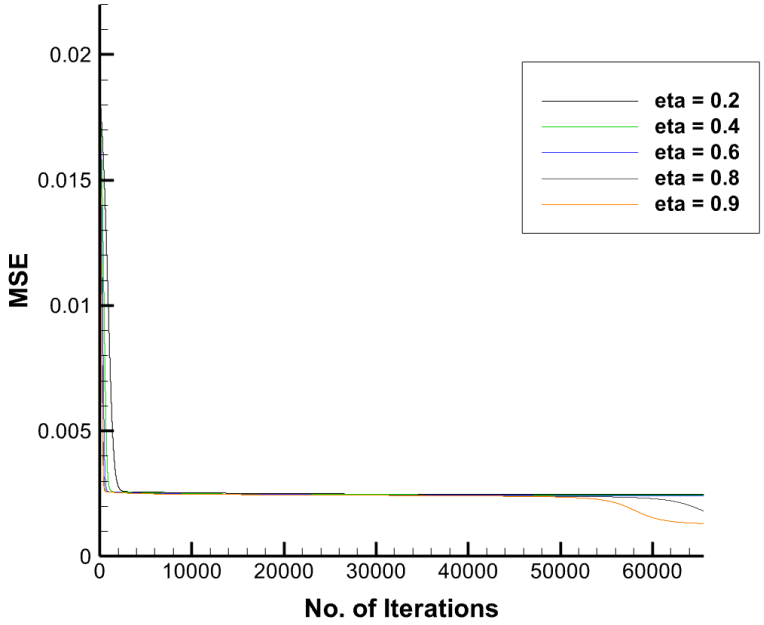


MSE vs No. of hidden neurons

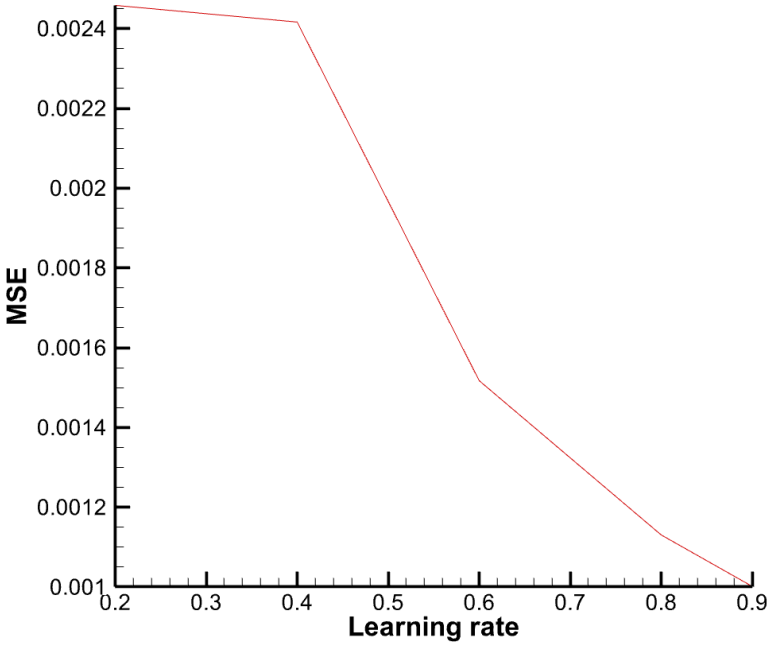


- As the number of hidden neurons increases, the final mean square error decreases.
- Increase in the number of hidden neurons also smoothens the curve and there are fewer oscillations for the same values of learning rate.
- As proper fitting of both testing and training is obtained at 4 hidden neurons.
- Optimum No. of hidden neurons are 4

MSE vs No. of Iterations



MSE vs learning rate

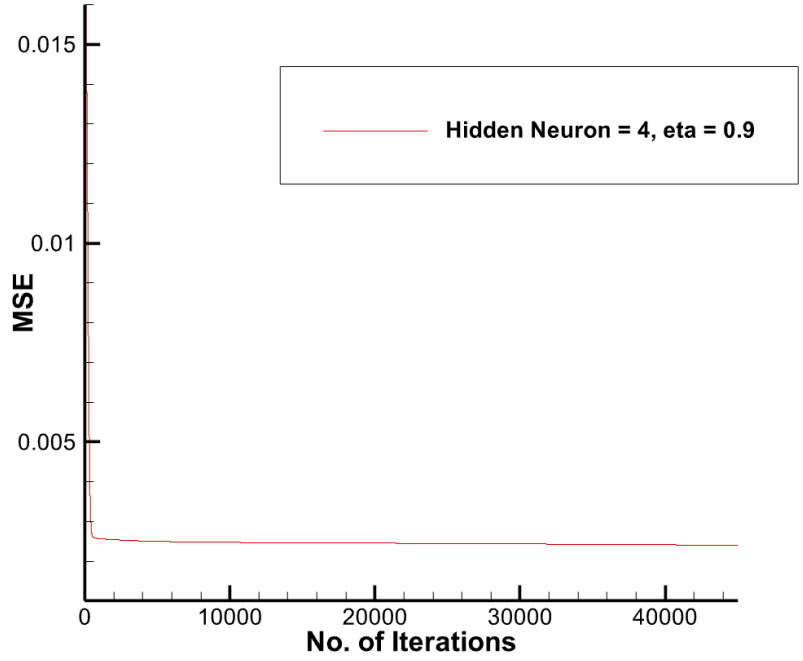


- As the value of learning rate (eta) increases, the minima is reached faster i.e. fewer iterations are required.
- The mean square error flattens at nearly 0.0025 with negligible change as iterations progress after for eta=0.9 after 50000 iterations value drops to minimum value.
- Optimum eta was taken as 0.9 as it gives minimum mean square error.

Final Optimized ANN gives following results:

Parameter	Value
Input Neurons	2
Hidden Neurons	4
Output Neurons	1
Learning rate	0.9
Mean Square error (Training)	0.001
Mean Square error (Testing)	0.009594
Number of iterations	90633

MSE vs No. of Iterations



6.Output of Network

Testing Pattern No.	Output of Network - oblique shock wave angle	Actual output - oblique shock wave angle
1	20.861647	8.86
2	16.480795	8.68
3	12.764171	8.51
4	10.067827	8.36
5	8.319168	8.23
6	7.282415	8.1
7	6.725706	7.99
8	6.473462	7.88
9	6.406542	7.78
10	6.448228	7.69
11	23.906391	13.91
12	20.404091	13.77
13	17.389183	13.65
14	15.166511	13.54
15	13.70375	13.44
16	12.824152	13.35
17	12.342175	13.26
18	12.112691	13.19
19	12.03547	13.12
20	12.045464	13.06
21	23.705532	19.41
22	21.380405	19.32
23	19.653004	19.23
24	18.506166	19.15
25	17.809025	19.09
26	17.41951	19.02
27	17.224531	18.97
28	17.145201	18.91
29	17.130167	18.87
30	17.147234	18.82
31	26.364399	25.37
32	24.644258	25.29
33	23.366379	25.21
34	22.515394	25.15
35	21.994328	25.09

Testing Pattern No.	Output of Network - oblique shock wave angle	Actual output - oblique shock wave angle
36	21.697945	25.04
37	21.542233	24.99
38	21.468508	24.95
39	21.438551	24.91
40	21.428501	24.88
41	27.199278	31.51
42	26.28949	31.45
43	25.684792	31.39
44	25.313097	31.34
45	25.098337	31.29
46	24.980318	31.25
47	24.917198	31.21
48	24.881824	31.17
49	24.857283	31.14
50	24.833309	31.11

7.Conclusion

A fully connected feed forward back propagating 3 layer ANN model is used to predict oblique shock wave angle using 2 input parameters. The ANN has been trained using 150 training patterns and 50 testing patterns with 4 hidden layer neurons. The learning rate is set to 0.9.

Training provides a mean square error of 0.001 and this trained ANN can be used to predict oblique shock wave angle for given inputs with a mean square error of 0.009594.

Code Section:

```c
// Defining library
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>


int main()
{
    float MSE;
    FILE *in;
    FILE *out;
    FILE *gra;
    in = fopen("input.txt","r");
    out = fopen("output.txt","w");
    gra = fopen("Graph2.txt","w");


    // Defining the variables
    int L,M,N,P,TP;
    float eta, alp,numt;
    int p,count=0;

    // Taking values from input file
    fscanf(in,"%d",&L);
    fscanf(in,"%d",&M);
    fscanf(in,"%d",&N);
    fscanf(in,"%d",&P);
    fscanf(in,"%f",&eta);
    fscanf(in,"%d",&TP);


    //Defining Input and Weight matrices
    float I[L+1][P+1],V[L+1][M+1],W[M+1][N+1];
    //Defining the target outputs for each pattern
    float TO[N+1][P+1];
    float NOT[N][P];
    //Taking values for input matrix from input text file
    int i,j,k;
    for(i=1;i<=P;i++)
    {
        for(j=0;j<=L;j++)
        {
            if(j==0)
            {
                I[j][i] = 1;
            }
            else
            {
                fscanf(in,"%f",&I[j][i]);
            }
        }
    }

    /*for(i=1;i<=P;i++)
```

```
    {
        for(j=0;j<=L;j++)
        {
            printf("%f\t",I[j][i]);
        }
        printf("\n");
    }*/

    //Taking random values for V and W weight matrices
    double randf;
    //for matrix V
    for(i=0;i<=L;i++)
    {
        for(j=1;j<=M;j++)
        {
            if(i==0)
            {
                V[i][j] = 0;
            }
            else
            {
                numt = rand()%10;
                V[i][j] = numt/10;
            }
        }
    }

    //now for matrix W
        for(i=0;i<=M;i++)
    {
        for(j=1;j<=N;j++)
        {
            if(i==0)
            {
                W[i][j] = 0;
            }
            else
            {
                numt = rand()%10;
                W[i][j] = numt/10;
            }
        }
    }

    //Taking target output matrix from text file
    for(p=1;p<=P;p++)
    {
        for(k=1;k<=N;k++)
        {
            //printf("TO[%d][%d] = ",k,p);
            fscanf(in,"%f",&TO[k][p]);
        }
    }

    //Normalisation of Input and output data
```

```
111:     //first finding max and min values
112:     float Max[L+1],Min[L+1];
113:     for(i=1;i<=L;i++)
114:     {
115:         Max[i] = I[i][1];
116:         Min[i] = I[i][1];
117:         for(p=1;p<=P;p++)
118:         {
119:             if(I[i][p]>Max[i])
120:             {
121:                 Max[i] = I[i][p];
122:             }
123:             if(I[i][p]<Min[i])
124:             {
125:                 Min[i] = I[i][p];
126:             }
127:         }
128:     }
129:
130:     for(i=1;i<=L;i++)
131:     {
132:         for(p=1;p<=P;p++)
133:         {
134:             I[i][p] = 0.1 + (0.8*((I[i][p] - Min[i])/(Max[i] - Min[i])));
135:         }
136:     }
137:
138:     float MaxT[N+1],MinT[N+1];
139:     for(k=1;k<=N;k++)
140:     {
141:         MaxT[k] = TO[k][1];
142:         MinT[k] = TO[k][1];
143:         for(p=1;p<=P;p++)
144:         {
145:             if(TO[k][p]>MaxT[k])
146:             {
147:                 MaxT[k] = TO[k][p];
148:             }
149:             if(TO[k][p]<MinT[k])
150:             {
151:                 MinT[k] = TO[k][p];
152:             }
153:         }
154:     }
155:
156:
157:     for(k=1;k<=N;k++)
158:     {
159:         for(p=1;p<=P;p++)
160:         {
161:             TO[k][p] = 0.1 + (0.8*((TO[k][p] - MinT[k])/(MaxT[k] - MinT[k])));
162:         }
163:     }
164:
165:     //defining the IH matrix(input to the hidden neuron)
```

```
166:     float IH[M+1][P+1],OH[M+1][P+1];
167:
168:     //defining the input and output matrix for output layer
169:     float IO[N+1][P+1],OO[N+1][P+1];
170:
171:     // defining the delta values
172:     float delW[M+1][N+1],debydw[M+1][N+1];
173:
174:     // defining required values
175:     float delV[L+1][M+1],dedv1[L+1][M+1],dedv2[L+1][M+1];
176:
177:     //Using loop
178:     do
179:     {
180:         // Now starting forward pass calculations
181:         //calculation for input to hidden neuron for each pattern
182:         for(p=1;p<=P;p++)
183:         {
184:             for(j=1;j<=M;j++)
185:             {
186:                 IH[j][p] = 0;
187:                 for(i=0;i<=L;i++)
188:                 {
189:
190:                     IH[j][p] = IH[j][p] + (I[i][p]*V[i][j]);
191:                 }
192:
193:                 // calculating output of hidden neuron
194:                 float temp = 0;
195:                 temp = exp(-IH[j][p]);
196:                 OH[j][p] = (1/(1+temp));
197:             }
198:         }
199:
200:         //bias for hidden neuron defined as 1
201:         for(p=1;p<=P;p++)
202:         {
203:             OH[0][p] = 1;
204:         }
205:
206:         //now solving for output layer
207:         //calculation for input to output neuron for each pattern
208:         for(p=1;p<=P;p++)
209:         {
210:             for(k=1;k<=N;k++)
211:             {
212:                 IO[k][p] = 0;
213:                 for(j=0;j<=M;j++)
214:                 {
215:                     IO[k][p] = IO[k][p] + (OH[j][p]*W[j][k]);
216:                 }
217:
218:                 // calculating output of output neuron
219:                 float t1 = 0, t2 = 0;
220:                 //temp = exp((-1)*(IH[j][p]));
```

```
221:                 t1 = exp(IO[k][p]);
222:                 t2 = exp((-1)*(IO[k][p]));
223:                 OO[k][p] = ((t1 - t2)/(t1 + t2));
224:             }
225:         }
226:
227:         for(p=1;p<=P;p++)
228:         {
229:             for(k=1;k<=N;k++)
230:             {
231:                 //printf("%f\n",OO[k][p]);
232:             }
233:         }
234:
235:         //Calculating the mean square error for given training patterns
236:         //printf("\n Mean sqaure error MSE is as follows\n");
237:         float sum = 0;
238:         MSE = 0;
239:         for(p=1;p<=P;p++)
240:         {
241:             for(k=1;k<=N;k++)
242:             {
243:                 sum = sum + ((TO[k][p]-OO[k][p])*(TO[k][p]-OO[k][p])*0.5);
244:             }
245:         }
246:
247:         //printf("%f",sum);
248:
249:         MSE = sum/P;
250:         printf("MSE = %f\n",MSE);
251:         fprintf(gra,"%f\n",MSE);
252:         //fprintf(out," The mean sqaure errpor is = %f\n",MSE);
253:         //fprintf(out,"%f\n",MSE);
254:
255:         //updating the "W" weight values
256:
257:         for(j=0;j<=M;j++)
258:         {
259:             for(k=1;k<=N;k++)
260:             {
261:                 debydw[j][k] = 0;
262:                 for(p=1;p<=P;p++)
263:                 {
264:                     float dd1,dd2,dd2t1,dd2t2,dd2t3,dd3;
265:                     dd1 = -(TO[k][p]-OO[k][p]);
266:                     dd2t1 = exp(IO[k][p]);
267:                     dd2t2 = exp(-IO[k][p]);
268:                     dd2t3 = dd2t1+dd2t2;
269:                     dd2 = 4/(dd2t3*dd2t3);
270:                     dd3 = OH[j][p];
271:                     debydw[j][k] = debydw[j][k] + (dd1*dd2*dd3);
272:                 }
273:                 //calculating delta w
274:                 //printf("\ndebydw for %d is %f\n",k,debydw[j][k]);
275:                 delW[j][k] = ((-eta)/P)*debydw[j][k];
```

```
276:            }
277:        }
278:        //now printing the delta w values
279:        /*printf("\nbelow are the delta w values\n");
280:        for(j=1;j<=M;j++)
281:        {
282:            for(k=1;k<=N;k++)
283:            {
284:                printf("delW[%d][%d] = %f\n",j,k,delW[j][k]);
285:            }
286:        }*/
287:
288:        //now updating V matrix values by taking avg error from the output layer
289:
290:        for(i=0;i<=L;i++)
291:        {
292:            for(j=1;j<=M;j++)
293:            {
294:                dedv1[i][j] = 0;
295:                //dedv2[i][j] = 0;
296:                for(p=1;p<=P;p++)
297:                {
298:                    for(k=1;k<=N;k++)
299:                    {
300:                        float dv1,dv2,dv3,dv4,dv5;
301:                        dv1 = -(TO[k][p]-OO[k][p]);
302:                        dv2 = 4/((exp(IO[k][p])+exp(-IO[k][p]))*(exp(IO[k][p])+exp(-IO[k][p])
303:                        dv3 = W[j][k];
304:                        //dv4 = OH[j][p]*(1-OH[j][p]);
305:                        dv4 = (exp(-IH[j][p]))/((1+exp(-IH[j][p]))*(1+exp(-IH[j][p])));
306:                        dv5 = I[i][p];
307:                        dedv1[i][j] = dedv1[i][j] + (dv1*dv2*dv3*dv4*dv5);
308:                        //printf("dedv1 value = %f\n",dedv1[i][j]);
309:                    }
310:                    //dedv2[i][j] = dedv2[i][j] + dedv1[i][j];
311:                    //printf("dedv2 value = %f\n",dedv2[i][j]);
312:                }
313:                //printf("dedv1 value = %f\n",dedv1[i][j]);
314:                delV[i][j] = (-eta)*(1/((float)N*(float)P))*(dedv1[i][j]);
315:                //printf("delVvalue = %f\n",delV[i][j]);
316:            }
317:        }
318:        //printf("%f",delV[1][1]);
319:        //now printing delV values
320:        //printf("\nbelow are the delta V values\n");
321:        for(i=1;i<=L;i++)
322:        {
323:            for(j=1;j<=M;j++)
324:            {
325:                //printf("delV[%d][%d] = %f\n",i,j,delV[i][j]);
326:            }
327:        }
328:
329:        //now updating and printing V and W values
330:        //printing W values
```

```
331:         //printf("\nFollowing are the updated W values\n");
332:         for(j=1;j<=M;j++)
333:         {
334:             for(k=1;k<=N;k++)
335:             {
336:                 W[j][k] = W[j][k] + delW[j][k];
337:                 //printf("W[%d][%d] = %f\n",j,k,W[j][k]);
338:             }
339:         }
340:
341:         //printing V values
342:         //printf("\nFollowing are the updated V values\n");
343:         for(i=1;i<=L;i++)
344:         {
345:             for(j=1;j<=M;j++)
346:             {
347:                 V[i][j] = V[i][j] + delV[i][j];
348:                 //printf("V[%d][%d] = %f\n",i,j,V[i][j]);
349:             }
350:         }
351:
352:         //fprintf(out,"%f\n",V[1][1]);
353:         count++;
354:         //fprintf(gra,"%d\n",count);
355:
356:     }while(MSE>0.001&&count<110000);
357:     fprintf(out,"MSE = %f\n",MSE);
358:     fprintf(out,"No of Iteration = %d",count);
359:
360:     //now getting output for testing patterns
361:     //inputs from testing pattern
362:     for(i=1;i<=TP;i++)
363:     {
364:         for(j=0;j<=L;j++)
365:         {
366:             if(j==0)
367:             {
368:                 I[j][i] = 1;
369:             }
370:             else
371:             {
372:                 fscanf(in,"%f",&I[j][i]);
373:             }
374:         }
375:     }
376:
377:     //Taking target output matrix from text file of testing pattern
378:     for(p=1;p<=TP;p++)
379:     {
380:         for(k=1;k<=N;k++)
381:         {
382:             fscanf(in,"%f",&TO[k][p]);
383:         }
384:     }
385:
```

```
386:     //Normalisation of Input and output data of testing pattern
387:     //first finding max and min values
388:     float MaxTP[L+1],MinTP[L+1];
389:     for(i=1;i<=L;i++)
390:     {
391:         MaxTP[i] = I[i][1];
392:         MinTP[i] = I[i][1];
393:         for(p=1;p<=TP;p++)
394:         {
395:             if(I[i][p]>MaxTP[i])
396:             {
397:                 MaxTP[i] = I[i][p];
398:             }
399:             if(I[i][p]<MinTP[i])
400:             {
401:                 MinTP[i] = I[i][p];
402:             }
403:         }
404:     }
405:
406:     for(i=1;i<=L;i++)
407:     {
408:         for(p=1;p<=TP;p++)
409:         {
410:             I[i][p] = 0.1 + (0.8*((I[i][p] - MinTP[i])/(MaxTP[i] - MinTP[i])));
411:         }
412:     }
413:
414:     float MaxTTP[N+1],MinTTP[N+1];
415:     for(k=1;k<=N;k++)
416:     {
417:         MaxTTP[k] = TO[k][1];
418:         MinTTP[k] = TO[k][1];
419:         for(p=1;p<=TP;p++)
420:         {
421:             if(TO[k][p]>MaxTTP[k])
422:             {
423:                 MaxTTP[k] = TO[k][p];
424:             }
425:             if(TO[k][p]<MinTTP[k])
426:             {
427:                 MinTTP[k] = TO[k][p];
428:             }
429:         }
430:     }
431:
432:
433:     for(k=1;k<=N;k++)
434:     {
435:         for(p=1;p<=TP;p++)
436:         {
437:             TO[k][p] = 0.1 + (0.8*((TO[k][p] - MinTTP[k])/(MaxTTP[k] - MinTTP[k])));
438:         }
439:     }
440:
```

```
441:         // Now starting forward pass calculations for testing pattern
442:         //calculation for input to hidden neuron for each pattern
443:         for(p=1;p<=TP;p++)
444:         {
445:             for(j=1;j<=M;j++)
446:             {
447:                 IH[j][p] = 0;
448:                 for(i=0;i<=L;i++)
449:                 {
450:
451:                     IH[j][p] = IH[j][p] + (I[i][p]*V[i][j]);
452:                 }
453:
454:                 // calculating output of hidden neuron
455:                 float temp = 0;
456:                 temp = exp(-IH[j][p]);
457:                 OH[j][p] = (1/(1+temp));
458:             }
459:         }
460:
461:         //bias for hidden neuron defined as 1
462:         for(p=1;p<=TP;p++)
463:         {
464:             OH[0][p] = 1;
465:         }
466:
467:         //now solving for output layer
468:         //calculation for input to output neuron for each pattern
469:         for(p=1;p<=TP;p++)
470:         {
471:             for(k=1;k<=N;k++)
472:             {
473:                 IO[k][p] = 0;
474:                 for(j=0;j<=M;j++)
475:                 {
476:                     IO[k][p] = IO[k][p] + (OH[j][p]*W[j][k]);
477:                 }
478:
479:                 // calculating output of output neuron
480:                 float t1 = 0, t2 = 0;
481:                 //temp = exp((-1)*(IH[j][p]));
482:                 t1 = exp(IO[k][p]);
483:                 t2 = exp((-1)*(IO[k][p]));
484:                 OO[k][p] = ((t1 - t2)/(t1 + t2));
485:             }
486:         }
487:
488:         //Output of training pattern
489:         for(p=1;p<=TP;p++)
490:         {
491:             for(k=1;k<=N;k++)
492:             {
493:                 printf("%f\n",OO[k][p]);
494:             }
495:         }
```

```
496:
497:
498:         //Calculating the mean square error for given testing patterns
499:         //printf("\n Mean sqaure error MSE is as follows\n");
500:         float sumT = 0,MSET;
501:         MSET = 0;
502:         for(p=1;p<=TP;p++)
503:         {
504:             for(k=1;k<=N;k++)
505:             {
506:                 sumT = sumT + ((TO[k][p]-OO[k][p])*(TO[k][p]-OO[k][p])*0.5);
507:             }
508:         }
509:
510:         MSET = sumT/TP;
511:         fprintf(out," \nThe mean sqaure error of testing pattern is = %f\n",MSET);
512:
513:     //Denormalisation of output
514:     fprintf(out,"\nDenormalised output is as follows\n");
515:     for(p=1;p<=TP;p++)
516:     {
517:         for(k=1;k<=N;k++)
518:         {
519:             NOT[k][p] = (((OO[k][p] - 0.1) * (MaxTTP[k] - MinTTP[k])) / 0.8) + MinTTP[k];
520:             fprintf(out,"%f\n",NOT[k][p]);
521:         }
522:     }
523:
524:
525:
526:
527:     return 0;
528:
529: }
530:
```