Experiments with the Back-Propagation Algorithm

Berg, Andreas

1992

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280–5316
ISRN LUTFD2/TFRT--7489--SE

Experiments with the Back-Propagation Algorithm

Andreas Berg

Department of Automatic Control
Lund Institute of Technology
April 1992

Experiments with the Back-Propagation Algorithm

Andreas Berg

March 20, 1992

Experiments with the Back-Propagation Algorithm

Andreas Berg

March 20, 1992

1 Introduction

This is a report for a project in the Neural Network course. A program for training of a neural network with the back-propagation algorithm was developed (see Appendix A). The network is meant to have binary inputs and targets, and the weights are initialized to randomized values.

Experiments have been carried out with, for instance, the following problems:

- Inverter:
  A 1-1-1 network (one neuron in the input layer, one in the hidden layer, and one in the output layer) was trained to invert its output. The 2 possible patterns were used in the training and the output of the network were within 0.1 from the targets after 359 cycles of training.

- XOR:
  A 2-2-1 network was trained to implement the XOR function. The 4 possible patterns were used in the training and the output of the network were within 0.1 from the targets after 683 cycles of training.

- Symmetry:
  A 5-2-1 network was trained to detect symmetrical inputs (e.g. 10001). 26 patterns were used in the training and the output of the network were within 0.1 from the targets after 321 cycles of training. The remaining 6 patterns were used for test purposes and the output of the network were within 0.2 from the expected value.

- Digits:
  A 15-4-2 network was trained to detect images of three digits (1, 2, and 3). 12 patterns were used in the training (4 slightly different images of each digit) and the output of the network were within 0.1 from the targets after 50 cycles of training. The following test was done with images of 1, 2, and 3, that were similar to the training patterns. In some, but not all, cases the network was able to give a correct answer. An answer was defined as correct when the outputs were within 0.2 from the expected values.

2 Manual

2.1 Setting up the network

The program must be called with a set-up file that contains the following parameters:

- *seed* = the seed for the randomize generator ($seed > 0$)
- η = the learning rate for the back-propagation algorithm ($0 < \eta \geq 1$)
- ι = the accepted error after training ($0 < \iota < 1$)
- $nodes[i]$ = the number of neurons in layer i (1 ... MAX_NODES), where $i = 1$ ... LAYERS.
- *patterns* = the number of patterns (1 ... MAX_PATTERNS)

Furthermore, the patterns, together with the corresponding target outputs, must also be specified in the set-up file:

$$x_1 \ldots x_{nodes[1]} \qquad out_1 \ldots out_{nodes[n]}, \quad \text{(pattern 1)}$$
$$\ldots$$
$$x_1 \ldots x_{nodes[1]} \qquad out_1 \ldots out_{nodes[n]}, \quad \text{(pattern } n\text{)}$$

Please note that only the values, in the order stated above, should be given in the set-up file. However, after the values can anything be written.

2.1.1 Example of a set-up file

The set-up file for a three-layer xor-network:

```
387 1.0 0.1
2 2 1
4
0 0   0
1 0   1
0 1   1
1 1   0
```

2.2 Output of the program

When the simulation is complete, a menu with the following options is presented in the program:

- h: Prints a help message.
- a: Shows the network with symbols for all training patterns.
- o: Shows the network with symbols for one training pattern. The patterns number (1 ... patterns) is asked for.
- s: Shows the network with symbols for a specified pattern. The pattern (0's and 1's separated by spaces) is asked for.
- p: Prints the final values of the weights (see below).
- q: Quits the program.

The network is presented like this three-layer example:

```
threshold        threshold to output neurons
    | input neurons |
   (T) (*) (o) (o) (T) <.> [*] <-- weights hid->out
   [-] [x] [-] <x> (o) [-] [-]
   <x> weights [-] (.) <---------- hidden neurons
   [.] in->hid [-] (x) <o> [x]
   [-] <.> [*] [-] (*) [-] [-]
                       (*) (o) <-- output neurons

Representations:       | Symbols: o :  0  <= y <  0.2
 () neuron             |          . : 0.2 <= y <  0.4
 [] positive weight    |          - : 0.4 <= y <  0.6
 <> negative weight    |          x : 0.6 <= y <  0.8
 T  threshold (always  |          * : 0.8 <= y <= 1.0
    locked to '1')     |
```

where y = output of neuron or normalized value of weight, which is calculated by dividing each weight with the absolute value of the largest weight in the same weight layer.

The values of the weights, weight(weight layer, input neuron, output neuron), are printed with weights connected to different input neurons separated by ':', and weights in different weight layers separated by '|'. The following order is used:

weight(0,0,1) ... weight(0,0, number of neurons in layer 1) : ... weight(0, number of neurons in layer 0, number of neurons in layer 1) |
... weight(last weight layer, number of neurons in second last layer, number of neurons in last layer)|

2.3 Output in the logfile

Data from the simulation is written in a file that is named after the set-up file with the extension '.log'. The following is written in the log file:

- Set-up data:
  Learning rate and accepted error.
  Number of neurons in each layer.
  Training patterns with corresponding targets.
  Initial randomized values of the weights.
- Final data:
  Number of cycles required for training.
  Training patterns with final values of output neurons.
  Final values of the weights.

The weights, weight(weight layer, input neuron, output neuron), are printed in the same order as described in Section 2.2.

2.4 Changeable definitions in the program

- LAYERS : Sets the number of layers in the network.
- MAX_NODES : Sets the maximum number of neurons in each layer.
- MAX_PATTERNS : Sets the maximum number of training patterns.
- MAX_CYCLES : Sets the maximum number of iteration cycles.
- PRINT_CYCLES : Sets the number of cycles between messages.

2.5 Hint

If the network fails to converge it might be caused by the initial values of the weights. Try changing the seed in the set-up file.

A Program

```
/*
 * Training of a Feed-Forward Back-Propagation Network.
 * The program must be called with a set-up file that
 * contains the following parameters:
 *   seed = the seed for the randomize generator (seed > 0)
 *   eta = the learning rate for the bp-algorithm (0 > eta <= 1)
 *   iota = the accepted error after training (0 > iota < 1)
 *   nodes[i] = the number of neurons in layer i (1 ... MAX_NODES),
 *     where i = 1 ... LAYERS.
 *   patterns = the number of patterns (1 ... MAX_PATTERNS)
 *  Furthermore, the patterns, together with the corresponding target
 * outputs, must be specified according to:
 *   x_1 ... x_<in_nodes>     out_1 ... out_<output_nodes> (pattern 1)
 *                          ...
 *   x_1 ... x_<in_nodes>     out_1 ... out_<output_nodes> (pattern n)
 *
 *  Example, the set-up file for a three-layer xor-network:
 * 387    1.0    0.1
 * 2     2    1
 * 4
 * 0 0    0
 * 1 0    1
 * 0 1    1
 * 1 1    0
 *
 *  Please note that only the values, in the order stated above, should be
 * given in the set-up file. After the values can anything be written.
 *
 *  Version 1.0: Andreas Berg / 920309
 */

# include <stdio.h>
# include <math.h>
# include <strings.h>

# define LAYERS 3          /* Number of layers in the network */
# define IN 0              /* The input layer's number */
# define OUT_LAYERS - 1    /* The output layer's number */
# define MAX_NODES 20      /* Maximum number of neurons in each layer */
# define MAX_PATTERNS 40   /* Maximum number of training patterns */
# define MAX_CYCLES 100000 /* Maximum number of iteration cycles */
# define PRINT_CYCLES 100  /* Number of cycles between messages */
# define MAX_NAME 30       /* Maximum number of letters in file name */
```

```
enum boolean { FALSE, TRUE };

int nodes[LAYERS], patterns; /* nodes[i] = number of neurons in layer i */
float neuron[LAYERS][MAX_NODES + 1]; /* neuron(layer, node) */
float weight[LAYERS - 1][MAX_NODES + 1][MAX_NODES + 1]; /* w(layer, in, out) */
float update[LAYERS - 1][MAX_NODES + 1][MAX_NODES + 1]; /* u(layer, in, out) */
float delta[LAYERS - 1][MAX_NODES + 1]; /* delta(weight_layer, neuron) */
float epsilon, eta, iota; /* error, learning rate, accepted error */
int input[MAX_PATTERNS + 1][MAX_NODES + 1]; /* i(pattern, input_neuron) */
int target[MAX_PATTERNS + 1][MAX_NODES + 1]; /* i(target, output_neuron) */

main(int argc, char **argv)
{
    int i, j, k, number = 0; /* Counters */
    unsigned int seed;       /* Seed for the randomizing generator */
    char *name;              /* Name of program */
    char *setname;           /* Name of set-up file */
    FILE *infile;            /* Set-up file (input) */
    char logname[MAX_NAME];  /* Name of log file */
    FILE *outfile;           /* Log file (output) */
    enum boolean quit;       /* Flag */
    float last(int pattern), absmax(float x, float y); /* Functions */
    enum boolean options(void);
    char symbol(float x);
    void show(int pattern), print_weights(int previous, FILE *pfile);
    void revise(int previous), introduce(int pattern);
    void initiate(int previous);
    void forward(int previous), backward(int previous);

    if ((name = rindex(*argv, '/')) \= NULL)    /* PART 1: INPUT */
name++;
    else
name = *argv;                              /* Name of program */
    if (argc \= 2) {
printf("Usage: %s 'set-up file'\n", name);
exit(1);
    }
    setname = *++argv;                             /* Name of set-up file */
    if ((infile = fopen(setname, "r")) == (FILE *)NULL){
printf("Couldn't open %s\n", setname);  /* Open set-up file */
exit(1);
    }
    fscanf(infile, "%d %f %f", &seed, &eta, &iota); /* Read set-up file */
    if (seed <= 0) {                               /* Check randomizer seed */
printf("Wrong value of seed: %d\n", seed);
exit(1);
    }
```

```
    if (eta <= 0 || eta > 1) {                  /* Check learning rate */
printf("Wrong value of eta: %f\n", eta);
exit(1);
    }
    if (iota <= 0 || iota >= 1) {               /* Check accepted error */
printf("Wrong value of iota: %f\n", eta);
exit(1);
    }
    for (i = IN; i <= OUT; i++) {
fscanf(infile, "%d", &nodes[i]);        /* Check number of nodes */
if (nodes[i] <= 0 || nodes[i] > MAX_NODES) {
    printf("Wrong number of nodes in layer %d: %d\n", i, nodes[i]);
    exit(1);
}
    }
    fscanf(infile, "%d", &patterns);            /* Check number of patterns */
    if (patterns <= 0 || patterns > MAX_PATTERNS) {
printf("Wrong number of patterns: %d\n", patterns);
exit(1);
    }
    for (i = 1; i <= patterns; i++) {
for (j = 1; j <= nodes[IN]; j++) {
    fscanf(infile, "%d", &input[i][j]); /* Check input patterns */
    if (input[i][j] \= 0 && input[i][j] \= 1) {
printf("Wrong value of input %d in pattern %d: %d\n",
       j, i, input[i][j]);
    exit(1);
    }
}
for (j = 1; j <= nodes[OUT]; j++) {     /* Check targets */
    fscanf(infile, "%d", &target[i][j]);
    if (target[i][j] \= 0 && target[i][j] \= 1) {
printf("Wrong value of target %d in pattern %d: %d\n",
       j, i, target[i][j]);
    exit(1);
    }
}
    }
    if (fclose(infile) == EOF) {                /* Close set-up file */
printf("Couldn't close %s\n", setname);
exit(1);
    }

    srand(seed);                /* PART 2: OUTPUT OF INITIAL DATA */
    for (i = IN; i < OUT; i++)
initiate(i);                    /* Randomize the initial weights */
    strcpy(logname, strcat(setname, ".log")); /* Name of output file */
```

```
    if ((outfile = fopen(logname, "w")) == (FILE *)NULL){
printf("Couldn't open %s\n", logname);
exit(1);
    }                                   /* Print set-up data to log file */
    fprintf(outfile, "* The weights, w(layer, in, out), are printed, with ");
    fprintf(outfile, "different inputs\n");
    fprintf(outfile, "* separated by ':', and layers separated by '|', ");
    fprintf(outfile, "in the following order:\n");
    fprintf(outfile, "* w(0,0,1)...w(0,0,%d):...w(0,%d,%d)|...w(%d,%d,%d)|\n",
    nodes[IN + 1], nodes[IN], nodes[IN + 1], LAYERS - 2,
    nodes[OUT - 1], nodes[OUT]);
    fprintf(outfile, "Learning rate = %4.2f, accepted error = %4.2f\n",
    eta, iota);
    fprintf(outfile, "Number of neurons in each layer = %d", nodes[IN]);
    for (i = IN + 1; i <= OUT; i++)
fprintf(outfile, "-%d", nodes[i]);
    fprintf(outfile, "\n");
    for (i = 1; i <= patterns; i++) { /* Print inputs and targets */
fprintf(outfile, "Pattern %d: ", i);
for (j = 1; j <= nodes[IN]; j++)
    fprintf(outfile, "(%d)", input[i][j]);
fprintf(outfile, " --> ");
for (j = 1; j <= nodes[OUT]; j++)
    fprintf(outfile, "(%d)", target[i][j]);
fprintf(outfile, "\n");
    }
    fprintf(outfile, "Initial randomized weights:\n");
    for (i = IN; i < OUT; i++)
print_weights(i, outfile);    /* Print initial weights */

    do {                              /* PART 3: TRAINING OF THE NETWORK */
number++;
epsilon = 0;
for (i = IN; i < OUT; i++)            /* Update the weights */
    revise(i);
for (i = 1; i <= patterns; i++) {  /* Do all patterns */
    introduce(i);                     /* Introduce the pattern */
    for (j = IN; j < OUT; j++)        /* Forward pass */
forward(j);
    epsilon = last(i);                /* Calculate output errors */
    for (j = OUT - 1; j > IN; j--) /* Backward pass */
backward(j);
}
if (number % PRINT_CYCLES == 0)    /* Time for printout */
    printf(" %d cycles of training.\n", number);
    } while ((epsilon > iota) && (number < MAX_CYCLES));
```

```
                                    /* PART 4: OUTPUT OF FINAL DATA */
    if (number >= MAX_CYCLES) {
    printf("No convergence after %d cycles.\n", number);
    exit(1);
}
    printf("Training completed after %d cycles.\n", number);
    fprintf(outfile, "Training completed after %d cycles.\n", number);
    for (i = 1; i <= patterns; i++) {    /* Print final data to log file */
fprintf(outfile, "Pattern %d: ", i);
introduce(i);
for (j = IN; j < OUT; j++)
    forward(j);
for (j = 1; j <= nodes[IN]; j++) /* Print neurons after training */
    fprintf(outfile, "(%d)", input[i][j]);
fprintf(outfile, " --> ");
for (j = 1; j <= nodes[OUT]; j++)
    fprintf(outfile, "(%4.2f)", neuron[OUT][j]);
fprintf(outfile, "\n");
    }
    fprintf(outfile, "Final weights:\n", number);
    for (i = IN; i < OUT; i++)           /* Print final weights */
print_weights(i, outfile);
    if (fclose(outfile) == EOF) {        /* Close log file */
printf("Couldn't close %s\n", logname);
exit(1);
    }
    printf("Written %s\n", logname);     /* End of final data */

    do                                   /* PART 5: USER-FRIENDLY INTERFACE */
quit = options();
    while (quit == FALSE);
    exit(0);
} /* main */

                                    /* PART 6: FUNCTIONS */
/* initiate: randomizes and initiates the weights */
void initiate(int previous)
{
    int i, j;

    for (i = 0; i <= nodes[previous]; i++)
    for (j = 1; j <= nodes[previous + 1]; j++) {
weight[previous][i][j] = (rand() % 200 - 100) / 100.0;
update[previous][i][j] = 0;
    }
} /* initiate */
```

```
/* print_weights: prints all weights in a layer */
void print_weights(int previous, FILE *pfile)
{
    int i, j, nr = 0, radmax;

    radmax = 10;
    for (i = 0; i <= nodes[previous]; i++) {
for (j = 1; j <= nodes[previous + 1]; j++) {
    fprintf(pfile, "%5.2f ", weight[previous][i][j]);
    nr++;
    if (nr == radmax && j \= nodes[previous + 1]) {
fprintf(pfile, "\n");
nr = 0;
    }
}
if (i \= nodes[previous]) {
    fprintf(pfile, ": ");
    if (nr == radmax) {
fprintf(pfile, "\n");
nr = 0;
    }
}
    }
    fprintf(pfile, "|\n");
} /* print_weights */

/* introduce: introduces the pattern to the inputs */
void introduce(int pattern)
{
    int i;

    for (i = 1; i <= nodes[IN]; i++)
neuron[IN][i] = input[pattern][i]; /* Pattern */
    for (i = 0; i < OUT; i++)
neuron[i][0] = 1.0; /* Thresholds are trained from locked neurons */
} /* introduce */

/* revise: updates the weights */
void revise(int previous)
{
    int i,j;

    for (i = 0; i <= nodes[previous]; i++)
for (j = 1; j <= nodes[previous + 1]; j++) {
    weight[previous][i][j] += update[previous][i][j];
    update[previous][i][j] = 0;
    }
```

```
} /* revise */

/* absmax: returns maximum of two absolute values */
float absmax(float x, float y)
{
    if (x < 0)
x = (-1) * x;
    if (y < 0)
y = (-1) * y;
    return (x > y) ? x : y;
} /* absmax */

/* maxweight: returns absolute value of the largest weight in a layer */
float maxweight(int previous)
{
    int i, j;
    float big = 0, absmax(float x, float y);

    for (i = 0; i <= nodes[previous]; i++) {
for (j = 1; j <= nodes[previous + 1]; j++)
    big = absmax(big, weight[previous][i][j]);
    }
    return big;
} /* maxweight */

/* forward: makes a forward pass */
void forward(int previous)
{
    int i, j;
    float sum, sig(float x);

    for (i = 1; i <= nodes[previous + 1]; i++) {
sum = 0;
for (j = 0; j <= nodes[previous]; j++) {
    sum += neuron[previous][j] * weight[previous][j][i];
}
neuron[previous + 1][i] = sig(sum);
    }
} /* forward */

/* last: calculates the error between outputs and targets */
float last(int pattern)
{
    int i, j;
    float diff, d(float x), absmax(float x, float y);

    for (i = 1; i <= nodes[OUT]; i++) {
```

```
diff = target[pattern][i] - neuron[OUT][i];
delta[OUT - 1][i] = d(neuron[OUT][i]) * diff;
for (j = 0; j <= nodes[OUT - 1]; j++)
    update[OUT - 1][j][i] +=
eta * delta[OUT - 1][i] * neuron[OUT - 1][j];
    }
    return absmax(epsilon, diff);
} /* last */

/* backward: makes a backward pass */
void backward(int previous)
{
    int i, j;
    float sum, d(float x);

    for (i = 1; i <= nodes[previous]; i++) {
sum = 0;
for (j = 1; j <= nodes[previous + 1]; j++)
    sum += delta[previous][j] * weight[previous][i][j];
delta[previous - 1][i] = d(neuron[previous][i]) * sum;
for (j = 0; j <= nodes[previous - 1]; j++)
    update[previous - 1][j][i] +=
eta * delta[previous - 1][i] * neuron[previous -1][j];
    }
} /* backward */

/* show: shows the network */
void show(int pattern)
{
    int i, j, k, l;
    char answer[30], symbol(float x);
    void introduce(int pattern), forward(int previous);
    float big[OUT - 1], maxweight(int previous);

    introduce(pattern);
    for (i = IN; i < OUT; i++) {
forward(i);
big[i] = maxweight(i);
    }
    printf("\n");
    if (pattern == 0)
printf("Pattern: ");
    else
printf("Pattern %d: ", pattern);
    for (i = 1; i <= nodes[IN]; i++)  /* print value of input neurons */
printf("(%d)", input[pattern][i]);
    printf(" --> ");
```

```
    for (i = 1; i <= nodes[OUT]; i++) /* print value of output neurons */
printf("(%4.2f)", neuron[OUT][i]);
    printf("\n");
    printf("\n");

    for (i = IN; i <= OUT; i++) {
for (j = i; j > IN + 1; j = j - 2)
    for (k = 0; k <= nodes[j - 2]; k++)
printf("    ");
if (i % 2 == 0) {
    if (i < OUT)
printf("(T) ");
    else
printf("    ");
    for (j = 1; j <= nodes[i]; j++)
printf("(%c) ", symbol(neuron[i][j]));
    if (i < OUT - 1) {
printf("(T) ");
for (j = 1; j <= nodes[i + 2]; j++) { /* thresholds */
    if (weight[i + 1][0][j] > 0)
printf("[%c] ", symbol(weight[i+1][0][j] / big[i+1]));
    else
printf("<%c> ", symbol(-weight[i+1][0][j] / big[i+1]));
}
    }
    printf("\n");
}
else {
    for (j = 1; j <= nodes[i]; j++) { /* all I2H */
for (k = 0; k <= nodes[i - 1]; k++) {
    if (weight[i - 1][k][j] > 0)
printf("[%c] ",
       symbol(weight[i - 1][k][j] / big[i - 1]));
    else
printf("<%c> ",
       symbol(-weight[i - 1][k][j] / big[i - 1]));
}
printf("(%c) ", symbol(neuron[i][j]));
if (i < OUT) {
    for (k = 1; k <= nodes[i + 1]; k++) { /* H2O */
if (weight[i][j][k] > 0)
    printf("[%c] ",
   symbol(weight[i][j][k] / big[i]));
else
    printf("<%c> ",
   symbol(-weight[i][j][k] / big[i]));
    }
```

```
}
printf("\n");
    }
}
    }
    printf("<RETURN>");
    gets(answer);
} /* show */

/* sig: calculates the sigmoid value */
float sig(float x)
{
    return (1 / (exp(-x) + 1));
/*    return (2 / (exp(-x) + 1) - 1); */
} /* sig */

/* d: calculates the derivate value */
float d(float x)
{
    return (x * (1 - x));
} /* d */

/* symbol: selects the appropriate symbol */
char symbol(float x)
{
    char c;

    if (x > 0.8)
c = '*';
    else if (x > 0.6)
c = 'x';
    else if (x > 0.4)
c = '-';
    else if (x > 0.2)
c = '.';
    else
c = 'o';
    return c;
} /* symbol */

/* options: prints a menu and carries out the desired task */
enum boolean options(void)
{
    enum boolean again, sense, q = FALSE;
    int i, nr;
    char c, answer[30];
    void help(void), show(int pattern);
```

```
    void print_weights(int previous, FILE *pfile);

    printf("\n");
    printf("   Available Options\n");
    printf("   -----------------\n");
    printf("(H)elp\n");
    printf("Show network for:\n");
    printf("   (A)ll training patterns\n");
    printf("   (O)ne training pattern\n");
    printf("   (S)pecified pattern\n");
    printf("(P)rint weights\n");
    printf("(Q)uit\n");
    printf("\n");
    printf("Option:");
    do {
again = FALSE;
gets(answer);
switch (*answer) {
case 'h': case 'H': /* help */
    help();
    again = TRUE;
    break;
case 'a': case 'A': /* all training patterns */
    for (i = 1; i <= patterns; i++)
show(i);
    break;
case 'o': case 'O': /* one training pattern */
    printf("Give the pattern's number:");
    scanf("%d", &nr);
    gets(answer); /* empties the input buffer */
    if (nr > 0 && nr <= patterns)
show(nr);
    else
printf("Pattern does not exist.\n");
    break;
case 's': case 'S': /* specified pattern */
    sense = TRUE;
    printf("Give the pattern in 0's and 1's separated by spaces:");
    for (i = 1; i <= nodes[IN]; i++) {
scanf("%d", &input[0][i]);
if (input[0][i] \= 0 && input [0][i] \= 1) {
    printf("Input '%d' does not make sense.\n",
   input[0][i]);
    sense = FALSE;
    break;
}
    }
```

```
    gets(answer); /* empties the input buffer */
    if (sense)
show(0);
    break;
case 'p': case 'P': /* print weights */
    printf("\n");
    for (i = IN; i < OUT; i++) {
print_weights(i, stdout);
    }
    printf("\n");
    printf("<RETURN>");
    gets(answer); /* empties the input buffer */
    break;
case 'q': case 'Q': /* quit */
    q = TRUE;
    break;
default:
    printf("Option '%c' not available. Please try again:", *answer);
    again = TRUE;
}
    } while (again == TRUE);
    return q;
} /* options */

/* help: prints some helpful information */
void help(void)
{
    printf("\n");
    printf("   Help\n");
    printf("   ----\n");
    printf("h: Prints this message.\n");
    printf("\n");
    printf("a: Shows the network with symbols for all training patterns.\n");
    printf("\n");
    printf("o: Shows the network with symbols for one training pattern.\n");
    printf("   The patterns number (1 ... patterns) is asked for.\n");
    printf("\n");
    printf("s: Shows the network with symbols for a specified pattern.\n");
    printf("   The pattern (0's and 1's separated by spaces) is asked for.\n");
    printf("\n");
    printf("   The network is presented like this three-layer example:\n");
    printf("\n");
    printf("   threshold        threshold to output neurons\n");
    printf("       | input neurons |\n");
    printf("      (T) (*) (o) (o) (T) <.> [*] <-- weights hid->out\n");
    printf("      [-] [x] [-] <x> (o) [-] [-]\n");
    printf("      <x> weights [-] (.) <---------- hidden neurons\n");
```

```
    printf("      [.] in->hid [-] (x) <o> [x]\n");
    printf("      [-] <.> [*] [-] (*) [-] [-]\n");
    printf("                          (*) (o) <-- output neurons\n");
    printf("\n");
    printf("   Representations:       | Symbols: o :  0  <= y <  0.2\n");
    printf("    () neuron             |          . : 0.2 <= y <  0.4\n");
    printf("    [] positive weight    |          - : 0.4 <= y <  0.6\n");
    printf("    <> negative weight    |          x : 0.6 <= y <  0.8\n");
    printf("    T  threshold (always  |          * : 0.8 <= y <= 1.0\n");
    printf("       locked to '1')     | where y = output of neuron or\n");
    printf("                          | normalized value of weight.\n");
    printf("\n");
    printf("p: Prints the final values of the weights, w(layer,in,out).\n");
    printf("   Different inputs are separated by ':' and layers\n");
    printf("   are separated by '|'. The following order is used:\n");
    printf("\n");
    printf("   w(0,0,1)...w(0,0,n_1):...w(0,n_0,n_1)|...w(n_l,n_N-1,n_N)|\n");
    printf("\n");
    printf("q: Quits the program.\n");
    printf("\n");
    printf("Option:");
} /* help */
```