9.2 Applications and Utilities

9.2.1 tlearn

tlearn is a software package written by Jeff Elman and others. It simulates "layers" of neurons interacting with user-specified connections, and can perform training by back-propagation. This portion of the technical report describes the inner workings of tlearn and the components that make up the software. The intent is for the reader to be able to reverse-engineer the software in order to extend it.

An example of such an extension is **tlavq** (described in section 9.2.11), which extends **tlearn** by adding an on-line clustering module with associated command-line switches.

Networks modeled by tlearn

Neurons in the network are grouped into layers, and connections between these layers are specified by the user in the command file (which usually has a .cf extension). The default connectivity is for complete connections from one layer to another. This can be changed to copy-back connections for recurrent nets. The default neuron transfer function is a sigmoid, but can also be specified to be linear. Links are by default included in any training performed, but can be set to fixed untrainable weights if desired. Trained networks can be run through the network to extract their outputs as well as their hidden unit activations.

tlearn network configuration files

The structure of the neural network for simulation is defined at run-time using a configuration (.cf) file.

example.cf

`NODES:`	
`nodes = 12`	Specifies the number of neurons in the network. Inputs do **not** count as nodes.
`inputs = 4`	Specifies the number of inputs into the network.
`outputs = 4`	Specifies the number of outputs from the network. Output neurons are included in the node count.
`output nodes are 5-8`	Specifies which nodes are to be sampled for their outputs. (Also used for training)
`CONNECTIONS:`	
`groups = 0`	Specifies the number of groups of nodes. Groups are constrained to have the same weights on their connections. This is not terribly useful, and is usually 0.
`1-4 from i1-14`	Specifies that nodes 1 through 4 are fully connected from inputs 1 through 4.
`5-8 from 1-4`	Specifies that the output nodes (5 through 8) are fully connected from hidden units 1 through 4.
`9-12 from 1-4 = 1.0 & 1.0 fixed one-to-one`	This line specifies that nodes 9 through 12 are "context" neurons. Their connections from nodes 1 through 4 are constrained to have weights of 1.0. Their activations are copied from nodes 1 through 4 on each sweep.
`SPECIAL:`	
`linear = 9-12`	Specifies that neurons 9 through 12 use a linear output

	function.
`selected = 1-4`	Specifies that when performing a run to probe the network's hidden units, the activations from nodes 1 through 4 will be extracted.

This configuration file describes the network shown below.

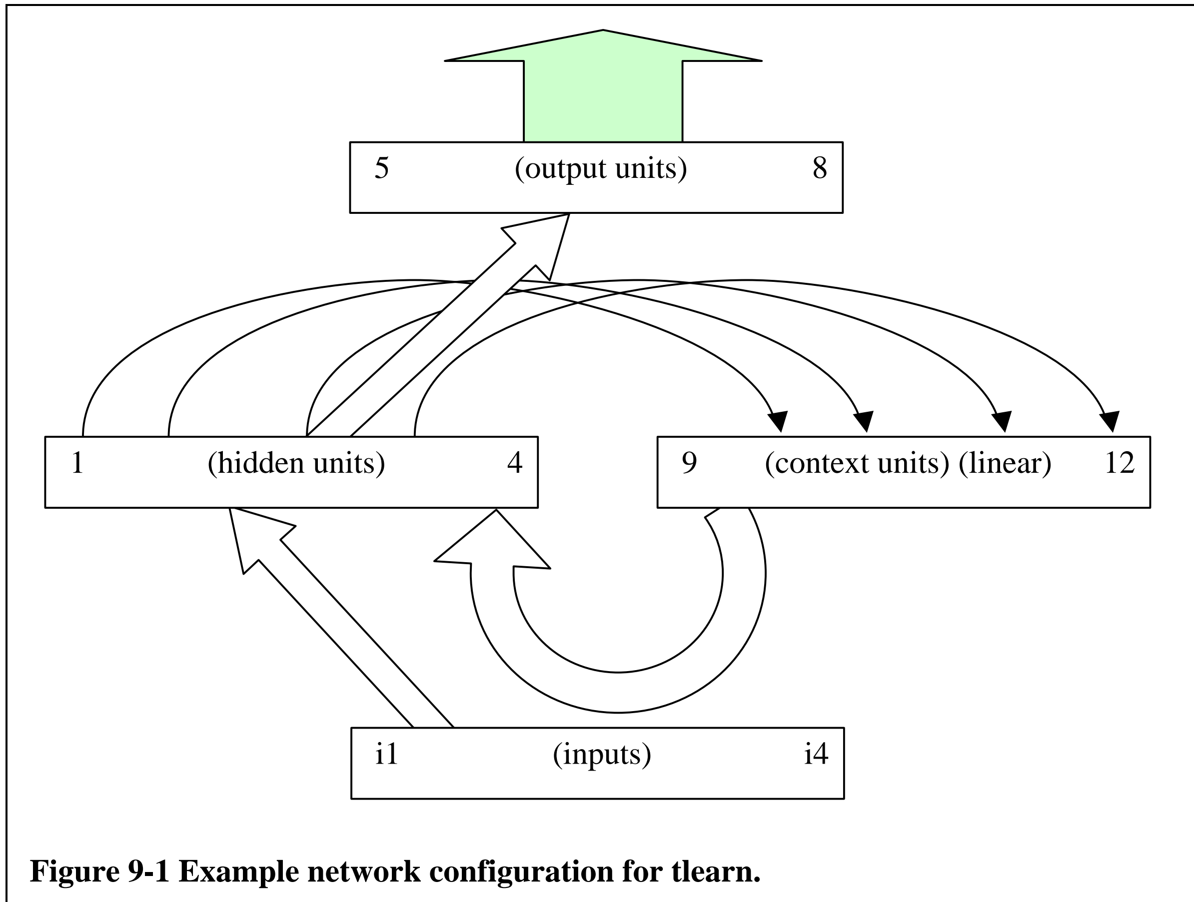

Figure 9-1 Example network configuration for tlearn.

For a more detailed description of the configuration options, see the tlearn manual. The next page is a quick reference guide to **tlearn**'s command-line switches.

tlearn

–f<fileroot>

–r<learning.rate>

–m<momentum>

–s<sweeps>

–l<old.weights.file>

–X (use.reset)

–M<RMSerr.target>

–E<RMSerr.checkpoint>

–C<weights.checkpoint>

–S<RNG.seed>

–V<Verify learning>

–R (randomise.input)

–P (probe.selected)

Figure 9-2 tlearn quick reference page.

9.2.2 tlearn's Internal Structure

tlearn is written in the Kernighan and Ritchie dialect of C. It has no function prototypes and uses a non-ANSI method for declaring function parameters, which can make understanding the source code difficult at times. This section contains the documentation resulting from the reverse-engineering process, including function prototypes and descriptions as well as the global variable definitions.

Source code files

tlearn.c	• Reads command-line switches • Opens configuration file • Loads weights from file (if specified) • Initialises biases • Main sweep loop
parse.c	• Reads and interprets configuration file • Sets up network configuration arrays
update.c	• Loads inputs and outputs from disk • Manages updating of targets and input values • Manages updating of weights (but not their desired values)
activate.c	• Calculates network activations
compute.c	• Calculates error measures • Calculates error signals and desired weight deltas • Determines weight updates required for learning
arrays.c	• Allocates arrays
weights.c	• Saves and loads weights to/from disk
subs.c	• Various misc. output functions • Network reset functions
exp.c	• (Separate program) creates a lookup table for exponential function

Source file module descriptions

tlearn.c

Structures

```
struct   cf {           /* Connection information */
    int   con,          /* FLAG: This connection exists */
          fix,          /* FLAG: This connection has a fixed weight */
          num           /* Group number this connection belongs to */
          lim;          /* FLAG: This connection has weight limits imposed */
    float min,          /* Minimum allowed weight */
          max;          /* Maximum allowed weight */
};
```

This structure is used in a two-dimensional array **cinfo**. Each cell holds the connection information between two nodes:

```
struct cf    **cinfo[to][from].
```

High-dimensional arrays in tlearn are indexed separately along their dimensions. For example,

```
struct cf    *ci = cinfo + to;
*(ci + from) = data;
```

```
struct   nf {           /* Node information */
```

```
        int    func,       /* Which output function this node uses */
               dela,       /* FLAG: This node has a delayed output */
               targ;       /* FLAG: This node is an output node (i.e. a target for learning */
    };
```

This structure is used in the array **ninfo**. Each cell holds the node information:

```
  struct nf    *ninfo[node].
```

Exported variables

```
    /* Node counts */
        int    nn,          /* Number of "real" nodes (i.e. not including inputs or bias nodes) */
               ni,          /* Number of input nodes */
               no,          /* Number of designated output nodes */
               nt,          /* Total number of nodes (nn + ni + 1) */
               np;          /* Index to first "real" node (ni + 1) */

    /* Information arrays */
        struct cf **cinfo;  /* 2D ARRAY[nn][nt]: Connection information between all nodes */
        struct nf *ninfo;   /* ARRAY[nn]: Node information for all nodes */

    /* Node flag arrays */
        int    *outputs,    /* ARRAY[no]: Indices of output nodes */
               *selects,    /* FLAG ARRAY[nn + 1]: This node is selected for probe output */
               *linput;     /* ARRAY[ni]: Localist input array */

    /* Network state arrays */
        float  *znew,       /* ARRAY[nt]: Activations of nodes at time t + 1 */
               *zold,       /* ARRAY[nt]: Activations of nodes at time t */
               *zmem,       /* ARRAY[nt]: Saved activations of nodes at time t */
               **wt,        /* 2D ARRAY[nn][nt]: Weight of connection [to][from] */
               **dwt,       /* 2D ARRAY[nn][nt]: Delta weight to rectify error signal */
               **winc,      /* 2D ARRAY[nn][nt]: Actual desired weight increase for learning */
               *target,     /* ARRAY[no]: Target output values */
               *error,      /* ARRAY[nn]: Error signals (output – target) */
               ***pnew,     /* 3D ARRAY[nn][nt][nn]: P-variable at time t+1 */
               ***pold;     /* 3D ARRAY[nn][nt][nn]: P-variable at time t */

    /* Network parameters */
        float  rate,        /* Learning rate (default: 0.1) */
           momentum,        /* Momentum for learning (default: 0.0) */
           weight_limit,    /* Bound for random weight initialisation limit (default: 1.0) */
           criterion,       /* RMS error target for end of learning (default: 0.0) */
           init_bias;       /* Offset for bias weight initialisation (default: 0.0) */
        long   sweep,       /* Current sweep */
               tsweeps,     /* Total number of sweeps on this network */
               report;      /* Output RMS error every 'report' sweeps (default: 0.0) */
        int    ngroups;     /* Number of groups */

    /* Network flags */
        int    backprop,    /* FLAG: Use standard back-propagation? (default: YES) */
               teacher,     /* FLAG: Feed back targets during learning? (default: NO) */
               localist,    /* FLAG: Use localist-type input values? */
               randomly,    /* FLAG: Present inputs in random order? (default: NO) */
```

```
        limits,          /* FLAG: Limit weight values? (default: NO) */
        ce;              /* FLAG: Use cross-entropy error collection? (default: NO) */

/* Files */
    char  root[128],     /* Root portion of filename (i.e. 'root'.data, .teach, .cf, .reset, etc) */
          loadfile[128]; /* Filename of saved weights file to initialise network with */
    FILE *cfp;           /* File pointer for .cf file */
```

Function prototypes

```
int main(IN int argc, IN char **argv);
/* -- Function main
 * Pre: TRUE
 * Post: TRUE
 * Purpose: Initialise network, main sweep loop
 */

void usage(void);
/* -- Function usage
 * Pre: TRUE
 * Post: The command-line usage has been output to stderr
 */

void intr(IN int sig);
/* -- Function intr
 * Pre: sig is the received termination signal form the OS
 * Post: The current set of weights has been saved and tlearn has exit
 */
```

Annotated source

```
/* tlearn.c - simulator for arbitrary networks with time-ordered input */

/*------------------------------------------------------------------------

This program simulates learning in a neural network using either
the classical back-propagation learning algorithm or a slightly
modified form derived in Williams and Zipser, "A Learning Algo-
rithm for Continually Running Fully Recurrent Networks."  The
input is a sequence of vectors of (ascii) floating point numbers
contained in a ".data" file.  The target outputs are a set of
time-stamped vectors of (ascii) floating point numbers (including
optional "don't care" values) in a ".teach" file.  The network
configuration is defined in a ".cf" file documented in tlearn.man.

------------------------------------------------------------------------*/

#include <math.h>
#include <stdio.h>
#include <signal.h>
#ifdef ibmpc
#include "strings.h"
#include <fcntl.h>
#else
#include <string.h>
#include <sys/file.h>
#endif
#include <sys/types.h>
#include <sys/stat.h>

#ifdef ibmpc
#define random(x) rand(x)
#define srandom(x) srand(x)
#endif
```

Node count variables

```
int   nn;      /* number of nodes */
int   ni;      /* number of inputs */
int   no;      /* number of outputs */
int   nt;      /* nn + ni + 1 */
int   np;      /* ni + 1 */
```

Information structures

```
struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight-limits flag */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};
```

Information arrays

```
struct   cf **cinfo; /* (nn x nt) connection info */
struct   nf *ninfo;     /* (nn) node activation function info */
```

Node flag arrays

```
int   *outputs;   /* (no) indices of output nodes */
int   *selects;   /* (nn+1) nodes selected for probe printout */
int   *linput; /* (ni) localist input array */
```

Activations, weights, target, error signal and p-variable arrays

```
float *znew;      /* (nt) inputs and activations at time t+1 */
float *zold;      /* (nt) inputs and activations at time t */
float *zmem;      /* (nt) inputs and activations at time t */
float **wt;    /* (nn x nt) weight TO node i FROM node j*/
float **dwt;      /* (nn x nt) delta weight at time t */
float **winc;     /* (nn x nt) accumulated weight increment*/
float *target; /* (no) output target values */
float *error;     /* (nn) error = (output - target) values */
float ***pnew; /* (nn x nt x nn) p-variable at time t+1 */
float ***pold; /* (nn x nt x nn) p-variable at time t */
```

Network parameters

```
float rate = .1;  /* learning rate */
float momentum = 0.; /* momentum */
float weight_limit = 1.;   /* bound for random weight init */
float criterion = 0.;   /* exit program when rms error is less than this */
float init_bias = 0.;   /* possible offset for initial output biases */
```

```
long  sweep = 0;  /* current sweep */
long  tsweeps = 0;   /* total sweeps to date */
long  report = 0; /* output rms error every "report" sweeps */

int   ngroups = 0;   /* number of groups */
```

Network flags

```
int   backprop = 1;  /* flag for standard back propagation (the default) */
int   teacher = 0;   /* flag for feeding back targets */
int   localist = 0;  /* flag for speed-up with localist inputs */
int   randomly = 0;  /* flag for presenting inputs in random order */
int   limits = 0; /* flag for limited weights */
int   ce = 0;     /* flag for cross_entropy */
```

Files

```
char  root[128];  /* root filename for .cf, .data, .teach, etc.*/
char  loadfile[128]; /* filename for weightfile to be read in */

FILE  *cfp;    /* file pointer for .cf file */

void intr();

extern int load_wts();
extern int save_wts();
extern int act_nds();
```

Function **main**

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

   FILE  *fopen();
   extern   char *optarg;
   extern   float rans();
   extern   long time();
/* extern   int intr(); */
```

Time variables

ttime: sweeps since start of .data file

tmax: number of elements in .data

```
   long  nsweeps = 0;   /* number of sweeps to run for */
   long  ttime = 0;  /* number of sweeps since time = 0 */
   long  utime = 0;  /* number of sweeps since last update_weights */
   long  tmax = 0;   /* maximum number of sweeps (given in .data) */
   long  umax = 0;   /* update weights every umax sweeps */
   long  rtime = 0;  /* number of sweeps since last report */
   long  check = 0;  /* output weights every "check" sweeps */
   long  ctime = 0;  /* number of sweeps since last check */
```

Flags and indices

```
   int   c;
   int   i;
   int   j;
   int   k;
   int   nticks = 1; /* number of internal clock ticks per input */
   int   ticks = 0;  /* counter for ticks */
   int   learning = 1;  /* flag for learning */
   int   reset = 0;  /* flag for resetting net */
   int   verify = 0; /* flag for printing output values */
   int   probe = 0;  /* flag for printing selected node values */
   int   command = 1;   /* flag for writing to .cmd file */
   int   loadflag = 0;  /* flag for loading initial weights from file */
   int   iflag = 0;  /* flag for -I */
   int   tflag = 0;  /* flag for -T */
   int   rflag = 0;  /* flag for -x */
   int   seed = 0;   /* seed for random() */
```

Measured error values

```
   float err = 0.;   /* cumulative ss error */
   float ce_err = 0.;   /* cumulate cross_entropy error */
```

Weight and p-variable array indices

```
   float *w;
   float *wi;
   float *dw;
   float *pn;
   float *po;
```

Connection information array index

```
   struct   cf *ci;
```

.cf and .cmd filenames

```
   char  cmdfile[128];  /* filename for logging runs of program */
   char  cfile[128]; /* filename for .cf file */
```

.cmd file pointer

```
   FILE  *cmdfp;
```

Set up signal handlers

```
   signal(SIGINT, intr);
#ifndef ibmpc
   signal(SIGHUP, intr);
```

```
   signal(SIGQUIT, intr);
   signal(SIGKILL, intr);
#endif

#ifndef ibmpc
   exp_init();
#endif

   root[0] = 0;

   while ((c = getopt(argc, argv, "f:hil:m:n:r:s:tC:E:ILM:PRS:TU:VXB:H:")) != EOF) {
      switch (c) {
         case 'C':
            check = (long) atol(optarg);
            ctime = check;
            break;
         case 'f':
            strcpy(root,optarg);
            break;
         case 'i':
            command = 0;
            break;
         case 'l':
            loadflag = 1;
            strcpy(loadfile,optarg);
            break;
         case 'm':
            momentum = (float) atof(optarg);
            break;
         case 'n':
            nticks = (int) atoi(optarg);
            break;
         case 'P':
            probe = 1;
            learning = 0;
            break;
         case 'r':
            rate = (float) atof(optarg);
            break;
         case 's':
            nsweeps = (long) atol(optarg);
            break;
         case 't':
            teacher = 1;
            break;
         case 'L':
            backprop = 0;
            break;
         case 'V':
            verify = 1;
            learning = 0;
            break;
         case 'X':
            rflag = 1;
            break;
         case 'E':
            report = (long) atol(optarg);
            break;
         case 'I':
            iflag = 1;
            break;
         case 'M':
            criterion = (float) atof(optarg);
            break;
         case 'R':
            randomly = 1;
            break;
         case 'S':
            seed = atoi(optarg);
            break;
         case 'T':
            tflag = 1;
            break;
         case 'U':
            umax = atol(optarg);
            break;
         case 'B':
            init_bias = atof(optarg);
            break;
         /*
          * if == 1, use cross-entropy as error;
          * if == 2, also collect cross-entropy stats.
```

Get options from command line

```
         */
        case 'H':
           ce = atoi(optarg);
           break;
        case '?':
        case 'h':
        default:
           usage();
           exit(2);
        break;
     }
  }

  if (nsweeps == 0){
     perror("ERROR: No -s specified");
     exit(1);
  }
```

Get options from command line

```
  /* open files */

  if (root[0] == 0){
     perror("ERROR: No fileroot specified");
     exit(1);
  }

  if (command){
     sprintf(cmdfile, "%s.cmd", root);
     cmdfp = fopen(cmdfile, "a");
     if (cmdfp == NULL) {
        perror("ERROR: Can't open .cmd file");
        exit(1);
     }
     for (i = 1; i < argc; i++)
        fprintf(cmdfp,"%s ",argv[i]);
     fprintf(cmdfp,"\n");
     fflush(cmdfp);
  }

  sprintf(cfile, "%s.cf", root);
  cfp = fopen(cfile, "r");
  if (cfp == NULL) {
     perror("ERROR: Can't open .cf file");
     exit(1);
  }
```

Open files

```
  get_nodes();
  make_arrays();
  get_outputs();
  get_connections();
  get_special();
```

Read network configuration and set up network arrays

```
  if (!seed)
     time(&seed);
  srandom(seed);
```

Initialise random number generator

```
  if (loadflag)
     load_wts();
  else {
     for (i = 0; i < nn; i++){
        w = *(wt + i);
        dw = *(dwt+ i);
        wi = *(winc+ i);
        ci = *(cinfo+ i);
        for (j = 0; j < nt; j++, ci++, w++, wi++, dw++){
           if (ci->con)
              *w = rans(weight_limit);
           else
              *w = 0.;
           *wi = 0.;
           *dw = 0.;
        }
     }
     /*
      * If init_bias, then we want to set initial biases
      * to (*only*) output units to a random negative number.
      * We index into the **wt to find the section of receiver
      * weights for each output node.  The first weight in each
      * section is for unit 0 (bias), so no further indexing needed.
      */
     for (i = 0; i < no; i++){
        w = *(wt + outputs[i] - 1);
        ci = *(cinfo + outputs[i] - 1);
```

Initialise network weights either to random values or to a saved state, if **loadflag** is set.

```
      if (ci->con)
         *w = init_bias + rans(.1);
      else
         *w = 0.;
   }
}
zold[0] = znew[0] = 1.;
for (i = 1; i < nt; i++)
   zold[i] = znew[i] = 0.;
if (backprop == 0){
   make_parrays();
   for (i = 0; i < nn; i++){
      for (j = 0; j < nt; j++){
         po = *(*(pold + i) + j);
         pn = *(*(pnew + i) + j);
         for (k = 0; k < nn; k++, po++, pn++){
            *po = 0.;
            *pn = 0.;
         }
      }
   }
}
```

Initialising weights to random values

```
nsweeps += tsweeps;
for (sweep = tsweeps; sweep < nsweeps; sweep++){
```

Main sweep loop start

```
   for (ticks = 0; ticks < nticks; ticks++){
```

Tick loop start

```
      update_reset(ttime,ticks,rflag,&tmax,&reset);
```

Retrieve reset flag from .reset file

```
      if (reset){
         if (backprop == 0)
            reset_network(zold,znew,pold,pnew);
         else
            reset_bp_net(zold,znew);
      }
```

Reset entire network if required (if **reset** flag is set)

```
      update_inputs(zold,ticks,iflag,&tmax,&linput);
```

Get input values from .data file

```
      if (learning || teacher || (report != 0))
         update_targets(target,ttime,ticks,tflag,&tmax);
```

Update target outputs if learning

```
      act_nds(zold,zmem,znew,wt,linput,target);
```

Update network activations

```
      if (learning || (report != 0))
         comp_errors(zold,target,error,&err,&ce_err);
```

Calculate error measures if required

```
      if (learning && (backprop == 0))
         comp_deltas(pold,pnew,wt,dwt,zold,znew,error);
      if (learning && (backprop == 1))
         comp_backprop(wt,dwt,zold,zmem,target,error,linput);
```

Calculate weight changes for learning if required

```
      if (probe)
         print_nodes(zold);
   }
```

Output selected nodes activations if required

```
   if (verify)
      print_output(zold);
```

Output output nodes activations if required

```
   if (report && (++rtime >= report)){
      rtime = 0;
      if (ce == 2)
         print_error(&ce_err);
      else
         print_error(&err);
   }
```

Report the error measures if required

```
   if (check && (++ctime >= check)){
      ctime = 0;
      save_wts();
   }
```

Save current weights to a checkpoint file if required

```
   if (++ttime >= tmax)
      ttime = 0;
```

Next data value, check for roll-over to start of .data file

```
   if (++utime >= umax){
      utime = 0;
      update_weights(wt,dwt,winc);
   }
```

Next update sweep, check whether to update weights and do so if required

```
}
```

If training occurred, save the weights.

```
    if (learning)
       save_wts();
    exit(0);

}
```

Function **usage**

```
usage() {
  fprintf(stderr, "\n");
  fprintf(stderr, "-f fileroot:\tspecify fileroot <always required>\n");
  fprintf(stderr, "-l weightfile:\tload in weightfile\n");
  fprintf(stderr, "\n");
  fprintf(stderr, "-s #:\trun for # sweeps <always required>\n");
  fprintf(stderr, "-r #:\tset learning rate to # (between 0. and 1.) [0.1]\n");
  fprintf(stderr, "-m #:\tset momentum to # (between 0. and 1.) [0.0]\n");
  fprintf(stderr, "-n #:\t# of clock ticks per input vector [1]\n");
  fprintf(stderr, "-t:\tfeedback teacher values in place of outputs\n");
  fprintf(stderr, "\n");
  fprintf(stderr, "-S #:\tseed for random number generator [random]\n");
  fprintf(stderr, "-U #:\tupdate weights every # sweeps [1]\n");
  fprintf(stderr, "-E #:\trecord rms error in .err file every # sweeps [0]\n");
  fprintf(stderr, "-C #:\tcheckpoint weights file every # sweeps [0]\n");
  fprintf(stderr, "-M #:\texit program when rms error is less than # [0.0]\n");
  fprintf(stderr, "-X:\tuse auxiliary .reset file\n");
  fprintf(stderr, "-P:\tprobe selected nodes on each sweep (no learning)\n");
  fprintf(stderr, "-V:\tverify outputs on each sweep (no learning)\n");
  fprintf(stderr, "-R:\tpresent input patterns in random order\n");
  fprintf(stderr, "-I:\tignore input values during extra clock ticks\n");
  fprintf(stderr, "-T:\tignore target values during extra clock ticks\n");
  fprintf(stderr, "-L:\tuse RTRL temporally recurrent learning\n");
  fprintf(stderr, "-B #:\toffset for offset biasi initialization\n");
  fprintf(stderr, "\n");
}
```

Function **intr**

```
void
intr(sig)
    int   sig;
{
    save_wts();
    exit(sig);
}
```

parse.c

Function prototypes

```
void get_nodes(void);
/* -- Function get_nodes
 * Pre: "root.cf" file has been opened
 *      cfp is a global file pointer to the configuration file
 *      The stream is at the start of the file
 *Post: Sets nn, ni, no, nt and np according to configuration file
 */

void get_outputs(void);
/* -- Function get_outputs
 * Pre: "root.cf" file has been opened
 *      cfp is a global file pointer to the configuration file
 *      get_nodes has been called
 * Post: Reads "outputs" line from "NODES" section of configuration file
 *       Fills outputs[] array according to configuration file
 */

void get_connections(void);
/* -- Function get_connections
 * Pre: "root.cf" file has been opened
 *      cfp is a global file pointer to the configuration file
 *      get_outputs has been called
 * Post: Reads "CONNECTIONS" section of configuration file
 *       Fills cinfo[] structure array according to network layout
 */

void get_special(void);
/* -- Function get_special
 * Pre: "root.cf" file has been opened
```

```
 *      cfp is a global file pointer to the configuration file
 *      get_connections has been called
 * Post: Reads "SPECIAL" section of configuration file
 *       Sets weight_limit global variable
 *       Sets selects[] array for probing hidden nodes
 *       Sets flags in ninfo[] structure array
 */

void get_str(IN FILE *fp, OUT char buf[], IN char str[]);
/* -- Functon get_str
 * Pre: fp is a file pointer to a file opened for reading
 *      buf[] is a pre-allocated string buffer to read data into
 *      str[] is the whitespace character expected after the next string
 * Post: A whitespace-terminated string was read into buf
 */

void get_nums(IN char str[], IN int nv, IN int offset, OUT int vec[]);
/* -- Function get_nums
 * Pre: str[] contains a string to interpret as a numeric set of numbers
 *      nv is the length of vec
 *      offset is the offset to "real" nodes in the node arrays
 *      vec[] is a pre-allocated array of flags for selection
 * Post: The numeric set in str was interpreted
 *       vec[] is a set of flags.  The cell i was set to YES if i was in
 *        the set given in str[]
 * Notes: Numeric sets are of the form a,b,c-d,e,… etc
 */

void parse_err(void);
/* -- Function parse_err
 * Pre: TRUE
 * Post: An error was printed to stderr
 */
```

update.c

Function prototypes

```
void update_inputs(   OUT float aold[], IN int tick, IN int flag,
                      OUT long *maxtime, OUT int *local[]);
/* -- Function update_inputs
 * Pre: the global variable root[] contains the root filename
 *      aold[] is the current activations in which to place the new
 *       input values
 *      tick is the current tick in the current sweep
 *      flag specifies that the inputs should be set to zero after
 *       the first tick in a sweep (-I switch)
 *      maxtime is the number of input vectors in the .data file
 *      local is the input array for localist inputs
 * Post: The first time this function is called, it loads the .data file
 *        and sets maxtime.  On subsequest executions, it loads the next
 *        input vector from the sequence into either aold[] or local[],
 *        depending on the global flag localist.
 */

void update_targets(  OUT float atarget[], IN long time, IN int tick,
                      IN int flag, IN long *maxtime);
/* -- Function update_targets
 * Pre: the global variable root[] contains the root filename
 *      atarget[] is the pre-allocated array of targets to update
 *      time is the index of the current input vector in the .data file
 *      tick is the current tick in the current sweep
 *      flag specifies that the targets should be set to zero after
 *       the first tick in a sweep (-T switch)
 *      maxtime is the number of input vectors in the .data file
```

```
 * Post: The first time this function is called, it loads the .teach file.
 *        On subsequent executions, it updates the targets in atarget[] to
 *        match the current training vector represented by time.
 */

void update_reset( IN long time, IN int tick, IN int flag,
                   IN long *maxtime, OUT int *now);
/* -- Function update_reset
 * Pre: the global variable root[] contains the root filename
 *      time is the index of the current input vector in the .data file
 *      tick is the current tick in the current sweep
 *      flag specifies that resets should be loaded from the .reset file
 *       (-X switch)
 *      maxtime si the number of input vectors in the .data file
 * Post: The first time this funciton is called, it loads the .reset
 *        file if required.  On subsequent executions, it updates the flag
 *        now to indicate if the network activations should be reset
 *        immediately before this learning sweep.
 */

void update_weights(  IN OUT float awt[][], IN OUT float adwt[][],
                      OUT float awinc[][]);
/* -- Function update_weights
 * Pre: awt[][] contains the current array of network weights
 *      adwt[][] contains the desired weight deltas for this learning sweep
 *      awinc[][] is a pre-allocated 2D ARRAY[nn][nt]
 * Post: Non-fixed weights have been updated according to the global
 *        learning rate rate and the global learning momentum momentum.
 *       Group weights have been averaged together.
 *       Limited weights have had their limits enforced.
 *       winc[][] contains the actual weight changes
 *       awt[][] contains the new weights
 *       adwt[][] has been zeroed
 */
```

activate.c

Function prototypes

```
void act_nds(  IN float aold[], OUT float amem[], OUT float anew[],
               IN float awt[][], IN int *local, IN float atarget[]);
/* -- Function act_nds
 * Pre: aold[] contains the network activations for the previous time step
 *      amem[] is an allocated ARRAY[nt]
 *      anew[] is an allocated ARRAY[nt]
 *      awt[][] contains the current network weights
 *      local is a FLAG: the input is in localist format
 *      atarget[] has been loaded with the current learning targets
 * Post: All network activations have been updated and placed in anew[]
 *       aold[] has been copied to amem[]
 *       learning targets have been fed back if the global flag teacher is
 *        set (-t switch)
 */
```

compute.c

Function prototypes

```
void comp_errors(  INT float aold[], INT float atarget[],
                   OUT float aerror[], IN OUT float *e,
                   IN OUT float *ce_e);
/* -- Function comp_errors
 * Pre: aold[] contains the current network activations
 *      atarget[] contains the current learning targets
 *      aerror[] is an allocated ARRAY[nn]
```

```
 *      e contains the accumulated error
 *      ce_e contains the accumulated cross-entropy error
 * Post: The error signals from atarget to aold have been calculated,
 *        and placed in aerror
 *       Either e has been updated or ce_e has been updated, depending
 *        on the global flag ce
 */

void comp_deltas(  IN OUT float apold[][][], OUT float apnew[][][],
                   IN float awt[][], IN OUT float adwt[][],
                   IN float aold[], IN float anew[], IN float aerror[]);
/* -- Function comp_deltas
 * Pre: apold[][][] contains the current p-variables for the network
 *      apnew[][][] is a pre-allocated 3D ARRAY[nn][nt][nn]
 *      awt[][] contains the current network weights
 *      adwt[][] contains the accumulated weight deltas
 *      aold[] contains the previous network activations
 *      anew[] contains the new network activations
 *      aerror[] contains the calculated error signals for the network
 * Post: apold[][][] has been updated
 *       apnew[][][] has been updated
 *       adwt[][] contains the updated weight deltas
 * Note: I don't really know what this function does.  Is it part of the
 *        Williams-Zipser algorithm for temporally recurrent learning?
 */

void comp_backprop(  IN float awt[][], IN OUT float adwt[][],
                     IN float aold[], IN float amem[],
                     IN float atarget[], OUT float aerror[],
                     IN int *local);
/* -- Function comp_backprop
 * Pre: awt[][] contains the current network weights
 *      adwt[][] contains the accumulated weight deltas
 *      aold[] contains the current network activations
 *      amem[] contains the previous network activations
 *      atarget[] contains the current learning targets
 *      aerror[] is a pre-allocated ARRAY[nn]
 *      local is a FLAG indicating that the inputs are in localist format
 * Post: The error signals from atarget[] to aold[] have been calculatd and
 *        placed in aerror[]
 *       The back-propagation learning algorithm has been used to
 *        calculate the desired weight deltas, which have been placed
 *        in adwt[][]
 */
```

arrays.c

Function prototypes

```
void make_arrays(void);
/* -- Function make_arrays
 * Pre: The network configuration file has been parsed
 * Post These global arrays have been allocated:
 *       zold[nt], zmem[nt], znew[nt]
 *       target[no], error[nn], outputs[no]
 *       selects[nt], linput[ni],
 *       wt[nn][nt], dwt[nn][nt], winc[nn][nt],
 *       cinfo[nn][nt], ninfo[nn]
 */

void make_parrays(void);
/* -- Function make_parrays
 * Pre: The network configuration file has been parsed
 * Post: These global arrats have been allocated:
 *        pold[nn][nt][nn], pnew[nn][nt][nn]
```

```
 */
```

weights.c

Function prototypes

```
void save_weights(void);
/* -- Function save_weights
 * Pre: The global variable root[] contains the root filename
 *      The global array wt[][] contains the network weights
 *      The global varibale sweep contains the current sweep number
 * Post: The network weights were saved to a file "root[].sweep.wts",
 *        in a manner that can be loaded by load_wts
 */

void load_wts(void);
/* -- Function load_wts
 * Pre: The global variable loadfile[] contains the filename of the saved
 *       state to load
 *      The global array wt[][] has been allocated
 * Post: The saved weights were loaded into wt[][]
 */
```

subs.c

Exported variables

/* Exp function lookup table */

float *__exp_array__; /* table for lookup */

Function prototypes

```
float rans(IN float w);
/* -- Function rans
 * Pre: w contains a range to generate random numbers: [-w..w]
 * Post: A random number in the range [-w..w] was returned
 */

void exp_init(void);
/* -- Function exp_init
 * Pre: TRUE
 * Post: The global array exp_table[] was allocated
 *       The data from the exp lookup file was read into exp_table[]
 */

void print_nodes(IN float aold[]);
/* -- Function print_nodes
 * Pre: aold[] contains the current network activations
 * Post: The network activations have been written to stdout
 */

void print_output(IN float aold[]);
/* -- Function print_output
 * Pre: aold[] contains the current network activations
 * post: The network outputs have been written to stdout
 */

void print_error(IN float *e);
/* -- Function print_error
 * Pre: The global variable root[] contains the root filename
 *      e contains the current accumulated error
 * Post: The first time this functon is called, it opens and initialises
 *        the error output file "root[].err".  On subsequent executions,
 *        it calculates either the RMS error or the cross-entropy error,
 *        depending on the global flag ce, and writes this error to the
```

```
 *        file.
 */

void reset_network(OUT float aold[], OUT float anew[],
                   OUT float apold[][][], OUT float apnew[][][]);
/* -- Function reset_network
 * Pre: aold[][] is a pre-allocated ARRAY[nn]
 *      anew[][]is a pre-allocated ARRAY[nn]
 *      apold[][][] is a pre-allocated 3D ARRAY[nn][nt][nn]
 *      apnew[][][]is a pre-allocated 3D ARRAY[nn][nt][nn]
 * Post: All cells in all parameter arrays have been zeroed
 */

void reset_bp_net(OUT float aold[], OUT float anew[]);
/* -- Function reset_bp_net
 * Pre: aold[] is a pre-alocated ARRAY[nn]
 *      anew[]is a pre-alocated ARRAY[nn]
 * Post: All cells in all parameter arrays have been zeroed
 */
```

exp.c

This stand-alone program generates a lookup table for the **exp** function, and writes it to **stdout**. When piped to a file, this is read by **init_exp** in *subs.c* to accelerate tlearn’s calculation of network activations.