1. A backprop network for phoneme recognition. This program is not designed for the ultimate in neural network design flexibility. At present, the only configurations supported are fully-connected feed-forward types. There is enough flexibility in the node design to implement other types of networks without too much trouble, I hope. The basic program outline follows. In addition to the standard I/O header file, include the math and library header files.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
  ⟨ Global Definitions 4 ⟩
  ⟨ Function Definitions 2 ⟩
  ⟨ Main Program 5 ⟩
```

2. The squashing function $\phi(v)$ is the hyperbolic tangent which produces values ranging from -1 to 1. The function is scaled up slightly by factor A, and the central slope is modified by B:

$$\phi(v) = A\tanh(Bv) = A\left(\frac{e^{Bv} - e^{-Bv}}{e^{Bv} + e^{-Bv}}\right)$$

Constants A and B are taken from *Neural Networks*, page 160, in the discussion about improving backprop performance. Additionally, the value ϵ represents the difference between the maximum output value possible and the target value. There be magic here.

```
#define A ((double) 1.716)
#define B ((double) 2.0/3.0)
#define EPSILON ((double) 0.716)
```

⟨ Function Definitions 2 ⟩ ≡

```
  double phi(float v)
  {
    double eval_pos, eval_neg;
    eval_pos ← exp((double)(B * v));
    eval_neg ← exp((double)(−B * v));
    return A * (eval_pos − eval_neg)/(eval_pos + eval_neg);
  }
```

See also sections 3, 9, 10, 11, 14, 16, 18, 22, 23, 26, 27, 28, 29, 34, 35, 37, 39, 40, 41, 43, 44, 45, 46, 47, 48, 50, 52, and 53.

This code is used in section 1.

3. The derivative of the output (squashing) function is also required for back propagation learning. The constants A and B are as before.

$$\phi'(v) = \frac{d}{dv}A\tanh(Bv) = AB\,\mathrm{sech}^2(Bv) = \frac{4AB}{(e^{Bv} + e^{-Bv})^2}.$$

⟨ Function Definitions 2 ⟩ +≡

```
  double phi_prime(float v)
  {
    double bv;
    double e_val;
    bv ← B * v;
    e_val ← exp(bv) + exp(−bv);
    return (4.0 * A * B)/(e_val * e_val);
  }
```

4. Basic Node Structure. A fully-connected feed-forward network means that every node's output in one layer is distributed to every node of the next layer. This also means that recipricating pointers from upper nodes to lower nodes will be fully-connected for purposes of the back-propagation algorithm. For input layer nodes, the only data space required is for the output value and the forward node pointers. Nodes in all other layers will require a set of weights for incoming connections, a threshhold value, and an activation value. For back-propagation it is convenient for nodes to have a space for the node's local gradient. Nodes also have local values for learning rate and momentum terms, but these are not currently varied on a node-by-node basis.

Architectural Notes:

1. Since layers are fully-connected, forward and backward pointers may be shared by an entire layer. Separate these pointers from the node proper so that they can be manipulated external to the node and shared amongst nodes of the same layer (i.e., one set of forward and backward pointers for each layer).
2. The learning rule (weight adjustment) uses a term for momentum which requires that weights of a previous iteration need to be stored (i.e., two sets of weights for each node).
3. I am using a trainable threshhold value. I choose to deal with this value as a separate entity, rather than as part of the weight vector, even though the threshhold behaves as a weight for all intents and purposes.
4. Since many node parameters do not pertain to the input layer, I have chosen to separate these out to save some space given that I intend to have a lot of input nodes.
5. I have chosen to create an array containing all the nodes. Each node then has an index, and these indices are the node pointers.

⟨ Global Definitions 4 ⟩ ≡

```
struct node_pointer {
  int n_nodes;   /* number of pointers */
  int *ptrs;   /* node element numbers */
};
struct upper {
  int nweights;   /* Number of weights */
  float *weight, *last_weight;   /* Weight vectors */
  float threshhold, old_t;   /* Threshhold values */
  float activation;   /* Activation value */
  float eta, alpha;   /* Learning rate and momentum */
  float delta;   /* Local gradient */
  struct node_pointer *backward;   /* pointers to the previous layer */
};
struct node {
  float output;   /* Output (squashed from activation) */
  struct upper *u;   /* Additional params for non-input nodes */
  struct node_pointer *forward;   /* pointers to the next layer */
};
```

See also sections 12, 13, 15, 17, 21, and 49.

This code is used in section 1.

5. The Main Program. Most of the program function will be discussed in later sections. All of the program's run-time configuration happens by means of parameters stored in what I am calling an experiment file. In this file is described the network configuration, the training datasets to be used, the criteria for stopping training, learning rates, etc.

One very convenient feature of this program is that most of the network state is saved periodically (if specified by experiment file parameters). At startup, part of the initialization involves reading the network state back in from saved files. In this way, training can be halted so that new datasets may be added, connection weights can be adjusted, and learning rates may be tweaked, and then the network may be re-started, resuming training where it left off.

```
⟨ Main Program 5 ⟩ ≡
  main(int argc, char *argv[])
  {
    ⟨ Local Definitions 6 ⟩
    ⟨ Command line argument processing 7 ⟩
    ⟨ Initialization 20 ⟩
    ⟨ Network Training 31 ⟩
    ⟨ Free Run Testing 51 ⟩
  }
```

This code is used in section 1.

6. Declare a file descriptor for reading the experiment file, and a character pointer for saving the program invocation name.

```
⟨ Local Definitions 6 ⟩ ≡
  FILE *paramfile;
  char *progname;
  int i, j;
```

See also section 32.

This code is used in section 5.

7. The only command-line argument to process is the name of the experiment file. All other run-time configuration information is read from there. Save the program file name for error reporting, and then open the named experiment file. Call the function for scanning the file.

```
⟨ Command line argument processing 7 ⟩ ≡
  progname ← *argv;    /* Save the file name by which we were invoked */
  argc−−;    /* Skip past the invocation name. */
  argv++;
  if (¬argc) {    /* Need at least one argument... */
    fprintf(stderr, "Usage:␣␣%s␣experiment_file_name\n", progname);
    exit(1);
  }
  if ((paramfile ← fopen(*argv, "r")) ≡ Λ) {
    fprintf(stderr, "Unable␣to␣open␣experiment␣file␣\"%s\"\n", *argv);
    exit(1);
  }
  parse_exp_file(paramfile);    /* Parse the experiment file */
  fclose(paramfile);    /* Our work is done here */
```

This code is used in section 5.

8. The Experiment File. Within this beast are the parameters for defining the network, the training datasets, the cross-validation files, the stopping criteria, the output files, and any other tweakable parameter I may want to whip up. In a different lifetime, these values might be entered via X-friendly control panels, but not now. I don't want to build a full-blown parser, so the format of the file is not too flexible.

Comment lines begin with a '`#`' character in typical UNIX style. Keywords may occur one per line with optional leading white space. Most keywords expect values which are specified by an '`=`' followed by the string, integer, or float value. Some keywords expect lists of values, in which case the keyword value is the number of elements in the list. The list items are then specified on succeeding lines, one value per line. The following table defines the currently recognized keywords and their values. Keywords marked by '*' are required. The "PRM" column is the type of the keyword parameter, and the "LIST" column is the type of the list items.

KEYWORD	PRM	LIST	DESCRIPTION
`*layers`	*int*	*int*	The number of layers in the network. The list that follows contains the number of nodes in each layer, from the input layer to the output layer. This keyword may be omitted if a checkpoint file exists, in which case the network description is taken from there.
`learning rate`	*float*	—	The initial learning rate assigned to each node. Defaults to 0.01.
`momentum`	*float*	—	The initial momentum term assigned to each node. Defaults to 0.8.
`rate adj epochs`	*int*	—	The number of epochs after which the learning rate and momentum terms are scaled. Disabled by default.
`scaling factor`	*float*	—	The amount by which to scale the learning rate and momentum term after a certain number of epochs.
`checkpoints`	*int*	—	The number of training epochs after which the checkpoint file is written. If *checkpoints* = *0*, then checkpointing is disabled.
`checkpoint file`	*string*	—	The checkpoint file's name. If this file exists before training begins, then the previous network's weights and learning rates are loaded and training will resume with the the next epoch after the last checkpoint.
`input frames`	*int*	—	The number of dataset input frames that constitute the inputs to the network. The number of nodes in the network's input layer should be this multiple of the datasets' *framesize*. Defaults to 1.
`input bias`	*float*	—	This value is added to frame file input values before scaling. This may be used to shift data from the interval $[0, 1]$ to $[-1, 1]$. Defaults to 0.0.
`input scale`	*float*	—	This value scales input data after the bias is added. This may be used to shift data from the interval $[0, 1]$ to $[-1, 1]$. Defaults to 1.0.
`*training files`	*int*	*string*	The list of strings are the filenames of the input datasets. The input datasets are annotated frame files from which sequences are drawn for training the network.
`training err file`	*string*	—	The filename where training error is recorded after every epoch.
`validation files`	*int*	*string*	The list of strings are the filenames of the cross-validation datasets. These are annotated frame files from which sequences are drawn for testing network generalization.
`validation err file`	*string*	—	The filename where cross-validation error is recorded after every epoch. This keyword is ignored if there are no validation files specified.
`max epochs`	*int*	—	The maximum number of epochs for which to train.
`*output strings`	*int*	*string*	The output strings represented by each output node. The first element of the list should be the string that is output when no output node is active. The number of strings should equal *output nodes* + 1.

9. Search through a list of keywords for a match to some string s. If found, return the value associated with that string. If not found, return a -1. If keywords are added to the list that have substrings in common, be sure that the longer keyword is entered into the list before the shorter keyword. Function *getkw*(s) returns the value associated with the keyword s or -1 if s is not a keyword. *getkeystr*(n) returns a pointer to the keyword string associated with value n or the Λ-string if n does not represent a keyword value.

```
#define KEY_LAYERS 1
#define KEY_FRAMES 2
#define KEY_STRINGS 3
#define KEY_TFILES 4
#define KEY_GFILES 5
#define KEY_CHKPT 6
#define KEY_EPOCHS 7
#define KEY_CHKFILE 8
#define KEY_ETA 9
#define KEY_ALPHA 10
#define KEY_RSCALE 11
#define KEY_REPOCH 12
#define KEY_EFILE 13
#define KEY_VFILE 14
#define KEY_IBIAS 15
#define KEY_ISCALE 16
```

⟨ Function Definitions 2 ⟩ +≡

```
struct kw {
  char *ks;
  int kv;
} keywords[] ← {{"layers",KEY_LAYERS},{"input␣frames",KEY_FRAMES},{"input␣bias",KEY_IBIAS},
    {"input␣scale",KEY_ISCALE},{"output␣strings",KEY_STRINGS},{"training␣files",
    KEY_TFILES},{"validation␣files",KEY_GFILES},{"checkpoint␣file",KEY_CHKFILE},
    {"checkpoints",KEY_CHKPT},{"max␣epochs",KEY_EPOCHS},{"learning␣rate",KEY_ETA},
    {"momentum",KEY_ALPHA},{"scaling␣factor",KEY_RSCALE},{"rate␣adj␣epochs",KEY_REPOCH},
    {"training␣err␣file",KEY_EFILE},{"validation␣err␣file",KEY_VFILE},{"",-1}};
int getkw(char *s)
{
  struct kw *k ← keywords;
  extern char *skipwhite();
  s ← skipwhite(s);
  while (strlen(k→ks) > 0) {
    if (strncmp(k→ks, s, strlen(k→ks)) ≡ 0) return k→kv;
    k++;
  }
  return −1;
}
char *getkeystr(int keyval)
{
  struct kw *k ← keywords;
  while (strlen(k→ks) > 0) {
    if (k→kv ≡ keyval) return k→ks;
    k++;
  }
  return "";
}
```

10. Define some auxiliary functions for parsing the experiment file. *skipwhite*( ) returns a pointer to the first non-white space character. *getval*( ) retrieves the **double** float value following the '=' character, which may be cast to type **int**. *getstring*( ) returns the string beginning with the first non-white space character following a '=' and terminated by white space.

```
⟨Function Definitions 2⟩ +≡
  char *skipwhite(char *ptr)
  {
    while (*ptr ≡ '␣' ∨ *ptr ≡ '\t' ∨ *ptr ≡ '\n') ptr++;
    return ptr;
  }
  double getval(char *ptr, int key)
  {
    while (*ptr ≠ '\0' ∧ *ptr ≠ '=') ptr++;
    if (*ptr ≠ '=') {
      fprintf(stderr, "Missing␣'='␣for␣keyword␣\"%s\"\n", getkeystr(key));
      return −1;
    }
    ptr ← skipwhite(++ptr);
    if ((*ptr ≥ '0' ∧ *ptr ≤ '9') ∨ *ptr ≡ '-' ∨ *ptr ≡ '.') return atof(ptr);
    else {
      fprintf(stderr, "Missing␣value␣after␣'='␣for␣keyword␣\"%s\"\n", getkeystr(key));
      return (double) −1.0;
    }
  }
  char *getstring(char *ptr, int key)
  {
    char *tmp;
    while (*ptr ≠ '\0' ∧ *ptr ≠ '=') ptr++;
    if (*ptr ≠ '=') {
      fprintf(stderr, "Missing␣'='␣for␣keyword␣\"%s\"\n", getkeystr(key));
      return Λ;
    }
    ptr ← skipwhite(++ptr);
    tmp ← (char *) malloc(strlen(ptr) + 1);
    strcpy(tmp, ptr);
    ptr ← tmp + strlen(tmp) − 1;
    while ((*ptr ≡ '\n' ∨ *ptr ≡ '␣' ∨ *ptr ≡ '\t') ∧ ptr ≥ tmp) ptr−−;
    if (ptr < tmp) {
      fprintf(stderr, "Missing␣a␣string␣following␣keyword␣\"%s\"\n", getkeystr(key));
      free(tmp);
      return Λ;
    }
    else {
      *(++ptr) ← '\0';
      return tmp;
    }
  }
```

11. Read the experiment file, skipping lines beginning with '#' characters. Keywords are specified one per line. Keywords expecting lists call auxiliary routines.

```
⟨ Function Definitions 2 ⟩ +≡
  parse_exp_file(FILE *file)
  {
    char theline[256], *ptr;
    int i, fatal_count ← 0, keyval;
    fgets(theline, sizeof (theline), file);
    while (¬feof(file)) {
      ptr ← skipwhite(theline);
      if (*ptr ≠ '#') {      /* Ignore any comment lines */
        keyval ← getkw(ptr);
        switch (keyval) {
        case KEY_LAYERS:
          if ((layers ← (int) getval(ptr, KEY_LAYERS)) < 0) fatal_count++;
          else do_layers(file, layers, &layer_spec);
          break;
        case KEY_FRAMES:
          if ((input_frames ← (int) getval(ptr, KEY_FRAMES)) < 0) fatal_count++;
          break;
        case KEY_STRINGS:
          if ((output_strings ← (int) getval(ptr, KEY_STRINGS)) < 0) fatal_count++;
          else do_output_strings(file, output_strings);
          break;
        case KEY_TFILES:
          if ((training_sets ← (int) getval(ptr, KEY_TFILES)) ≤ 0) fatal_count++;
          else do_datasets(file, training_sets, &tsets);
          break;
        case KEY_GFILES:
          if ((cross_sets ← (int) getval(ptr, KEY_GFILES)) < 0) fatal_count++;
          else do_datasets(file, cross_sets, &csets);
        case KEY_CHKPT:
          if ((checkpoint ← (int) getval(ptr, KEY_CHKPT)) < 0) {
            fprintf(stderr, "Bad␣check␣point␣value:␣%d.␣Checkpoint␣disabled\n", checkpoint);
            checkpoint ← 0;
          }
          break;
        case KEY_EPOCHS:
          if ((maximum_epochs ← (int) getval(ptr, KEY_EPOCHS)) < 0) {
            fprintf(stderr, "Bad␣max␣epochs␣value:␣%d.␣Parameter␣disabled\n", maximum_epochs);
            maximum_epochs ← 0;
          }
          break;
        case KEY_CHKFILE:
          if ((chkpt_file ← getstring(ptr, KEY_CHKFILE)) ≡ Λ) {
            fprintf(stderr, "Bad␣checkpoint␣file␣name.␣␣Checkpoint␣disabled.\n");
            checkpoint ← 0;
          }
          break;
        case KEY_ETA:
          if ((eta ← getval(ptr, KEY_ETA)) ≤ 0.0) {
            fprintf(stderr, "Bad␣learning␣rate:␣%g.␣Using␣default␣%g.\n", eta, ETA);
```

```
            eta ← ETA;
          }
          break;
        case KEY_ALPHA:
          if ((alpha ← getval(ptr, KEY_ALPHA)) < 0.0) {
            fprintf(stderr, "Bad␣momentum␣term:␣%g.␣Using␣default␣%g", alpha, ALPHA);
          }
          break;
        case KEY_RSCALE:
          if ((rate_factor ← getval(ptr, KEY_RSCALE)) < 0.0) {
            fprintf(stderr, "Bad␣rate␣scaling␣factor:␣%g.␣␣Disabling.\n", rate_factor);
            rate_factor ← 1.0;
          }
          break;
        case KEY_REPOCH:
          if ((rate_reduction ← (int) getval(ptr, KEY_REPOCH)) < 0) {
            fprintf(stderr, "Bad␣rate␣adj␣epochs␣value:␣%d.␣␣Disabling.\n", rate_reduction);
            rate_reduction ← 0;
          }
          break;
        case KEY_EFILE: terr_file ← getstring(ptr, KEY_EFILE);
          break;
        case KEY_VFILE: verr_file ← getstring(ptr, KEY_VFILE);
          break;
        case KEY_IBIAS: input_bias ← getval(ptr, KEY_IBIAS);
          break;
        case KEY_ISCALE: input_scale ← getval(ptr, KEY_ISCALE);
          if (input_scale ≡ 0.0) {
            fprintf(stderr, "Bad␣input␣scaling␣value:␣%g.␣Resetting␣to␣1.0\n", input_scale);
            input_scale ← 1.0;
          }
          break;
        default: ptr += strlen(ptr) − 1;
          while ((*ptr ≡ '\n' ∨ *ptr ≡ '␣' ∨ *ptr ≡ '\t') ∧ ptr ≥ theline) ptr −−;
          *(ptr + 1) ← '\0';
          ptr ← skipwhite(theline);
          if (*ptr) fprintf(stderr, "Unrecognized␣experiment␣file␣keyword:␣\"%s\"\n", ptr);
        }
      }
      fgets(theline, sizeof (theline), file);
    }
    if (fatal_count) {
      fprintf(stderr, "%d␣fatal␣errors␣parsing␣experiment␣file.\n", fatal_count);
      exit(2);
    }
    ⟨Print experiment file verification 19⟩
  }
```

12. And now for some miscellaneous definitions of variables read from the experiment file:

```
⟨ Global Definitions 4 ⟩ +≡
  int checkpoint;   /* Update the checkpoint file every checkpoint epochs */
  char *chkpt_file;   /* Checkpoint file name */
  int epoch;   /* epoch counter */
  int input_frames ← 1;   /* Number of input frames per pattern */
  int maximum_epochs ← 0;   /* epoch stopping criterion */
  char *terr_file;   /* Training error file name */
  char *verr_file;   /* Cross-validation error file name */
  int rate_reduction ← 0;   /* scale learning after this many epochs */
  float rate_factor ← 1.0;   /* rate scaling factor */
  float input_bias ← 0.0;   /* Pre-scaling bias of input data values */
  float input_scale ← 1.0;   /* Post-bias scaling of input data values */
```

13. Define space for the layer definitions. *layer_spec* will eventually become an array with *layers* elements, each of which will contain the number of nodes in the layer. Layers are defined from input (element zero) to output. Start by defining some symbolic values for referencing particular layers.

```
#define INPUT_LAYER 0
#define HIDDEN_LAYER 1
#define OUTPUT_LAYER (layers − 1)
⟨ Global Definitions 4 ⟩ +≡
  int layers;
  int *layer_spec;
```

14. The *do_layers*( ) function will finish up the layers parsing by reading from the experiment file the number of nodes for every layer. For now, this information is just stored for later in the variable *layer_spec*, which has space allocated to it here. Ignore comment lines.

```
⟨ Function Definitions 2 ⟩ +≡
  do_layers(FILE *file, int nlayers, int **layer_spec)
  {
    int i;
    char theline[128];
    *layer_spec ← (int *) calloc(nlayers, sizeof(int));
    i ← 0;
    while (i < nlayers) {
      fgets(theline, sizeof (theline), file);
      if (feof(file)) break;
      if (theline[0] ≠ '#') {
        (*layer_spec)[i] ← atoi(theline);
        i++;
      }
    }
  }
```

15. For displaying the output of the network, define a list of strings which will be assigned to each output node. When the output node is active, its string will be displayed. Variable *output_strings* has the number of outputs strings stored, and *node_strings* will contain the list of actual strings.

```
⟨ Global Definitions 4 ⟩ +≡
  int output_strings;
  char **node_strings;
```

16. Read the node-specific output strings from the experiment file. Comments are not allowed in this section since '#' characters may be the desired output for a node.

```
⟨ Function Definitions 2 ⟩ +≡
  do_output_strings(FILE *file, int nstrings)
  {
    int i;
    char theline[256];
    node_strings ← (char **) calloc(nstrings, sizeof(char *));
    i ← 0;
    while (i < nstrings) {
      fgets(theline, sizeof (theline), file);
      if (feof(file)) break;
      node_strings[i] ← (char *) malloc(strlen(theline) − 1);
      theline[strlen(theline) − 1] ← '\0';    /* wipe the linefeed char */
      strcpy(node_strings[i], theline);
      i++;
    }
  }
```

17. Define structures for storing training set and cross validation set data. Very little needs to be maintained here. Data frames consist of a target value to be learned by the network, and *framesize* input parameters.

```
⟨ Global Definitions 4 ⟩ +≡
  struct frame {
    int target;
    float *input;
  };
  struct dataset {
    char *filename;
    int framesize;
    int nframes;
    struct frame *frames;
  } *tsets, *csets;
  int training_sets, cross_sets;
```

18. Read the the list of training datasets from the experiment file. For now just read in the file names. Comment lines are allowed in this section, so file names beginning with '#' characters won't work unless whitespace occupies the first character of the line. Leading whitespace is deleted, and the filename is terminated by the first whitespace encountered.

```
⟨Function Definitions 2⟩ +≡
  do_datasets(FILE *file, int nsets, struct dataset **setvar)
  {
    int i;
    char theline[256], *s1, *s2;    /* Allocate structures for the training set files. */
    *setvar ← (struct dataset *) calloc(nsets, sizeof(struct dataset));
    i ← 0;
    while (i < nsets) {
      fgets(theline, sizeof (theline), file);
      if (feof(stdin)) break;
      if (theline[0] ≠ '#') {
        s1 ← skipwhite(theline);    /* strip leading white space */
        (*setvar)[i].filename ← (char *) malloc(strlen(s1));
        s2 ← (*setvar)[i].filename;
        while (*s1 ≠ '␣' ∧ *s1 ≠ '\n' ∧ *s1 ≠ '\t') *s2++ ← *s1++;    /* Copy the filename */
        *s2 ← '\0';
        i++;
      }
    }
  }
```

19. Once the experiment file has been successfully parsed, give the poor user (me!) an opportunity to see what is going on.

⟨ Print experiment file verification 19 ⟩ ≡

```
printf("\n␣␣Experiment␣File␣Summary\n␣␣=======================\n\n");
if (layers) {
  printf("␣␣Network␣architecture:␣");
  for (i ← 0; i < layers; i++) printf("%d%s", layer_spec[i], i < layers − 1 ? "-" : "\n");
}
printf("␣␣Initial␣Learning␣Rate:␣%g␣␣Initial␣Momentum␣Term:␣%g\n", eta, alpha);
if (rate_reduction)
  printf("␣␣Learning␣rate␣reduced␣by␣factor␣%g␣after␣%d␣epochs.\n", rate_factor, rate_reduction);
printf("␣␣%d␣consecutive␣frames␣used␣for␣input\n", input_frames);
printf("␣␣Input␣bias:␣%g␣␣␣Input␣Scaling:␣%g\n", input_bias, input_scale);
if (maximum_epochs) printf("␣␣Maximum␣number␣of␣training␣epochs:␣%d\n", maximum_epochs);
if (output_strings) {
  printf("␣␣Node␣Output␣Strings␣(%d):\n", output_strings);
  for (i ← 0; i < output_strings; i += 4) {
    printf("␣␣␣␣%15-s␣%15-s␣%15-s␣%15-s\n", node_strings[i],
        i + 1 < output_strings ? node_strings[i + 1] : "", i + 2 < output_strings ? node_strings[i + 2] : "",
        i + 3 < output_strings ? node_strings[i + 3] : "");
  }
  printf("\n");
}
if (checkpoint) printf("␣␣Checkpoint:␣every␣%d␣epoch%s␣␣File:␣%s\n\n", checkpoint,
      checkpoint > 1 ? "s" : "", chkpt_file);
if (terr_file ≠ Λ) printf("␣␣Training␣error␣written␣to␣file␣\"%s\"\n", terr_file);
if (verr_file ≠ Λ ∧ cross_sets)
  printf("␣␣Generalization␣error␣written␣to␣file␣\"%s\"\n", verr_file);
if (training_sets) {
  printf("␣␣Training␣Datasets:\n");
  for (i ← 0; i < training_sets; i += 3) {
    printf("␣␣␣␣%20-s␣%20-s␣%20-s\n", tsets[i].filename,
        i + 1 < training_sets ? tsets[i + 1].filename : "", i + 2 < training_sets ? tsets[i + 2].filename : "");
  }
  printf("\n");
}
if (cross_sets) {
  printf("␣␣Cross-validation␣Datasets:\n");
  for (i ← 0; i < cross_sets; i += 3) {
    printf("␣␣␣␣%20-s␣%20-s␣%20-s\n", csets[i].filename, i + 1 < cross_sets ? csets[i + 1].filename : "",
        i + 2 < cross_sets ? csets[i + 2].filename : "");
  }
}
printf("\n");
```

This code is used in section 11.

20. Initialization. Once the experiment file has been parsed, all of the information should be available for initialization. At this point, the network may be built, the training sets may be read, and any checkpoint information or previous weight sets may be read back into the network. The complete initialization task involves building the network, reading any checkpoint initialization data, and finally reading the training and cross-validation datasets. If layers have not been defined by the experiment file, then we may assume that they are being defined instead by the checkpoint files. It is an error of none of these conditions hold.

```
⟨ Initialization 20 ⟩ ≡
  if (layers) build_network();
  if (checkpoint)
    if (input_checkpoint() < 0) exit(2);
  if (¬layers) {    /* If there are no layers by now, there's nothing to do. */
    fprintf(stderr, "No␣layers␣defined!\n");
    exit(3);
  }
  if (training_sets) input_datasets(training_sets, tsets);
  if (cross_sets) input_datasets(cross_sets, csets);
```

See also section 30.

This code is used in section 5.

21. All of the nodes for all of the layers will be kept in one contiguous array, *nodes*. A layer will be defined by a starting and ending index value into this array, kept in *layer_def*. Variable *total_nodes* will contain the number of nodes in the entire network. The variable *weights_set* is a flag which will be non-zero if weights have been read in from the checkpoint file.

```
#define ETA (0.01)
#define ALPHA (0.8)
⟨ Global Definitions 4 ⟩ +≡
  struct node *nodes;
  int total_nodes;
  int weights_set;
  float eta ← ETA, alpha ← ALPHA;
  struct layerdef {
    int begin;    /* starting index value of this layer */
    int end;    /* ending index value of this layer */
    int count;    /* count of nodes in this layer */
  };
  struct layerdef *layer_def;
```

22. Build the network from the number of nodes specified for each layer in *layer_specs*. A more detailed description of each layer is built in *layer_def*. Then connections are established between layers by creating the *forward* and *backward* arrays for each layer, which every node in the layer shares. Finally, each node must have its own unique weight arrays allocated to it. The global variable *total_nodes* is set in this function.

```
⟨ Function Definitions 2 ⟩ +≡
  build_network( )
  {
    int i, j;
    struct node_pointer *forw, *back;
    layer_def ← (struct layerdef *) calloc(layers, sizeof(struct layerdef));
      /* Obtain the total number of nodes in the network, and setup the layer descriptors */
    total_nodes ← 0;
    for (i ← 0; i < layers; i++) {
      if (layer_spec[i] ≤ 0) {
        fprintf(stderr, "Invalid␣layer␣specification:␣%d\n", layer_spec[i]);
        exit(2);
      }
      layer_def[i].begin ← total_nodes;
      total_nodes += layer_def[i].count ← layer_spec[i];
      layer_def[i].end ← total_nodes − 1;
    }
    nodes ← (struct node *) calloc(total_nodes, sizeof(struct node));
    for (i ← 0; i < layers; i++) {      /* Build the forward and backward node pointer arrays */
      if (i > INPUT_LAYER) {      /* Only for non-input layers */
        back ← (struct node_pointer *) malloc(sizeof(struct node_pointer));
        back→n_nodes ← layer_def[i − 1].count;
        back→ptrs ← (int *) calloc(back→n_nodes, sizeof(int));
        for (j ← 0; j < back→n_nodes; j++) back→ptrs[j] ← j + layer_def[i − 1].begin;
      } else back ← Λ;
      if (i < OUTPUT_LAYER) {      /* Only for non-output layers */
        forw ← (struct node_pointer *) malloc(sizeof(struct node_pointer));
        forw→n_nodes ← layer_def[i + 1].count;
        forw→ptrs ← (int *) calloc(forw→n_nodes, sizeof(int));
        for (j ← 0; j < forw→n_nodes; j++) forw→ptrs[j] ← j + layer_def[i + 1].begin;
      } else forw ← Λ;
      for (j ← layer_def[i].begin; j ≤ layer_def[i].end; j++) {
        nodes[j].forward ← forw;
        if (i > INPUT_LAYER) {
          nodes[j].u ← (struct upper *) malloc(sizeof(struct upper));
          nodes[j].u→backward ← back;
          nodes[j].u→threshhold ← 0.0;
          nodes[j].u→old_t ← 0.0;
          nodes[j].u→nweights ← back→n_nodes;
          nodes[j].u→weight ← (float *) malloc(back→n_nodes * sizeof(float));
          nodes[j].u→last_weight ← (float *) malloc(back→n_nodes * sizeof(float));
        } else nodes[j].u ← Λ;
      }
    }
    weights_set ← 0;
  }
```

23. The checkpoint file can be very important since it contains the weight arrays for every node at a particular stage of training. It will therefore be useful for pre-loading a network after a program or system crash before continued training or for production work once training is completed. It will also be useful in tracing the state of the network while training is happening.

A checkpoint file has the *layers* parameter in common with an experiment file, and I will also allow comments. Otherwise, the format of the file is limited. The first line of a checkpoint file must contain a line beginning with the characters "`#checkpoint`". If the network architecture has been specified by the experiment file already, then the same architecture must be represented by the checkpoint file or a fatal error is reported. Variables *chkpt_layers* and *chkpt_layer_spec* are the checkpoint file's counterpart of the experiment file's *layers* and *layer_spec*.

```
⟨ Function Definitions 2 ⟩ +≡
  input_checkpoint( )
  {
    FILE *file;
    char theline[256], *ptr;
    int i, j, a_node;
    int chkpt_layers, *chkpt_layer_spec;
    float thresh, eta, alpha;
    extern float floatvalue(char **s);
    if ((file ← fopen(chkpt_file, "r")) ≡ Λ) return 0;    /* No error, just nothing to read yet */
    fgets(theline, sizeof (theline), file);
    if (strncmp(theline, "#checkpoint", 11) ≠ 0) {
      fprintf(stderr, "Checkpoint␣file␣\"%s\"␣has␣invalid␣header.\n", chkpt_file);
      goto FileError;
    }
    while (theline[0] ≡ '#' ∧ ¬feof(file)) fgets(theline, sizeof (theline), file);
    ⟨ Retrieve the checkpoint layer spec 24 ⟩
    do fgets(theline, sizeof (theline), file); while (theline[0] ≡ '#' ∧ ¬feof(file));
    ptr ← skipwhite(theline);
    if (strncmp(ptr, "epoch", 5) ≠ 0) {
      fprintf(stderr, "Missing␣the␣\"epoch\"␣spec␣in␣checkpoint␣file.\n");
      goto FileError;
    }
    ptr += 5;
    epoch ← (int) getval(ptr, 0);
    do fgets(theline, sizeof (theline), file); while (theline[0] ≡ '#' ∧ ¬feof(file));
    ⟨ Read in the network nodes 25 ⟩
    fclose(file);
    return 0;
  FileError: fprintf(stderr, "Error␣in␣format␣of␣checkpoint␣file␣\"%s\"\n", chkpt_file);
    fclose(file);
    return −1;
  }
```

24. Reading and comparing the layer specification is slightly involved, so I separated it out here. There are several scenarios to consider. If we got to this point, then the checkpoint file is not empty: it should therefore contain a valid layer specification, in which case *chkpt_layers* > 0, and a valid number of nodes will be specified for each layer. If the experiment file did not have a valid layer specification (i.e., *layers* ≤ 0), then the checkpoint file should override the experiment file. If the experiment file *does* contain a valid network specification, then it must agree exactly with what is located in the checkpoint file. If it does not, then an error condition exists, and the program should abort.

```
⟨ Retrieve the checkpoint layer spec 24 ⟩ ≡
  ptr ← skipwhite(theline);
  if (strncmp(ptr, "layers", 6) ≠ 0) {
    fprintf(stderr, "Missing␣\"layers\"␣spec␣in␣checkpoint␣file.\n");
    goto FileError;
  }
  ptr += 6;
  chkpt_layers ← (int) getval(ptr, 0);
  if (chkpt_layers ≤ 0) {
    fprintf(stderr, "Bad␣or␣missing␣checkpoint␣layers␣value\n");
    goto FileError;
  }
  else {    /* Valid chkpt_layers */
    do_layers(file, chkpt_layers, &chkpt_layer_spec);
        /* Do sanity checks between experiment file definition and checkpoint file definition of the
        network */
    if (layers > 0 ∧ (chkpt_layers ≠ layers)) {
      fprintf(stderr, "Experiment/Checkpoint␣network␣architecture␣disagree.\n");
      goto FileError;
    }
    if (layers ≡ 0) {    /* Since there was apparently no network spec given by the experiment file,
          rebuild the network from the checkpoint file network spec. */
      layers ← chkpt_layers;
      layer_spec ← chkpt_layer_spec;
      build_network();
    } else {    /* See that the rest of the network description agrees */
      for (i ← 0; i < chkpt_layers; i++) {
        if (chkpt_layer_spec[i] ≠ layer_spec[i]) {
          fprintf(stderr, "Experiment/Checkpoint␣network␣architecture␣disagree.\n");
          goto FileError;
        }
      }
    }
  }
```

This code is used in section 23.

25. Once the network is built, space will have been allocated to a node for its weight vector. At this point, we can scan the rest of the checkpoint file for node descriptors and weights. A one-line description of the node's state is given by its number, followed by a threshhold value, followed by its learning rate, and finally by the momentum term. All subsequent lines contain values for the weight vector. The number of weights must equal the number of nodes in the previous layer. There is not enough error checking in the reading of weights. We'll just have to wait for the next release.

```
⟨ Read in the network nodes 25 ⟩ ≡
  i ← layer_def[HIDDEN_LAYER].begin;
  while (¬feof(file) ∧ i < total_nodes) {
    if (sscanf(theline, "n␣%d␣t␣%g␣eta␣%g␣alpha␣%g", &a_node, &thresh, &eta, &alpha) ≠ 4 ∨ i ≠ a_node)
      {
      fprintf(stderr, "Scanning␣nodes␣from␣checkpoint␣file␣out␣of␣sync.\n");
      fprintf(stderr, "␣␣Last␣file␣node␣id:␣%d,␣current␣node␣id:␣%d\n", a_node, i);
      fprintf(stderr, "␣␣Reading␣line:␣%s\n", theline);
      goto FileError;
    }
    nodes[i].u→old_t ← nodes[i].u→threshhold ← thresh;
    nodes[i].u→eta ← eta;
    nodes[i].u→alpha ← alpha;
    ptr ← theline + strlen(theline);      /* Get to end of line */
    for (j ← 0; j < nodes[i].u→nweights; j++) {
      if (*ptr ≡ '\0') {      /* EOL, refill the buffer */
        do fgets(theline, sizeof (theline), file); while (¬feof(file) ∧ theline[0] ≡ '#');
        ptr ← skipwhite(theline);
      }
      nodes[i].u→last_weight[j] ← nodes[i].u→weight[j] ← floatvalue(&ptr);
    }
    do fgets(theline, sizeof (theline), file); while (theline[0] ≡ '#' ∧ ¬feof(file));
    i++;
  }
  if (feof(file) ∧ i < total_nodes) {
    fprintf(stderr, "Insufficient␣nodes␣recorded␣in␣checkpoint␣file.\n");
    fprintf(stderr, "␣␣Last␣file␣node␣id:␣%d,␣current␣node␣id:␣%d\n", a_node, i);
    fprintf(stderr, "␣␣Reading␣line:␣%s\n", theline);
    goto FileError;
  }
  weights_set ← 1;      /* Set the flag to indicate valid weights have been loaded. */
```

This code is used in section 23.

26. Define function for retrieving successive weight values from a checkpoint file. It is assumed that $*ptr$ is positioned at the beginning of the next numeric value. After the value is scanned, $*ptr$ is updated to point at the next non-white space character or `'\0'`.

```
⟨ Function Definitions 2 ⟩ +≡
  float floatvalue(char **ptr)
  {
    float thevalue;
    thevalue ← (float) atof(*ptr);
    while (**ptr ≠ '␣' ∧ **ptr ≠ '\n' ∧ **ptr ≠ '\t' ∧ **ptr ≠ '\0') (*ptr)++;
    *ptr ← skipwhite(*ptr);      /* Chew up trailing white space */
    return thevalue;
  }
```

27. While we're dealing with checkpoint files, we should go ahead and define the checkpoint file writing routine. This routine is complete atomic: it opens and closes the checkpoint file with each invocation in case of program crashes.

```
#define WTS_PER_LINE 7

⟨ Function Definitions 2 ⟩ +≡
  write_checkpoint(int epoch)
  {
    FILE *file;
    int i, j, count;
    if ((file ← fopen(chkpt_file, "w")) ≡ Λ) {
      fprintf(stderr, "Could␣not␣open␣checkpoint␣file␣\"%s\"␣for␣writing.\n", chkpt_file);
      fprintf(stderr, "Checkpointing␣disabled!\n");
      checkpoint ← 0;
      return;
    }
    fprintf(file, "#checkpoint␣file\n");
    fprintf(file, "layers␣=␣%d\n", layers);
    for (i ← 0; i < layers; i++) fprintf(file, "%d\n", layer_spec[i]);
    fprintf(file, "#␣epoch␣error␣=␣%lg\n", epoch_error);
    fprintf(file, "epoch␣=␣%d\n", epoch);
    i ← layer_def[HIDDEN_LAYER].begin;
    while (i < total_nodes) {
      fprintf(file, "n␣%d␣t␣%g␣eta␣%g␣alpha␣%g␣delta␣%g\n", i, nodes[i].u→threshhold, nodes[i].u→eta,
          nodes[i].u→alpha, nodes[i].u→delta);
      if (nodes[i].u→nweights) {
        count ← 0;
        for (j ← 0; j < nodes[i].u→nweights; j++) {
          fprintf(file, "%g␣", nodes[i].u→weight[j]);
          count++;
          if (count ≥ WTS_PER_LINE) {
            fprintf(file, "\n");
            count ← 0;
          }
        }
        if (count) fprintf(file, "\n");
      }
      i++;
    }
    fclose(file);
  }
```

28. The format of training datasets and cross-validation datasets is the same. The first two lines of the frame files should have the lines '`framesize =` n' and '`nframes =` m'. Then *nframes* of frame data should follow. First, the target value is read, and then *framesize* input values are read, one value per line.

```
⟨ Function Definitions 2 ⟩ +≡
  input_datasets(int n, struct dataset *dset)
  {
    int i, j, k;
    FILE *file;
    char theline[256];
    for (i ← 0; i < n; i++) {
      if ((file ← fopen(dset[i].filename, "r")) ≡ Λ) {
        fprintf(stderr, "Couldn't␣open␣dataset␣\"%s\"\n", dset[i].filename);
        goto FileError;
      }
      fgets(theline, sizeof (theline), file);
      if (strncmp(theline, "framesize", 9) ≠ 0) {
        fprintf(stderr, "Dataset␣format␣error,␣file␣\"%s\",␣expected␣\"framesize\"\n",
              dset[i].filename);
        goto FileError;
      }
      dset[i].framesize ← (int) getval(theline + 9, 0);
      if (dset[i].framesize * input_frames ≠ layer_spec[0]) {
        fprintf(stderr, "Dataset␣\"%s\"␣framesize␣is␣incompatible:␣%d\n", dset[i].framesize);
        goto FileError;
      }
      fgets(theline, sizeof (theline), file);
      if (strncmp(theline, "nframes", 7) ≠ 0) {
        fprintf(stderr, "Dataset␣format␣error,␣file␣\"%s\",␣expected␣\"nframes\"\n",
              dset[i].filename);
        goto FileError;
      }
      dset[i].nframes ← (int) getval(theline + 7, 0);
      dset[i].frames ← (struct frame *) calloc(dset[i].nframes, sizeof(struct frame));
      for (j ← 0; j < dset[i].nframes; j++) {
        dset[i].frames[j].input ← (float *) malloc(sizeof(float) * dset[i].framesize);
        fgets(theline, sizeof (theline), file);    /* Read frame target value */
        dset[i].frames[j].target ← atoi(theline);
        for (k ← 0; k < dset[i].framesize; k++) {
          fgets(theline, sizeof (theline), file);    /* Read an input value */
          dset[i].frames[j].input[k] ← input_scale * (atof(theline) + input_bias);
        }
      }
      fclose(file);
      continue;    /* Get the next file */
    FileError: fclose(file);
      dset[i].framesize ← dset[i].nframes ← 0;
      dset[i].frames ← Λ;
    }
  }
```

29. One more thing needs to be done as part of the initialization step. If weights have not been read from the checkpoint file, then an initial setting of weights needs to be created. According to *Neural Networks*, the weight values need to be set from the range

$$\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i}\right)$$

where F_i is the *fan-in* (total number of inputs) to a node. The system function *drand48*( ) will be used to generate pseudo-random values. First define the *random_weight* function.

```
#define W_OFFSET 2.4
⟨ Function Definitions 2 ⟩ +≡
  float random_weight(int n)
  {
    float base1;
    base1 ← W_OFFSET/(float) n;
    return (float)(base1 ∗ (2.0 ∗ drand48( ) − 1.0));
  }
```

30. Now generate random weights if necessary. But first, initialize the random number generator with a static seed value so that results may be duplicated from run to run. Establish the initial learning rate *eta* and momentum parameter *alpha*. If these have not been re-assigned by the experiment file, then they are set by default to ETA and ALPHA.

```
⟨ Initialization 20 ⟩ +≡      /∗ Generate random weight values for all non-input layer nodes. ∗/
  srand48(0);      /∗ Random number seed value. ∗/
  if (¬weights_set) {
    for (i ← layer_def[HIDDEN_LAYER].begin; i < total_nodes; i++) {
      for (j ← 0; j < nodes[i].u→nweights; j++) {
        nodes[i].u→last_weight[j] ← nodes[i].u→weight[j] ← random_weight(nodes[i].u→nweights);
      }
      nodes[i].u→eta ← eta;
      nodes[i].u→alpha ← alpha;
    }
  }
```

31. Training. Once all the datasets are read in, training may begin. Weight adjustment occurs pattern-by-pattern. An epoch consists of one presentation of all available training patterns. Presentation order is randomized between epochs. At the end of every epoch, the average error is calculated. Generalization error is also calculated after each epoch if cross-validation datasets have been supplied.

Here's the rundown: first, find the number of patterns available over all datasets, and then begin an epoch. First, initialize the activation level of every neuron above the input layer. Then choose a pattern and transfer the input parameters to the input nodes. Then starting with the input layer and using the *forward* node pointers, update the activation level of every attached neuron. Once one layer's activations have been calculated, run the squashing function on every activation level in the layer and compute the output value. Propagate the output level to the next layer, an so on, until the last layer's output values have been calculated. From the target value, calculate the error signal, and propagate the error back through the network. Adjust the weights for every layer as the error is moved back through the net. Choose the next pattern of the epoch and repeat this process. Training will stop when the maximum number of epochs is reached, or when some other error criterion is met.

```
⟨ Network Training 31 ⟩ ≡
  ⟨ Training Initialization 33 ⟩
  if (epoch < maximum_epochs) {      /* Kludge-o-rama */
    do {
      if (training_sets) {      /* Do training */
        ⟨ Training Epoch Loop 36 ⟩
      }
      epoch++;
      if (cross_sets) {      /* Cross-validation */
        ⟨ Validation Epoch Loop 54 ⟩
      }
      if (terr ≠ Λ) fprintf(terr, "%d␣%lg\n", epoch, epoch_error);
      if (verr ≠ Λ) fprintf(verr, "%d␣%lg\n", epoch, cross_error);
      if (checkpoint ∧ (epoch % checkpoint ≡ 0)) {
        write_checkpoint(epoch);
        if (terr ≠ Λ) fflush(terr);
        if (verr ≠ Λ) fflush(verr);
      }
      if (rate_reduction ∧ (epoch % rate_reduction ≡ 0)) {
        do_rate_tuning(layer_def[HIDDEN_LAYER].begin, total_nodes − 1, rate_factor);
      }
    } while (¬end_of_training(epoch));
  }
```

This code is used in section 5.

32. Let's define some more variables.

```
⟨ Local Definitions 6 ⟩ +≡
  int total_frames;      /* Frames available for training */
  int cross_frames;      /* Frames available in cross-validation sets */
  unsigned int *framelist;      /* A list of datasets and available frames */
  int frames_left;      /* The number of examples remaining in an epoch */
  int pattern;      /* a dataset/frame index */
  FILE *terr, *verr;      /* File descriptors for training and validation error */
```

33. Before beginning training, I have to find out how many training frames I will be dealing with. This number is slightly different from the total number of frames available since several frames may be used for input at one time, given by variable *input_frames*. Allocate space for the list of frames. Initialize the epoch counter if weights have not been read from a checkpoint file.

```
⟨ Training Initialization 33 ⟩ ≡
  if (¬weights_set) epoch ← 0;
  total_frames ← 0;
  cross_frames ← 0;
  for (i ← 0; i < training_sets; i++) total_frames += tsets[i].nframes − (input_frames − 1);
  for (i ← 0; i < cross_sets; i++) cross_frames += csets[i].nframes − (input_frames − 1);
      /* Allocate storage according to the larger of total_frames and cross_frames */
  if (total_frames > cross_frames) framelist ← (unsigned int *) calloc(total_frames, sizeof(int));
  else framelist ← (unsigned int *) calloc(cross_frames, sizeof(int));
  if (terr_file ≠ Λ) {
    if ((terr ← fopen(terr_file, "w")) ≡ Λ) {
      fprintf(stderr, "Couldn't␣open␣training␣error␣file␣\"%s\"\n", terr_file);
      terr_file ← Λ;
      terr ← Λ;
    }
  }
  if (cross_sets ∧ verr_file ≠ Λ) {
    if ((verr ← fopen(verr_file, "w")) ≡ Λ) {
      fprintf(stderr, "Couldn't␣open␣validation␣error␣file␣\"%s\"\n", verr_file);
      verr_file ← Λ;
      verr ← Λ;
    }
  }
```

This code is used in section 31.

34. Now that we know how many starting frames are available, I will build a list of datasets and frames to be presented as patterns. As the epoch progresses, a pattern is randomly selected and removed from the list. This list is rebuilt before every new epoch. The dataset index and the frame index are packed into one 4-byte word, 2 bytes for each value.

```
#define DSET_SHIFT (16)
#define FRAME_MASK ((1 ≪ DSET_SHIFT) − 1)
⟨ Function Definitions 2 ⟩ +≡
  build_framelist(unsigned int *list, struct dataset *sets, int nsets, int iframes)
  {
    int i, j, k, dset;
    k ← 0;
    for (i ← 0; i < nsets; i++) {
      if (sets[i].nframes) {
        dset ← i ≪ DSET_SHIFT;      /* Shift the dataset number to the upper half */
        for (j ← (int)(iframes/2); j < sets[i].nframes − (int)(iframes/2); j++)
          list[k++] ← (unsigned)(dset | (j & FRAME_MASK));
      }
    }
  }
```

35. Choosing a pattern from the framelist means simply selecting the element randomly from the list, and then removing the element so that it won't be selected again. Note that *count* is actually the index value of the last entry of the table, so that multiplying by *drand48*( ) will not extend beyond the end of the table, even after rounding to the nearest integer.

```
⟨ Function Definitions 2 ⟩ +≡
  choose_pattern(unsigned int *framelist, int count)
  {
    int index, pattern, i;    /* Choose an item from a list of count + 1 items */
    index ← (int)(drand48( ) * (float) count + 0.5);
    pattern ← framelist[index];
    for (i ← index; i < count; i++) framelist[i] ← framelist[i + 1];
    return pattern;
  }
```

36. Run an epoch by building the frame list, and then repeatedly choose items from the list. Variable *frames_left* is the maximum index value of the *framelist* array after decrementing.

```
⟨ Training Epoch Loop 36 ⟩ ≡
  build_framelist(framelist, tsets, training_sets, input_frames);
  frames_left ← total_frames;
  epoch_error ← 0.0;
  while (frames_left−−) {
    pattern ← choose_pattern(framelist, frames_left);
    copy_input(pattern, tsets);
    ⟨ Forward Propagation 38 ⟩
    ⟨ Backward Propagation 42 ⟩
    adjust_weights( );
    if (debugging) show_network(epoch, pattern);
  }
  epoch_error /= (double) total_frames;
```

This code is cited in section 51.

This code is used in section 31.

37. To copy the selected pattern into the input layer, simply decode the dataset and frame number from the *pattern*, and copy the requested number of frames (*input_frames*) to the input layer. The frame *fnum* extracted from *pattern* is the central frame. Additional input is taken from either side of the central frame. The desired output value is decoded from the central frame's *target* parameter. Since no squashing function is applied to the input values, they are copied directly to the *output* variable of the input nodes.

```
⟨ Function Definitions 2 ⟩ +≡
  copy_input(int pattern, struct dataset *sets)
  {
    int dset, fnum, i;
    int frame_el, inode;

    dset ← pattern ≫ DSET_SHIFT;
    fnum ← pattern & FRAME_MASK;
    inode ← 0;      /* For each frame... */
    for (i ← fnum − (int)(input_frames/2); i ≤ fnum + (int)(input_frames/2); i++) {
        /* For each element of a frame... */
      for (frame_el ← 0; frame_el < sets[dset].framesize; frame_el++) {
        nodes[inode].output ← sets[dset].frames[i].input[frame_el];
        inode++;
        if (inode > layer_def[INPUT_LAYER].end) break;
      }
    }
  }
```

38. The forward pass of training proceeds from the input layer to the output layer, a layer at a time. First, initialize the activation levels of all nodes above the input layer. Then, beginning with the input nodes, add the output level times the input weight to the activation level of nodes in the next layer. When all of the input to a layer has been propagated, set the output values of all nodes in the next layer by running the activation value through the squashing function. Then proceed to propagate the output values to the next layer.

```
⟨ Forward Propagation 38 ⟩ ≡
  init_activation(layer_def[HIDDEN_LAYER].begin, layer_def[OUTPUT_LAYER].end);
  for (i ← INPUT_LAYER; i < OUTPUT_LAYER; i++) {
    send_forward(i);
    squash(i + 1);
  }
```

This code is used in sections 36, 51, and 54.

39. The *init_activation*( ) function is straight-forward: the threshhold value is subtracted from the activation value immediately. Note that *end* specifies the final index value, and so must be included.

```
⟨ Function Definitions 2 ⟩ +≡
  init_activation(int start, int end)
  {
    int i;

    for (i ← start; i ≤ end; i++) nodes[i].u→activation ← −nodes[i].u→threshhold;
  }
```

40. Function *send_forward*( ) is also not complicated, but slightly tricky. Index value i references the input node. Each input node will point to some number of nodes indexed by j where output should be sent. The index value of each of these connected nodes is referenced by index nn. Finally, the particular weight index which applies to any input value is tracked by index wc, so wc changes with every change in i. For example, the weight at index 0 is applied to the first input node from the previous layer. Eeek.

```
⟨ Function Definitions 2 ⟩ +≡
  send_forward(int i_layer)
  {
    int i, j;
    int nn, wc;
    wc ← 0;
    for (i ← layer_def[i_layer].begin; i ≤ layer_def[i_layer].end; i++) {
      for (j ← 0; j < nodes[i].forward→n_nodes; j++) {
        nn ← nodes[i].forward→ptrs[j];
        nodes[nn].u→activation += nodes[i].output * nodes[nn].u→weight[wc];
      }
      wc++;
    }
  }
```

41. Apply the squashing function $\phi(\nu)$ to the activation level of a neuron to obtain the output value. The activation level already includes the threshhold value for the node.

```
⟨ Function Definitions 2 ⟩ +≡
  squash(int i_layer)
  {
    int i;
    for (i ← layer_def[i_layer].begin; i ≤ layer_def[i_layer].end; i++)
      nodes[i].output ← phi((double) nodes[i].u→activation);
  }
```

42. The backward pass begins analagously to forward propagation. First, all the local gradient values for nodes in the first hidden layer through the last hidden layer (i.e., below the output layer) are reset. Next, the error signal is determined at the output layer from the target value for the central frame, and then the error is distributed back to the first hidden layer of the network.

```
⟨ Backward Propagation 42 ⟩ ≡
  zero_delta(layer_def[HIDDEN_LAYER].begin, layer_def[OUTPUT_LAYER − 1].end);
  epoch_error += calculate_error(pattern);
  for (i ← OUTPUT_LAYER; i > HIDDEN_LAYER; i−−) {
    send_backward(i);
    local_gradient(i − 1);
  }
```

This code is used in section 36.

43. Reset the deltas for a range of nodes. Nothing fancy here except that *end* is a legitimate index value.

```
⟨ Function Definitions 2 ⟩ +≡
  zero_delta(int start, int end)
  {
    int i;
    for (i ← start; i ≤ end; i++) nodes[i].u→delta ← 0.0;
  }
```

44. To calculate error, retrieve the target value from the central frame of the current pattern. This target value represents the output node which should be active. A target value of 0 means that no output nodes should be active. The derivative of the squashing function $\phi'(\nu)$ is used in the calculation of the δ of the output layer according to Eq. 6.33 in *Neural Networks*:

$$\delta_j(n) = e_j(n)\phi'_j\left(v_j(n)\right),$$

where n refers to the iteration value, and j refers to the output node index.

```
⟨ Function Definitions 2 ⟩ +≡
  float calculate_error(int pattern)
  {
    int dset, fnum;
    int i, active_node, onode;
    float error_val, acc_err;

    dset ← pattern ≫ DSET_SHIFT;
    fnum ← pattern & FRAME_MASK;
    active_node ← tsets[dset].frames[fnum].target;
    acc_err ← 0.0;
    onode ← 1;
    for (i ← layer_def[OUTPUT_LAYER].begin; i ≤ layer_def[OUTPUT_LAYER].end; i++) {
      error_val ← (onode ≡ active_node ? (A − EPSILON) − nodes[i].output :
          (−A + EPSILON) − nodes[i].output);
      acc_err += (error_val ∗ error_val) ∗ 0.5;
      nodes[i].u→delta ← error_val ∗ phi_prime((double) nodes[i].u→activation);
      onode++;
    }
    return acc_err;
  }
```

45. Accumulate a node's local gradient information from the layer above it. There are subtle differences between this function and *send_forward*( ).

```
⟨ Function Definitions 2 ⟩ +≡
  send_backward(int i_layer)
  {
    int i, j, nn;

    for (i ← layer_def[i_layer].begin; i ≤ layer_def[i_layer].end; i++) {
      for (j ← 0; j < nodes[i].u→backward→n_nodes; j++) {
        nn ← nodes[i].u→backward→ptrs[j];
        nodes[nn].u→delta += nodes[i].u→delta ∗ nodes[i].u→weight[j];
      }
    }
  }
```

46. Update the *delta*s on a layer by multiplying the derivative of the squashing function by the accumulated value in a *delta*. This completes the terms given by Eq. 6.34 in *Neural Networks*:

$$\delta_j(n) = \phi_j'\left(v_j(n)\right) \sum_k \delta_k(n) w_{kj}(n),$$

where n is the iteration, j is the index of the hidden node, and k is the index of a node in the layer above the node indexed by j. w_{kj} is the weight connecting the output of node j to node k.

```
⟨ Function Definitions 2 ⟩ +≡
  local_gradient(int i_layer)
  {
    int i;
    for (i ← layer_def[i_layer].begin; i ≤ layer_def[i_layer].end; i++) {
      nodes[i].u→delta *= phi_prime((double) nodes[i].u→activation);
    }
  }
```

47. Assuming that both the forward and backward passes have been completed, the weights for all nodes may now be adjusted using the generalized delta rule (page 156):

$$w_{ji}(n+1) = w_{ji}(n) + \alpha\left[w_{ji}(n) - w_{ji}(n-1)\right] + \eta\,\delta_j(n) y_i(n).$$

```
⟨ Function Definitions 2 ⟩ +≡
  adjust_weights( )
  {
    int i, j, nn;
    float old_weight;
    struct node *np;
    for (i ← layer_def[HIDDEN_LAYER].begin; i ≤ layer_def[OUTPUT_LAYER].end; i++) {
      np ← &(nodes[i]);
      old_weight ← np→u→threshhold;
      np→u→threshhold += np→u→alpha * (np→u→threshhold − np→u→old_t) −
          (np→u→eta * np→u→delta);      /* update the threshhold */
      np→u→old_t ← old_weight;
      for (j ← 0; j < np→u→nweights; j++) {
        nn ← np→u→backward→ptrs[j];
        old_weight ← np→u→weight[j];
        np→u→weight[j] += np→u→alpha * (np→u→weight[j] − np→u→last_weight[j]) +
            np→u→eta * np→u→delta * nodes[nn].output;
        np→u→last_weight[j] ← old_weight;
      }
    }
  }
```

48. Here's where the stopping criteria are determined. All stopping tests should go here, and the function should return `TRUE` if any of the criteria have been met, and `FALSE` otherwise.

```
⟨ Function Definitions 2 ⟩ +≡
  end_of_training(int epochs)
  {
    if (maximum_epochs ∧ epochs ≥ maximum_epochs) return 1;
    else return 0;
  }
```

49. Start messing with error stuff.

```
⟨ Global Definitions 4 ⟩ +≡
  double epoch_error;
  double cross_error;
```

50. Do some rate reduction after a certain number of epochs.

```
⟨ Function Definitions 2 ⟩ +≡
  do_rate_tuning(int start, int end, double factor)
  {
    int i;
    for (i ← start; i ≤ end; i++) {
      nodes[i].u→eta *= factor;
      nodes[i].u→alpha *= factor;
    }
  }
```

51. Testing. Once the network is finished training (finished according to the stopping criteria, at least), run through the training sets once again, displaying the frame data and the solution that the network calculates. This may have to be modified once I start working with larger datasets. This is basically copied from the ⟨ Training Epoch Loop 36 ⟩, but without the back-propagation and weight adjustments.

```
⟨ Free Run Testing 51 ⟩ ≡
  build_framelist(framelist, tsets, training_sets, input_frames);
  for (frames_left ← 0; frames_left < total_frames; frames_left++) {
    pattern ← framelist[frames_left];
    copy_input(pattern, tsets);
    ⟨ Forward Propagation 38 ⟩
    show_frame(frames_left);
  }
```

This code is used in section 5.

52. Displaying the frame output means translating output nodes into printable values from *node_strings*.

```
⟨ Function Definitions 2 ⟩ +≡
  show_frame(int framenum)
  {
    int i, oc, result;
    float highest_output;

    highest_output ← −A;
    result ← 0;    /* best output code so far (null case) */
    oc ← 1;    /* output code */
    for (i ← layer_def[OUTPUT_LAYER].begin; i ≤ layer_def[OUTPUT_LAYER].end; i++) {
      if (nodes[i].output > 0.5 ∧ nodes[i].output > highest_output) {
        result ← oc;
        highest_output ← nodes[i].output;
      }
      oc++;
    }
    if (output_strings) printf("␣␣RESULT:␣%s\n", node_strings[result]);
    else printf("␣␣RESULT:␣node␣%d␣val␣%g\n", result, highest_output);
  }
```

53. Show network state (debugging function). This dumps the state of the whole network in excruciating detail.

```
⟨ Function Definitions 2 ⟩ +≡
  int debugging ← 0;
  show_network(int epoch, int pattern)
  {
    int i, j, k, nc;
    fprintf(stderr, "EPOCH␣%d␣DSET␣%d␣FRAME␣%d\n", epoch, pattern ≫ 16, pattern & #ffff);
    for (i ← 0; i < layers; i++) {
      fprintf(stderr, "␣␣LAYER␣%d\n", i);
      for (j ← layer_def[i].begin, nc ← 0; j ≤ layer_def[i].end; j++, nc++) {
        fprintf(stderr, "␣␣␣␣Node␣%d,␣(%d␣in␣layer)\n", j, nc);
        if (i > INPUT_LAYER) {
          for (k ← 0; k < nodes[j].u→nweights; k++)
            fprintf(stderr, "␣␣␣␣␣␣From␣node␣%d␣with␣weight␣%g␣(lw=%g,dw=%g)\n",
                  nodes[j].u→backward→ptrs[k], nodes[j].u→weight[k], nodes[j].u→last_weight[k],
                  nodes[j].u→weight[k] − nodes[j].u→last_weight[k]);
        }
        if (nodes[j].forward ≠ Λ)
          for (k ← 0; k < nodes[j].forward→n_nodes; k++)
            fprintf(stderr, "␣␣␣␣␣␣Connected␣to␣node␣%d␣with␣weight␣%g\n",
                  nodes[j].forward→ptrs[k], nodes[nodes[j].forward→ptrs[k]].u→weight[nc]);
        if (i > INPUT_LAYER) fprintf(stderr, "␣␣␣␣␣␣thresh=%g␣active=%g␣output=%g␣delta=%g\n",
                nodes[j].u→threshhold, nodes[j].u→activation, nodes[j].output, nodes[j].u→delta);
        else fprintf(stderr, "␣␣␣␣␣␣output=%g\n", nodes[j].output);
      }
    }
  }
```

54. Run the equivalent of an epoch on the cross-validation datasets, but without back-propagation of error or weight adjustment.

```
⟨ Validation Epoch Loop 54 ⟩ ≡
  build_framelist(framelist, csets, cross_sets, input_frames);
  frames_left ← cross_frames;
  cross_error ← 0.0;
  while (frames_left−−) {
    pattern ← choose_pattern(framelist, frames_left);
    copy_input(pattern, csets);
    ⟨ Forward Propagation 38 ⟩
    cross_error += calculate_error(pattern);
  }
  cross_error /= (double) cross_frames;
```

This code is used in section 31.

55. Index. Variables, function names, and structure definitions are listed here with the section numbers where they are referenced. Definitions occur in underlined sections.

NETWORK