INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Connectionist Layered Object-Oriented Network Simulator (CLONES): User's Manual

Phil Kohn

TR-91-073

3 March 1992

Abstract

CLONES is a object-oriented library for constructing, training and utilizing layered connectionist networks. The CLONES library contains all the object classes needed to write a simulator with a small amount of added source code (examples are included). The size of experimental ANN programs is greatly reduced by using an object-oriented library; at the same time these programs are easier to read, write and evolve. The library includes database, network behavior and training procedures that can be customized by the user. It is designed to run efficiently on data parallel computers (such as the RAP [6] and SPERT [1]) as well as uniprocessor workstations. While efficiency and portability to parallel computers are the primary goals, there are several secondary design goals:

1. minimize the learning curve for using CLONES,
2. minimize the additional code required for new experiments,
3. allow heterogeneous algorithms and training procedures to be interconnected and trained together.

Within these constraints we attempt to maximize the variety of artificial neural network algorithms that can be supported.

1 Overview

Continuing experimentation with Artificial Neural Networks (ANNs)[3] has made it increasingly clear that:

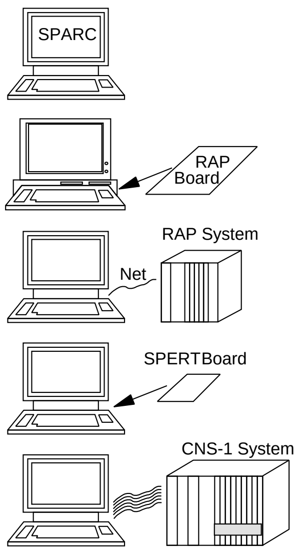
1. Because of the diversity and continuing evolution of ANN algorithms, the programming environment must be both powerful and flexible.

2. These algorithms are very computationally intensive when applied to large databases of training patterns.

Ideally we would like to implement and test ideas at about the same rate that we come up with them. We have approached this goal both by developing application specific parallel hardware, the Ring Array Processor (RAP) [6, 2, 5], and by building an object-oriented software environment, the Connectionist Layered Object-oriented Network Simulator (CLONES). By using an object-oriented library, the size of experimental ANN programs can be greatly reduced while making them easier to read, write and modify. CLONES is written in C++ and utilizes libraries previously written in C and assembler. It is a completely integrated system including object classes for databases and training procedures with default implementations that can be easily adapted by users. An interface to a Graphical User Interface (or GUI) library for viewing CLONES data structures is under development. This GUI interface will also be used by other projects.

Researchers often generate either a proliferation of versions of the same basic program, or one giant program with a large number of options and many potential interactions and side-effects. Some simulator programs include (or worse, evolve) their own language for describing networks. We feel that a modern object-oriented language (such as C++) has all the functionality needed to describe, build and train ANNs. By using an object-oriented design, we attempt to make the most frequently changed parts of the program small and well localized. The parts that rarely change are in a centralized library. One of the many advantages of an object-oriented library for experimental work is that any part can be specialized by making a new class of object that inherits the desired operations from a library class.

Our ANN research currently encompasses two hardware platforms and several languages, shown in Figure 1. Two new hardware platforms, the SPERT board [1] and the CNS-1 system are in design (unfilled check marks). The SPERT design is a custom VLSI parallel processor installed on an SBUS card plugged into a SPARC workstation. Using variable precision fixed point arithmetic, a single SPERT board will have performance comparable to a 10 board RAP system with 40 TMS320C30 digital signal processors (each of which runs at 32 million operations per second). This is to be accomplished by using very wide instructions (128 bits), wide memory buses (128 bits), eight parallel datapaths (each with a multiplier, shifter and adder), and very fast SRAM. The CNS-1 system will be based on multiple custom VLSI parallel processors interconnected by high speed communication buses.



System	Performance	Languages Supported				
		Assem	C	C++	Sather	pSather
SparcStation 2	2 MFLOP	✔	✔	✔	✔	✔
Desktop RAP + Sun 4/330 Host	100 MFLOP	✔	✔ &	✔ &	✔ &	
Networked RAP (1-10 Boards)	1 GFLOP		⚡	⚡	⚡	
SparcStation + SPERT Board	1 GOP	✓	↯	↯	↯	
CNS-1 System	200 GOP	✓	↯	↯	↯	↯

Source Compatible

Linker Compatible

Native Support:	✔	Completed	✓	In Design
RPC Support:	⚡	Completed	↯	In Design

Figure 1: Hardware and software configurations

Because the investment in software is generally large, we *insist on* source compatibility of CLONES programs across hardware platforms. This is accomplished by hiding the hardware configuration in a set of library classes. Because it is based on a common library interface, the same CLONES source code can be compiled for the RAP or UNIX workstations simply by selecting UNIX or TI compiler and library files. The implementation of these classes will be different for each hardware configuration. These libraries include matrix and vector classes that hide the distribution of data and computation among the processors from the user.

There are two ways to run CLONES programs that utilize the RAP:

1. The program can be compiled to run completely (except for file input and output) on the processors of the RAP. In this case, the host SPARC machine only handles communication with the world outside of the RAP. So long as external input and output are not a bottleneck, this method minimizes communication between the host and RAP machine.

2. CLONES code runs on the host SPARC except when library routines are called. A "RPC" SPARC library is used; this library is based on a very fast shared memory remote procedure call (RPC) mechanism. Any function in the RAP library can be remotely called by the host SPARC including memory allocation routines. For example, creating a matrix or vector on the SPARC causes the RPC library to request memory for it on all of the RAP processors. When data inside one of these objects is manipulated with **put** or **get** functions, the RPC library copies the data (and converts representations if required).

In the case of SPERT and CNS-1, we plan to (at least initially) only support the second (RPC) method above.

It is also considered important to allow routines in different languages to be linked together. This includes support for Sather, an object-oriented language that has been developed at ICSI for workstations. The parallel version of Sather, called pSather, may be supported on the CNS-1. Both of these languages utilize the special purpose hardware as a remote compute server (via an RPC interface); they are not compiled to run on that hardware.

CLONES is seen as an ANN researcher's interface to this multiplatform, multilanguage environment. Although CLONES is an application written specifically for ANN algorithms, it's object-orientation gives it the ability to easily include previously developed libraries. CLONES currently runs on UNIX workstations and the RAP.

The four most important priorities of the CLONES design were:

1. Preserve the efficiency of the hardware. Making efficiency a primary goal differentiates this design from many others[7] where flexibility, generality and ease of representation are the most important design goals. However, there are other new designs that do provide a significant increase in efficiency over previous simulators[8, 9].

2. Allow the same CLONES source code to utilize uniprocessor workstations and single program data parallel machines (such as RAP or SPERT), and perhaps other vector processing, systolic array or SIMD (Single Instruction stream, multiple data streams) machines. To accomplish this, CLONES is based on abstract matrix and vector classes that enforce a uniform interface. Each hardware configuration has its own matrix and vector classes that inherit from these abstract classes. In the case where the platform has a C compiler, the complete CLONES program can run on the parallel machine; this reduces the host-server communication to requests for external data files and communication with the user.

3. Make it easy to customize any aspect of the network behavior and training procedure. For example, abstract classes for **Layer** and **Connect** leave the data structure used to represent activations, errors, and weights totally unspecified. The behavior of network components and their training procedures can be customized without changing any CLONES source files: a new class that inherits from a library class is declared and then some of the functions are redefined. Since we have traded some flexibility for efficiency on these hardware platforms, we do not claim that all connectionist algorithms can be cleanly implemented in CLONES.

4. Heterogeneous algorithms and training procedures can be interconnected, then trained and run as a single network.

5. The trained networks can be easily embedded into other programs. These programs might use the network as part of another algorithm (such as dynamic programming for speech recognition), or CLONES nets might be embedded in another ANN simulator that is better for other processing.

2 CLONES overview

To make CLONES easier to learn, we restrict ourselves to a subset of the many features of C++. Excluded features include multiple inheritance, operator overloading (however, function overloading is used) and references. Since the multiple inheritance feature of C++ is not used, CLONES classes can be viewed as a collection of simple inheritance trees. This means that all classes of objects in CLONES either have no parent class (top of a class tree) or inherit the functions and variables of a single parent class.

Users customize the behavior of network components and training procedures by creating a new class that inherits from a library class and then redefining some of its functions. This allows the user to modify the behavior of CLONES objects without changing the CLONES library.

Figure 2 shows the overall inheritance structure of CLONES classes. An overview of these classes is presented here. Each class is described in detail in the following

Figure 2: Class tree for CLONES

sections.

CLONES consists of a library of C++ classes that represent networks (**Net**), their components (**Net_part**) and training procedures. There are also utility classes used during training such as: databases of training data (**Database**), tables of parameters and arguments (**Param**), and performance statistics (**Stats**). **Database** and **Param** do not inherit from any other class. Their class trees are independent of the rest of CLONES and each other. The **Stats** class inherits from **Net_behavior**.

The top level of the CLONES class tree is a class called **Net_behavior**. It defines function interfaces for many general object functions including file save or restore and debugging. It also contains behavior functions that are called during different phases of running or training a network. For example, there are functions that are called before or after a complete training run (**pre_training**, **post_training**), before or after a pass over the database (**pre_epoch**, **post_epoch**) and before or after a forward or backward run of the network (**pre_forw_pass, post_forw_pass, pre_back_pass, post_back_pass**). The **Net**, **Net_part** and **Stats** classes inherit from this class.

All network components used to construct ANNs are derived from the two classes **Layer** and **Connect**. Both of these inherit from class **Net_part**. A CLONES network can be viewed as a graph where the nodes are **Layer** objects and the arcs are **Connect** objects. Each **Connect** connects a single input **Layer** with a single output **Layer**. A **Layer** holds the data for a set of units (such as an activation vector), while a **Connect** operates on the data as it passes between **Layers**. Data flows along **Connects** between the pair of **Layers** by calling **forw_propagate** (input to output) or **back_propagate** (output to input) behavior functions in the **Connect** object.

For efficiency, CLONES does not have objects that represent single units (or artificial neurons). Instead, **Layer** objects are used to represent a set of units. Little flexibility is lost since **Layers** with a single unit can be used when required. However, many very small **Layers** will not run as efficiently as fewer larger ones. Also, the elements of the activation vector of a **Layer** need not be computed by the same function. **Layers** are *not* restricted to representing a vector of units that must be treated identically. Because arrays of units are passed down to the lowest level routines, most of the computation time is focused into a few inner loops. These assembly coded loops fit into the processor instruction cache thereby reducing the overhead of instruction fetches. Time spent in all of the levels of control code that call these loops becomes less significant as the size of the **Layer** is increased.

The **Layer** class does not restrict the representation of its internal information. For example, the representation for activations may be a floating (or fixed) point number for each unit (**Analog_layer**), or it may be a set of unit indices, indicating which units are active (**Binary_layer**). **Analog_layer** and **Binary_layer** are built into the CLONES library as subclasses of the class **Layer**. The **Analog_layer** class specifies the representation of activations, but it still leaves open the procedures that use and update the activation array. These procedures may or may not treat all the units of the **Layer** the same way. **BP_analog_layer** is a subclass of **Analog_layer** that specifies the procedures for the back-propagation algorithm. Subclasses of **Analog_layer** may also add new data structures to hold extra internal state such as the error vector in the case of **BP_analog_layer**. The **BP_Analog_layer** class has subclasses for various transfer functions such as **BP_sigmoid_layer** and **BP_linear_layer**.

Layer classes have behavior functions that are called in the course of running the network. For example, one of these functions (called **pre_forw_propagate**) initializes the **Layer** for a forward pass, perhaps by clearing its activation vector. After all of the connections coming into it are run, another **Layer** behavior function (called **post_forw_propagate**) is called that computes the activation vector from the partial results left by these connections. For example, this function may apply a transfer function such as the sigmoid to the accumulated sum of all the input activations. These behavior functions can be changed by making a subclass.

Layers also have functions (**get** and **put**) that allow access to internal activation data. A subclass of **Layer** may support multiple **get** routines for accessing the same activation data, each with its own datatype for activations and imposing its own structuring on these activations. The default **get** functions include floating point and integer activations that are organized in one, two or three dimensional arrays. New access functions can be added as new activation datatypes and organizational structures are researched.

For example, the **BP_analog_layer** class leaves open the activation transfer function (or squashing function) and its derivative. Subclasses define new transfer functions to be applied to the activations. A new class of back-propagation layer with a customized transfer function (instead of the default sigmoid) can be created with the following C++ code:

```
class My_new_BP_layer_class : public BP_analog_layer {

  My_new_BP_layer_class(int number_of_units)
    : BP_analog_layer(number_of_units);    // constructor

  // apply forward transfer function to activation vector
  // (note that Fvec is the floating point vector class)
  void transfer(Fvec *activation) {
    int i;
    for(i=0; i<activation->n_ele; i++) // for each element of activation vector
      activation->put(i, my_transfer_function(activation->get(i)));
  }

  // apply backward error transfer function to error vector (given activation vector)
  void d_transfer(Fvec *activation, Fvec *error)  {
    int i;
    for(i=0; i<error->n_ele; i++)  // for each element of error vector
      error->put(i,
                 derivative_of_my_transfer_function(activation->get(i), error->get(i)));
  }
};
```

A **Connect** class includes two behavior functions: one that transforms activations from the input **Layer** into partial results in the output **Layer** (**forw_propagate**) and one that takes outgoing errors and generates partial results in the input **Layer** (**back_propagate**). The structure of a partial result is part of the **Layer** class. The subclasses of **Connect** include: **Bus_connect** (one to one), **Full_connect** (all to all) and **Sparse_connect** (some to some).

Each subclass of **Connect** may contain a set of internal parameters such as the weight matrix in a **BP_full_connect**. Subclasses of **Connect** also specify which pairs of **Layer** subclasses can be connected. When a pair of **Layer** objects are connected, type checking by the C++ compiler insures that the input and output **Layer** subclasses are supported by a **Connect** class.

In order to do its job efficiently, a **Connect** must know something about the internal representation of the layers that are connected. By using C++ overloading, the **Connect** function selected depends not only on the class of **Connect**, but also on the classes of the two layers that are connected. Not all **Connect** classes are defined for all pairs of **Layer** classes. However, **Connects** that convert between **Layer** classes can be utilized to compensate for missing **Connect** classes.

CLONES allows the user to view layers and connections much like tinker-toy wheels and rods. ANNs are built up by creating **Layer** objects and passing them to the create functions of the desired **Connect** classes. Changing the interconnection pattern does not require any changes to the **Layer** classes or objects and vice-versa.

At the highest level, a **Net** object delineates a subset of a network and controls its training. Operations can be performed on these subsets by calling functions on their **Net** objects. The **Layers** of a **Net** are specified by calling one of the following routines (of the **Net** object) for each **Layer**: **new_input_layer**, **new_hidden_layer**, or **new_output_layer**. A **Net** can have any number of input, hidden and output **Layers**. Given the **Layers**, the **Connects** that belong to the **Net** are deduced by the **Net_order** objects (see below). **Layer** and **Connect** objects can belong to any number of **Nets**.

The **Net** labels all of its **Layers** as either input, output or hidden. These labels are used by the **Net_order** objects to determine the order in which the behavior functions of the **Net_parts** are called. For example, a **Net** object contains **Net_order** objects called **forward_pass_order** and **backward_pass_order** that control the execution sequence for a forward or backward pass. The **Net** object also has functions that call a function by the same name on all of its **Layers** and **Connects** (for example **set_learning_rate**).

Figure 3 shows an example network with five layers and five connections. This figure also shows two **Net** objects. A **Net** object deliniates a subset of a network. As shown in the figure these subsets may overlap. Net1 refers to Layer1, Layer2, Layer3, and the **Connects** between Layer1 and Layer2 and between Layer2 and Layer3. Net2 refers to the all of **Layer** and **Connect** objects in the figure. Operations can be performed on these subsets by calling functions on the Net1 or Net2 object.

The networks depicted in figure 3 could be used, for example, to incorporate

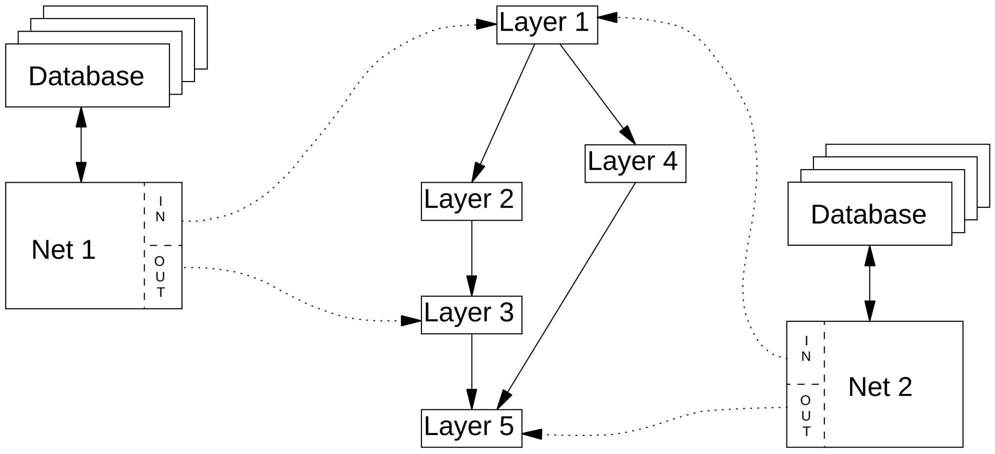


Figure 3: Example CLONES network

knowledge into an ANN by training a network to produce useful intermediate results. In this example, Net1 is trained first to produce an output at Layer3 that is a useful input for a later training step. Then **set_learning_mode(0)** is called on Net1 to freeze its parameters. Calling **run_training** on Net2 causes **Connects** between **Layers** 1 and 4, 3 and 5, 4 and 5 to be trained with the aid of the unadaptive input from Layer3.

Often there are several constructors for the same class of **Net**. **Net** constructors may create new **Layer** and **Connect** objects. For example, a subclass of **Net** may automatically build a three **Layer** structure when it is created. Its arguments might be the sizes of the input, hidden and output layers. **Net** constructors may also take existing **Layer**, **Connect** or **Net** object pointers as arguments. This allows new **Net** objects to add structure to an existing **Net**. The **Net** constructor can also call the constructors of other (or the same) **Net** class to create modular hierarchically structured topologies. Such a **Net** might manage these internal **Net** objects and hide them from external view.

The **build** function of the **Net_order** object scans the connectivity of the **Net**. The rules that relate topology to order of execution are centralized and encapsulated in subclasses of **Net_order**. Changes to the topology of the **Net** are localized to just the code that creates the **Layers** and **Connects**; one does not need to update separate code that contains explict knowledge about the order of execution when the **Net** topology changes.

The training procedure is divided into a series of steps, each of which is a call to a function in the **Net** object. At the top level, calling **run_training** on a **Net** performs a complete training run. It starts by calling **pre_training** and ends by calling the **post_training** behavior functions. It calls **run_epoch** in a loop until the the **next_learning_rate** function returns zero. The **run_epoch** function calls **run_forward** and **run_backward**.

The **Net** object contains global variables for use by all of its components. These global variables of **Net** include: the current pattern, the correct target output, the epoch number, **Stats** objects, **Database** objects and **Param** object. A pointer to the **Net** object is always passed to all of its **Layer** and **Connect** behavior functions when they are called.

The **Net** object contains one or more **Database** object pointers. More than one **Net** can share the same **Database**. The **Net** contains functions that interface the **Databases** to the **Layers** of the **Net**. For example, the **set_input** function sets the activations of the input **Layers** for a given pattern number of the database. Another of these sets the error vector of the output layer (**set_error**). Some of these functions, such as **is_correct** evaluate the performance of the **Net** on the current pattern.

Another global variable in the **Database** is a **Param** object that contains a table of parameter names, each with a list of values. Again, multiple **Nets** can share a single **Param** object. These parameters usually come from the command line and/or parameter files. New parameters can be added to a CLONES program by adding a single line to the parameter table (file parameters.h). The line contains the parameter name and a single line of documentation for the help menu.

3 Example: backpropagation

Before describing CLONES classes and training procedures in more detail, it might be helpful to look a simple CLONES program from the user's point of view. The following program creates an input, hidden and output layer and then connects input to hidden and hidden to output. Backpropagation with a sigmoid transfer function is used for this example.

```
void
main(int argc, char **argv)
{
Param param;                                  // declare a parameter list
BP_sigmoid_layer *input, *hidden, *output;    // declare 3 layers
BP_full_connect *input_hidden, *hidden_output; // declare 2 connections
BP_net *net;                                  // declare a net object
Sentence_database *db;                        // declare a training database

FILE *db_file;
char *file_mode;

// read command line parameters into parameter object (param)
param.parse(argc, argv);

// create the database object with defaults of 1 frame of input context
// and 1 category label per frame
db = new RAM_database(param.get_int("frames_per_window",1),
  param.get_int("labels_per_frame",1));

// set the reading mode to binary if this is a binary database
file_mode = param.get_bool("binary_database")? "rb" : "r";

// read in the database
db_file = fopen(param.get_string("database_file_name"), file_mode);
if (db_file == NULL) { perror("can not open database"); exit(-1) }
db->io(db_file, file_mode, param.get_int("number_of_sentences"));
fclose(db_file);

// create the layers; get the input and output layer sizes from the database object
// the default hidden layer is 512 units unless specified by -hidden_layer_size
input = new BP_sigmoid_layer(db->get_pattern_size());
hidden = new BP_sigmoid_layer(param.get_int("hidden_layer_size",512));
output = new BP_sigmoid_layer(db->get_n_label());

// create the connections
input_hidden = BP_full.connect(input, hidden);
hidden_output = BP_full.connect(hidden, output);

// create the Net object
net = new BP_net(&param, db, new Stats);

// define the input, output and hidden layers for this net
net->new_input_layer(input);
net->new_hidden_layer(hidden);
net->new_output_layer(output);

// build the net (updates order objects, etc.)
net->build();

// train it up!
net->run_training();
}
```

What the **run_training** function of the **Net** actually does depends on the subclass of **Net**. This section outlines the default training procedure that comes with the **Net** object and its subclasses (unless it is overriden by writing new functions). The training procedure is broken down into a series of steps. Each step involves calling a function of the **Net** class. Since there are several levels of control functions (with **run_training** at the top), the user can customize the training procedure anywhere from slightly to completely by redefining functions in a new subclass of **Net**.

All of the training functions listed below can be found in the file **run.cc**; they all belong to class **Net** or one of its subclasses. Here are the basic steps of the training procedure **run_training**:

1. The **pre_training** function is called on the **Net** and all of its **Net_parts**. This is used to initialize the network. For example, the **pre_training** function of a **Connect** object might randomize its weight matrix.

2. Optionally, there is a **pre_training** epoch, in which the complete database is run through the network with the **stage** variable of the **Net** object set to **PRE_TRAIN** [1]. This is used by algorithms that need to use the database to initialize the network for training. For example, an RBF[4] network might calculate the covariance matrix during the **PRE_TRAIN** epoch.

3. Call **reset_learning_rate** to initialize the learning control parameters or temperature schedule.

4. Inside the main training loop, call **run_epoch** for each **Database** until the **next_learning_rate** function returns **FALSE**. Each call to **run_epoch** will present the network with as many patterns as are in the **Database** objects of the **Net**. In this loop the **stage** variable of the **Net** object is set to **TRAIN** to adapt the network to the patterns, or **TEST** for cross-vaildation. Inside the **run_epoch** function:

    (a) The **pre_epoch** function is called on the **Stats** object and all **Net_parts**.

    (b) Loop over all patterns in the database. The loop index is a pattern sequence number running from zero to the number of patterns in the database minus one. Inside the **run_epoch** loop over the pattern sequence number:

[1]Other values of the **stage** variable include: **TRAIN**, **TEST** and **POST_TRAIN**.

   i. Call **pattern_order(seq_num)** to translate the sequence number into a database pattern_id. Changing the **pattern_order** function of the **Net** to modifies the order of pattern presentation. The default **pattern_order** function uses the **db_order** variable of the **Net** to select an ordering. If **db_order** is **REAL** then the pattern_id returned is just the sequence number; the patterns are presented in the order specified by the database. If **db_order** is **RANDOM_PATTERN**, the pattern_id returned is a random number between zero and the number of patterns minus one. There are other pattern orderings that are documented in the file **clones.h** where the enumeration **DB_order** is declared.
   ii. Call **pattern_select(pattern_id)**. If it returns **FALSE** then the pattern is skipped.
   iii. Call **set_input(pattern_id)** to load data into the input layer(s) of the network. The default **set_input** loads all of the pattern data from the database **get_pattern** function into the input layer(s) activation vector using the **Net put_input** function.
   iv. Call **set_target** to set the **Net** target variable(s) to some representation of the desired output.
   v. Call one of the functions: **train**, **test**, **pre_train** or **post_train**, depending on the value of the **stage** variable of the **Net**. This example assumes that **stage** is set to **TRAIN**, so the function **train** is called. These functions are called by function **train**:
      A. Call **run_forward(pattern_id)** to setup and run the forward pass of the network. The details of what **run_forward** calls are shown in a separate outline below.
      B. Call **set_error** to set the error vector in the output layer(s) based on the target values in the **Net** object.
      C. Call **pre_back_pass** on the **Stats** object and **Net_parts**.
      D. Call **run_back_order** to run the backward_order object of the **Net**.
      E. Call **post_back_pass** on the **Stats** object and **Net_parts**.
   vi. The **post_epoch** function is called on all **Net_parts** and statistics object(s).

5. Call **next_learning_rate** to adjust the learning variable(s) of the **Net** object (it may utilize performance data in the Stats object for this). If it returns zero then training is complete, otherwise loop back to step (a) and run another epoch.

6. The **post_training** function is called on the **Net** and all of its **Net_parts**.

7. Optionally, there is a post_training epoch in which the complete database is run with the **stage** variable of the **Net** object set to **POST_TRAIN**. This can be used to collect performance statistics on the fully trained network.

Inside the function **run_forward** above, the following steps occur:

1. Call **pre_forw_pass** on the **Stats** object and **Net_parts**.
2. Call **run_forw_order** to execute the calls held by the **forward_order** object. This in turn calls the behavior on network components. Note that the functions called depends on the subclass of **Net_order** object. Also note that the functions in this list are the only **run_training** fuctions that are not in the file **run.cc**. The defaults for these functions for backpropagation are in files **bp.cc** and **bp.h**. For a simple three layer feed-forward network and the default forward order (from the **Forw_order** class), the sequence of behavior function calls is:

   (a) hidden_layer . pre_forward_propagate()

   (b) input_hidden_connect . forward_propagate()

   (c) hidden_layer . post_forward_propagate()

   (d) output_layer . pre_forward_propagate()

   (e) hidden_output_connect . forward_propagate()

   (f) output_layer . post_forward_propagate()
3. Call **get_output** to get the output layer(s) activation vector(s) into an output array in the **Net** object.
4. Call **pre_back_pass** on the **Stats** object and **Net_parts**.

A listing of all of the above functions is included in the appendix.

4 CLONES Classes

To improve readability, all global symbols (including class names) start with a capital letter. As a general rule, CLONES does not have any global variables (the one exception is the connection creation objects such as **BP_full**). All global information is contained in the **Net** object and its subclasses. **Net** objects are passed explicitly to functions that need global information about the network and the state of the training run. This allows multiple "global" name environments to be maintained independently for each network.

The following subsections describe the CLONES classes in more detail.

4.1 Net_behavior

The CLONES training procedure consists of calling various behavior functions that can be found in a number of object classes and their subclasses, including: **Net**, **Layer**, **Connect** and **Stats**. **Net_behavior** is the parent of all these classes. It contains functions that are called before any training is done and after training is completed: **pre_training** and **post_training**. An epoch involves running a single pass over the database(s). **Net_behavior** also has functions that are called before and after running an epoch: **pre_epoch** and **post_epoch**.

Also, all CLONES objects have a function called **io** that allows the object's internal data structures to be saved or restored from a disk file. The first argument to **io** is a pointer to a FILE structure that was returned from the standard C function **fopen**. Data is transferred without first rewinding the file; this allows a series of **io** calls to read or write a sequence of objects in the same file. The second argument selects file reading or writing by using the same mode character string that was used as the second argument to **fopen** when the file was opened. [2] Often the **io** function simply calls the **io** functions of its components; in these cases the **io** function uses the same code for both reading are writing itself.

Some of the functionality of the **Net_behavior** class is summarized below:

Net_behavior Function	Description
print_name()	Print the symbolic name
set_name(char *name)	Set the symbolic name
print_class_name()	Print the symbolic name of the class
char *get_class_name()	Return the symbolic name of the class
Bool is_class(Class_tag *class)	Quick test of object class
Bool is_class(char *class_name)	Slow test of object class (strcmp)
Bool is_subclass(Class_tag *class)	Quick test of object parents
Bool is_subclass(char *class_name)	Slow test of object parents (strcmp)
Class_tag *get_class_tag()	Get the unique number for this class
Class_tag *get_parent_class_tag()	Get the unique number for parent
print_ident()	Print name, class, size, address, etc.
dump(int level)	Dump internal data structures
print(FILE* file, int level)	Pretty print internal data structures
io(FILE *file, char *mode)	Read or write object from/to a file
file_io(char *file_name, char *mode)	Open, read or write, then close
pre_training(Net*)	Called before a training run begins
post_training(Net*)	Called after a training run is complete
pre_epoch(Net*)	Called before a pass over the database
post_epoch(Net*)	Called after each pass over the database
pre_forw_pass(Net*)	Called before doing a forward pass on Net
post_forw_pass(Net*)	Called after a forward pass is complete
pre_back_pass(Net*)	Called before doing a backward pass on Net
post_back_pass(Net*)	Called after a backward pass is complete
set_learning_rate(float rate)	Set rate of adaptation (0.0 for no learning)
set_learning_mode(int mode)	Set the learning_mode variable (0 for no learning)

[2] Note that "rb" or "wb" must be used to open a file for reading or writing in BINARY mode. See the RAP software users manual for more details.

4.2 Net_part

A **Net_part** is a modular component used to build a network. Currently, **Layer** and **Connect** are the only subclasses. All network components respond to certain global commands from the **Net** object. These include routines to adjust the rate of adaptation during training (**set_learning_rate** and **set_learning_mode**).

There are also routines to dynamically change the number of units in a **Layer** and its **Connects** (**add_units** and **del_units**). When the size of a **Layer** is changed, by default it automatically calls functions **add_units** or **del_units** on its **Connects** so that they can adjust to the new number of units. Since these two functions are virtual, the **Layer** need not know the subclass of **Connect** being called.

All network components used to construct ANNs are derived from the two classes **Layer** and **Connect**. **Net_part** defines functions that are shared by all network components:

Net_part Function	Description
add_units(Net*,int n)	Increase the number of units (by n)
del_units(Net*,int n)	Decrease the number of units (by n)
int get_fixed_point()	Get the position of the binary point (SPERT only)
set_fixed_point(int)	Set the position of the binary point (SPERT only)
float get_avg_fan_in()	Return average input connections per unit
float get_avg_fan_out()	Return average output connections per unit
forw_propagate(Net*)	**Connect** adjusts output **Layer** based the input **Layer**
back_propagate(Net*)	**Connect** adjusts self based on output **Layer** error
forw_pre_propagate(Net*)	**Layer** setup for input **Connect forw_propagate** calls
back_pre_propagate(Net*)	**Layer** setup for output **Connect back_propagate** calls
forw_post_propagate(Net*)	**Layer** compute after all input **Connect forw_propagate** calls
back_post_propagate(Net*)	**Layer** compute after all output **Connect back_propagate** calls

Note that these functions define the abstract interface to a **Net_part**. Subclasses of **Net_part** such as **Layer** or **Connect** must provide the C++ code that defines their behavior.

The **Net_part** class contains propagation behavior functions that are used by both **Layer** and **Connect** subclasses. One may wonder why these functions are defined in the **Net_part** class instead of in the **Layer** and **Connect** classes. The primary reason is to allow a sequence of calls to these functions to be maintained as a table (in subclasses of **Net_order**) with entries of a single data type: pointer to member function of **Net_part**. This allows very efficient calling of these behavior (propagate) functions without the need to first check if it is a **Layer** or **Connect** object.

Net_part has member variables that control the learning process in general. These include the integer **learning_mode** and the floating point **learning_rate**. The meaning of these variables is defined in subclasses of **Net_part**, but it is universal that a **learning_mode** of zero freezes any adaptation of that network component. New learning control variables are currently being added (such as **weight_decay_mode** and **decay_rate**).

4.3 Layer

The network behavior of a **Layer** class is specified by functions that initialize it for a forward or backward pass (**pre_forw_propagate** and **pre_back_propagate**), and functions that produce activations or errors from the partial results left from running connections to the **Layer** (**post_forw_propagate** and **post_back_propagate**).

The number of units in a **Layer** is returned by **get_n_unit**. Each **Layer** contains two lists of **Connect** object pointers: **Connects** coming into the **Layer** and **Connects** going out of the **Layer**. The size of these lists in returned by **get_n_in_connect** and **get_n_out_connect**. Input **Connects** of a **Layer** are returned from function **get_in_connect(connect_number)**. Output **Connects** of a **Layer** are returned by function **get_out_connect(connect_number)**. The **connect_number** above starts at zero for the first connection.

The functions of a **Layer** are summarized in this table:

Layer Function	Description
new Layer(int)	Create a layer with (int) units
copy(Layer* lay)	Copy all internal parameters of lay into this **Layer**
int get_n_unit()	Get the number of units in **Layer**
Connect *get_in_connect(int)	Get the n^{th} input connection to **Layer**
Connect *get_out_connect(int)	Get the n^{th} output connection from **Layer**
int get_n_in_connect(int)	Get number of input connects
int get_n_out_connect(int)	Get number of output connects
put(float*,int,int)	Set activation 1d vector (array,size,offset)
get(float*,int,int)	Get activation 1d vector (array,size,offset)
put(float*,int,int,int)	Set activation 2d matrix (array,size,x,y)
get(float*,int,int,int)	Get activation 2d matrix (array,size,x,y)
put(float*,int,int,int,int)	Set activation 3d array (array,size,x,y,z)
get(float*,int,int,int,int)	Get activation 3d array (array,size,x,y,z)
put(int*,int,int)	Set activation 1d vector (array,size,offset)
get(int*,int,int)	Get activation 1d vector (array,size,offset)
put(int*,int,int,int)	Set activation 2d matrix (array,size,x,y)
get(int*,int,int,int)	Get activation 2d matrix (array,size,x,y)
put(int*,int,int,int,int)	Set activation 3d array (array,size,x,y,z)
get(int*,int,int,int,int)	Get activation 3d array (array,size,x,y,z)

A **Layer** can define **put** and **get** functions that allow the units to be accessed as a sequence of floating point numbers. Both of these functions take as their arguments: a pointer to an array of floating point numbers (float*), the number of units to transfer (int) and the starting unit index to transfer (int). If only the first argument is used, the other two default so that the entire activation array will be transferred. If only the first two arguments are specified, the starting unit index defaults to zero.

Multidimensional **put** and **get** functions are also available allowing the same **Layer** to be accessed as a one, two or three dimensional array of activations. For each of the **put** and **get** functions above there is a function for integer activations as well as floating point. A **Layer** that represents activations in one way will have functions that convert from their representation to other supported representations. For example, a **Layer** that internally represents activations as an array of fixed point or integer numbers will include functions that provide access to these activations as a floating point array.

The **Analog_layer** class specifies the representation of activations, but it still leaves open [3] the procedures that use and update the activation array. The class **BP_analog_layer** is a subclass of **Analog_layer** that defines behavior (propagate) functions for the backpropagation algorithm.

4.4 Connect

Each **Connect** object connects two **Layer** objects. The two **Layers** are refered to as "incoming" and "outgoing" since often during a forward pass the **Connect** moves data from incoming to outgoing. The **Connect** has two behavior functions: one that transforms activations from the incoming **Layer** into partial results in the outgoing **Layer** (**forward_propagate**), and one that takes outgoing **Layer** errors and generates partial results in the incoming **Layer** (**backward_propagate**). The structure of a partial result is part of the **Layer** class. In order to do its job efficiently, a **Connect** must know something about the internal representation of the **Layers** that are connected. The **Connect** function selected depends not only on the class of **Connect**, but also on the classes of the two layers that are connected. This is accomplished by connect creation objects. These are the only global objects in CLONES. Their only purpose is to create a **Connect** object of the correct type given the classes of the two **Layers** being connected. For example, a backpropagation connect creation class for analog or binary full connections might look like:

```
// The only purpose of class BP_full_select is to create new
// instances of BP_full_aa, BP_full_ba, etc. depending on the layer classes.
// There is only one instance of this class called BP_full.

class BP_full_select {
 public:
  BP_full *connect(BP_analog_layer *a1, BP_analog_layer *a2)
    { return((BP_full*) new BP_full_aa(a1,a2)); }

  BP_full *connect(BP_binary_layer *b1, BP_analog_layer *a2)
    { return((BP_full*) new BP_full_ba(b1,a2)); }

  BP_full *connect(BP_binary_layer *b1, BP_binary_layer *a2)
    { return((BP_full*) new BP_full_bb(b1,a2)); }
};

// create the global BP_full_connect (of class BP_full_select)
BP_full_select BP_full_connect;
```

To create a fully connected backpropagation connection between two analog or binary **Layers**:

[3]These functions are specified in C++ as virtual. This means that redefining the function in a subclass is visible even when an object of that subclass is passed as an abstract object of the parent class, or grand-parent class, etc.

```
BP_full_connect.connect(input_layer, output_layer);
```

A **Connect** object contains a pointer to the input **Layer** and a pointer to the output **Layer** of the connection. These are returned by functions **get_in_layer** and **get_out_layer**.

Connect objects also may define the member function **share**; this allows two compatible **Connect** objects to share parameters. For example, a **BP_analog_connect** object can share weights with another object of the same class. Again, compatibility between the **Connects** is enforced by compile time type checking by the C++ compiler. Several (overloaded) **share** functions allow a **Connect** to share parameters with any number of other **Connect** classes.

The functions of the **Connect** are summarized below:

Connect Function	Description
Connect *Connect(Layer*,Layer*)	Create a new connection between two layers
copy(Connect* conn)	Copy all internal parameters of conn into this **Connect**
Layer *get_in_layer()	Get the input layer of this **Connect**
Layer *get_out_layer()	Get the output layer of this **Connect**
share(Connect_subclass*,Net*)	Share parameters between two **Connects**

4.5 Net_order

The order in which behavior functions in **Layer** and **Connect** objects are called is determined at run-time by utilizing a **Net_order** object. Each **Net_order** object is associated with a single **Net** object. The **build** function of a **Net_order** object prepares it for calls to its **run** function (this may involve a scan of the **Net** topology). The **run** function executes a sequence of **Net_part** behavior functions. Note that the order of network component execution may be statistical; Network components may be executed more or less than once per pass. A subclass of **Net_order**, called **List_order**, contains an ordered list of **Net_part** pointers. This subclass is currently used for running the same sequence of behavior calls repeatedly. The following code demonstrates how to iterate over the **Net_parts** of a **Net_order** to set the learning rate for each **Layer** and **Connect**:

```
Net_part *part;
Net_order *order;
int part_index;

for(part_index = 0; part_index < order->get_n_parts(); part_index++) {
  part = order->get_part(part_index);
  // do something to each part of this net ordering
  // in this case, set the learning rate for all connections to zero
  // (layers may still have non-zero learning rate for biases)
  if (order->is_connect(part_index))
    part->set_learning_rate(0.0);
}
```

A**Net** object contains three **Net_order** objects: one for the forward pass order, one for the backward pass order and one that lists all the network parts once for use as a parts list. It is often useful to loop over all **Net_part** objects in a **Net**. The **Net** object has functions **get_n_parts** and **get_part** with the same interface as a **Net_order** object. In the above example, **Net_order** could be replaced with **Net** to set the learning rate of all network connections. Since iterating over all parts of a net happens frequently in CLONES, there is a macro called **LOOP_OVER_ALL_PARTS** that allows such a loop with a single line body to be written in one line. For example, to set the learning rate of all parts to zero:

```
LOOP_OVER_ALL_PARTS(net_or_order_object,
    part->set_learning_rate(0.0));
```

The macro has three local variables: **part** is a pointer to the current **Net_part**, **part_id** is the index number of the **Net_part** and **n_part** is the total number of parts in the **Net**. Here is the definition of this macro:

```
#define LOOP_OVER_ALL_PARTS(NET,FUNC) \
{ \
  Net_part *part; \
  int part_id, n_part; \
  n_part = (NET)->get_n_part(); \
  for(part_id=0; part_id<n_part; part_id++) { \
    part = (NET)->get_part(part_id); \
    FUNC; \
  } \
}
```

A **Net_order** object is prepared for use by calling its **build** function. This function takes a **Net** and scans it to make an internal list of **Net_parts**. Different subclasses of **Net_order** have redefined **build** to create different orderings. For example, **Forw_order** and **Back_order** classes generate orderings used by feed-forward backpropagation networks. Note that these orderings need not contain all the **Net_parts** of the **Net**, and may contain multiple entries for the same **Net_part** (often with a different operation code).

Each **Net_part** in a **Net_order** object also has an operation code number. This number can be used to indicate what should be done to the **Net_part** when the **run** function is called on the **Net_order** object. For example, there are different operation codes for running **pre_forward** and **post_forward** functions on a **Layer** object. These codes and their meaning are hidden inside the **build** and **run** functions of the **Net_order** class. Calling **run** on a **Net_order** class causes functions to be called on each **Net_part** depending on its operation code.

This table summerizes member functions of the **Net_order** class:

Net_order Function	Description
int get_n_part()	Return total number of parts
Net_part *get_part(int)	Return the n^{th} part
int get_operation_code(int)	Return the n^{th} operation code
build(Net*)	Build internal data structures for network
run()	Run each **Net_part** in order, function called depends on operation code
int is_layer(int n)	Return true if n^{th} **Net_part** is a **Layer**
int is_connect(int n)	Return true if n^{th} **Net_part** is a **Connect**

4.6 Net

The **Net** object contains three lists of **Layer** pointers: input, hidden and output. Each list may contain references to any number of **Layers**. Several **Net** objects may refer to the same layers, which is useful for training and testing subnets in a larger **Net**. A **Net** object also has **Net_order** objects that are responsible for determining the order in which **Layer** and **Connect** functions are called to run a forward or backward pass.

New **Net** objects are created by calling one of the constructor functions in a subclass of **Net**. In C++ this is accomplished by the following syntax:

```
new_net_object = new Net_subclass_name(
  arguments such as: sizes, Layers, Connects, or Nets)
```

For example, the constructor for a three layer **Net** might look like:

```
new_net_object =
  new Three_layer_MLP(
    // param is a standard parameter list holding object (class Param)
    param,
    // NULL layer below creates the layer
    input_layer, hidden_layer, output_layer,
    // size required below if NULL above for layer above
    input_size, hidden_size, output_size
    // the size arguments are optional when existing layers are used
  );
```

Often there are several constructors for the same class of **Net**. **Net** constructors may create new **Layer** and **Connect** objects. They may also take existing **Layer**, **Connect** or **Net** object as arguments allowing new **Net** objects to add structure to an existing network. The **Net** constructor can also call the constructors of other (or the same) **Net** class to create modular hierarchically structured topologies.

The **Net** class inherits all functions in class **Net_behavior**. It adds these functions:

Net Function	Description
new_input_layer(Layer*)	Add another input layer to input layer list
new_hidden_layer(Layer*)	Add another hidden layer to hidden layer list
new_output_layer(Layer*)	Add another output layer to output layer list
build()	Setup for current set of inputs, hiddens and outputs
int get_n_input_layer()	Return number of input layers
int get_n_hidden_layer()	Return number of hidden layers
int get_n_output_layer()	Return number of output layers
Layer *get_input_layer(int)	Return n^{th} input layer
Layer *get_hidden_layer(int)	Return n^{th} hidden layer
Layer *get_output_layer(int)	Return n^{th} output layer
Bool is_input_layer(Layer*)	True if argument is an input layer
Bool is_hidden_layer(Layer*)	True if argument is a hidden layer
Bool is_output_layer(Layer*)	True if argument is an output layer
del_input_layer(Layer*)	Delete an existing input layer from list
del_hidden_layer(Layer*)	Delete an existing hidden layer from list
del_output_layer(Layer*)	Delete an existing output layer from list
int get_n_part()	Return number of **Layers** and **Connects**
Net_part *get_part(int)	Return the n^{th} **Layer** or **Connect**
Bool is_layer(Net_part*)	Return true if **Net_part** is a **Layer**
Bool is_connect(Net_part*)	Return true if **Net_part** is a **Connect**
int get_n_input_unit()	Return total number of units in all input **Layers**
int get_n_output_unit()	Return total number of units in all output **Layers**
put_input(float*)	Put activation vector into all input **Layers**
get_output(float*)	Get output activation from all output **Layers**
put_error(float*)	Put error vector into output **Layers**
run_training()	Run a complete training
run_epoch(stage)	Run a pass over the database
run_forw(int)	Run a forward pass for database pattern sequence number
run_back()	Run the backward pass for current pattern
void select_database(Stage)	Select database(s) for current phase of training
int pattern_order(int)	Get database pattern number from sequence number
int pattern_select(int)	Return 0 to skip presentation of pattern number
set_input(int)	Set input **Layers** for a pattern number from **Database**
set_target()	Set the desired **Net** output from the **Database(s)**
set_error()	Set the error vector based on **Net** output and target(s)
Bool is_correct()	Returns true if output is “correct”

The **build** function of a **Net** object prepares the current network topology and configuration for training or running. It is responsible for calling the **build** functions of all internal **Net_order** and **Net** objects. It also sets up arrays that are used as staging areas for the input, output and error vectors. Any time the topology changes, **build** must be called at least once to update the **Net** object.

Since CLONES does not have global variables, the **Net** object serves as a place to put data structures that are of general use to a particular network. The advantage over global variables is that **Nets** with their own global environments can coexist in the same program without name conflicts. The table below briefly describes the variables that are in a **Net**:

Net Variable	Description
param	Parameter object pointer for command line or parameter file variables
db_order	Select pattern order algorithm for training (REAL, RANDOM_PATTERN, ...)
epoch_num	Number of passes over database that have been run
pattern_num	Pattern number of current pattern being run
stage	Current phase of training (PRE_TRAIN, TRAIN, TEST, POST_TRAIN)
target	Current target category for output (categorization tasks only)
input	Floating point array of current input data for the **Net**
output	Floating point array of current output data from **Net**

Subclasses of **Net** often add their own variables. For example **BP_net** adds these variables:

BP_net Variable	Description
learning_rate	Current learning rate
ramping_learning_rate	Flag indicating that learning rate is ramping down
label_output_unit	Array of output unit number to train for each category
error	Floating point array of current errors for each output

4.7 Database

The **Database** class is independent of the rest of CLONES and can be used separately. It serves as an abstract interface that allows training examples (represented as floating point vectors) to be retieved by pattern number. The training pattern consists of one or more frames of data from the database. There is also a member function to get "labels" that identify (or classify) each frame. Since **Database** is abstract, it can not be constructed. A subclass of **Database** called **Sentence_database** implements this model very specifically for speech tasks. It adds many functions to the interface that implement a sentence level of structure.

A database object allows access to a sequence of frames. Each frame has a fixed number of floating point fields. The total number of frames in a database object can be obtained by calling **get_n_frames**. The number of fields in a frame is returned by **get_frame_size**. Not all of the fields of a frame hold input data for training the net, some may be used for identification of the pattern, the correct classification or target output, and other information needed by the trainer. Note that all fields of a frame are floating point; integer information is stored in floating point and converted as required.

Each frame has a unique frame_id from zero to **get_n_frames()-1**. To get a complete frame, call **get_frame(frame_id, output_array)**. To get just the fields of the frame that contain input data for the network, call **get_data(frame_id, output_array)**.

One or more frames are combined to make a complete input pattern for the input **Layers** of the **Net**. Patterns are numbered by a pattern_id from zero to **get_n_patterns()-1**. The number of patterns is often different than the number of frames. To get the complete network input data for pattern pattern_id, call: **get_pattern(pattern_id, output_array)**. This is usually called from the **set_input** function of the **Net** class. The default **get_pattern** function assembles a context window of frames around the center frame:

```
void
Sentence_database::get_pattern(int pattern_num, float *array)
{
  int frame_id, i, index;

  frame_id = pattern_center_frame_id(pattern_num) - half_window_size;
  index = 0;
  for(i=0; i<window_size; i++) {
    get_data(frame_id + i, &array[index]);
    index += data_size;
  }
}
```

Objects of class **Database** can not be created since some of its member functions are declared as null slots; it is an abstract class. The subclass **Sentence_database** adds functionality to support speech databases. The only functions that are not defined in **Sentence_database** are the low level functions that access the data. These include: **get_frame**, **put_frame**, **set_size**, and **data_io**. **Sentence_database** has two subclasses: **Disk_database** and **RAM_database**. These two classes are fully specified; their constructors can be called to create new objects. The only difference between them is that **Disk_database** holds the frame data in a binary datafile on disk while **RAM_database** holds this data in a distributed matrix (in DRAM).

Objects of these classes can be easily created:

```
database = new Disk_database(number_of_frames_in_window, number_of_labels_per_frame);

database = new RAM_database(number_of_frames_in_window, number_of_labels_per_frame);
```

The **Disk_database** data file must be opened in binary mode ("rb"). Since the file handle is used during training, the **fclose** function should not be called until one is finished using the **Disk_database**. In the case of **RAM_database**, **fclose** can be called on the file as soon as all calls to the **io** function are completed.

A **Sentence_database** contains a sequence of numbered sentences. The number of sentences is returned from **get_n_sentences**. Each sentence has a number of frames returned by **sent_n_frames(sentence_id)**. The first frame_id of a sentence is returned by **sent_first_frame_id(sentence_id)** and the last frame_id is returned by **sent_last_frame_id(sentence_id)**. The absolute sentence_id of a frame is **get_sent_id(frame_id)**. Note that the sentence_id is a field of the **Database** and may not begin at zero and may skip around. The relative sentence_number counts the sentences in order starting at zero. It can be obtained by calling the function **get_sent_num(frame_id)**. To get the relative number of frames into the sentence from an absolute frame_id, use **get_frame_num(frame_id)**.

The relationship between pattern_id and frame_id depends on the number of frames in the pattern window. If the pattern window is one frame then the pattern_id is equal to the frame_id. If the window were three, the first pattern_id (zero) corresponds to the first three frames of the first sentence (centered on frame one). The pattern window always includes frames from the same sentence. To get the center frame_id for a given pattern_id, use **pattern_center_frame_id(pattern_id)**. The inverse, the pattern_id of frame_id is **frame_pattern_id(frame_id)**. The first or last pattern_id in sentence sent_id is returned by **sent_first_pattern_id(sent_id)** or **sent_last_pattern_id(sent_id)**.

Each **Database** frame has a fixed number of category labels. The number of categories associated with a frame is returned by **get_label_per_frame**. The default is one label per frame. Each label field is an integer ranging from zero to **get_n_label(label_field_num)**. The integer label value for label label_field_num of frame_id is **get_frame_label(frame_id,label_field_num)**. To get the category for label_num of the center frame in pattern_id, use **get_pattern_label(pattern_id)**.

All frames with a given label can be quickly obtained using hidden index tables of the **Sentence_database**. A frame_id with label_value in the first label field is returned by **label_frame_id(label_value, example_num)**. where example_num ranges from zero to **label_n_frames(label_value)-1**.

To read or write a database file from a **Sentence_database** object, the function **io** is called:

```
sentence_database_object->io(
  file_pointer, mode_string, number_of_sentences)
```

The mode_string is one of the following: "r" (read ascii), "w" (write ascii), "rb" (read binary) or "wb" (write binary). Binary databases are about half the size of ascii database files and can be read many times faster since ascii to floating point conversion is not required. A **Disk_database** object can only be created for a binary database. All floating point numbers in binary files use the IEEE 32-bit standard.

The functions of a **Database** object include:

Database Function	Description
int get_n_frame()	Return total number of frames
int get_frame_size()	Return number of fields in a frame
int get_data_size()	Return number of data fields in a frame
int get_label_per_frame()	Return number of label fields in a frame
get_frame(int,float*,int,int)	Get frame_id into floating array (size and offset)
float get_frame(int,int)	Return field value given frame_id and field number
float get_frame_label(int,int)	Return label field value given frame_id and label field number
put_frame(int,float*,int,int)	Put frame_id from floating array (size and offset)
get_data(int,float*)	Get the data part of frame_id into array
io(FILE*,const char*,int)	Read or write database in binary or ascii
int get_pattern_size()	Return the number of fields in a pattern
int get_n_pattern()	Return the total number of patterns in database
get_pattern(int,float*)	Get data for a complete pattern given pattern_id
int get_n_label(int)	Return number of categories given label field number
int get_pattern_label(int,int)	Return category number given pattern_id

Sentence_database adds these functions to the interface:

Sentence_database Function	Description
Sentence_database(int,int)	Create **Sentence_database** given window size and labels per frame
int get_window_size()	Return number of frames in a context window
int get_n_sentence()	Return number of sentences in database
int get_n_label_per_frame()	Return number of labels on each frame
int get_frame_label(int,int)	Return label for frame_id and label number
int get_sent_id(int)	Return absolute database sentence_id field for frame_id
int get_sent_num(int)	Return relative sentence number of frame_id in this database
int get_frame_num(int)	Return frame offset within sentence of frame_id
int sent_first_frame_id(int)	Return first frame_id of relative sentence number
int sent_last_frame_id(int)	Return last frame_id of relative sentence number
int sent_n_frames(int)	Return number of frames in relative sentence number
int label_n_frames(int)	Return number of frames with given category number
int label_frame_id(int,int)	Return frame_id with given category and example index number
int pattern_center_frame_id(int)	Return center frame_id given pattern_id
int sent_first_pattern_id(int)	Return first pattern_id of given sentence number
int sent_last_pattern_id(int)	Return last pattern_id of given sentence number
int sent_n_pattern(int)	Return number of patterns in sentence number
int frame_pattern_id(int)	Return pattern_id for a given frame_id
set_select_input(int,int*)	Select fields and frames to use in pattern
compute_stats()	Computes means and variances for each data field
normalize_data()	Scale data (in place) to be normal(0,1)
copy_stats(Sentence_database*)	Copy the statistics from another database

4.8 Param

Like the **Database** class and its subclases, the **Param** classes are independent of the rest of CLONES and can be used separately. **Param** objects contain a symbol table that associates each parameter name with a list of one or more string values. Parameters from the command line can be entered into a **Param** object by calling the **parse** function:

```
parse(int argc, char **argv, Param_mode mode)
```

Or, parameters can be taken a file by calling the **parse** function:

```
parse(char *file_name, Param_mode mode).
```

The **Param_mode** argument has three possible values: **OVERRIDE**, **UNDERRIDE** and **APPEND**. These determine what happens when a parameter that already has a value in the **Param** object, is encountered again. **OVERRIDE** throws out the old value before adding new values. **UNDERRIDE** keeps the old value and ignores the new ones. **APPEND** adds the new value to the end of the existing list of strings.

Parameters in files or on the command line have the same format. They consist of a sequence of parameter declarations. Each declaration has a parameter name preceded with a minus sign followed by an (optional) list of parameter values. The list of values is delimited by the next parameter declaration or the end of file. When parsing from a file, a line that begins with "#" will be skipped as a comment. For example, this is a simple command line with two parameters:

```
-db database -hidden_layers 16 -256
```

The above would set parameter db to the one entry list "database" and parameter "hidden_layers" to the two entry list: "16 -256".

Parameter values that are parsed before any minus prefixed parameter name are associated with a default parameter name. For example, to use the first parameter values in the command line as a list of file names containing more parameters:

```
  // Set the default parameter name for values that occur before any
  // -parameter_name tokens to "parameter_files".
  param.set_default_param_name("parameter_files");

  // Add the parameters from the command line to the parameter list.
  param.parse(argc,argv,OVERRIDE);

  // Any values that come before the first parameter name are added as
  // "parameter_files".  In this case, we use these arguments
  // as file names that contain more parameters.
  for(i = 0; param.get_string(param_files,NULL,i) != NULL; i++) {
    param.parse(param.get_string(param_files,NULL,i),UNDERRIDE);
  }
```

The value of parameters can be returned in many ways. To get the n^{th} string associated with parameter xyz:

```
get_string("xyz", default_value, n)
```

The **default_value** is returned if there is no such parameter. The last argument indexes the array of strings for the parameter; it defaults to 0, returning the first string listed under parameter xyz.

Other routines that get parameter values include:

Param Function	Description
Param()	Create a new parameter object

parse(int, char**, Param_mode)
parse(char*, Param_mode)
	Parse a C style command line argument list
	Parse from a file given the file name and OVERRIDE control

set_default_param_name(char*)	Set parameter name for values that proceed a parameter -flag
int get_bool(char*)	Return 1 if name exist (may not have any string values)
int get_size(char*)	Return the number of strings associated with name
int get_int(char*,int,int) float get_float(char*,float,int) char *get_string(char*,char*,int) char *copy_string(char*,char*,int)	Get integer parameter given name, default value and index Get floating parameter given name, default value and index Get string parameter given name, default value and index Same as above but returns copy of string (user must free it)
int *get_int_array(char*) int *get_float_array(char*)	Get all integers associated with name as an array Get all floats associated with name as an array
int *get_range_array(char*,int,int,int)	Get array of ranges as a table of bools
int *get_mapping(char*,int*,int)	Get a table of pairs of indexes for an index mapping
print(FILE*)	Print out a list of current parameters and their values

Parameters can be checked against a master parameter table to avoid misspelling. The table also contains documentation used to produce a usage message in the case of a command line or parameter file error. This master table is kept in a file called parameters.h. To add a new parameter to CLONES, one line must be added to this file. The line contains a pair of string constants: the name of the parameter and one line of documentation. Here is an example of this file:

```
char* Parameters[ ] = {
  "-debug","debug level (int)",
  "-db","database file name (string)",
  "-net_read", "read weights from file",
  "-net_write", "write weights to a file",
  "-window", "number of frames in an input pattern",
  "-output_size", "size of output layer",
  "-hidden_size", "size of hidden layer(s)",
  "-db_order", "order of presentation of patterns during training",
  "-train_sent", "number of training sentences",
  "-test_sent", "number of testing sentences",
  "-label_per_frame", "number of labels (categories) for each frame",
  "-normalize", "normalize all data based on stats for whole training set",
  "-initial_learning_rate", "initial rate of adaptation",
  "-min_random_bias", "smallest random number for initial biases",
  "-max_random_bias", "largest random number for initial biases",
  "-min_random_weight", "smallest random number for initial weights",
  "-max_random_weight", "largest random number for initial weights",
  NULL, NULL, // mark end of table
};
```

4.9 Stats

A **Stats** object holds performance statistics accumulated over the course of training and testing epochs. The **Net** object contains two **Stats** objects, one for the training database and one for the database used for testing generalization. Here is an example that demonstrates how simple statistics can be obtained by using the functions inherited from **Net_behavior**.

```
// Stats objects are used to collect statistics about the training
class Stats : public Training_behavior {
 private:
  int total_try, total_correct;            // counters
  float correct, prev_correct;             // % correct on this and prev cycle
 public:
  Stats() { reset(); }
  virtual void reset()
    { correct = prev_correct = total_try = total_correct = 0; }

  // at the beginning of an epoch, zero the counters
  void pre_epoch(Net*) {
    total_try = total_correct = 0;
  }

  // after each forward pass, keep count of if it was correct
  void post_forw_pass(Net*) {
    total_try++;
    if (net->is_correct())
      total_correct++;
  }

  // at the end of an epoch, print some stats
  void post_epoch(Net*) {
    enum Stage stage;
    stage = net->stage;
    if (stage != TRAIN && stage != TEST)
      return;
    prev_correct = correct;
    correct = (float)total_correct * 100.0 / (float)total_try;
    printf("%s%7.3f%% %s correct (%d out of %d)\n",
      stage != TEST? "        " : "",
      correct, stage == TEST? "test" : "train",
      total_correct, total_try);
  }
};
```

5 CONCLUSIONS

CLONES is a useful set of tools for training ANNs especially when working with large training databases and networks. It runs efficiently for networks of interest on parallel hardware we have developed. Extending CLONES to cover a wider variety of connectionist algorithms is currently in progress. The source code for CLONES will become available through anonymous FTP.

ACKNOWLEDGEMENTS

Special thanks to Steve Renals for daring to be the first CLONES user. Others who provided valuable input to this work were: Jerry Feldman, Krste Asnovic, Nelson Morgan, James Beck and Chuck Wooters. Support from the International Computer Science Institute is gratefully acknowledged.

6 Appendix: CLONES code

```
// do a complete training run
// run patterns until training criterion is reached
void
BP_net::run_training()
{
  // run pre_training function on all Net components
  pre_training(this);

  test_stats->pre_training(this);
  train_stats->pre_training(this);

  if (!param->get_bool("skip_pre_train_pass")) {
    select_database(TRAIN);
    run_epoch(PRE_TRAIN);
    select_database(TEST);
    run_epoch(PRE_TRAIN);
  }

  reset_learning_rate();

  // run test set first for baseline needed by next_learning_rate
  select_database(TEST);
  run_epoch(TEST);

  do {
    select_database(TRAIN);
    run_epoch(TRAIN);
    select_database(TEST);
    run_epoch(TEST);
  } while(next_learning_rate());

  if (!param->get_bool("skip_post_train_pass")) {
    select_database(TRAIN);
    run_epoch(POST_TRAIN);
    select_database(TEST);
    run_epoch(POST_TRAIN);
  }

  post_training(this);

  test_stats->post_training(this);
  train_stats->post_training(this);
}

/**********************************************************************/

void
Net::pre_training(Net *net)
{
  training_epoch_num = epoch_num = 0;
  LOOP_OVER_ALL_PARTS(net, part->pre_training(net));
}


// run a complete pass
void
Net::run_epoch(Stage new_stage)
{
  int pat, npat;
  Net *net = (Net*) this;

  epoch_num++;
  if (stage == TRAIN)
    training_epoch_num++;
  stage = new_stage;
  stats->pre_epoch(net);
  pre_epoch(net);

  npat = get_n_patterns();

  for(pat = 0; pat < npat; pat++) {
    // get order of pattern presentation (from sequence number pat)
    pattern_num = pattern_order(pat);

    // see if this pattern is selected for presentation (this time around)
    if (! pattern_select(pattern_num))
      continue;

    // user's routine to set input layers of net
    set_input(pattern_num);

    // set Net target variable
    set_target();

    switch(stage) {
    case TRAIN:
      train(pat);     break;
    case TEST:
      test(pat);     break;
    case PRE_TRAIN:
      pre_train(pat);     break;
    case POST_TRAIN:
      post_train(pat);     break;
    default:
      printf("bad stage %d\n", stage);
      exit(-1);
    }
  }

  stats->post_epoch(net);

  post_epoch(net);
}

// Reset the learning rate schedule for a new training run.
void
BP_net::reset_learning_rate()
{
  if (param->get_bool("learning_rates")) {
    learning_rate = param->get_float("learning_rates",0.0);
  } else {
    learning_rate = param->get_float("initial_learning_rate",0.05);
  }
  printf("initial_learning_rate = %f\n", learning_rate);
  set_learning_rate(learning_rate);
  ramping_learning_rate = 0;
}
```

```
// Set next learning rate based on stats from last epoch
// returns 0 when training is complete
int
BP_net::next_learning_rate()
{
  if (param->get_bool("learning_rates")) {
    learning_rate = param->get_float("learning_rates", 0.0, training_epoch_num);
  } else {

    if (test_stats->correct - test_stats->prev_correct <
        param->get_float("ramp_threshold",0.5))
    {
      // Improvement in the test score was less than 0.5% (or rap_threshold if set)
      // If we are already ramping the learning rate, then we are done.
      if (ramping_learning_rate)
        return(0); // done training

      // Start ramping learning rate
      ramping_learning_rate = 1;
    }

    if (ramping_learning_rate)
      learning_rate = learning_rate / param->get_float("divide_learning_rate",2.0);
  }

  set_learning_rate(learning_rate);
  printf("       learning rate = %f\n", learning_rate);

  if (learning_rate == 0.0)
    return(0);

  return(1); // keep training
}

void
Net::post_training(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->post_training(net));
}


/**********************************************************************/


void
Net::pre_epoch(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->pre_epoch(net));
}


void
BP_net::train(int pattern_num)
{
  run_forward();
  set_error();
  stats->pre_back_pass(this);
  run_back_order();
  stats->post_back_pass(this);
}


void
Net::test(int pattern_num)
{
  run_forward();
}


void
Net::post_epoch(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->post_epoch(net));

  if (param->get_bool("learning_rates")
  || (stage == TEST && improved()))
    write_weights();
}

int
Net::improved()
{
  return(test_stats->correct == test_stats->best_correct);
}

/*********************************************************************/


// run forward
void
Net::run_forward()
{
  pre_forw_pass(this);
  stats->pre_forw_pass(this);

  // actually run the forward pass of the net
  run_forw_order();

  // get output from output layer(s)
  get_output(output);

  post_forw_pass(this);
  stats->post_forw_pass(this);
}

// set_error sets the error vectors for Net output layers.
// This can be redefined to change the way errors are generated
// from the database and target.

void
```

```
BP_net::set_error()
{
  assert(target >= 0 && target < n_output);

  // set the errors for each output unit
  copy_v_v(sizeof(float) * n_output, output, error);
  error[target] = error[target] - 1.0;

  // put the errors into the output layer(s)
  put_error(error);
}

// return true if Net output is considered "correct".
// this is called by the stats routine (post_forw_pass)
int
BP_net::is_correct()
{
  float max;
  int max_output;

  // get the max element index in the output array of floats
  max_output = max_v(n_output, output, &max);

  if (max_output == target)
    return(1);

  return(0);
}

void
Net::pre_back_pass(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->pre_back_pass(net));
}

void
Net::post_back_pass(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->post_back_pass(net));
}


/*********************************************************************/


// set_input sets the Net input layers based on database pattern pat_num.
// This function can be redefined to select and process data
// from the database(s) before putting it into the input layers.
void
Net::set_input(register int pattern_num)
{
  // be sure we got a valid pattern number
  assert(pattern_num >= 0 && pattern_num < db->get_n_patterns());

  // get the pattern from the database
  db->get_pattern(pattern_num, input);

  // if you want hack the pattern data, redefine transform_data
  transform_data(input);

  // put the pattern into the input layers of the net
  put_input(input);
}


// Get pattern number given a sequence number from 0 to n_pattern-1
// The db_order variable in BP_net is used to select a pattern order.
// See the comment in clones.h where the enumeration DB_order is defined.

int
BP_net::pattern_order(int seq_num)
{
  register int pattern_num;
  int label, n_pat;
  DB_order pat_order = db_order;
  Sentence_database *sdb = (Sentence_database*) db;

  if (stage != TRAIN)
    pat_order = REAL;

  switch(pat_order) {
  // choose a random pattern from the database
  case RANDOM_PATTERN:
    pattern_num = db->random(db->get_n_patterns());
    break;

  // get patterns in database order
  case REAL:
    pattern_num = seq_num;
    break;

  default:
    panic("bad DB order %d\n", db_order);
  }
  return(pattern_num);
}

int
BP_net::pattern_select(register int pattern_id)
{
  int label;

  if (label_output_unit != NULL) {
    label = db->get_pattern_label(pattern_id, target_label_num);
    if (label >= n_label_output_unit)
      return(0);
    if (label_output_unit[label] >= 0)
      return(1);
    else
      return(0);
  }
  return(1);
}

// set_target sets the target variable for the current
// This can be redefined to process the target numbers given by the
// database(s).
void
BP_net::set_target()
{
  Sentence_database *sdb = (Sentence_database*) db;
  assert(target_label_num >= 0 && target_label_num < sdb->n_label_per_frame);
  target = db->get_pattern_label(pattern_num, target_label_num);

  // check if there is a mapping in effect
  // between target labels and output units
  if (label_output_unit != NULL) {
    assert(target < n_label_output_unit);
    target = label_output_unit[target];
    assert(target >= 0 && target < n_output);
  }
}

void
Net::pre_forw_pass(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->pre_forw_pass(net));
}

void
Net::post_forw_pass(Net *net)
{
  LOOP_OVER_ALL_PARTS(net, part->post_forw_pass(net));
}



/***********************************************************************/

void
Forw_order::run()
{
  register int i, n;
  register Net_part *s;  // connect or layer to run

  n = order.size;
  for(i=0; i<n; i++) {
    s = (Net_part*)order[i];
    switch((Order_call)ops[i]) {
    case CONNECT:  // connection
      s->forw_propagate(net);
      break;
    case PRE_CONNECT:  // layer
      s->forw_pre_propagate(net);
      break;
    case POST_CONNECT: // layer
      s->forw_post_propagate(net);
      break;
    default:
      panic("bad op code in Net_order %d", (int)ops[i]);
    }
  }
}

void
Back_order::run()
{
  register int i, n;
  register Net_part *s;  // connect or layer to run

  n = order.size;
  for(i=0; i<n; i++) {
    s = (Net_part*)order[i];
    switch((int)ops[i]) {
    case CONNECT:
      s->back_propagate(net);
      break;
    case PRE_CONNECT:
      s->back_pre_propagate(net);
      break;
    case POST_CONNECT:
      s->back_post_propagate(net);
      break;
    default:
      panic("bad op code in Net_order %d", (int)ops[i]);
    }
  }
}

/**********************************************************************/

// BP_analog_layer defines the CLONES functions required for basic
// analog backprop (no momentum, etc.).
// The activation and error transfer functions are still left open here.
class BP_analog_layer : public Analog_layer {
  Fvec *error;               // error vector
  Fvec *bias;                // bias vector
  float bias_learning_rate;  // learning rate for bias vector

  // Transfer function run on unit activations (usually forw_post_connect)
  virtual void transfer(Fvec *out);

  // forw_pre_propagate is run before any connections into this layer.
  // In this case it copies the bias vector into the activation vector.
  void forw_pre_propagate(Net *) { output->copy(bias); }

  // forw_post_propagate is run after all forw_propagate()
  // to this layer have run.
  // In this case it applies the transfer function to the activation vector.
  void forw_post_propagate(Net *) { transfer(output); }
```

```
};

// BP_sigmoid is a BP_analog_layer with the transfer functions set
// for a sigmoid function.
class BP_sigmoid : public BP_analog_layer {
  void transfer(Fvec *out) { out->sigmoid(out); }
};

// BP_full_aa defines the CLONES functions for a backprop connection
// between two analog layers.
class BP_full_aa : public BP_full {

  // forw_propagate is called to adjust the output layer activations based
  // on the input layer activations.
  void forw_propagate(Net *) {
    // output_layer_activation += weight_matrix * input_layer_activation
    output_layer()->output->muladd(weight, input_layer()->output);
  }
};
```

References

[1] K. Asanović, J. Beck, B. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek. SPERT: A VLIW/SIMD microprocessor for artificial neural network computations. Technical Report TR-91-072, International Computer Science Institute, 1991.

[2] J. Beck. The ring array processor (RAP): Hardware. Technical Report TR-90-048, International Computer Science Institute, 1990.

[3] H. Bourlard and N. Morgan. Connectionist approaches to the use of Markov models for continuous speech recognition. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 3. Morgan Kaufmann, San Mateo CA, 1991.

[4] D. S. Broomhead and D. Lowe. Multi-variable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[5] N. Morgan, J. Beck, P. Kohn, and J. Bilmes. Neurocomputing on the RAP. In K. W. Przytula and V. K. Prasanna, editors, *Digital Parallel Implementations of Neural Networks*. Prentice-Hall, Englewood Cliffs NJ, 1992.

[6] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. The RAP: a ring array processor for layered network calculations. In *Proceedings IEEE International Conference on Application Specific Array Processors*, pages 296–308, Princeton NJ, 1990.

[7] H. Schmidt and B. Gomes. ICSIM: An object-oriented connectionist simulator. Technical Report TR-91-048, International Computer Science Institute, 1991.

[8] D. van Camp, T. Plate, and Geoffrey Hinton. Xerion. Technical Report TR-??, University of Toronto, 1991.

[9] A. Wieland, R. Leighton, and W. Morgart. Aspirin for migraines. In *Proceedings of the 1988 International Neural Network Society Conference*, 1988.