The MikeNet Neural Network Simulator

Overview

MikeNet is a simulation environment for building connectionist (PDP) models. It implements backprop, backprop through time, and continuous time backprop through time.

MikeNet is implemented as a 'c' library. You build a simulation by writing a short program in 'c', including the header files and linking to the library. The library itself consists of routines to create and run the network.

MikeNet has been tested on a variety of unix platforms: linux (i386 and alpha), Cray T3E, Convex Exemplar, IBM AIX, Hewlett Packard HPUX, SGI's unix, Solaris, BSD-386, NetBSD, early versions of SunOS, Dec's Ultrix. The code is vanilla 'c' code and ought to port to any unix platform. It uses no scripting language nor graphical interface; just 'c' code.

General Design

MikeNet has five main types of objects. They are: a network (containing a set of groups and connection blocks and parameters), a group (a set of units with the same layout; a "layer" in normal PDP parlance), a connection block (array of weights connecting two groups), an example, and an example set. They are detailed below.

The general format of a MikeNet simulation has these parts in a 'c' file:

- Define the network objects (net and groups)
- Add groups tho the net object
- Connect groups, and add connection objects to the net
- Randomize connections
- Train:
    - Pick an example
    - Forward propagate
    - Backward Propagate error
    - Adjust weights
    - Repeat until done
- Then, optionally save weights and/or print out status.

A very simple simulation (for xor) in mikenet looks like this: simple.c

Nice Features (Propaganda)

The design of MikeNet allows for some things that are harder to do in other simulators. One thing you can do is instantiate multiple network objects, and have groups and connections shared between networks. Training (forward or backward propagation) works over network objects, so only the groups in the network object are affected. This allows you to train up subnetworks easily, as parts of larger networks.

MikeNet is designed for large networks, and is most fast for such large projects. It is very memory-efficient and can hold very large example files without taking up too much memory.

MikeNet is written as a 'c' library, and as such, all output is totally customizable.

There are many 'hooks' into the internals of MikeNet, for example allowing one to change a training example during processing at the example or tick granularity. The overhead of such things is generally negligible.

MikeNet does not have notions of "input" and "output" groups per se. The mathematics of backprop through time allow a given group (or unit, for that matter) to have clamps at any time sample, or targets at any time sample. Often, one wants to build networks that perform mappings in two directions, such as spelling to sound, and sound to print. The idea of an "input" and "output" group is overly constraining. In MikeNet, any group can have clamps or targets, or both, or neither.

Building the Network

A Net Object

A net object is basically a structure that holds pointers to arrays of group objects and connection objects. As stated above, network operations such as forward and backward propagation take place over net objects. To instantiate a net object, first define the variable for it somewhere:

```
Net * mynet;
```

Then, initialize it. The parameter is the maximum number of time slices the network will be run for.

```
mynet = create_net(5);
```

This creates a network that could run for up to 5 time slices.

The Group Object

Having created a network object, we create groups and add them to the network.

First, define our group variables (this has to go in the variable declaration part of your 'c' program, of course).

```
Group *input,*hidden,*output,*bias;
```

This declares four groups: an input, hidden, output and bias group (a bias group is special: it has one unit, that always outputs a constant value).

We initialize the normal (non-bias) groups by calling the init_group function.

```
input = init_group("MyInput",2,5);
```

The first parameter is a text name for the group. The example file will refer to this text name. The second parameter (2, in this case) is the number of units in the group). The third one is the maximum number of ticks the group might be run for (5 in this case).

For biases, we initialize them as such:

```
bias = init_bias(1.0,5);
```

The first argument is the default output of the bias unit (1.0 here). The 2nd one is the number of samples that group may be run for.

Having created our groups, we now add them to the network object we created above.

```
bind_group_to_net(mynet,input);
bind_group_to_net(mynet,hidden);
bind_group_to_net(mynet,output);
bind_group_to_net(mynet,bias);
```

It doesn't matter what order you do this.

The next step is to connect our groups, using the connection object.

The Connections Object

You begin by declaring the connection objects. You need one for every set of connections between layers.

```
Connections *c1,*c2,*c3,*c4;
```

Then, you connect groups:

```
c1 = connect_groups(input,hidden);
c2 = connect_groups(hidden,output);
c3 = connect_groups(bias,hidden);
c4 = connect_groups(bias,output);
```

The syntax of the connect_groups command is simply the 'from' group object and the 'to' group object.

Next, we add our connections to our network object.

```
bind_connection_to_net(mynet,c1);
bind_connection_to_net(mynet,c2);
bind_connection_to_net(mynet,c3);
bind_connection_to_net(mynet,c4);
```

Lastly, we randomize our connections to initial weight values.

```
randomize_connections(c1,0.5);
```

... and so on. The first argument is the connection object, the 2nd is the weight range. Here, weights are randomized between +/- 0.5.

Example Files

General Format

Basically, the format for an example file is a set of individual examples. Each example consists of a TAG line, which is like a comment for that example. This is optionally followed by a PROB line which gives the probability for which that example will be presented. Then, there are a set of CLAMP and TARGET lines, defining a group to get a clamp or target, what time ticks it gets that clamp or target for, followed by the word

SPARSE or FULL, and a set of values, then a comma. SPARSE examples will be discussed in subsequent revisions to this document. For now, we'll only consider FULL. With FULL examples, a value is given for each unit in the given group.

Here is a typical example entry:

```
TAG Word: dog
PROB 0.2
CLAMP MyInput 0-2 FULL
1 0 1 1 0 0 0 ,
TARGET 1-2 FULL
0 0 0 1 1 0 1 ,
;
```

The time specification (0-2 in the CLAMP line above) can be of the form - to give a range, or just . The word ALL can be substituted, which means "for all time ticks".

You load in an example set by first defining the ExampleSet object:

```
ExampleSet *examples;
```

Remember, this definition goes with other variables in the 'c' file.

Then, you can load them in like this:

```
examples = load_examples("xor.ex",5);
```

The first argument to load_examples is the file name, the second is the maximum number of time ticks that example file may be run for.

Individual Examples

You train a network by selecting an individual example from an example set for training, one at a time. The Example object is first defined:

```
Example *ex;
```

Then, you can pick an example. To work through examples in sequence, do something like this:

```
for(i=0;i < examples->numExamples;i++)
{
ex = &examples->examples[i];
...
```

This picks the ith example from the example set.

You can also sample examples probabilistically, using:

```
ex = get_random_example(examples);
```

The examples will be sampled according to their probabilities. Always make probabilities in the example file between 0.0 and 1.0.

Training the Network

The basic way you train a network is as follows. You make a loop for however many iterations you want. Then, you pick an example, call bptt_forward to forward propagate activity, then bptt_compute_gradients to accumulate the gradient terms, then bptt_apply_deltas to change the weights. You can use online or batch learning depending on how you pick examples and apply deltas, as discussed below.

Online Learning

This is pretty simple. You loop for as many examples as you want to process. On each iteration, pick an example (usually at random). Then, do bptt_forward, then bptt_compute_gradients, then bptt_apply_deltas. Repeat as often as necessary. Like this:

```
for(i=0;i < 10000;i++)
{
ex = get_random_example(examples);
bptt_forward(mynet,ex);
bptt_compute_gradients(mynet,ex);
bptt_apply_deltas(mynet);
}
```

The bptt_forward function takes two arguments: the network, and the example. Same with bptt_compute_gradients. The bptt_apply_deltas function just takes the net object. It also zeros out the gradients once it changes the weights.

Batch Learning

This is similar to online, except that you iterate over (usually) all of the examples before applying the deltas. Like this:

```
for(i=0;i < 10000;i++)
{
for(j=0;j < examples->numExamples;j++)
{
ex=&examples->examples[j];
bptt_forward(mynet,ex);
bptt_compute_gradients(mynet,ex);
} /* end of loop over examples */
bptt_apply_deltas(mynet);
} /* end of 'i' loop */
```

Now, you may want to keep track of the sum squared error as you train, and print out informative messages so you know if the thing is training or not. You can call the compute_error function to get the error for the network's current state on the current example. Call it after calling bptt_forward, like this:

```
error += compute_error(mynet,ex);
```

It takes two arguments: the network and the example.

Other Useful Commands

You'll probably want to save weights from runs of the simulation. Call the save_weights function to do that:

```
save_weights(mynet,"foo.weights");
```

The first argument (predicatbly) is the network object, the second is a file name. Note that MikeNet will attempt to compress weight files when it saves them, so the above code will probably result in foo.weights.gz sitting in your directory.

To load weights (to, say, evaluate a network or test it), you just call load_weights with the same arguments. Remember that you gotta do this *after* building the network (and don't call randomize_connections or you'll re-randomize all the weights you loaded in.

Building The Simulator Package

First, get the gzipp'd tar file of MikeNet version 8.0: mikenet_v8.tar.gz

You might want to use the latest (development, not totally tested) one: latest build.. untested

Then, stick it in your home directory. Untar it:

gunzip mikenet_v8.tar tar xvf mikenet_v8.tar

This will create a directory called "Mikenet-v8.0" with a bunch of stuff in it. Next, you have to tell the simulator where it is located. Put this in your .cshrc startup file:

setenv MIKENET_DIR ${HOME}/Mikenet-v8.0

The MIKENET_DIR environment variable is used by makefiles to find stuff. Before you build the libraries, execute that statement and make sure it's set. Then, build the libraries.

```
cd Mikenet-v8.0/src
make clean
make cc
```

The above line makes a generic, "cc" version of the simulator. Instead of "make cc" you can have "make linux", "make alpha", "make gcc" (recommended if you have access to the gnu c compiler). Check the Makefile in ~/mikenet/src to see what different options there are.

This should build the thing, copy header files over to ~/mikenet/include and libraries over to ~/mikenet/lib/${ARCH}. The ARCH environment variable is optional, but allows you have different versions of MikeNet for different architectures (you may have to do mkdir ~/mikenet/lib/${ARCH} before building the thing so the directory is there).

Ok. Now you're ready to try a demo. Change directories over to ~/mikenet/demos/xor and do rm xor; make xor. It should build the "xor" demo. Then run it by simply typing ./xor and you should see it run. It will kick out the iteration number and sum squared error; the error should go down, and when its below 0.01 it should stop and kick out the actual unit outputs for each example.

Go to ~/mikenet/demos/tutorial and look at the code in the tutorial.c file. It has lots of comments and useful things. The "xor.c" file (and "xor_online.c", an online learning version) are simpler ones.

The directory "cxor" has a continuous-time version of the xor problem. And the "benchmark" directory has a time benchmark so you can see how quick different machines are.

To build your own simulation, start with one of the existing demos (like xor), and copy all of the files into your own directory. Change xor.c and xor.ex as you see fit, and then do a "make" to build your simulation. You don't need to recompile the libraries, just your own simulation .c file(s) when you change things.

Parallel Processing Extensions

MikeNet has some code that can optionally be compiled in that will call routines in the public domain MPI parallel library. MPI is basically a set of routines that let you build a single executable and spawn copies of it on multiple machines over RSH or SSH connections. They can talk to each other, and do basic parallel operations like "take this vector from each machine and return the sum of all elements from all machines to a vector on the head machine".

Download mpitest.tar.gz (80mb!) for some demo programs. If you want to run this demo, you should move or delete the existing weight files or it'll choke as it tries to overwrite them. This demo learns phon -> sem mappings for about 6600 words, and takes about 3 days to do so on a cluster of 17 machines. It uses batch learning.. that works best with parallel code. See also the "mpi" demo in the "demos" directory of the MikeNet distribution. Note: all of these demos were built using the MPICH mpi library. You need to compile MikeNet with the USE_MPI directive compiled in, and the include dir for mpi stuff. See the entry for "linux_mpi" in the Makefile in the MikeNet src directory.

BLAS Extensions

BLAS is a library of routines (originally in Fortran, though there are C interfaces) that do matrix algebra. Simulations use a lot of vector/matrix operations, so any fast, specialized libraries to do such things are a win. There is a tool called ATLAS which will create a BLAS library specially tuned to your machine. It does clever things like

carefully managing the cache, and using special instructions on your machine (like SSE on certain intel chips, or 3dNow on certain Athalons).

Get the ATLAS library from http://math-atlas.sourceforge.net/. Then, for your target machine, see if it has the SSE or 3dNow extensions (on linux, do cat /proc/cpuinfo and look at the "flags" line for sse or 3dnow.) Then, build the atlas stuff according to its instructions.. build it on the machine you plan to run stuff on.

Then, go into the MikeNet src directory and edit the Makefile so that BLAS_INC points to your atlas include dir. Compile MikeNet with blas (do "make clean; make linux_blas"). Then try the "blas_test" stuff in the "demos" directory (there's a README in there). See also "mpi_blas" in the mpitest.tar.gz package mentioned above for mixing mpi and blas.

Source: http://www.cnbc.cmu.edu/~mharm/research/tools/mikenet/