Web Site of Robert John Morton

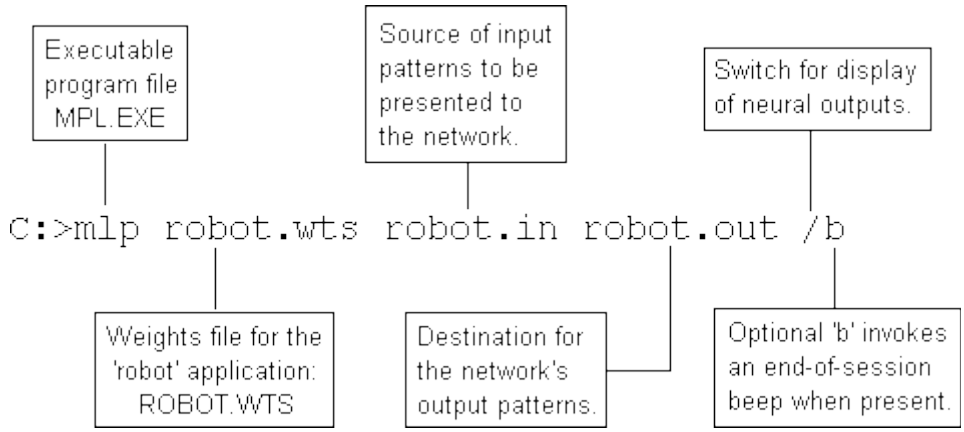
Neural Networks: A Complete MLP Program

Commentary on the multi-layer perceptron program MLP.EXE whose source code is listed in the file LP.C. Topics: command line, input arguments, argument validation, dynamic allocation, the .WTS file, the perceptron functions, input/output, training manager, main().

Command Line

To create a practical multi-layer perceptron we must implement it as an executable file with a name such as mlp. When it is run it must be told where to find details of its size (number of neurons in each layer) and the values of its inter-neural weights. It also needs to know where to find its input and where to send its output.

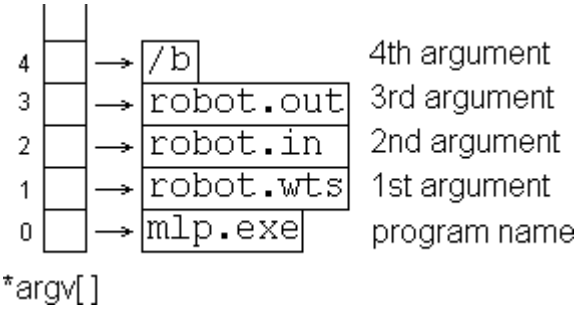
We should therefore design the executable file to take a set of command line arguments as follows:



The '/' before the 'b' means that neural outputs will not be displayed. If this is replaced by a digit 0 to 9, the output of each neuron in each layer will be displayed and updated for each new input pattern. The actual value of the digit 0 to 9 determines the number of seconds that are to elapse between processing and outputs display for successive input patterns.

Input Arguments

When the program is started, the program name and its arguments are automatically put into a set of character arrays pointed to by the elements of a pointer array traditionally called argv[]. The name 'argv' stands for 'argument vectors'. The start address of argv[] is passed to the main() function



together with another argument traditionally called argc, which contains the number of items on the command line.

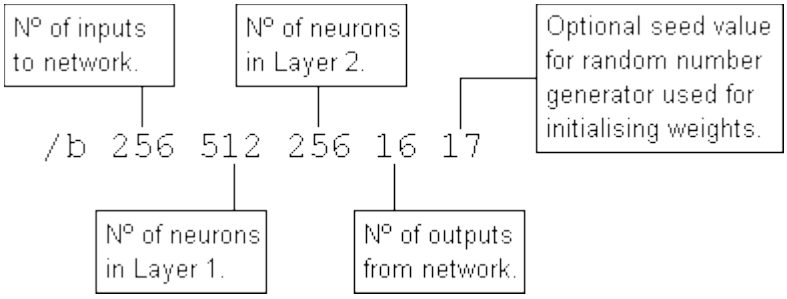
```
main(int argc, char *argv[]){

}
```

So for example to open the network's input source (robot.in in this case) we use a statement such as:

```
FILE *inh = fopen(*(argv + 1), "rb");  //open application file
```

Additional arguments are required for training sessions as follows:



These specify the size and layout of a new network and allow the weights to be re-initialised differently when the network fails to converge.

Argument Validation

Three of the command line arguments are filespecs. The OS checks any filespecs before attempting to open a file. However, there are other checks specific to this program which we must do beforehand. These special checks are performed by the CheckFileName() function listed in mlp.c. This function first checks the overall length of the presented file name. It then checks the presented filename's extension against the expected one, or adds the 'expected' extension if omitted on the command line.

This function is called by CheckFileArgs() which ensures that the name of the weights file entered on the command line either has a .wts extension or no extension (in which case the .wts is added). No specific extension is imposed for the names of the input and output files.

Assuming all file names are valid, CheckFileArgs() opens the files in the appropriate modes. For normal operation, the weights and input files are opened for read-only and the output file for write-only. For training, the input and output files are opened for read-only (the output file contains the target outputs during training), and the weights file is opened for write-only so that the weights can be stored when training has been completed.

Dynamic Allocation

In mlp.htm the weights and layer arrays were declared in the program source file. This means that if we want to change the size of the network, we must edit and recompile the program.

It is far better however to have a single executable file which can be used for any mlp application irrespective of the size and shape of the network. To do this we must allocate the weight and output arrays dynamically at run-time instead of statically at compile time. For this we must include the memory allocation function malloc():

```
#include <malloc.h>   /* contains the prototype for
                         the malloc() function */
```

Since the number of layers is fixed by Kolmogorov's Theorem we keep the original definitions of NL and NAL:

```
#define NL  4   //number of network layers
#define NAL 3   //number of active network layers
```

Also, we must keep our declarations of those arrays whose size depends only on the number of layers:

```
short N[NL],    //Neurons per layer
     *L[NL],    //Access to the layer outputs
    **W[NL],    //Access to all weights
```

As discussed in train.htm we need the following additional pointer arrays for the program to work in training mode:

```
     *E[NL],    //Access to the layer error arrays
    **M[NL];    //Access to all delta-weights
```

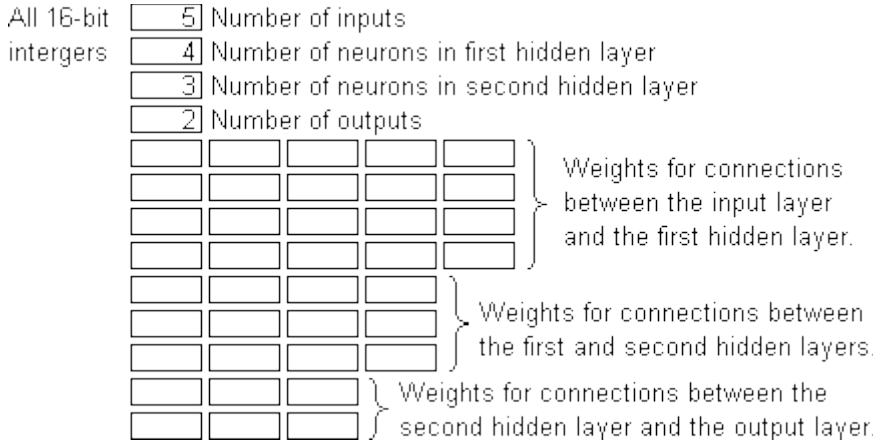
Finally, we need to replace the array declarations for the weights, inputs, outputs, errors and delta-weights by a function called Walloc() listed in mlp.c which allocates corresponding heaps of integers and pointers at run-time.

Note that Walloc() only allocates memory for error and delta-weight arrays when the extra training arguments have been given on the command line.

In normal mode, ie when not in training mode, Walloc() also loads in the weights from the .wts file as it allocates memory in which to store them.

The .wts File

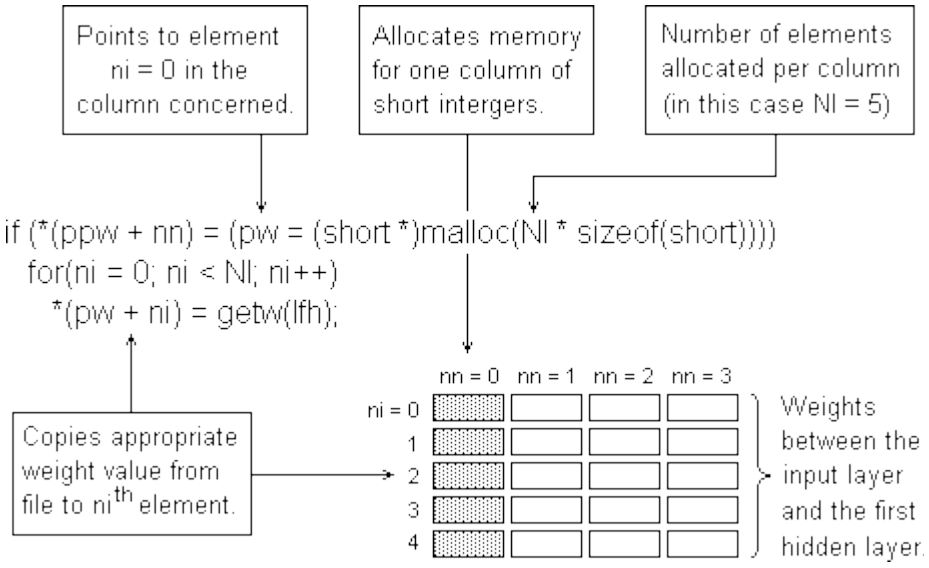
Before it can load the weights, Walloc() needs to know how many neurons there are in each layer of the network. In other words, it needs the infill for the array N[ ]. This is held with the weights in the .wts file as follows:



The .wts file is normally created by the training algorithm. Its contents do not change once training has been completed. In normal mode, the mlp program simply reads the .wts file as part of its initialisation process.

To exercise the mlp program initially, however, it is necessary to generate a test file robot.wts using the generator mlptest1.c (simply run mlptest1.exe). This will also generate a sample input file robots.in. An alternative set of weights and input values can be generated using mlptest2.c (simply run mlptest2.exe).

Walloc() essentially allocates memory for a heap of short integers and then loads into them the input weights for a given neuron nn from the weights file as follows:



However, in the final version of Walloc() this for() loop also provides for loading random starting values for the weights when operating in training mode. It obtains random numbers by calling the random series initialiser srand() and the random number generator rand().

The Perceptron Functions

mlp() was finally implemented in mlp.c as a void function because it was decided that it was more convenient for it to pick up the pointer to its input directly from the first element of the global pointer array L[ ].

mlp() was also made to call a display function ShowOut() when the display switch is set on the command line. ShowOut() displays a table showing the outputs from each layer of neurons in the network in response to each input pattern presented to mlp().

mlptrain() has also been implemented without its input and target output pointers passed as arguments. The input pointer is again obtained directly from the global pointer array L[ ]. The pointer to each corresponding target output pattern required for training has been implemented as a global pointer 'pt'. The weight gain term and the momentum term shift factors are still passed to mlptrain() as input arguments.

Input/Output

Input and output has been made to work only from and to disk files. The next complete input pattern for the network is obtained each time by a call to GetInp(). In normal operation of the network the resulting output pattern is sent to the output file by a call to PutOut(). When the network is being trained however, instead of outputting the output pattern, a target (correct) output pattern is read in for comparison with the obtained output by calling the GetOut() function.

GetInp() and GetOut() also call ShowOut() when the output display switch is set on the command line. This is necessary because mlp() only displays the outputs from each active layer and so does not show the input values.

To operate the network in real-time with input and output coming from and going to a communications port, the input/output section of mlp.c must be re-written in an event-driven form such as a message processor under ROBOS, which would itself run under the host OS.

Training Manager

Control of the network's training mode is done by TrainingManager(). The essence of TrainingManager() is an infinite for(;;) loop which can only be broken by either the network's error function for each sample input pattern becoming acceptably low, or by the user pressing a key.

The infinite loop cycles through the training data then halves the gain term h, rewinds back to the beginning of the training data and cycles through it again. For each sample pattern in the training data it calls GetInp() to read in the pattern and passes it through mlp() to get the output pattern. It then calls GetOut() to read in the corresponding correct output pattern from the training data. It then calls GetErrFun() which subtracts from it vectorially the observed output to get the Error Function. If this is too large, mlptrain() is called to adjust the network's weights thereby to reduce it and sets the flag to indicate that the network has not yet converged.

If at the end of a given pass of the training data the error function has not exceeded the acceptable limit for every pattern, then we break out of the infinite for(;;) loop, store the final weight values for the converged network and exit TRUE. If on the other hand the Error Function is still too high for at least one of the training data input patterns, the gain factor is halved, the training data files are 'rewound' and the for(;;) loop is repeated.

ShowTrainPass() is called in the outer for(;;) loop to display the number of passes of the whole training data that have been made so far. In the inner loop, ShowPatCnt() and ShowErrFun() are called respectively to show the number of the input pattern currently being processed and the magnitude of the Error Function for that pattern.

main()

Overall co-ordination of the program is of course done by main(). On start-up, main() calls SigLoad() to load in the sigmoid function's look-up table used by mlp(). The sigmoid look-up table is generated independently by a separate program called siggen.c which produces the look-up table in a file called sigmoid.dat. Main() then calls CheckNonFileArgs() and CheckFileArgs() to read in and validate the command line arguments as discussed in detail earlier. It then calls Walloc() to allocate memory and read in the weights again as discussed earlier. If any of these functions fails it displays an error message giving the reason and main() terminates after displaying a 'cannot continue' message.

Assuming none of the above function calls fails, main() uses the 'Training' flag set by CheckNonFileArgs() to determine whether the program is to run in ordinary mode or in training mode. If it is training mode, main() calls TrainingManager(), otherwise it calls ShowLayers() to display the number of neurons in each layer of the network and then sets up a for() loop which executes for the number of patterns in the input file. For each input pattern it first checks for an aborting keystroke. If not aborted it then displays the number of the current pattern, reads it in, processes it by calling mlp() and then sends the resulting output pattern to the output file.
