Spatial Disparity Program

Stephen Eglen

March 16, 2021

Abstract

These are just a few notes on how to run and extend the spatial disparity model.

1 Conventions

JVS All file names are in **bold**. All C procedure and variable names are in *italics*. All C arrays are indexed from 0 to $n-1$, where n is the number of array elements.

2 Running the Program

To run the program, you need the program `testnet` in your path. To start the program type:

```
testnet disp.prm
```

where disp.prm is the parameter file that controls the behaviour of the program. JVS This file specifies a network with a 5×1 input array, and 1000 inputs. As the program runs, it will print up the progress of Conjugate gradient descent (CG).

Two files are actually needed to run the program which are now described.

2.1 Parameter Files

The parameter file stores all the relevant parameters of the program that can be changed. Each parameter corresponds to a global variable within the program, and so when modifying the program, the parameters should be accessible everywhere. Each parameter has a default value, which can be changed by the parameter file. (Otherwise the parameter keeps its default value.)

To change a value of a parameter, simply write the parameter name and its new value. The amount of whitespace should not matter, and neither should the order in which the parameters are listed. Placing a # indicates that the rest of a line is a comment. C style comments can also be used in the file.

A list of all the parameters can be found in the file dispvars.h, along with documentation. The default values are set in the procedure *setParamDefaults()* in the file **testnet.c**. An example parameter file is given in Appendix A.

Any text (eg parameter names misspellt) not recognised by the parser is simply copied to stdout. If you have misspellt a parameter name, it is worth checking the output as it comes up on the screen to see if the parameter has the value you expect. (look for the text `*** System Parameters ***` in the output)

2.2 Network Files

A second file is needed to tell the program how the network is arranged. This is called the network file. An example network file is given in Appendix B

Comments are allowed in this file by using a # only at the start of the line. The order of entries in the network file is important, and should be as follows:

2.2.1 Number of layers

`nLayers` n

where n is the number of layers in the network. For a network with one input layer, one hidden layer and one output layer, $n = 3$. The number of layers can be any value greater than 1. (A value of two will just create one input layer and one output layer.)

2.2.2 Number of cells in each layer

For each layer $0, 1 \ldots n-1$, you need to specify how many cells there are:

`layer i x y`

means that layer i has $x \times y$ cells, arranged in a grid of x rows by y columns. Note that this number of cells does not include the bias cells.

2.2.3 Layer information

For each layer, you need to specify what kind of cells are in the layer, and whether they provide a bias for the next layer. This needs to be done for layers $0..n-1$ in order.

`layer i actfn bias?`

Activation function: actfn = LINEAR or TANH to indicate the kind of activation function each unit in this layer has. LINEAR is a bad choice of name, and maybe it should be renamed to IDENTITY at some point in time?

Bias: bias? = bias or nobias to indicate whether a bias cell should be put into this layer. This bias cell will project to all cells in the next layer if it is present. When telling the program the number of cells in a layer you do not need to include the bias cell. For example, if you have 5x2 cells in layer 0 and there is also a bias cell in layer 0, then just say `layer 0 5 2` as normal. The activation and output of the bias cell is always set to 1.0.

Note that this program assumes that the output layer only has one unit, and that this unit's output is copied into an array of virtual output units, to produce the effect of shared weights across an array of output units.

2.2.4 Connectivity

This part of the network file says how cells in the source layer are connected to cells in the destination layer. Cells in layer i can only receive input from cells in layer $i-1$.

Connectivity needs to be specified for each layer $1..n-1$, in order. There are two types of connectivity: full and local.

Full connectivity: To specify full connectivity to layer i from layer $i-1$, you simply put:

```
connections to layer i
full
```

Local connectivity:

To specify local connectivity, you need to say for each cell in the destination layer, the rectangle of units in the source layer that it receives input from.

For example, if layer 2 has two cells, and layer 1 has fifteen cells arranged in 5 columns and 3 rows, then the following:

```
connections to layer 2
0 0 3 3
2 3 5 3
```

says that the first cell in layer 2 receives input from layer 1 cells in the region [0,0 to 3,3] and the second cell in layer 2 receives input from layer 1 cells in the region [2,3 to 5,3].

2.3 Training the network

Once you have a parameter and network file, the program can then be run. (The parameter *checker* allows you to switch between CG and the checker module.)

2.4 Files produced during training

As the program proceeds, several files are created.

2.4.1 Correlation

After every line search, the correlation is tested, and written to the file *corrn.dat*, which can be plotted. The correlation is also written on the screen.

2.4.2 Merit function

The merit function is written every line search to the file *dispoutputs*, so that the merit function can be checked.

2.4.3 z, zbar and ztilde

z, zbar and ztilde are periodically written at a line search *i* (normally every 10 line searches) to the files *z.i, zbar.i, ztilde.i*.

Depending on the topology of the output, these files can either be viewed as a graph (1D) or as an image (2D). In both cases however, these files are raw text.

To view the file, eg *z.50*, as a graph, I normally do the following in gnuplot

```
gnuplot> plot "<sw z.50"
```

where *sw* is an auxiliary program.

If the file is to be viewed as an image, then I convert it to an image with the following

```
snpgm z.50
```

where *snpgm* is another program. This will show the weights as a PGM file within xv.

2.4.4 Weights files

The weights are also periodically written out to the file *wts.i*, where *i* is the line search number. This is mainly so that they can then be used for later testing of the network. The format of these files is one weight per line in ASCII format.

Interpreting the weights files

A simple script, `ew`, has been written to extract weights from a weight file and put them in a suitable raw format such that they can be converted to an image by `snpgm`. For example, to extract weights 0 to 8 (the first 9 lines of the weights file wts.1000, since weights start from 0), and convert them into a 3 by 3 image, we do:

```
ew 0 8 3 wts.1000 hidden0
```

This will create the raw file hidden0, containing 3x3 weights, and an image `hidden0.pgm` which can be viewed.

The general usage is: ew *first last wid wts opfile*, where

first is the number of the first weight to extract. *last* is the number of the last weight to extract. *wid* is the width of the image to create. (The image is filled up on a row by row basis.) *wts* is the name of the weights file to extract the weights from. *opfile* is the name of the file to write the weights into. Another file opfile.pgm will be created with the image in it.

The image is created by filling in pixels on a row by row basis. If there are not enough weights to complete the last row, padding pixels (set to the maximum weight found in the image) are included so that the image can be made.

See Appendix E for an example of how to extract weights.

2.4.5 Mask files

To check that the convolution masks $\bar{\Phi}, \tilde{\Phi}, \bar{\Phi}', \tilde{\Phi}'$ have been created. These masks can be viewed with either gnuplot or xv, in the same way as the other output files.

The masks are assumed here to be exponential, with a cut of at 4 half lives. The size of the masks for the short and long range means can be given in terms of λ or in terms of the half life t_h. The form of the exponential used here is:

$$A = A_0 e^{-\lambda t}$$

When at the half life, t_h, $A_h = \frac{1}{2}A_0$:

$$\begin{aligned} \frac{1}{2}A_0 &= A_0 e^{-\lambda t_h} \\ \ln \frac{1}{2} &= -\lambda t_h \\ t_h &= \frac{\ln 2}{\lambda} \end{aligned}$$

2.5 Testing the network

At the end of training, you are left with a set of weights and other output files. These weights can then be fed back into the program to test the program on another set of images that weren't seen during training.

To do this, you need to create another parameter file which tells it which images you want. (The network file will stay the same.) An example parameter file is given in Appendix C.

To switch off learning, set $doLearning = 0$, and set *results* to the name of the file where you wish the outputs of the network to be written. Set $initWts$ to the name of the weights file you wish to load into the network to test on the new images.

The program is then run, with this new parameter file. The outputs of the network will be written to the results file, and the correlation between the outputs and the shifts array will be printed on the screen.

(Relevant function: void *checkNetPerformance()* in **testnet.c**)

2.6 Sampling the input images

$totalInputWid$ and $totalInputHt$ describe how big the two input images and the shift file are. Input vectors for the network are created by cutting the input images (and the shift image) into boxes of size $(inputWid + inputSkipX) \times (inputHt \times inputSkipY)$ For each box, an input vector of size $inputWid \times inputHt$ is extracted from the top left hand corner of this box. Similarly, the appropriate shift value is taken from the centre of each box over the shifts image.

As a diagnostic, the input vectors created are written to the file *inputs.test*, one per line of the file. The corresponding disparity values from the shifts image are stored in `shifts.test`.

2.6.1 Input Image Format

The input image format is slightly different to the images output by Jim's program that generates the images. The basic difference is that the first two lines of the image are deleted (this is the image size and a blank line), and a trailing newline is added. (This was done so that the image format is fairly raw: if the image is x columns and y rows, then the file is y lines long, with x floating point numbers on each line. Once in the raw format, I can then view the file as an image with `snpgm`.)

The function that reads in images is void readInputFile(char *inputFile, Array arr).

To save you editing the files by hand, a simple script `conjimim` will convert the image into the proper format. To use this, simply do `conjimim image1.txt > im1.txt`, and im1.txt will be in the proper format.

All of the data files have been kept in the directory `/rsunq/vision/stephene/data/JIMS/`.

2.6.2 Input Layer Size

If the input images are being chopped up into vectors of size $inputWid \times inputHt$, then because we have two input vectors (one from each image) entering the network, the network file should have the line `layer 0 x y`, where $x = inputWid$ and $y = 2 \times inputHt$.

2.7 Size of virtual output layer

The topology of the output layer is defined by the parameters $outputWid$ and $outputHt$. If $outputHt == 1$, it is assumed that the output units are arranged in a one dimensional chain, otherwise the units will be in a 2d grid of size $outputWid \times outputHt$.

The number of units in the output layer must not be less than the number of input vectors (*numInputVectors*), and ideally should be the same (i.e $outputWid \times outputHt == numInputVectors$). If there are more output units than input vectors, the remaining output units will have an output of 0.

Also, the following should hold:

$$outputWid = \frac{totalInputWid}{inputWid + inputSkipX}$$

$$outputHt = \frac{totalInputHt}{inputHt + inputSkipY}$$

2.8 Run time

For most of the 1d examples, the program will take just a couple of minutes to run on a workstation. The 2d examples will take much longer however. In the largest experiment set up, (gauss600c.prm) using a 100x100 output array, it took around a day to do 500 epochs on a fast workstation! However, the smaller 2d examples (eg gauss600.prm - a 20 by 20 output layer) also run quite quickly.

3 Updating the program

All of the source files are stored in the directory ~stephene/disparity/.

3.1 General Program Description

3.1.1 Initialisation

First of all, the parameter file is read in to set the parameters of the network. Then the network file is read in, and all of the activation data structures are created, along with the weights. The input images are chopped into input vectors.

3.1.2 Conjugate Gradient

Conjugate gradient does most of the hard work, calling the evaluation function and then creating the derivative vectors. When the evaluation function is worked out, it will also create the derivative vectors, and therefore some work is done but not taken advantage of here.

For each call to determine the evaluation function, all of the input vectors are presented to the network, one at a time. The output of the network is then stored in the virtual array z of output units. (The output of the network to the ith input vector is stored in z(i).) After all of the inputs have been presented, z is then convolved with the two masks to create $\bar{z}$ and $\tilde{z}$. From there, the errors for the top layer of virtual cells z are created, and then propagated backwards to the hidden layer cells. These errors are then passed back to conjugate gradient.

Before using CG, it is worthwhile to use the gradient checker, by setting the *checker* parameter to 1, and watching to see if the ratio calculated comes to 1.000.

3.2 Relevant functions

This section gives the names of the major functions that will be useful to follow when extending the program.

netmain() is the main function of the program and calls the initialisation routines, and then either tests the network, or trains it using conjugate gradient.

3.2.1 Initialisation

Initialisation is done by the function *setUpNetwork()*. This reads in the parameter file (*getParams()*) and then creates the network data structures (*readnet()*). Weights are initialised to random values (*initWtsRnd()*). The input vectors and actual disparity values are read by *createInputVectorsAndShifts()*.

readnet() is quite a large function which does several things. Firstly, it allocates space for the weights (*createWeights()*). It allocates space for the activations array actnInfo (which stores current network activity) and the structure *allActns* (which stores all of the activations for one iteration - i.e for each input vector).

The main task for this function is to then read in the network file, and decide how many weights will be needed to set the network up, and so forth. This is done on a layer by layer basis. For each cell in each layer, we find out from the network file which cells it is taking input for, allocating weights contiguously. (So if a cell in layer 1 receives inputs from 5 layer 0 cells, then the 5 weights are contiguous in the weight vector.) This is done using *connectCells()*. At the same time, the preCellInfo and cellInfo structures are updated. (The preCellInfo structure says which cells in layer $i+1$ are connected to by a cell in layer i. Conversely, cellInfo says which cells in layer i provide input to a cell in layer $i+1$.)

The global arrays for $z, \bar{z}, \tilde{z}$ and so on are created using *createZs()*. Similarly, the error vector arrays dw and onedw are created by *createdw()*.

The final job of initialisation is to create either 1D or 2D masks for calculating U and V. (Using either *createMasks()* or *createMasks2()*.)

3.2.2 Testing Network Performance

If no learning is to take place, then the routine *checkNetPerformace()* is used to see how well the net performs on data that it hasn't seen before.

To do this, it reads in results from the initWts parameter using *readWts()*. Each input vector is then inserted into the activations array (getNextInputVector), and the activation of the output cells for that input calculated. All of the activations are then stored into the *allActns* structure by *storeActivations()*.

Once all of the inputs have been presented, the correlation is determined by *Rvec_correlate()* and the z array output to the results file using *writeArray()*. After this, the program exits, as no learning is to take place.

3.2.3 Conjugate Gradient and Checker Routine

If learning is to take place, then the conjugate gradient routine (cg_williams) is called. Importantly, the merit function is called *evalFn()* and the function returning the weight change vector is called *evalPartials()*.

The program normally runs for a fixed number of line searches (set by the maxiterations parameter), but also will print the correlation between output and image disparity. This is done by *finishedFn()*.

Both *evalFn()* and *evalPartials()* are simple wrappers around the function *calcMeritAndPartials()*, which does most of the hard work, evaluating the function and then creating the weight change vector.

Network Activation

The function *clearActivationArray()* sets all of the units activations and outputs to 0.0. *setBiases()* then sets the activation and output of the cells to 1.0. The activation and outputs of the inputs is then set by copying across the relevant input vector.

Once the input cells have an activation and input, the activation of all layers is calculated by *calcAllActivations()*.

For this learning rule, we normally need to do batch learning. So, all of the activations and outputs of each cell needs to be remembered for later use for calculating the error vector and creating the array of virtual outputs. *storeActivations()* does this job of copying the cell activations into the *allActns* structure.

3.3 Global Variables

Several global variables have been used in the program. All of the system parameters are global variables, and are described in the file **dispvars.h**. Not all of the global variables however are parameters (i.e. changeable by the parameter file) – these variablesare documented in the file **dispglobals.h**.

JVS Variables listed in *.prm file can be changed directly by user to alter values in **dispvars.h**. In contrast, variables in **dispglobals.h** are changed by program as a consequence of user altering values in *.prm file.

Finally, four global variables – `weightInfo, netInfo,actInfo, layerInfo` are declared in **dispnet.h**. They are declared here, rather than in **dispglobals.h** since these variables are of complex types - these types also being defined and documented in this file.

3.4 Main Data structures

Most of the data structures discussed here are defined in the file **dispnet.h**.

3.4.1 Array

The array data type stores a 2d array as a 1 dimensional vector of length $wid \times ht$. By convention if the array is 1D, then $ht = 1$.

JVS The *data* slot in *Array* is a pointer to the first element of an array with $ht \times wid$ elements.

3.4.2 layerInfo

For each layer i of the network, layerInfo[i] is a structure containing the details of each layer: the number of cells, their activation type, and whether this layer also provides a bias input to the next layer.

For each cell j in a layer, preCellInfo[j], a component of the layerInfo stores the details about the weights and cells in layer $i+1$ that this cell connects to. Hence, by looking at this structure, it will allow you to work out the error for this cell, given the errors for cells in layer $i+1$.

3.4.3 cellInfo

For cell j (numbered relative to the number of cells in this layer) in layer i, layerInfo[i].cellInfo[j] shows which cells in layer $i-1$ provide input to cell j. This structure is used by the function *calcActivation()* to calculate the activation and output of the cell given some input.

3.4.4 weightInfo

WeightInfo stores the actual weights of the network as a 1d vector in data, starting from 0. preCell[i] tells you the number of the presynaptic cell that this weight is used for. postCell[i] tells you the number of the postsynaptic cell that this weight is used for.

3.4.5 Bias cells

If cells in layer i need bias input as well as input from cells in layer $i-1$, then an extra cell is placed in layer $i-1$.

3.4.6 activationInfo

Each cell in the network is given a unique number, starting from 0. These numbers are allocated contiguously from 0. For example, given the following network file:

```
# Simple 3 layer network
nLayers 3
layer 0 5 2
layer 1 5 2
layer 2 1 1

layer 0 LINEAR bias
layer 1 TANH nobias
layer 2 LINEAR nobias
connections to layer 1
full
connections to layer 2
full
```

These are the activation cell numbers (printed by *printNet()* and written to the file `actn.info` when the program has set up the network):

```
** Activations **

Layer 0
 0  1  2  3  4
 5  6  7  8  9
Bias 10

Layer 1
11 12 13 14 15
16 17 18 19 20
```

```
Layer 2
21
```

The *activationInfo* structure stores the internal activations $actn[i]$ and outputs $op[i]$ of each cell i. In the above example, units 0..9 will store the current input vector and unit 10 is the bias unit that projects to cells in layer 1. (These unit numbers are the unit numbers referred to in preCellInfo and cellInfo). actn[21] will store the activation of the output unit, and op[21] stores the output of the output cell.

In this context, the activation of a cell is the dot product of the weights and the inputs. The output of the cell is the result of passing the activation through the activation function (normally either linear or tanh).

3.4.7 allActns

JVS This is the main structure used to store the inputs, outputs and error terms of every cell for all input vectors. The *allActns* structure contains 5 slots. Each slot is a 2D array of dimensionality $dim[numInputVectors][numUnits]$. The 5 slots are:

num The length of this array. i subscript can range from 0 to $i-1$. Normally, i is equal to *numInputVectors* .

$allActs[i]$ Array of total input to each of *numUnits* units in network.

$allOps[i]$ Output value of each of *numUnits* units in network.

$errors[i]$: Array of error values, one per unit in network.

numUnits: Number of units in network.

If there are x input vectors, then each input vector is loaded into the input units of the activations array (by *getNextInputVector(int vecnum)*) and then the activation propagated through all layers to the output cell. (This is done by the procedure void *calcAllActivations()*.)

Once the activations, errors and outputs have been calculated for a given input vector, all of the activations and outputs of the net need to be stored for later use by the functions to calculate the merit function and error vector.

Hence, for input vector x, $allActs[x]$ is a pointer to a vector of activation values – these activation values are the values of all the cells in the network given the xth input vector. $allOps[x]$ stores the corresponding output values of the cells given that input. Finally, $errors[x]$ stores the vector of error measures given the xth input vector.

JVS Once all input vectors have been processed by the net, the output of each input vector is stored by $getZ()$ in $z.data$.

3.4.8 Calculating error vectors

The main function to present all the inputs and calculate the outputs and error vectors is *calcMeritAndPartials()* in **testnet.c**.

Once all of the input vectors have been input to the network, and the array of output values z have been created, $\tilde{z}$ and $\bar{z}$ arrays are created by convolving the output z with the short and long range masks (masks are created using the function void *createMasks()* in **dispmasks.c**)

U and V are then calculated by subtracting either $\tilde{z}$ or $\bar{z}$ from z. The error measures $\frac{\partial U}{\partial X}, \frac{\partial V}{\partial X}$ are also created by convolution - see Appendix D for details. These values then provide error measures for the top layers of cells, which are then stored in the *allActns* data structure using the routine *storeTopLayerErrors(da)*, where da is the array of error

vectors for the output cells only. These error values are then propagated back to earlier layers by the back prop method, implemented in *propagateErrors()*.

Finally, once the error values δ_a have been computed for all cells in the network, the actual weight changes for each unit need to be calculated using $\Delta w_{uv} = \delta_v z_u$, where δ_v is the error on cell v, z_u is the output of cell u and w_{uv} is the weight connecting cell u to v. If there are x input vectors, then we will get x weight change vectors, which are simply added up to produce the final weight change vector, which is stored in the global array dw. This is done by the function *createPartials()*.

3.4.9 Testing the network

When it comes to testing the network on new images, the program simply loads in some weights previously created during a training session, reads in the input vectors, and creates the array of virtual outputs z, which are then saved to the results file. The correlation between these outputs and the actual disparity outputs is then displayed on the screen.

3.5 Overview of how the weight sharing works

To create an array of virtual outputs z of size $outputWid \times outputHt$, there should be $outputWid \times outputHt$ input vectors. The ith input vector is presented to the network and the output of the output cell calculated. This output is then stored in $z(i)$. This assumes that the network has only one cell in the output layer.

If you want to have y cells in the output layer, then you will need to make several changes:

1. Change the size of z so that it is now an array of size $y \times numInputVectors$ (normally $numInputVectors == outputWid \times outputHt$). This is done by the function *createZs()* in **dispnet.c**.

2. *Change getZ()* in **dispnet.c**. This function will extract the output of the network for each input vector and store them in the z array. So if you want to change the number of values that are extracted from the network and put into the z array, do it here.

3.6 Source Files

Here is a list of all of the relevant source files and a brief description of their contents:

bp_check_deriv.c Checker routine to see that the derivative vector is being computed ok.

cg_williams_module.c and .h The conjugate gradient engine, adapted from Jim Stone.

convolve.c The 1d and 2d convolution routines (both assuming wrap around in the masks and input images).

dispcorrn.c Correlation routines.

disperrors.c Calculate the errors for the output cells and propagate them backwards to previous layers.

dispinputs.c Read in the input images and chop them up into input vectors to be presented to the network.

dispmasks.c Create the masks to be used for the convolutions.

dispnet.c Calculates the activations of units in the network given some input.

dispscan.c This file is generated automatically from the lex file dispscan.l. The lex file stores the instructions on how to parse the parameter file.

dispvars.h Header file that stores the definition of all of the parameters that can be changed by the parameter file.

dispwts.c Routines for handling the weights data structures.

readnet.c Read in the network file and create the network data structures.

testconvolve.c Test the convolution (1D and 2D) routines.

testnet.c and .h Contains all the high level functions to run the program. Most of the hard work is done in the other files!

3.7 The makefile

To recompile the program, type `make testnet`. The makefile also has rules to convert the lex file for scanning in the parameter file into the c file. This should all be automatic.

3.8 New parameters

To help add new parameters, run the script `newparam`. This will provide you with the relevant text to put in the files `dispvars.h, dispscan.l, testnet.c`. For example:

```
newparam int inputSkipY
```

3.9 Masks

At the moment, only exponential functions have been used as the masks (1d and 2d) for producing $\tilde{z}$ and $\bar{z}$ from z. To use other functions, you will need to modify the function *createMasks()*. The masks for the derivatives $(\bar{\Phi}', \tilde{\Phi}')$ are calculated from the original masks $(\bar{\Phi}, \tilde{\Phi})$, so you will only need to change the routines to create uMask and vMask. To make new masks, it is probably easiest to take a copy of the function *create2dExpMask()*.

3.10 TAGS

To help navigate around the C source functions, a TAGS file has been created, which is useful for finding functions in emacs. When the cursor is located on top of a function name, you should be able to type `M-.` and then return a couple of times to get the definition of that function.

The tags finding function will also find macro definitions and typedef statements, but not global variables unfortunately.

If new functions are added to the existing files, you will need to do `make TAGS` to update the tags file. If you add new files to the program, you will need to add this file to the TAGS dependency list in the makefile, and then remake the tags file.

3.11 Other Libraries

The program should not require any other libraries of mine to run.

3.12 How to not weight share

At the moment, weight sharing is implemented, so that the same weights are used for each input vector.

If we are not going to weight share, then we need to create one big weight array storing all of the weights. This can be in a separate array of size $numInputVectors \times nw$, where nw is the number of weights in the normal network (ie. weightInfo.numWts). Let us call this new array $allwts$ for example.

This array can be created once no more weights are required for the normal network (for example, after the *noMoreWeights()* function in *readNet()*). These weights will then need to be given random initial weights - currently this is done for the normal network by *initWtsRnd()* in the file **testnet.c**.

Once the weights have been allocated, the appropriate weights from $allwts$ need to be copied across to the weightInfo.data structure when the network activation is being calculated. So for example, in the function *calcMeritAndPartials()*, we will need to do something like *copyWeights(vecnum)* before we call *calcAllActivations()* for each input vector.

Next, we will also need to copy the weights across into $weightInfo.data$ when we are working out the weight change for each weight, in the function *calcErrors()*. This will need to be done before the line *wts = preCellInfo[unit].wts* in this function. To be clever, the call to *copyWeights()* should not be in the *calcErrors* routine, as this is called possibly several times per input vector. So, it is probably better to put the call to *copyWeights()* in the function *propagateErrors()*.

Other changes will also be needed - but these should be much simpler. For example, when calling conjugate gradient or the checker routine, we will needed to pass the new weight array $allwts.data$ rather than $weightInfo.data$.

Also, the weight I/O functions will need to be changed so that when we output the weights (mainly within the *cg_williams* routine), then we output $allwts$ rather than $weightInfo$.

3.12.1 Testing the net

Testing the network on inputs unseen during training is done by *checkNetPerformance()*. This will need to be modified in a similar fashion to *calcMeritAndPartials()*, so that when each new input is presented, the corresponding weights are also loaded in.

4 Results

Some example runs are given in Appendix F. A summary of all of the runs performed with this net is given in the file `/disparity/ResultsSummary`. All of the results are currently stored in the directory `/rsunq/vision/stephene/dispruns`, although most of them have been compressed to save space.

5 To do

1. At the moment, the error propagation routine *calcErrors()* in **disperrors.c** assumes that the layer has tanh activation function. This needs to be tested for, rather than assuming that we should take the derivative of *tanh()* for the cells. - **Done Fri Dec 15 1995**

2. For each input vector, the program needs to keep a copy of each units activation, output and error. For a large number of input vectors (eg 40,000 for a 200x200 network), this causes the program to come to a halt, complaining about not being able to allocate enough data.

3. Both evaluation calls within CG – to evaluate the merit function and to provide the error vector – evaluate both the merit function and the error vector for simplicity. However, it would be more efficient only to evaluate the functions if the weights have changed since the last call to these functions.

4. Bias cells. If a bias cell is included in a layer, then its activation and output are set to 1.0 by the routine *setBiases()*. However, this probably assumes that the bias cell has a linear activation function. If a bias cell is included in the hidden layer at the moment (ie. the output cell receives a bias input), then this cell will be treated as a tanh unit, when in fact it is a linear unit. This problem does not occur at the moment because there is no bias unit included in the hidden layer. To solve this, I guess *setBiases()* will need to check what activation function is used for the current layer, or calcErrors will need to be modified. **Done - Fri Dec 15 1995**

5. Input cells. The function *getNextInputVector()* also currently assumes that the activation function of the input cells is the identity function. **Fri Dec 15 1995 - Normally the input cells activation function will be the identity function, and so this function just prints an error and exits if the identity function is not used for input cells.**

6 Other programs

The following auxiliary programs will be useful. (I use these quite often.). All of them are scripts and can be found in `/disparity/dispbin`. One of the sub programs, `2drawtopgm` is a binary. If it needs recompiling, it can be remade in the `src/` subdirectory.

sw Strip Whitespace from files and throw out numbers one line at a time.

snpgm Show the Newest file as a PGM file. Converts raw data files into a PGM image, and then starts up xv. Will create a pgm file as a side effect.

newparam Generate all the relevant text to edit into the source files for creating a new parameter.

conjimim Convert Jims image files into a raw format.

get2dsize Tells you the dimension of raw image files (number of columns by number of rows).

ew Extract weights from the weights file. This is mainly useful for seeing what the weights going into a hidden / output unit look like.

A Example Parameter File

Filename: gauss600.40x40.prm

```
# Gauss 600 x 600.
# Take 3x3 vectors from each image.
# Sat Dec  9 1995
# $file ~/disparity/gauss600.40x40.prm$

# 40 x 40 output array.

netFile ~/proj/2021/as/disparity/disparity/gauss600.net

useHalf 1
uhalf 2
vhalf 100

#ulambda 0.021661
#vlambda 0.000021661

# dimensionality of output cells
outputWid 40
outputHt 40


image1File ~/proj/2021/as/disparity/disparity/data/JIMS/gauss_600_by_600/image1.txt
image2File ~/proj/2021/as/disparity/disparity/data/JIMS/gauss_600_by_600/image2.txt
shiftsFile ~/proj/2021/as/disparity/disparity/data/JIMS/gauss_600_by_600/shifts.txt
checker 0
cgmax 1

maxiterations 1000
```

B Example Network File

Filename: gauss600.net

```
# Simple 3 layer network
# Mon Dec 11 1995

# $file ~/disparity/gauss600.net$
# $date December 12, 1995$

# Number of layers
nLayers 3

# Number of cells in each layer (excluding bias)
layer 0 3 6
layer 1 1 5
layer 2 1 1

# Layer information

layer 0 LINEAR bias
layer 1 TANH nobias
layer 2 LINEAR nobias

# Connectivity
connections to layer 1
full
connections to layer 2
full
```

C Example parameter file for testing the net after training

Filename: checkegg.prm

```
# Gauss 600 x 600.
# Take 3x3 vectors from each image.
# Sat Dec  9 1995
# $file ~/disparity/checkegg.prm$
# 40 x 40 output array.

# Check the disparity of the egg box, after the weights have been
# trained on the gauss600 x 600 image.

netFile ~/disparity/gauss600.net
inputWid 3
inputHt 3

# these parameters refer to the egg box (ie, the testing set)
inputSkipX 1
inputSkipY 1
numInputVectors 2500
totalInputWid 200
totalInputHt 200


useHalf 1
uhalf 2
vhalf 100

# These will vary according to the
# dimensionality of output cells
outputWid 50
outputHt 50

image1File ~/rvs/data/JIMS/sin_200_by_200_period200/image1.txt
image2File ~/rvs/data/JIMS/sin_200_by_200_period200/image2.txt
shiftsFile ~/rvs/data/JIMS/sin_200_by_200_period200/shifts.txt
checker 0
cgmax 1

maxiterations 1000

# Testing related parameters.
doLearning 0
results eggresults.dat
initWts  wts.1000
```

D Merit Function and Learning Rule

D.1 Notation

Term	Meaning
x	total input to a cell
z	Output of a cell
$\bar{z}$	Long range mean of z
$\tilde{z}$	Short range mean of z
i	Subscript for input cells
j	Subscript for hidden cells
w_{ij}	Weight from input cell i to hidden layer cell j
w_{jk}	Weight from hidden layer cell j to output layer cell k
a, b, k	Subscript for output cells
$\tilde{\Phi}$	Short range kernel for convolution to form $\tilde{z}$
$\bar{\Phi}$	Long range kernel for convolution to form $\bar{z}$

The merit function F is defined as:

$$F = \log \frac{V}{U}$$

where V is the long range variance and U is the short range variance:

$$U = \frac{1}{2}\sum_k (\tilde{z}_k - z_k)^2$$
$$V = \frac{1}{2}\sum_k (\bar{z}_k - z_k)^2$$

$\bar{z}_k$ is the long range mean, and $\tilde{z}_k$ is the short range mean for an output cell k, defined as:

$$\bar{z}_k = \sum_a \bar{\Phi}_{a-k} z_a$$
$$\tilde{z}_k = \sum_a \tilde{\Phi}_{a-k} z_a$$

where a ranges over output cells.

To calculate the derivative of the merit function with respect to each weight, we need to determine:

$$\frac{\partial F}{\partial w} = \frac{1}{V}\frac{\partial V}{\partial w} - \frac{1}{U}\frac{\partial U}{\partial w}$$

Hence, after all of the input has been presented to the network, we calculate U and V. We then need to calculate $\frac{\partial U}{\partial W}$, or to use the back propagation approach, we need to calculate $\frac{\partial U}{\partial x_a}$.

D.2 Deriving the Error Derivatives

In a similar fashion to the Back Propagation algorithm, once the errors δ are known for the output layer, they can then be used to calculate weight changes between output

and hidden layer cells. The errors can also then be propagated back to the hidden layer cells so that the weights between input and hidden layer cells can be changed:

Given δ_k for output layer cells, we can then calculate:

$$\Delta w_{jk} = \delta_k z_j$$

We can then calculate δ_j from the δ_k:

$$\delta_j = g'(x_j) \sum_k w_{jk} \delta_k$$

where $g'(x)$ is the derivative of the activation function for the hidden layer cells.

D.2.1 Computing $\frac{\partial U}{\partial x_a}$

From the normal back propagation maths, $\delta_k = \frac{\partial E}{\partial x_a}$, and so we need to find $\frac{\partial E}{\partial x_a}$:

$$U = \frac{1}{2} \sum_k (\bar{z}_k - z)^2 \tag{1}$$

$$\frac{\partial U}{\partial x_a} = \sum_k (\bar{z}_k - z) \left(\frac{\partial \bar{z}_k}{\partial x_a} - \frac{\partial z_k}{\partial x_a} \right) \tag{2}$$

To find $\frac{\partial z_k}{\partial x_a}$:

$$z_k = f(x_k) \tag{3}$$

Here the activation function of the output cells is the identity function, and so:

$$\frac{\partial z_k}{\partial x_a} = \begin{cases} 1 & \text{if } a = k \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

To find $\frac{\partial \tilde{z}_k}{\partial x_a}$:

$$\tilde{z}_k = \sum_b \tilde{\Phi}_{b-k} z_b \tag{5}$$

$$\frac{\partial \tilde{z}_k}{\partial x_a} = \sum_b \tilde{\Phi}_{b-k} \frac{\partial z_b}{\partial x_a} \tag{6}$$

Using Equation 4, we can then see that equation 6 is zero unless $a = b$:

$$\frac{\partial \tilde{z}_k}{\partial x_a} = \tilde{\Phi}_{a-k} \tag{7}$$

The values for $\frac{\partial z_k}{\partial x_a}$ and $\frac{\partial \tilde{z}_k}{\partial x_a}$ from equations 4 and 7 can then be substituted back into equation 2 to give:

$$\frac{\partial U}{\partial x_a} = \begin{cases} \sum_k (\tilde{z}_k - z)(\tilde{\Phi}_{a-k} - 1) & \text{if } a = k \\ \sum_k (\tilde{z}_k - z)(\tilde{\Phi}_{a-k}) & \text{otherwise} \end{cases} \tag{8}$$

So, if we simply define a new kernel $\tilde{\Phi}'$:

$$\tilde{\Phi'}_a = \begin{cases} \tilde{\Phi}_a - 1 & \text{if } a = 0 \\ \tilde{\Phi}_a & \text{otherwise} \end{cases} \tag{9}$$

$$\frac{\partial U}{\partial x_a} = \sum_k (\tilde{z}_k - z_k)\tilde{\Phi'}_{a-k} \tag{10}$$

The δ_a can be computed as a convolution of the vector $(\tilde{z}_a - z)$. By symmetry, a similar expression can be derived for $\frac{\partial F}{\partial x_a}$, so that the final δ_a is:

$$\delta_a = \frac{\partial F}{\partial x_a} = \frac{1}{V}\frac{\partial V}{\partial x_a} - \frac{1}{U}\frac{\partial U}{\partial x_a} \tag{11}$$

And once δ_a is defined for the output cells, we can then back propagate the errors to the earlier layers, and then calculate the weight change. In general for a weight connecting cell u to cell v, the weight change for the weight w_{uv} is $\Delta w_{uv} = \delta_v z_u$.

E Example weight extraction

This section gives a brief example on how to extract the weight vectors.

If the network is run on the parameter file /disparity/gauss600.40x40.2.prm, then after training, the `weight.info` file will say how the weights are broken down. This network is set up as follows (taken from gauss600.5x5.net:)

```
...
layer 0 5 10
layer 1 1 3
layer 2 1 1

# Layer information

layer 0 LINEAR bias
layer 1 TANH nobias
layer 2 LINEAR nobias
...
```

Hence, there are 51 (25×5 input vectors + 1 bias) cells going into each of the three hidden units. Also, there are 3 weights from hidden to output cells.

The final weights (from `wts.1000` can be extracted as follows into the files hidden.[012].pgm

```
ew 0 49 5 wts.1000 hidden0
ew 51 100 5 wts.1000 hidden1
ew 102 151 5 wts.1000 hidden2
ew 153 155 3 wts.1000 op0
```

(Note that the bias cell is cell 50, and these are not printed, hence the jump of one between end values of one set of weights and the start of the next set of weights – jumping over weights 50, 101 and 152.)

F Example runs

This is the README file in the disparity directory. Afterwards are the resulting plots and images.

```
README file for Spatial Disparity Model
---------------------------------------

Tue Dec 12 1995
---------------

This readme shows you how to train and test the net on both 1D and 2D
images.

1D Example
----------

To run the program on sin data of period 1000, with a a virtual output
layer of size 1000 x 1.  We are taking 5x1 input samples for each eye.

Create a directory where you want to put results:

cd /rsunq/vision/stephene/dispruns/testthencheck
mkdir sin5x1000

Training:

Run the program in the directory where you wish to store the output
files.

cd sin5x1000
testnet ~/disparity/sin5x1000.period1000.prm

As the program is running, it will print up the progress of conj grad
and the correlation between network output and disparity values.

Once the network has finished running, you can then view the resulting
values of z, zbar and ztilde:

gnuplot
set title "results from ~/disparity/sin5x1000.period1000.prm"
plot "<sw z.1000", "<sw zbar.1000", "<sw ztilde.1000"
plot 'corrn.dat'
plot 'dispoutputs'
quit

(Final correlation on training data = 0.936587.)

(To print files from gnuplot do:)
set term post portrait
```

```
set output "sintrainz.ps"
replot
set term x11
```

Testing:

Once the network has been trained, the network can then be tested on data that it hasnt seen before. So, for example, if we want to test the net on the sin with period 200, we create a new parameter file, say ~/disparity/testsin5.prm. (It masy be easiest to copy the parameter file that you used for training...) In this file we need to say which weights to use, which new inputs to use, and to switch the learning off. Then run testnet again:

```
testnet ~/disparity/testsin5.prm
```

This will load in the weights and then run the net on the new image you have given it. As output, it prints the correlation between the disparity and the network output. It also creates the results file (in this case sin200results.dat) which can then be viewed:

```
gnuplot
plot "<sw sin200results.dat"
quit
```

(In this case, I get the correlation 0.941803 on the test data.)

2d Example

Training and testing the net on 2d data is not much different to the 1d training - except it may take a lot longer, and images are viewed with xv rather than using gnuplot.

Training the net on the 2d Gauss data - image size is 600 x600, and we are taking 3x3 inputs from each eye. The virtual output layer is 40x40. Will then be testing the net on the egg box data.

Training:

```
cd /rsunq/vision/stephene/dispruns/testthencheck/
mkdir gauss600.40x40
cd gauss600.40x40

cd /tmp/results
testnet ~/disparity/gauss600.40x40.prm
```

Once the program has finished, you can then expect the weights:

```
snpgm z.100
snpgm z.200
snpgm z.400
snpgm z.1000
```

(top tip: to see the latest image whilst it is running, you can type snpgm z.*)

And again see the correlations:

```
gnuplot
plot 'corrn.dat',
plot 'dispoutputs'
quit
```

(Final correlation for training data = 0.909241)

Testing:

To test the net on the egg box data, we then set up another parameter file (eg ~/disparity/checkegg.prm. Again, we switch learning off, tell it which weight file and input files to read in.

```
testnet ~/disparity/checkegg.prm
```

This time, the correlation for the egg box data is 0.891811, and the results can be viewed:

```
snpgm eggresults.dat
```

Finally, we can then extract the weights and view them. The actn.info file shows us how the cells are numbered:

```
cat actn.info
```

```
Layer 0
 0  1  2
 3  4  5
 6  7  8
 9 10 11
12 13 14
15 16 17
Bias 18

Layer 1
19
```

```
20
21
22
23

Layer 2
24

and the weight.info tells us which weights are used to connect cells
together. So to extract the weights for input to hidden and hidden to
output layers (excluding bias weights 18, 37

ew 0 17 3 wts.1000 hidden0
ew 19 36 3 wts.1000 hidden1
ew 38 55 3 wts.1000 hidden2
ew 57 74 3 wts.1000 hidden3
ew 76 93 3 wts.1000 hidden2
ew 95 99 5 wts.1000 op0

xv -expand 20 hidden0.pgm hidden1.pgm hidden2.pgm hidden3.pgm op0.pgm &



Compiling on DEC: Mon 24 Nov 97
-------------------------------

When I compiled the code on the DEC at Edinburgh, I received an error
that free should only take one argument. (File cg_williams_module.c,
line 404).  Second argument was redundant, so I've removed this.
```

F.1 1D Outputs

Figure 1: Above: z, zbar and ztilde after training on the sin 5x1000 data set. Below: Correlation and merit function.

1D Outputs - Test Results

Figure 2: Output after testing sin5x1000 period 200 data on network learnt using sin period 1000 data.

F.2 2D Outputs

Figure 3: Above: The final output array z after training on the gauss 600 x 600 data (gauss600.40x40.prm). Below: correlation and merit function.

2D Outputs - Test Results

Figure 4: The image file eggresults.dat.pgm - showing the output of the net trained on the gaussian data when presented with the egg box data (checkegg.prm).

F.2.1 Weights

These are the final weights as a result of learning:

Weights to hidden unit 0 min -1.283811 max 1.048920

Weights to hidden unit 1 min -1.421594 max 1.689905

Weights to hidden unit 2: min -0.413099 max 0.379919

Weights to hidden unit 3: min -1.274979 max 1.064472

Weights to hidden unit 4: min -1.985908 max 1.680910

Weights to output unit 0: min -0.470672 max 0.666059