Neural Computing Research Group
Dept of Computer Science & Applied Mathematics
Aston University
Birmingham B4 7ET
United Kingdom
Tel: +44 (0)121 333 4631
Fax: +44 (0)121 333 4586
http://www.ncrg.aston.ac.uk/

Mixture Density Network Training by Computation in Parameter Space

David J. Evans
evansdj@aston.ac.uk

Technical Report NCRG/98/016 August 3, 1998

Abstract

Training Mixture Density Network (MDN) configurations within the NETLAB framework takes time due to the nature of the computation of the error function and the gradient of the error function. By optimising the computation of these functions, so that gradient information is computed in parameter space, training time is decreased by at least a factor of sixty for the example given. Decreased training time increases the spectrum of problems to which MDNs can be practically applied making the MDN framework an attractive method to the applied problem solver.

1 Introduction

Mixture Density Networks (MDNs) provide a framework for modelling conditional probability densities $p(\boldsymbol{t}|\boldsymbol{x})$ (Bishop, 1995). The distribution of the outputs, $\boldsymbol{t}$, is described by a parametric model whose parameters are determined by the output of a neural network, which takes $\boldsymbol{x}$ as its inputs. The general model is described by equation 1 below:

$$p(\boldsymbol{t}|\boldsymbol{x}) = \sum_{j=1}^{M} \alpha_j(\boldsymbol{x})\phi_j(\boldsymbol{t}|\boldsymbol{x}) \tag{1}$$

Were $\alpha_j(\boldsymbol{x})$ represent the mixing coefficients (which depend on $\boldsymbol{x}$) and $\phi_j(\boldsymbol{t}|\boldsymbol{x})$ are the kernel distributions of the mixture model whose parameters also depend on $\boldsymbol{x}$.

Training of mixture density networks for modelling wind vectors requires data sets of at least three thousand examples, with a MDN complexity of at least two centres and fifteen hidden units. Using the NETLAB[1] toolbox for MATLAB, training MDNs of this complexity takes at least a week, but can be longer dependent on the machine configuration and loading.

The majority of training time is spent computing two functions, the gradient of the error function and the error function. The bottle neck in these functions is the MATLAB `for` loop which is poorly optimised. These two functions are re-engineered to take advantage of the MATLAB optimised matrix functionality.

2 Software Techniques for Computation in Parameter Space

This section describes software techniques used to facilitate computation of the error and error gradient of a MDN by matrix operations. For a complete discussion of the implementation of MDNs see (Bishop, 1994)[2]. The parameter space is defined as the outputs of the Multi-Layer Perceptron (MLP), after the inputs $\boldsymbol{x}$ have been forward propagated through the network. The outputs of the MLP are vectors which contain the parameters that define the coefficients of the mixture model conditional on the inputs $\boldsymbol{x}$. For spherical Gaussian mixture models the coefficients[3] are, $\alpha_{j,n}$ the mixing coefficient for the j^{th} kernel of pattern n, $\mu_{jk,n}$ the k^{th} element of the centre of the j^{th} kernel of pattern n and $\sigma^2_{j,n}$ the width or variance of the j^{th} kernel of pattern n. The order of the coefficients in the parameter vector have been changed from that in the current NETLAB implementation of the MDN to clarify the notation of the problem. The parameter vector for the n^{th} pattern is now described as:

$$[\underbrace{\alpha_{1,n},\alpha_{2,n},\cdots,\alpha_{j,n},\cdots,\alpha_{M,n}}_{M\text{ mixing coefficients}}, \underbrace{\mu_{11,n},\mu_{12,n},\cdots,\mu_{1c,n}}_{1^{st}\text{ kernel centre}},\cdots,\underbrace{\mu_{j1,n},\mu_{j2,n},\cdots,\mu_{jc,n}}_{j^{th}\text{ kernel centre}},\cdots,\underbrace{\mu_{M1,n},\mu_{M2,n},\cdots,\mu_{Mc,n}}_{M^{th}\text{ kernel centre}},\cdots, \underbrace{\sigma^2_{1,n},\sigma^2_{2,n},\cdots,\sigma^2_{j,n},\cdots,\sigma^2_{M,n}}_{M\text{ widths}}] \tag{2}$$

where M is the number of kernels (mixtures) in the model and c is the dimension of the target space (when modelling wind vectors $c = 2$). For all patterns we have a matrix of parameters $\mathbf{P}$,

[1]Available from `http://www.ncrg.aston.ac.uk/netlab/`

[2]Available from `http://www.ncrg.aston.ac.uk/Papers/`

[3]Throughout this document the subscript identifies the model parameter and the pattern for which the model parameter refers too. For example $\alpha_{j,n}$ is the mixing coefficient of the j^{th} kernel for the n^{th} pattern.

which is split into three sub-matrices defined by $\mathbf{P}^{\alpha}$ the mixing coefficients, $\mathbf{P}^{\mu}$ which describes the centres of each kernel and $\mathbf{P}^{\sigma}$ the parameters defining the variance of each kernel. Each row corresponds to a training pattern (total N):

$$\mathbf{P}^{\alpha} = \overbrace{\begin{bmatrix} \alpha_{1,1} & \alpha_{2,1} & \cdots & \alpha_{M,1} \\ \alpha_{1,2} & \alpha_{2,2} & \cdots & \alpha_{M,2} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{1,n} & \alpha_{2,n} & \cdots & \alpha_{M,n} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{1,N} & \alpha_{2,N} & \cdots & \alpha_{M,N} \end{bmatrix}}^{\text{dimension}M} \tag{3}$$

$$\mathbf{P}^{\mu} = \overbrace{\begin{bmatrix} \mu_{11,1} & \mu_{12,1} & \cdots & \mu_{1c,1} & \cdots & \mu_{M1,1} & \mu_{M2,1} & \cdots & \mu_{Mc,1} \\ u_{11,1} & u_{12,2} & \cdots & u_{1c,2} & \cdots & \mu_{M1,2} & \mu_{M2,2} & \cdots & \mu_{Mc,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mu_{11,n} & \mu_{12,n} & \cdots & \mu_{1c,n} & \cdots & \mu_{M1,n} & \mu_{M2,n} & \cdots & \mu_{Mc,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mu_{11,N} & \mu_{12,N} & \cdots & \mu_{1c,N} & \cdots & \mu_{M1,N} & \mu_{M2,N} & \cdots & \mu_{Mc,N} \end{bmatrix}}^{\text{dimension}Mc} \tag{4}$$

$$\mathbf{P}^{\sigma} = \overbrace{\begin{bmatrix} \sigma^2_{1,1} & \sigma^2_{2,1} & \cdots & \sigma^2_{M,1} \\ \sigma^2_{1,2} & \sigma^2_{2,2} & \cdots & \sigma^2_{M,2} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma^2_{1,n} & \sigma^2_{2,n} & \cdots & \sigma^2_{M,n} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma^2_{1,N} & \sigma^2_{2,N} & \cdots & \sigma^2_{M,N} \end{bmatrix}}^{\text{dimension}M} \tag{5}$$

There is the corresponding matrix $\boldsymbol{t}$ which describes the target values for each pattern:

$$\boldsymbol{t} = \overbrace{\begin{bmatrix} t_{1,1} & t_{2,1} & \cdots & t_{c,1} \\ t_{1,2} & t_{2,2} & \cdots & t_{c,2} \\ \vdots & \vdots & \vdots & \vdots \\ t_{1,n} & t_{2,n} & \cdots & t_{c,n} \\ \vdots & \vdots & \vdots & \vdots \\ t_{1,N} & t_{2,N} & \cdots & t_{c,N} \end{bmatrix}}^{\text{dimension}M} \tag{6}$$

2.1 Computing the Gaussian activations and probabilities

Each kernel within the MDN framework is implemented using a c dimensional Gaussian. The computation of a Gaussian requires the squared distance between the targets and the centres of the Gaussian to be computed. For each centre for each pattern we require:

$$\mathbf{d}_{j,n} = ||\boldsymbol{t}_n - \boldsymbol{\mu}_j(\boldsymbol{x}_n)||^2 \tag{7}$$

In computing the squared distance we are interested in the parameters which correspond to the centres of the Gaussian.

To compute the distance the following operation is computed for each centre of each Gaussian for each pattern:

$$\begin{pmatrix} t_{1,n} \\ t_{2,n} \\ \vdots \\ t_{c,n} \end{pmatrix} - \begin{pmatrix} \mu_{j1,n} \\ \mu_{j2,n} \\ \vdots \\ \mu_{jc,n} \end{pmatrix} \tag{8}$$

This operation can be completed as one matrix operation as follows:

$$\mathbf{D} = \overbrace{\begin{bmatrix} t_{1,1} & t_{2,1} & \cdots & t_{c,1} & \cdots & t_{1,1} & t_{2,1} & \cdots & t_{c,1} \\ t_{1,2} & t_{2,2} & \cdots & t_{c,2} & \cdots & t_{1,2} & t_{2,2} & \cdots & t_{c,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ t_{1,n} & t_{2,n} & \cdots & t_{c,n} & \cdots & t_{1,n} & t_{2,n} & \cdots & t_{c,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ t_{1,N} & t_{2,N} & \cdots & t_{c,N} & \cdots & t_{1,N} & t_{2,N} & \cdots & t_{c,N} \end{bmatrix}}^{\text{dimension}Mc}$$

$$- \overbrace{\begin{bmatrix} \mu_{11,1} & \mu_{12,1} & \cdots & \mu_{1c,1} & \cdots & \mu_{M1,1} & \mu_{M2,1} & \cdots & \mu_{Mc,1} \\ u_{11,1} & u_{12,2} & \cdots & u_{1c,2} & \cdots & \mu_{M1,2} & \mu_{M2,2} & \cdots & \mu_{Mc,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mu_{11,n} & \mu_{12,n} & \cdots & \mu_{1c,n} & \cdots & \mu_{M1,n} & \mu_{M2,n} & \cdots & \mu_{Mc,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mu_{11,N} & \mu_{12,N} & \cdots & \mu_{1c,N} & \cdots & \mu_{M1,N} & \mu_{M2,N} & \cdots & \mu_{Mc,N} \end{bmatrix}}^{\text{dimension}Mc} \tag{9}$$

That is

$$\mathbf{D} = \overbrace{\begin{bmatrix} t_{1,1} & t_{2,1} & \cdots & t_{c,1} & \cdots & t_{1,1} & t_{2,1} & \cdots & t_{c,1} \\ t_{1,2} & t_{2,2} & \cdots & t_{c,2} & \cdots & t_{1,2} & t_{2,2} & \cdots & t_{c,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ t_{1,n} & t_{2,n} & \cdots & t_{c,n} & \cdots & t_{1,n} & t_{2,n} & \cdots & t_{c,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ t_{1,N} & t_{2,N} & \cdots & t_{c,N} & \cdots & t_{1,N} & t_{2,N} & \cdots & t_{c,N} \end{bmatrix}}^{\text{dimension}Mc} - \mathbf{P}^{\mu} \tag{10}$$

Inspection of equation (9) reveals that the target data is repeated for each centre, and so by reshaping the target matrix the distances can be computed as matrix operations within MATLAB. The following MATLAB code reshapes the $\boldsymbol{t}$ vector into the form required in equation (9).

```
% Build t that suits parameters,
% that is repeat t for each centre

t = kron(ones(1,ncentres),t);
```

```
% Which gives results like the following
% --------------------------------------

t =
     1     2     3
     4     5     6
     7     8     9
t =

     1     2     3     1     2     3     1     2     3
     4     5     6     4     5     6     4     5     6
     7     8     9     7     8     9     7     8     9
```

The following code the completes the squared distance operation.

```
% Do subtraction
diff = t - centres;
% Square each result

diff2 = diff.^2;

% reshape and sum each component

diff2 = reshape(diff2',dim_target,(ntarget*ncentres))';

% This is the transformation after the reshape
% centres are zero for this illustration
% diff2 =
%
%     1     4     9     1     4     9     1     4     9
%    16    25    36    16    25    36    16    25    36
%    49    64    81    49    64    81    49    64    81
%
%
% diff2 =
%
%     1     4     9
%     1     4     9
%     1     4     9
%    16    25    36
%    16    25    36
%    16    25    36
%    49    64    81
%    49    64    81
%    49    64    81


sum2 = sum(diff2,2);

% Calculate the sum of distance, and reshape
% so that we have a distance for each centre per target
% i.e. ntarget * ncentres

dist2 = reshape(sum2,ncentres,ntarget)';
% This is the transformations after the reshape
% sum2 =
%
%    14
%    14
%    14
%    77
%    77
%    77
%   194
%   194
%   194
%
%
% dist2 =
%
%    14    14    14
%    77    77    77
%   194   194   194
```

Where

$$\text{dist2} = \overbrace{\begin{bmatrix} d_{1,1} & d_{2,1} & \cdots & d_{M,1} \\ d_{1,2} & d_{2,2} & \cdots & d_{M,2} \\ \vdots & \vdots & \vdots & \vdots \\ d_{1,n} & d_{2,n} & \cdots & d_{M,n} \\ \vdots & \vdots & \vdots & \vdots \\ d_{1,N} & d_{2,N} & \cdots & d_{M,N} \end{bmatrix}}^{\text{dimension}M} \tag{11}$$

and the equation (7) is now in matrix form. Now that the distance has been computed it is a natural progression to compute the activations of each Gaussian kernel.

$$\mathbf{A} = \overbrace{\begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{M,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{M,2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{M,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1,N} & a_{2,N} & \cdots & a_{M,N} \end{bmatrix}}^{\text{dimension}M} \tag{12}$$

where

$$a_{j,n} = \phi_j(\boldsymbol{t}_n|\boldsymbol{x}_n) = -\frac{1}{(2\pi\sigma_{j,n}^2)^{\frac{c}{2}}} \exp\left\{\frac{d_{j,n}}{2\sigma_{j,n}^2}\right\} \tag{13}$$

The probabilities of each Gaussian are then computed by multiplying each activation by the respective mixing coefficient:

$$\mathbf{Pr} = \overbrace{\begin{bmatrix} \alpha_{1,1}a_{1,1} & \alpha_{2,1}a_{2,1} & \cdots & \alpha_{M,1}a_{M,1} \\ \alpha_{1,2}a_{1,2} & \alpha_{2,2}a_{2,2} & \cdots & \alpha_{M,2}a_{M,2} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{1,n}a_{1,n} & \alpha_{2,n}a_{2,n} & \cdots & \alpha_{M,n}a_{M,n} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{1,N}a_{1,N} & \alpha_{2,N}a_{2,N} & \cdots & \alpha_{M,N}a_{M,N} \end{bmatrix}}^{\text{dimension}M} \tag{14}$$

These principles are implemented in a function called `f_prob` listed below. Where `mixparams.vars` refers to the matrix $\mathbf{P}^{\sigma}$, line 12 computes the squared distance, line 22 computes the matrix $\mathbf{A}$ and finally line 28 is the computation of $\mathbf{Pr}$

```
1     function [prob,a] = f_prob(net,mixparams,t)
2
3     ncentres   = net.mix.ncentres;
4     dim_target = net.mix.nin;
5     nparams    = net.mix.nparams;
6     ntarget    = size(t, 1);
7
8
9
10    % Calculate squared norm matrix, of dimension (ndata, ncentres)
11    % vector (ntarget * ncentres)
12    dist2 = f_dist2(net,mixparams,t);
13
14    % Calculate variance factors
15    variance = 2.*mixparams.vars;
16
17
18    % Compute the normalisation term
19    normal  = ((2.*pi).*mixparams.vars).^(dim_target./2);
20
21    % Now compute the activations
22    a = exp(-(dist2./variance))./normal;
23
24
25    % Accumulate negative log likelihood of targets
26
27
28    prob = mixparams.mixcoeffs.*a;
29
```

2.2 Computing the probability of a point, π_j

The probability of a point is defined as:

$$\pi_j = \frac{\alpha_j \phi_j}{\sum_{l=1}^{m} \alpha_l \phi_l} \tag{15}$$

The computation of equation (15) is implemented using row and column operations on the matrix described by equation (14):

$$\pi_{j,n} = \frac{pr_{j,n}}{pr_{1,n} + pr_{2,n} + \cdots + pr_{M,n}} \tag{16}$$

and

$$\Pi = \overbrace{\begin{bmatrix} \pi_{1,1} & \pi_{2,1} & \cdots & \pi_{M,1} \\ \pi_{1,2} & \pi_{2,2} & \cdots & \pi_{M,2} \\ \vdots & \vdots & \vdots & \vdots \\ \pi_{1,n} & \pi_{2,n} & \cdots & \pi_{M,n} \\ \vdots & \vdots & \vdots & \vdots \\ \pi_{1,N} & \pi_{2,N} & \cdots & \pi_{M,N} \end{bmatrix}}^{\text{dimension}M} \tag{17}$$

which is implemented in MATLAB as follows;

```
function [post, a] = f_post(net, mixparams, t)
%
% Check that inputs are consistent

[prob a] = f_prob(net,mixparams,t);

s = sum(prob, 2);
% Set any zeros to one before dividing
s = s + (s==0);
post = prob./(s*ones(1, net.mix.ncentres));
```

2.3 Reshaping the parameter matrix

When computing the derivative $\frac{\partial E_n}{\partial z^{\mu}_{jk}}$ its is necessary that each of the components of the kernel centres is operated on by its respective variance and posterior. To facilitate this operation as a single matrix operation one further reshape is required. This takes a matrix (say $\mathbf{P}^{\sigma}$) and rebuilds the columns so that the dimensions are the same as $\mathbf{P}^{\mu}$, and populated such that for each $\mu_{jk,n}$ there is a corresponding $\sigma^2_{j,n}$. An example would be as follows where each of the centre parameters is matched to their corresponding width parameter.

$$\mathbf{P}^{\mu} = \overbrace{\begin{bmatrix} \mu_{11,1} & \mu_{12,1} & \cdots & \mu_{1c,1} & \cdots & \mu_{M1,1} & \mu_{M2,1} & \cdots & \mu_{Mc,1} \\ u_{11,1} & u_{12,2} & \cdots & u_{1c,2} & \cdots & \mu_{M1,2} & \mu_{M2,2} & \cdots & \mu_{Mc,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mu_{11,n} & \mu_{12,n} & \cdots & \mu_{1c,n} & \cdots & \mu_{M1,n} & \mu_{M2,n} & \cdots & \mu_{Mc,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mu_{11,N} & \mu_{12,N} & \cdots & \mu_{1c,N} & \cdots & \mu_{M1,N} & \mu_{M2,N} & \cdots & \mu_{Mc,N} \end{bmatrix}}^{\text{dimension} Mc} \tag{18}$$

$$\text{corresponding widths} = \overbrace{\begin{bmatrix} \sigma^2_{1,1} & \sigma^2_{1,1} & \cdots & \sigma^2_{1,1} & \cdots & \sigma^2_{M,1} & \sigma^2_{M,1} & \cdots & \sigma^2_{M,1} \\ \sigma^2_{1,1} & \sigma^2_{1,2} & \cdots & \sigma^2_{1,2} & \cdots & \sigma^2_{M,2} & \sigma^2_{M,2} & \cdots & \sigma^2_{M,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma^2_{1,n} & \sigma^2_{1,n} & \cdots & \sigma^2_{1,n} & \cdots & \sigma^2_{M,n} & \sigma^2_{M,n} & \cdots & \sigma^2_{M,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma^2_{1,N} & \sigma^2_{1,N} & \cdots & \sigma^2_{1,N} & \cdots & \sigma^2_{M,N} & \sigma^2_{M,N} & \cdots & \sigma^2_{M,N} \end{bmatrix}}^{\text{dimension} Mc} \tag{19}$$

The following MATLAB code shows how to reshape the parameter matrix into the desired form,

```
z = [ 1 2 3;4 5 6; 7 8 9]

z = kron(ones(dim_target,1),z);
z = reshape(z,ntarget,(ncentres*dim_target));

% Gives results like this
% z =
%
%      1     2     3
%      4     5     6
%      7     8     9
%
%
% z =
%
%      1     1     1     2     2     2     3     3     3
%      4     4     4     5     5     5     6     6     6
%      7     7     7     8     8     8     9     9     9
```

3 Computing the error function in parameter space

The negative log likelihood error function for a MDN is defined as (Bishop, 1995; Bishop, 1994):

$$E = \sum_{n=1}^{N} - \ln \left\{ \sum_{j=1}^{m} \alpha_j(\mathbf{x}_n)\phi_j(\mathbf{t}_n|\mathbf{x}_n) \right\} \tag{20}$$

Then each element in equation (14) is defined as follows :

$$Pr_{j,n} = \alpha_j(\mathbf{x}_n)\phi_j(\mathbf{t}_n|\mathbf{x}_n) \tag{21}$$

and the implementation becomes row and column operations in MATLAB. The following code shows the function `f_mdnerr`, which implements equation (20).

```
1     function err = f_mdnerr(net, x, t)
2     %F_MDNERR       Evaluate error function for Mixture Density Network.
3
4     % Check arguments for consistency
5
6     errstring = consist(net, 'f_mdn', x, t);
7     if ~isempty(errstring)
8       error(errstring);
9     end
10
11    % Get the output mixture models
12    mixparams = f_mdnfwd(net, x);
13    probs     = f_prob(net,mixparams,t);
14    err       = sum( -log(max(eps,sum(probs,2))));
```

Line 13 returns a matrix of probabilities, and so the computation of the error for each pattern is a summation along the rows of `probs`, and the total error becomes a summation of the vector resulting from `sum(probs,2)`

4 Computing the gradient of the error function in parameter space

First forward propagate the inputs x through the MLP, which returns a matrix containing the parameters for each pattern (see Appendix A for source code listing)

```
[mixparams, z] = f_mdnfwd(net, x);
```

`mixparams` is a structure containing three matrices $\mathbf{P}^\alpha$, $\mathbf{P}^\mu$ and $\mathbf{P}^\sigma$ of the form described in equations (3), (4) and (5) respectively. Using techniques similar to those described in Section 2 all the derivatives are then computed with matrix operations.

4.1 Computing the error gradient with respect to the mixing coefficients, $\frac{\partial E_n}{\partial z^\alpha}$

The standard result for each centre is:

$$\frac{\partial E_n}{\partial z_j^\alpha} = \alpha_j - \pi_j \tag{22}$$

is simply computed as

$$\frac{\partial E_n}{\partial z^\alpha} = \Delta^\alpha = \mathbf{P}^\alpha - \Pi \tag{23}$$

4.2 Computing the error gradient with respect to the kernel centres, $\frac{\partial E_n}{\partial z^\mu_{jk}}$

The general result is

$$\frac{\partial E_n}{\partial z^\mu_{jk}} = \pi_j \left\{ \frac{\mu_{jk} - t_k}{\sigma_j^2} \right\} \tag{24}$$

Using techniques described in Section 2.3 matrices $\mathbf{P}^\sigma$ and Π can be reshaped, and the following operation is computed within Matlab:

$$\frac{\partial E_n}{\partial z^\mu_{jk}} = \Delta^\mu =$$

$$\overbrace{\begin{bmatrix}
\pi_{1,1}\left\{\frac{\mu_{11,1}-t_{1,1}}{\sigma^2_{1,1}}\right\} & \cdots & \pi_{1,1}\left\{\frac{\mu_{1c,1}-t_{c,1}}{\sigma^2_{1,1}}\right\} & \cdots & \pi_{M,1}\left\{\frac{\mu_{M1,1}-t_{1,1}}{\sigma^2_{M,1}}\right\} & \cdots & \pi_{M,1}\left\{\frac{\mu_{Mc,1}-t_{c,1}}{\sigma^2_{M,1}}\right\} \\
\pi_{1,2}\left\{\frac{\mu_{11,2}-t_{1,2}}{\sigma^2_{1,2}}\right\} & \cdots & \pi_{1,2}\left\{\frac{\mu_{1c,2}-t_{c,2}}{\sigma^2_{1,2}}\right\} & \cdots & \pi_{M,2}\left\{\frac{\mu_{M1,2}-t_{1,2}}{\sigma^2_{M,2}}\right\} & \cdots & \pi_{M,2}\left\{\frac{\mu_{Mc,2}-t_{c,2}}{\sigma^2_{M,2}}\right\} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\pi_{1,n}\left\{\frac{\mu_{11,n}-t_{1,n}}{\sigma^2_{1,n}}\right\} & \cdots & \pi_{1,n}\left\{\frac{\mu_{1c,n}-t_{c,n}}{\sigma^2_{1,n}}\right\} & \cdots & \pi_{M,n}\left\{\frac{\mu_{M2,n}-t_{2,n}}{\sigma^2_{M,n}}\right\} & \cdots & \pi_{M,n}\left\{\frac{\mu_{Mc,n}-t_{c,n}}{\sigma^2_{M,n}}\right\} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\pi_{1,N}\left\{\frac{\mu_{11,N}-t_{1,N}}{\sigma^2_{1,N}}\right\} & \cdots & \pi_{1,N}\left\{\frac{\mu_{1c,N}-t_{c,N}}{\sigma^2_{1,N}}\right\} & \cdots & \pi_{M,N}\left\{\frac{\mu_{M2,N}-t_{2,N}}{\sigma^2_{M,N}}\right\} & \cdots & \pi_{M,N}\left\{\frac{\mu_{Mc,N}-t_{c,N}}{\sigma^2_{M,N}}\right\}
\end{bmatrix}}^{\text{dimension} Mc} \tag{25}$$

4.3 Computing the error gradient with respect to the kernel widths $\frac{\partial E_n}{\partial z^\sigma_j}$

The general result:

$$\frac{\partial E_n}{\partial z^\sigma_j} = -\frac{\pi_{j,n}}{2}\left\{ \frac{\|\boldsymbol{t}_n - \boldsymbol{\mu}_j(\boldsymbol{x}_n)\|^2}{\sigma^2_{j,n}} - c \right\} \tag{26}$$

is computed using the functions and matrices defined previously. Using the Matlab operator `./` and `.*`, for element-wise division and multiplication respectively, the computation becomes:

$$\frac{\partial E_n}{\partial z^\sigma_j} = \Delta^\sigma = \frac{\Pi}{2}\left\{ \frac{\text{dist2}}{\mathbf{P}^\sigma} - \mathbf{C} \right\} \tag{27}$$

where $\mathbf{C}$ is a matrix of dimension $(npatterns, ncentres)$ with each element taking the value c, the dimension of the target space.

A full listing of the MATLAB function to compute the gradient of the error function is given in Appendix B

5 Testing

5.1 Training Accuracy

Tests using '`gradcheck`' from NETLAB toolbox show that, for the configurations tested, the implementation of the gradient function performs to specification.

Comparison of `demmdn1` and `f_demmdn1` produces interesting results. Initially the training errors appear to be identical (to the 6th decimal place). After the 36th iteration (`demmdn1` trains for 200) the errors diverge in the 6th decimal. Comparing scale, they are identical (to the 6th decimal place) until the 105th iteration, where `f_demmdn1` remains static for one iteration, there after the one step lagged scale of `f_demmdn1` is the same as `demmdn1`. An explanation of these differences is offered by inspecting the average delta[4] and the average of the modulus of delta for the results returned by `gradcheck` as shown in table 1.

MDN type	`mean(delta)`	`mean(abs(delta))`
`f_demmdn`	-3.4169e-009	4.0190e-008
`demmdn`	-1.6406e-009	4.1471e-008

Table 1: Results of running `gradcheck`

The mean delta for `f_demmdn1` is at least twice that of `demmdn1`, whilst the `mean(abs(delta))` are of the same magnitude but differ in the 9th decimal place. The scaled conjugate gradients optimisation algorithm (Bishop, 1995) uses information on the gradient of the error function to minimise the error function. It is suggested that the differences in computed gradient accumulates during training and accounts for the divergence of training errors between `demmdn1` and `f_demmdn1`.

5.2 Training Speed

The programme `demmdn1` was also used to illustrate the improvement in training time by comparing the results of the MATLAB `profile` function for each implementation. Two examples of `profile` reports are shown in Appendix C. Ten profile reports of each method where collected by running batch jobs (on a Silicon Graphics Challenge L, holding 4 x 200MHz R10000 CPUs, 512 Mb RAM, and running IRIX 6.2.). The summaries of these reports are tabulated in table 2. Note although the standard deviation of `demmdn1` seems large, both standard deviations relative to their means are of the same order. The difference in mean execution time illustrates the improvement in training time by computation of the error and error gradient functions in parameter space.

[4] *delta* is the difference between the computation of the error derivatives obtained from the analytic expressions and those calculated using finite differences (Bishop, 1994).

MDN type	`mean(execution time)` s	`std(execution_time)` s
`f_demmdn1`	`10.99`	`0.23`
`demmdn1`	`723.18`	`51.52`

Table 2: Summary results of running `demmdn1` Netlab package ten times.

6 Conclusions

The techniques presented here for training Mixture Density Networks show that training in parameter space leads to substantial gains in training time without loss of accuracy. Examination of the gradient information shows that differences in training errors are due to small differences in the computation of the gradient information. The example presented in this report shows an improvement in mean training time of at least a factor of sixty. The decreased training time allows us to tackle more complicated problems, which previously took too long to train to be of any practical use. Such an example, modelling wind vectors conditional on satellite information, discussed briefly in Section 1, shows training times improved from several days to a few hours.

Acknowledgements

I thank Dan Cornford, for his patient reading and constructive comments on the draft versions of this report, and Ian Nabney, for his constructive comments on the second draft of this report and suggesting to change `mixparams` from a matrix to a Matlab data structure and to use the Matlab function `kron` instead of complex matrix reshapes implemented in the first version of the software.

Appendices

A Listing of MDN forward propagation function

```
function [mixparams, z, a] = f_mdnfwd(net, x)
%F_MDNFWD Forward propagation through Mixture Density Network.
%
% Description
% MIXPARAMS = MDNFWD(NET, X) takes a mixture density network data
% structure NET and a matrix X of input vectors, and forward propagates
% the inputs through the network to generate a structure MIXPARAMS which
%       describe the parameters of a mixture model.  Each row of X represents
%       one input vector and the corresponding row of MIXPARAMS represents the
%       data structure vector of the corresponding mixture model parameters
%       for the conditional probability of target vectors.
%
% [MIXPARAMS, Z] = MDNFWD(NET, X) also generates a matrix Z of the
% hidden unit activations where each row corresponds to one pattern.
%
```

```
% [MIXPARAMS Z, A] = MLPFWD(NET, X) also returns a matrix A  giving the
% summed inputs to each output unit, where each row  corresponds to one
% pattern.
%
% See also
% GMM, MDN, F_MDNERR, F_MDNGRAD, MLPFWD, MDNMIX
%

% Copyright (c) Christopher M Bishop, Ian T Nabney (1996, 1997)
% Copyright (c) David J Evans (1998)

% Check arguments for consistency
errstring = consist(net, 'f_mdn', x);
if ~isempty(errstring)
  error(errstring);
end

% Extract mlp and mixture model descriptors
mlpnet = net.mlp;
mix    = net.mix;

ncentres   = mix.ncentres; % Number of components in mixture model
dim_target = mix.nin; % Dimension of targets
nparams    = mix.nparams;  % Number of parameters in mixture model

% Propagate forwards through MLP
[y, z, a] = mlpfwd(mlpnet, x);

% Compute the postion for each parameters in the whole
% matrix.  Used to define the mixparams structure
mixcoeff  = [1:1:ncentres];
centres   = [ncentres+1:1:(ncentres*(1+dim_target))];
variances = [(ncentres*(1+dim_target)+1):1:nparams];

% Convert output values into mixture model parameters

% Use softmax to calculate priors
% Prevent overflow and underflow: use same bounds as glmfwd
% Ensure that sum(exp(y), 2) does not overflow
maxcut = log(realmax) - log(ncentres);
% Ensure that exp(y) > 0
mincut = log(realmin);
temp = min(y(:,1:ncentres), maxcut);
temp = max(temp, mincut);
temp = exp(temp);
mixpriors = temp./(sum(temp, 2)*ones(1,ncentres));


% This is the dimension of the centres(1, ncentres*dim_target)
mixcentres = y(:,(ncentres+1):ncentres*(1+dim_target));

% Variances are exp of network outputs
mixwidths = exp(y(:,(ncentres*(1+dim_target)+1):nparams));
% Now build up all the mixture model weight vectors
```

```
ndata = size(x, 1);

% Return parameters

mixparams.mixcoeffs = mixpriors;
mixparams.centres   = mixcentres;
mixparams.vars      = mixwidths;
```

B Listing of the MDN error gradient implementation

```
function g = f_mdngrad(net, x, t)
%F_MDNGRAD Evaluate gradient of error function for Mixture Density Network.
%
% Description
% G = F_MDNGRAD(NET, X, T) takes a mixture density network data
% structure NET, a matrix X of input vectors and a matrix T of target
% vectors, and evaluates the gradient G of the error function with
% respect to the network weights. The error function is negative log
% likelihood of the target data.  Each row of X corresponds to one
% input vector and each row of T corresponds to one target vector.
%
% See also
% F_MDN, F_MDNFWD, F_MDNERR, MLPBKP, MDNMIX
%

% Copyright (c) Christopher M Bishop, Ian T Nabney (1996, 1997)
% Copyright (c) David J Evans (1998)

% Check arguments for consistency
errstring = consist(net, 'f_mdn', x, t);
if ~isempty(errstring)
  error(errstring);
end

[mixparams, z] = f_mdnfwd(net, x);

% Compute gradients at MLP outputs: put the answer in deltas
ncentres      = net.mix.ncentres; % Number of components in mixture model
dim_target    = net.mix.nin; % Dimension of targets
nmixparams    = net.mix.nparams;        % Number of parameters in mixture model
ntarget       = size(t,1);
deltas        = zeros(ntarget, net.mlp.nout);
e = ones(ncentres, 1);
f = ones(1, dim_target);


post = f_post(net,mixparams,t);

% Calculate prior derivatives
deltas(:,1:ncentres)  = mixparams.mixcoeffs - post;
```

```
% Calculate centre derivatives
long_t = kron(ones(1,ncentres),t);

centre_err = mixparams.centres - long_t;

% Get the post to match each ujk
% this array will be (ntarget,(ncentres*dim_target))
long_post = kron(ones(dim_target,1),post);
long_post = reshape(long_post,ntarget,(ncentres*dim_target));

% Get the variance to match each ujk
% this array will be ntarget*(ncentres*dim_target)
var = mixparams.vars;
var = kron(ones(dim_target,1),var);
var = reshape(var,ntarget,(ncentres*dim_target));

% Compute delta
deltas(:,(ncentres+1):(ncentres*(1+dim_target))) = ...
                       (centre_err.*long_post)./var;

% Compute variance derivatives
dist2            = f_dist2(net,mixparams,t);
c                = dim_target*ones(ntarget,ncentres);
deltas(:,(ncentres*(1+dim_target)+1):nmixparams) = ...
                      post.*((dist2./mixparams.vars)-c)./(-2);


g = mlpbkp(net.mlp, x, z, deltas);
```

C Timing comparisons

Example results from running `profile` function in MATLAB

Results for `f_demmdn1`

```
Total time in "~/Netlab/netopt.m": 10.89 seconds

100% of the total time was spent on lines:
        [38 35]

               34: % Extract weights from network as single vector
 0.01s,  0%    35: w = feval(pakstr, net);
               36:
               37: % Carry out optimisation
10.88s, 100%   38: [s{1:nargout}] = eval(optstring);
               39: w = s{1};
```

Results for `demmdn1`

```
Total time in "~/Netlab/netopt.m": 700.48 seconds

100% of the total time was spent on lines:
        [38 35]

                34: % Extract weights from network as single vector
  0.02s,  0%    35: w = feval(pakstr, net);
                36:
                37: % Carry out optimisation
700.46s, 100%   38: [s{1:nargout}] = eval(optstring);
                39: w = s{1};
```

References

Bishop, C. M. 1994. Mixture density networks. Technical Report NCRG/94/004, Department of Computer Science and Applied Mathematics, Aston University, Birmingham, B4 7ET, UK.

Bishop, C. M. 1995. *Neural Networks and Pattern Recognition.* Oxford University Press.