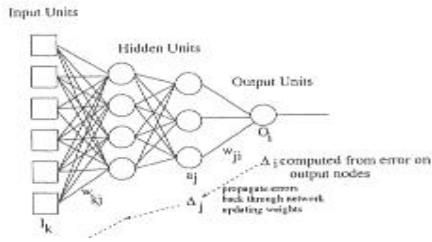


Non-Symbolic AI lecture 8

EASy

Backpropagation in a multi-layer perceptron



Non-Symbolic AI lec 8

Summer 2006

1

Jiggling the weights in each layer

EASy

When we present the training sets, for each Input we have an actual Output, compared with the Target gives the Error

$$\text{Error } E = T - O$$

Backprop allows you to use this error-at-output to adjust the weights arriving at the output layer

... but then also allows you to calculate the effective error '1 layer back', and use this to adjust the weights arriving there

... and so on **back-propagating** errors through any number of layers

Non-Symbolic AI lec 8

Summer 2006

2

Differentiable sigmoid

EASy

The trick is the use of a sigmoid as the non-linear transfer function

$$y = g(x) = \frac{1}{1 + e^{-x}}$$

Because this is nicely differentiable – it so happens that

$$\frac{dg}{dx} = g'(x) = g(x)(1 - g(x))$$

Non-Symbolic AI lec 8

Summer 2006

3

How to do it in practice

EASy

For the details, consult a suitable textbook

e.g. Hertz, Krogh and Palmer "Intro to the Theory of Neural Computation" Addison Wesley 1991

But here is a step by step description of how you have to code it up.

Non-Symbolic AI lec 8

Summer 2006

4

Initial decisions

EASy

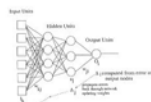
Decide how many inputs you have.

Add one more pseudo-input (clamped to value 1) for biases in next layer.

Decide how many hidden nodes you have. Add one more, clamped to value 1, for biases to next layer.

Probably you just have 1 hidden layer, tho in principle can be more.

Decide how many output nodes you have.



Non-Symbolic AI lec 8

Summer 2006

5

Weights

EASy

Now you can make vectors holding all the weights. Suppose there are a input units, b hidden units, c output units

```
float i_to_h_wts[a+1][b];
```

```
float h_to_o_wts[b+1][c];
```

The +1 in each case is to account for the biases.

Initialise all the weights to small random values.

Non-Symbolic AI lec 8

Summer 2006

6

Presenting the inputs

EASy

Now we are going to present members of the training set to the network, and see what outputs result – and how good they are.

We could present a whole **batch** from the training set, and calculate how to jiggle the weights on the basis of all the errors

Or we can do it incrementally, one at a time. This is what we shall do – present a single input vector, calculate the resulting activations at the hidden and output layers.

First this single pass forward.

Activations

EASy

We need vectors(or arrays) to hold the activations at each layer.

float inputs[a+1]; inputs[a]=1.0; /* bias */

float hidden[b+1]; hidden[b]=1.0; /* bias */

float outputs[c];

float targets[c]; /* what the outputs **should** be */

Pick out one of the training set, and set the input values equal to this member.

Now calculate activations in hidden layer

Forward to Hidden layer

EASy

```
for (i=0;i<b;i++) {
    sum=0.0;
    for (j=0;j<a+1;j++)
        sum += inputs[j] * i_to_h_wts[j][i];
    hidden[i] = sigmoid(sum);
}
```

Using a sigmoid function you have written to calculate

$$y = g(x) = \frac{1}{1 + e^{-x}}$$

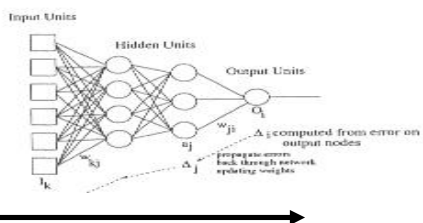
Forward to Output layer

EASy

```
for (i=0;i<c;i++) {
    sum=0.0;
    for (j=0;j<b+1;j++)
        sum += hidden[j] * h_to_o_wts[j][i];
    output[i] = sigmoid(sum);
}
```

End of forward pass

EASy



That has got us all the way forward.

Calculating delta's

EASy

Now for all the nodes, in all bar the first input layer, we are going to calculate deltas (appropriate basis for changes at those nodes).

float delta_hidden[b+1];

float delta_outputs[c];

The underlying formula used is $\partial_i = g'(x)[T_i - O_i]$

Which conveniently is $\partial_i = g(x)(1 - g(x))[T_i - O_i]$



Deltas on output layer

EASy

```
for (j=0;j<c;j++)
    delta_outputs[j] = outputs[j]*(1.0 - outputs[j]) *
                        (target[j] - outputs[j]);
```

Store these deltas for the final output layer, and also use this to propagate the errors backward (using the weights on the connections) through to the hidden layer

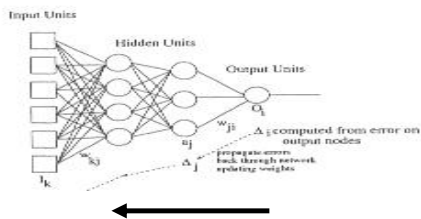
Deltas on Hidden layer

EASy

```
for (j=0;j<b+1;j++) {
    error = 0.0;
    for (k=0;k<c;k++)
        error += h_to_o_wts[j][k] * delta_outputs[k];
    delta_hidden[j] = hidden[j] * (1.0 - hidden[j]) * error;
}
```

End of backward pass

EASy



That has got us all the way backward, calculating deltas.

Now the weight-changes

EASy

OK, we have calculated the errors at all the nodes, including hidden nodes. Let's use these to calculate weight changes everywhere – using a learning rate parameter η

```
float delta_i_to_h_wts[a+1][b];
float delta_h_to_o_wts[b+1][c];
```

Calculate the weight-changes

EASy

```
for (j=0; j<c; j++)
    for (k=0; k<b+1; k++) {
        delta_h_to_o_wts[k][j] = eta *
            delta_outputs[j] * hidden[k];
        h_to_o_wts[k][j] += delta_h_to_o_wts[k][j];
```

That gives new values for all the hidden-to-output weights

Calculate the weight-changes (2)

EASy

```
for (j=0; j<b; j++)
    for (k=0; k<a+1; k++) {
        delta_i_to_h_wts[k][j] = eta *
            delta_hidden[j] * inputs[k];
        i_to_h_wts[k][j] += delta_i_to_h_wts[k][j];
```

That gives new values for all the inputs-to-hidden weights

How big is eta ?

EASy

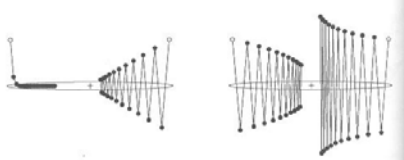
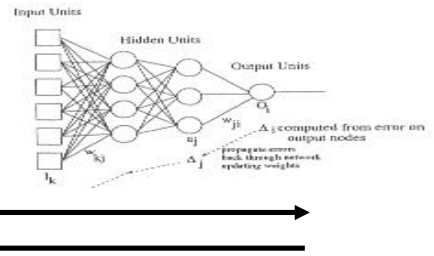


FIGURE 5.10 Gradient descent on a simple quadratic surface (the left and right parts are copies of the same surface). Four trajectories are shown, each for 20 steps from the open circle. The minimum is at the + and the ellipse shows a constant error contour. The only significant difference between the trajectories is the value of η , which was 0.02, 0.0476, 0.049, and 0.0505 from left to right.

For example, eta = 0.02 is a common rate to use.

End of forward and backward pass

EASy



Repeat many times

EASy

That was a single pass, based on a single member of the training set, and making small jiggles in the weights (based on the learning rate eta, e.g. $\gamma = 1.0$)

Repeat this lots of times for different members of the training set, indeed going back and using each member many times – each time making a small change in the weights.

Eventually (fingers crossed) the errors (Target – Output) will get satisfactorily small, and unless it has over-fitted the training set, the Black Box should generalise to unseen test data.

Wrapping up in a program

EASy

I presented the things-to-do in a pragmatic order, but for writing a program you have to wrap it up a bit differently.

Define all your weights, activations, deltas as arrays of floats (or doubles) first. Define your sigmoid function.

Write functions for a pass forward, and a pass backward.

Write a big loop that goes over many presentations of inputs.

All this is left as an exercise for the reader.

Problems ?

EASy

If there are, for instance, 100 weights to be jiggled around, then backprop is equivalent to gradient descent on a 100-dimensional error surface – like a marble rolling down towards the basin of minimum error.

(there are other methods, e.g. conjugate gradient descent, that might be faster).

What about worries that 'the marble may get trapped in a local optimum'?

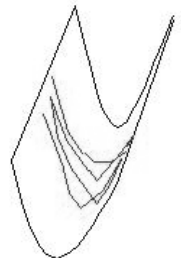
Actually, that rarely happens, though another problem is more frequent.

Valleys

EASy

Using the marble metaphor, there may well be valleys like this, with steep sides and a gently sloping floor.

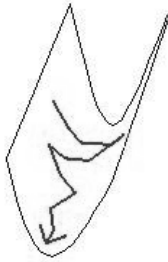
Gradient descent tends to waste time swooshing up and down each side of the valley (think marbles!)



Momentum

EASy

If you can add a **momentum** term, that tends to cancel out the back-and-forth movements and emphasise any consistent direction, then this will go down such valleys with gentle bottom-slopes much more successfully – faster.



Implementing momentum

EASy

This means keeping track of all the `delta_weight` values from the last pass, and making the new value of each `delta_weight` basically fairly similar to the previous value – i.e. give it momentum or 'reluctance to change'.

Look back a few slides to 'Calculate the weight changes' where I put a purple arrow →

Substitute

$$\text{delta_wts}[k][j] = \text{eta} * \text{delta_outputs}[j] * \text{hidden}[k] + \alpha * \text{old_delta_wts}[k][j];$$

Value for momentum

EASy

Alpha is the momentum factor, between 0 and 1. Typical value to use is 0.9.

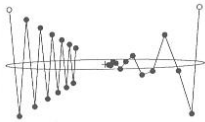


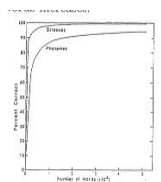
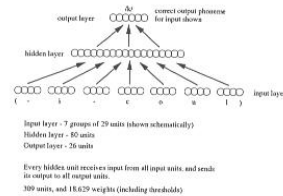
FIGURE 6.3 Gradient descent on the simple quadratic surface of Fig. 5.10. Both trajectories are for 12 steps with $\eta = 0.0476$, the best value in the absence of momentum. On the left there is no momentum ($\alpha = 0$), while $\alpha = 0.5$ on the right.

So common parameter setting rates (to be changed when appropriate) are: **Learning rate eta 0.02, momentum alpha 0.9**

Applications

EASy

You now have all the knowledge to write a back-prop program to tackle the most complex application – e.g. NETalk (Sejnowski and Rosenberg 1986)



Exercise for the reader – Seminars week 4

EASy

Construct an ANN with 2 input nodes, 2 hidden nodes and one output node.

Use binary inputs – so there are 4 possible input vectors.

Train it on the XOR rule so that for these

Inputs there are these Target outputs

00	0
01	1
10	1
11	0

Overfitting?

EASy

NB: with this particular problem, should you or should you not be worrying about overfitting?

See note in previous lecture on this.

Any Questions ?

EASy