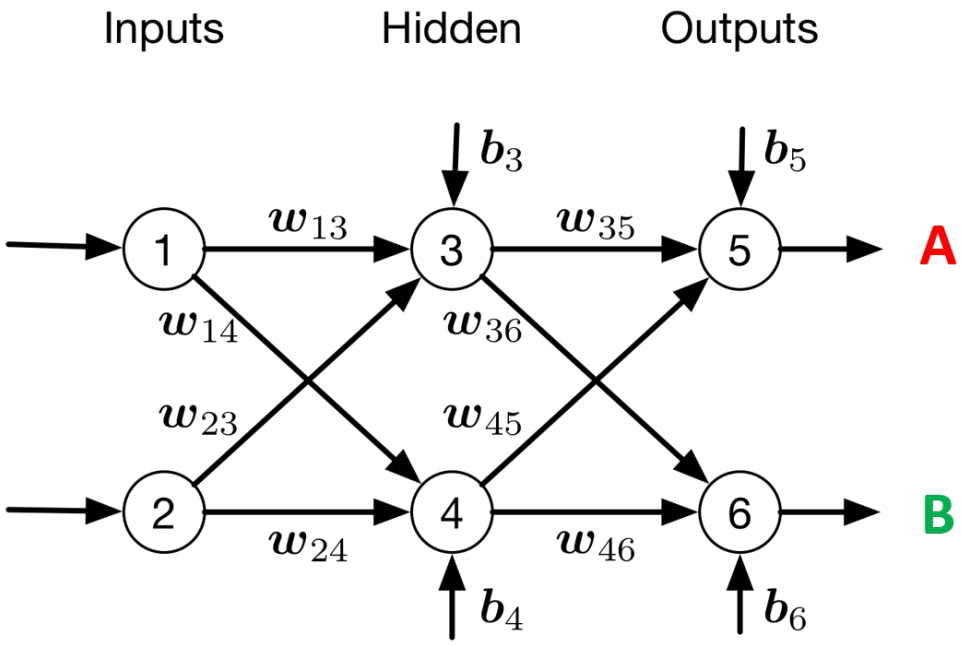
COMP307: Feedforward and Backprop for Neural Networks



In[ ]:=

Our Network

We have six nodes: two inputs, two hidden nodes (in one hidden layer), and two output nodes. This is a full connected network, where each node is connected to all nodes in the previous layer, and to all nodes in the next layer.
Each connection has a weight (w), and each node (except for the input nodes) has a bias (b).

Lets say that we have randomly initialised our weights and biases as follows:

In[•]:=
```
weights = Dataset[<|"w13" → 0.72, "w14" → -0.31,
    "w23" → 0.10, "w24" → -0.92, "w35" → -0.37, "w36" → 0.43, "w45" → -0.19,
    "w46" → 0.78, "b3" → 0.01, "b4" → 0.38, "b5" → -0.13, "b6" → 0.78|>]
```

Out[•]=

w_{13}	0.72
w_{14}	–0.31
w_{23}	0.1
w_{24}	–0.92
w_{35}	–0.37
w_{36}	0.43
w_{45}	–0.19
w_{46}	0.78
b_3	0.01
b_4	0.38
b_5	–0.13
b_6	0.78

We can then access our weights and biases like this:

In[•]:=
```
weights["w36"]
```
Out[•]=
```
0.43
```

In[•]:=
```
weights["b6"]
```
Out[•]=
```
0.78
```

Feedforward

Lets assume we have a single instance, with the two features (inputs) as follows:

In[•]:=
```
I1 = 0.90
```
Out[•]=
```
0.9
```

In[•]:=
```
NumberForm[0.9, 16]
```
Out[•]//NumberForm=
```
0.9
```

In[•]:=
```
I2 = -0.20
```
Out[•]=
```
-0.2
```

The output of an input node is simply its input. We do **not** apply the activation function to input nodes, as we want to “preserve” their true values for the first hidden layer. Hence:

In[•]:=
```
O1 = I1
```
Out[•]=
```
0.9
```

```
In[•]:= NumberForm[0.9, 16]
Out[•]//NumberForm=
0.9
```

```
In[•]:= O2 = I2
Out[•]= -0.2
```

We can now calculated the input to each of our two hidden nodes (3 and 4). We notate this as their 'z' value, which is equal to the weighted sum of the inputs, plus the node's bias:

$$z_j = b_j + \sum_i w_{ij} * O_i$$

```
In[•]:= z3 = weights["w13"] * O1 + weights["w23"] * O2 + weights["b3"]
Out[•]= 0.638
```

```
In[•]:= z4 = weights["w14"] * O1 + weights["w24"] * O2 + weights["b4"]
Out[•]= 0.285
```

We then apply our activation function to z to find the output of the two hidden nodes:

```
In[•]:= O3 = LogisticSigmoid[z3]
Out[•]= 0.654301
```

```
In[•]:= O4 = LogisticSigmoid[z4]
Out[•]= 0.570772
```

We can now repeat the process for the next layer of nodes (5 and 6), using the outputs of our previous hidden layer:

```
In[•]:= z5 = weights["w35"] * O3 + weights["w45"] * O4 + weights["b5"]
Out[•]= -0.480538
```

```
In[•]:= z6 = weights["w36"] * O3 + weights["w46"] * O4 + weights["b6"]
Out[•]= 1.50655
```

And again, we apply the sigmoid activation function to the weighted sum:

```
In[•]:= O5 = LogisticSigmoid[z5]
Out[•]= 0.382125
```

```
In[•]:= O6 = LogisticSigmoid[z6]
Out[•]= 0.81855
```

For a classification problem, we choose the class that has the highest corresponding output:

```
In[•]:= class = If[O5 > O6, "A", "B"]
Out[•]= B
```

We did it! You may like to play around with changing the weights and inputs, and re-evaluating to see how the outputs then change. How much do you need to change things to get class A instead of B?

Backpropagation

Now that we have calculated the outputs of our network, we can work backwards to update the weights and biases to be more accurate compared to the actual class labels.

Recall that we use the symbol β_j to represent the error term or the "beneficial factor" of node j.

Note that this is **not** the bias, it is the Greek letter beta, not the English letter 'b'! Our first step, is to calculate β_j for all of the non-input nodes in the network.

For our output nodes, we simply calculate β_j as the difference between the true/**desired** class label (d) and the actual output (O).

For this example, we will set d_5 and d_6 as follows :

```
In[•]:= d5 = 0
        d6 = 1
Out[•]= 0

Out[•]= 1
```

This is the same as saying that we want this instance to be class B, and not class A. In other words, the output of node 5 should be as small as possible (close to 0) and the output of node 6 should be as large as possible (close to 1).

We now calculate β_5 and β_6 :

```
In[•]:= β5 = d5 - O5
Out[•]= -0.382125

In[•]:= NumberForm[-0.382125, 16]
Out[•]//NumberForm=
        -0.3821250789718405

In[•]:= β6 = d6 - O6
Out[•]= 0.18145
```

Calculating β_j for the non-output nodes is more complicated. The equation from class is:

$$\beta_j = \sum_k w_{jk}\, O_k\, (1 - O_k)\, \beta_k$$

This is telling us to sum across each connection coming out of node j. Each of these connections will take a "turn" being the k. For example, Node 3 has two connections coming out of it, with weights w_{35} and w_{36} which go to nodes 5 and 6 respectively.

This is why we call it "backpropagation", as we take the error terms and outputs from layers further forward in the network, and then "propagate" their errors back to earlier layers.

Lets try it for node 3:

In[•]:= β_3 = (weights["w_{35}"] * O_5 * (1 – O_5) * β_5) + (weights["w_{36}"] * O_6 * (1 – O_6) * β_6)

Out[•]= 0.0449706

We have two *k* values (5 and 6), and so we are summing across two terms, which are shown in two sets of brackets.
For each k value, we multiply the current weight of that connection (e.g. w_{35}) with the *slope* of the output of node 5 ($O_5 * (1 - O_5)$) and with the error term of node 5 (β_5), to get an overall error term back-propagating through that connection.

Repeating for node 4:

In[•]:= β_4 = (weights["w_{45}"] * O_5 * (1 – O_5) * β_5) + (weights["w_{46}"] * O_6 * (1 – O_6) * β_6)

Out[•]= 0.0381633

We now repeat the process for the *next* hidden layer, working backwards. However, we only have one hidden layer in this network, so we are done! There is no need to calculate β_1 or β_2, as they are not needed when updating the weights. Note that if we did have another hidden layer, **we always use the OLD weights in our calculations**, as we do not change any of the weights until we have calculated all the changes.

We now have all our values of β. Based on these, we can now finally calculate the weight changes for each of the weights in our network, by using the equation:

$$\Delta w_{ij} = \eta O_i O_j \left(1 - O_j\right) \beta_j$$

When calculating the weight change, we want it to be proportional to the output of the node which the weight applies to, the slope of the activation function of the next node, and to the error term we have already calculated.
We also have a *learning rate* (η), which is a parameter that controls how aggressively our weights are updated, and so how quickly our network learns. There is a trade-off here - as we are doing gradient descent, we risk overshooting a good combination of weights with a high η, whereas with a low η, we will train very slowly.

To keep things simple here, we define:

In[•]:= η = 1

Out[•]= 1

Based on our weight change equation, we can calculate the change for each of our 8 weights (the order does not matter!). For the connections from the input to the hidden layer:

In[•]:= Δw_{13} = η * O_1 * O_3 * (1 – O_3) * β_3

Out[•]= 0.00915476

In[•]:= Δw_{14} = η * O_1 * O_4 * (1 – O_4) * β_4

Out[•]= 0.0084147

```
In[•]:= Δw23 = η * O2 * O3 * (1 - O3) * β3
Out[•]= -0.00203439

In[•]:= NumberForm[-0.00203439, 16]
Out[•]//NumberForm=
-0.002034391967164421

In[•]:= Δw24 = η * O2 * O4 * (1 - O4) * β4
Out[•]= -0.00186993
```

And then for the connections from the hidden to the output layer:

```
In[•]:= Δw35 = η * O3 * O5 * (1 - O5) * β5
Out[•]= -0.0590323

In[•]:= Δw36 = η * O3 * O6 * (1 - O6) * β6
Out[•]= 0.0176335

In[•]:= Δw45 = η * O4 * O5 * (1 - O5) * β5
Out[•]= -0.0514961

In[•]:= Δw46 = η * O4 * O6 * (1 - O6) * β6
Out[•]= 0.0153824
```

We also calculate the update for the biases in a similar manner. Note that our equation for updating biases only considers one output, since there is no “input” to the bias:

$$\triangle b_j = \eta O_j \left(1 - O_j\right) \beta_j$$

Our bias updates are:

```
In[•]:= Δb3 = η * O3 * (1 - O3) * β3
Out[•]= 0.010172

In[•]:= NumberForm[0.010172, 16]
Out[•]//NumberForm=
0.01017195983582211

In[•]:= Δb4 = η * O4 * (1 - O4) * β4
Out[•]= 0.00934967

In[•]:= Δb5 = η * O5 * (1 - O5) * β5
Out[•]= -0.0902218

In[•]:= Δb6 = η * O6 * (1 - O6) * β6
Out[•]= 0.0269501
```

Finally, we can go ahead and change our weights and biases of our network!:

In[ ]:=
```
nw13 = Δw13 + weights["w13"]
nw14 = Δw14 + weights["w14"]
nw23 = Δw23 + weights["w23"]
nw24 = Δw24 + weights["w24"]
nw35 = Δw35 + weights["w35"]
nw36 = Δw36 + weights["w36"]
nw45 = Δw45 + weights["w45"]
nw46 = Δw46 + weights["w46"]
nb3 = Δb3 + weights["b3"]
nb4 = Δb4 + weights["b4"]
nb5 = Δb5 + weights["b5"]
nb6 = Δb6 + weights["b6"]
```

Out[ ]= 0.729155

Out[ ]= −0.301585

Out[ ]= 0.0979656

Out[ ]= −0.92187

Out[ ]= −0.429032

Out[ ]= 0.447634

Out[ ]= −0.241496

Out[ ]= 0.795382

Out[ ]= 0.020172

Out[ ]= 0.38935

Out[ ]= −0.220222

Out[ ]= 0.80695

Further Analysis

Let’s compare this to our old weights:

In[•]:= **weights**

Out[•]=

w_{13}	0.72
w_{14}	–0.31
w_{23}	0.1
w_{24}	–0.92
w_{35}	–0.37
w_{36}	0.43
w_{45}	–0.19
w_{46}	0.78
b_3	0.01
b_4	0.38
b_5	–0.13
b_6	0.78

Our weights didn't change that much? There's a few reasons for this. Firstly, our classifier wasn't *that* wrong to start with. Secondly, this is only a single instance --- if we had 100 instances, then we'd add all 100 changes together.

We can see what output, we'd get with our new weights, though. All we have to do is do the forward pass again!

In[•]:= **nz_3 = nw_{13} * O_1 + nw_{23} * O_2 + nb_3**

Out[•]= 0.656818

In[•]:= **nz_4 = nw_{14} * O_1 + nw_{24} * O_2 + nb_4**

Out[•]= 0.302297

In[•]:= **nO_3 = LogisticSigmoid[nz_3]**

Out[•]= 0.658545

In[•]:= **ScientificForm[0.658545]**

Out[•]//ScientificForm=

6.58545×10^{-1}

In[•]:= **nO_4 = LogisticSigmoid[nz_4]**

Out[•]= 0.575004

In[•]:= **nz_5 = nw_{35} * nO_3 + nw_{45} * nO_4 + nb_5**

Out[•]= −0.64162

In[•]:= **nz_6 = nw_{36} * nO_3 + nw_{46} * nO_4 + nb_6**

Out[•]= 1.55909

In[•]:= nO_5 = LogisticSigmoid[nz_5]

Out[•]= 0.34488

In[•]:= nO_6 = LogisticSigmoid[nz_6]

Out[•]= 0.826222

So our output of node 5 changes from 0.382 to 0.345, and node 6 from 0.819 to 0.826. Node 5 had a bigger change as it was further from the desired result of '0', than Node 6 was from '1'.
If you repeated this process a bunch more times, you'd eventually approach outputs of 0 and 1 respectively!