$\hat{y} = f(W \cdot a) = \text{sigmoid}(0.1 \times 0.495) = 0.512.$

With the network output, we can start the backwards pass to calculate the weight updates for both layers. Using the fact that for the sigmoid function $f'(W \cdot a) = f(W \cdot a)(1 - f(W \cdot a))$, the error term for the output unit is

$$\delta^o = (y - \hat{y})f'(W \cdot a) = (1 - 0.512) \times 0.512 \times (1 - 0.512) = 0.122.$$

Now we need to calculate the error term for the hidden unit with backpropagation. Here we'll scale the error term from the output unit by the weight W connecting it to the hidden unit. For the hidden unit error term, $\delta^h_j = \sum_k W_{jk} \delta^o_k f'(h_j)$, but since we have one hidden unit and one output unit, this is much simpler.

$$\delta^h = W \delta^o f'(h) = 0.1 \times 0.122 \times 0.495 \times (1 - 0.495) = 0.003$$

Now that we have the errors, we can calculate the gradient descent steps. The hidden to output weight step is the learning rate, times the output unit error, times the hidden unit activation value.

$$\Delta W = \eta \delta^o a = 0.5 \times 0.122 \times 0.495 = 0.0302$$

Then, for the input to hidden weights w_i, it's the learning rate times the hidden unit error, times the input values.

$$\Delta w_i = \eta \delta^h x_i = (0.5 \times 0.003 \times 0.1, 0.5 \times 0.003 \times 0.3) = (0.00015, 0.00045)$$

From this example, you can see one of the effects of using the sigmoid function for the activations. The maximum derivative of the sigmoid function is 0.25, so the errors in the output layer get reduced by at least 75%, and errors in the hidden layer are scaled down by at least 93.75%! You can see that if you have a lot of layers, using a sigmoid activation function will quickly reduce the weight steps to tiny values in layers near the input. This is known as the **vanishing gradient** problem. Later in the course you'll learn about other activation functions that perform better in this regard and are more commonly used in modern network architectures.

Implementing in NumPy