where h is the input to the output unit,

$h = \sum_i w_i x_i$

Implementing with NumPy

For the most part, this is pretty straightforward with NumPy.

First, you'll need to initialize the weights. We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is $1/\sqrt{n}$ where n is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

```
weights = np.random.normal(scale=1/n_features**.5, size=n_features)
```

NumPy provides a function `np.dot()` that calculates the dot product of two arrays, which conveniently calculates h for us. The dot product multiplies two arrays element-wise, the first element in array 1 is multiplied by the first element in array 2, and so on. Then, each product is summed.

```
# input to the output layer
output_in = np.dot(weights, inputs)
```

And finally, we can update Δw_i and w_i by incrementing them with `weights += ...` which is shorthand for `weights = weights + ...`.

Efficiency tip!

You can save some calculations since we're using a sigmoid here. For the sigmoid function, $f'(h) = f(h)(1 - f(h))$. That means that once you calculate $f(h)$, the activation of the output unit, you can use it to calculate the gradient for the error gradient.