Applying Sigmoid Function

To get the signals emerging from the hidden node, we simply apply the sigmoid squashing function to each of these emerging signals.

Here's the formula of Sigmoid function:

$$O_{hidden} = sigmoid(X_{hidden})$$

That should be easy, especially if the sigmoid function is already defined in a handy Python library. It turns out it is! The *scipy* Python library has a set of special functions, and the sigmoid function is called `expit()`. Don't ask me why it has such a silly name. This *scipy* library is imported just like we imported the *numpy* library:

```
# scipy.special for the sigmoid function expit()
import scipy.special
```

Because we might want to experiment and tweak, or even completely change, the activation function, it makes sense to define it only once inside the neural network object when it is first initialized. After that, we can refer to it several times, such as in the `query()` function. This arrangement means we only need to change this definition once, and not have to locate and change the code anywhere an activation function is used.

The following defines the activation function we want to use in the neural network's initialization section.

```
# activation function is the sigmoid function
self.activation_function = lambda x: scipy.special.expit(x)
```

What is this code? It doesn't look like anything we've seen before. What is that *lambda*? Well, this may look daunting, but it really isn't. All we've done here is created a function like any other, but we've used a shorter way of writing it out. Instead of the usual `def()` definitions, we use the magic *lambda* to create

a function there and then, quickly and easily. The function here takes *x* and

returns `scipy.special.expit(x)` which is the sigmoid function. Functions created with lambda are nameless, or anonymous as seasoned coders like to call them, but here we've assigned it to the name `self.activation_function()`. All this means is that whenever someone needs to use the activation function, all they need to do is call `self.activation_function()`.

So, going back to the task at hand, we want to apply the activation function to the combined and moderated signals into the hidden nodes. The code is as simple as the following:

```
# calculate the signals emerging from hidden layer
hidden_outputs = self.activation_function(hidden_inputs)
```

That is, the signals emerging from the hidden layer nodes are in the matrix called *hidden_outputs*.

That got us as far as the middle hidden layer, what about the final output layer? Well, there isn't anything really different between the hidden and final output layer nodes, so the process is the same. This means the code is also very similar.

Have a look at the following code which summarises how we calculate not just the hidden layer signals but also the output layer signals too.

```
# calculate signals into hidden layer
hidden_inputs = numpy.dot(self.wih, inputs)
# calculate the signals emerging from hidden layer
hidden_outputs = self.activation_function(hidden_inputs)

# calculate signals into final output layer
final_inputs = numpy.dot(self.who, hidden_outputs)
# calculate the signals emerging from final output layer
final_outputs = self.activation_function(final_inputs)
```

If we took away the comments, there are just four lines of code shown in bold that do all the calculations we needed, two for the hidden layer and two for the final output layer.