Here's how I implemented the sigmoid function.

```
class Sigmoid(Node):
    def __init__(self, node):
        Node.__init__(self, [node])

    def _sigmoid(self, x):
        """
        This method is separate from `forward` because it
        will be used with `backward` as well.

        `x`: A numpy array-like object.
        """
        return 1. / (1. + np.exp(-x)) # the `.` ensures that `1` is a float

    def forward(self):
        input_value = self.inbound_nodes[0].value
        self.value = self._sigmoid(input_value)
```

It may have seemed strange that `_sigmoid` was a separate method. As seen in the derivative of the sigmoid function, Equation (4), the sigmoid function is actually *a part of its own derivative*. Keeping `_sigmoid`separate means you won't have to implement it twice for forward and backward propagations.

This is exciting! At this point, you have used weights and biases to compute outputs. And you've used an activation function to categorize the output. As you may recall, neural networks improve the **accuracy** of their outputs by modifying weights and biases in response to training against labeled datasets.

There are many techniques for defining the accuracy of a neural network, all of which center on the network's ability to produce values that come as close as possible to known correct values. People use different names for this accuracy measurement, often terming it **loss** or **cost**. I'll use the term *cost* most often.

For this lab, you will calculate the cost using the mean squared error (MSE). It looks like so:

$$C(w,b) = \frac{1}{m} \sum_x ||y(x) - a||^2$$

Equation (5)

Here *w* denotes the collection of all weights in the network, *b* all the biases, *m* is the total number of training examples, *a* is the approximation of *y(x)* by the network, both *a* and *y(x)* are vectors of the same length.

The collection of weights is all the weight matrices flattened into vectors and concatenated to one big vector. The same goes for the collection of biases except they're already vectors so there's no need to flatten them prior to the concatenation.

Here's an example of creating *w* in code:

```
# 2 by 2 matrices
w1  = np.array([[1, 2], [3, 4]])
w2  = np.array([[5, 6], [7, 8]])

# flatten
w1_flat = np.reshape(w1, -1)
w2_flat = np.reshape(w2, -1)

w = np.concatenate((w1_flat, w2_flat))
# array([1, 2, 3, 4, 5, 6, 7, 8])
```

It's a nice way to abstract all the weights and biases used in the neural network and makes some things easier to write as we'll see soon in the upcoming gradient descent sections.

NOTE: It's not required you do this in your code! It's just easier to do this talk about the weights and biases as a collective than consider them invidually.

The cost, *C*, depends on the difference between the correct output, *y(x)*, and the network's output, *a*. It's easy to see that no difference between *y(x)* and *a* (for all values of *x*) leads to a cost of 0.

This is the ideal situation, and in fact the learning process revolves around minimizing the cost as much as possible.

I want you to calculate the cost now.

You implemented this network in the forward direction in the last quiz.

As it stands right now, it outputs gibberish. The activation of the sigmoid node means nothing because the network has no labeled output against which to compare. Furthermore, the weights and bias cannot change and learning cannot happen without a cost.

Instructions

For this quiz, you will run the forward pass against the network in nn.py. I want you to finish implementing the `MSE` method so that it calculates the cost from the equation above.

I recommend using the `np.square` (**documentation**) method to make your life easier.

1. Check out nn.py to see how `MSE` will calculate the cost.
2. Open miniflow.py. Finish building `MSE`.
3. Test your network! See if the cost makes sense given the inputs by playing with nn.py.