Here's how I implemented MSE:

```
class MSE(Node):
    def __init__(self, y, a):
        """
        The mean squared error cost function.
        Should be used as the last node for a network.
        """
        # Call the base class' constructor.
        Node.__init__(self, [y, a])

    def forward(self):
        """
        Calculates the mean squared error.
        """
        # NOTE: We reshape these to avoid possible matrix/vector broadcast
        # errors.
        #
        # For example, if we subtract an array of shape (3,) from an array
        # (3,1) we get an array of shape(3,3) as the result when we want
        # an array of shape (3,1) instead.
        #
        # Making both arrays (3,1) insures the result is (3,1) and does
        # an elementwise subtraction as expected.
        y = self.inbound_nodes[0].value.reshape(-1, 1)
        a = self.inbound_nodes[1].value.reshape(-1, 1)
        m = self.inbound_nodes[0].value.shape[0]

        diff = y - a
        self.value = np.mean(diff**2)
```

The math behind MSE reflects Equation (5), where *y* is target output and *a* is output computed by the neural network. We then square the difference `diff**2`, alternatively, this could be `np.square(diff)`. Lastly we need to sum the squared differences and divide by the total number of examples *m*. This can be achieved in with `np.mean` or `(1 /m) * np.sum(diff**2)`.

Note the order of `y` and `a` doesn't actually matter, we could switch them around (`a - y`) and get the same value.

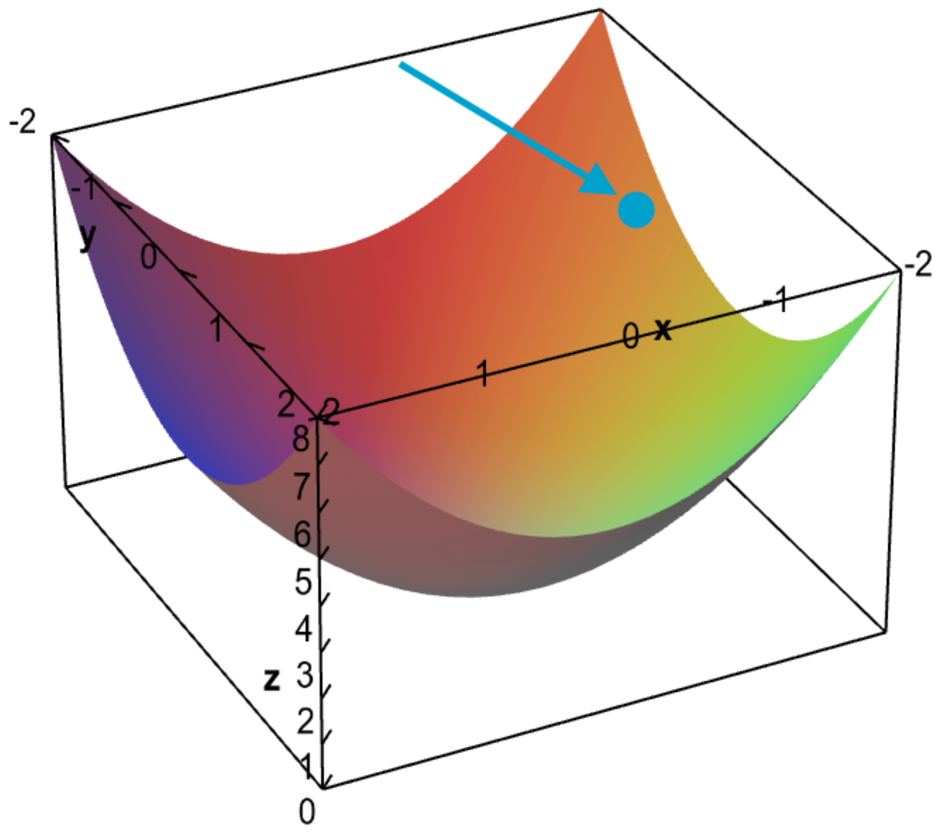
Backpropagation

Great! We've successfully calculated a full forward pass and found the cost. Next we need to start a backwards pass, which starts with backpropagation. Backpropagation is the process by which the network runs error values backwards.

During this process, the network calculates the way in which the weights need to change (also called the gradient) to reduce the overall error of the network. Changing the weights usually occurs through a technique called gradient descent.

Making sense of the purpose of backpropagation comes more easily after you work through the intended outcome. I'll come back to backpropagation in a bit, but first, I want to dive deeper into gradient descent.

Gradient Descent



A point on a three dimension surface.

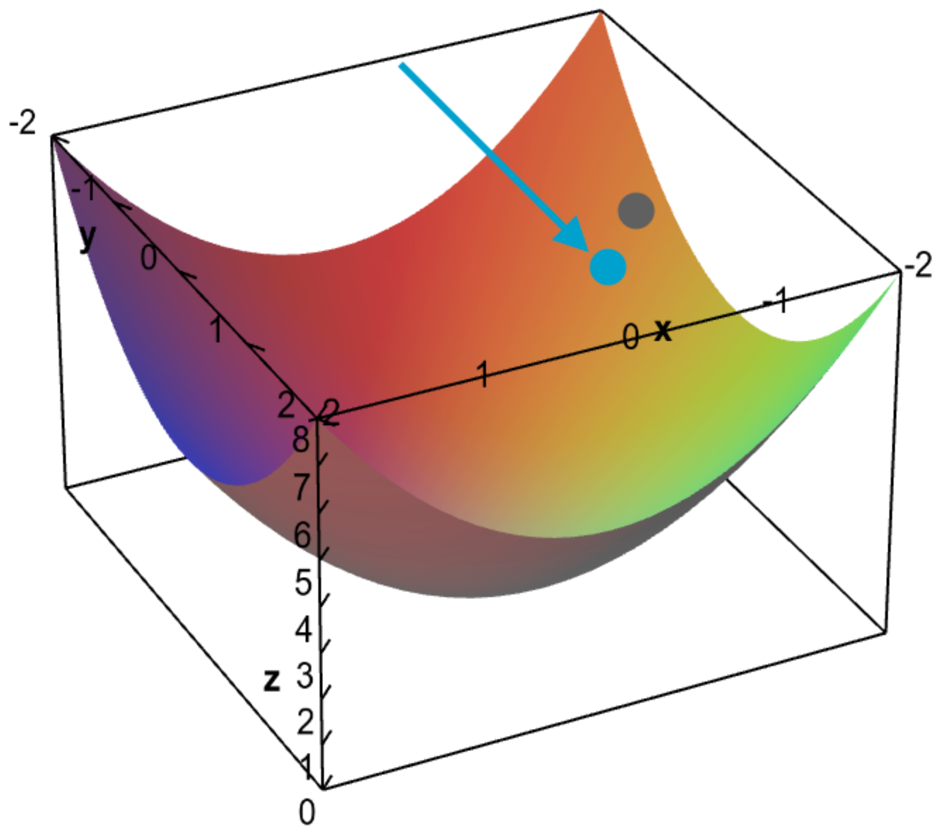
Imagine a point on a surface in three dimensional space. In real-life, a ball sitting on the slope of a valley makes a nice analogy. In this case, the height of the point represents the difference between the current output of the network and the correct output given the current parameter values (hence why you need data with known outputs). Each dimension of the plane represents another parameter to the network. A network with *m* parameters would be a hypersurface of *m* dimensions.

(Imagining more than three dimensions **is tricky**. The good news is that the ball and valley example describes the behavior of gradient descent well, the only difference between three dimensional and *n* dimensional situations being the number of parameters in the calculations.)

In the ideal situation, the ball rests at the bottom of the valley, indicating the minimum difference between the output of the network and the known correct output.

The learning process starts with random weights and biases. In the ball analogy, the ball starts at a random point near the valley.

Gradient descent works by first calculating the slope of the plane at the current point, which includes calculating the partial derivatives of the loss with respect to all of the parameters. This set of partial derivatives is called the **gradient**. Then it uses the gradient to modify the weights such that the next forward pass through the network moves the output lower in the hypersurface. Physically, this would be the same as measuring the slope of the valley at the location of the ball, and then moving the ball a small amount in the direction of the slope. Over time, it's possible to find the bottom of the valley with many small movements.



It moved a little bit!

While gradient descent works remarkably well, the technique isn't **guaranteed** to find the absolute minimum difference between the network's output and the known output. Why?