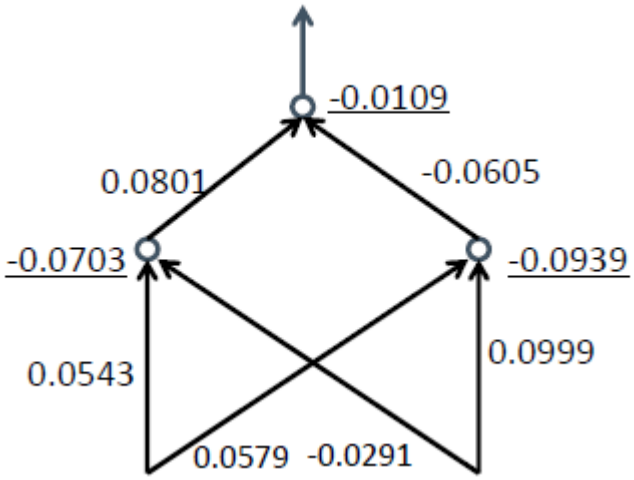
Report - Analysis of the neural network

> Introduction to the assignment: Use a neural network to accomplish an XOR problem with a learning rate of 0.6 and a loss less than 0.008. The network should have 2 input neurons, 2 hidden neurons and 1 output neuron with the activation function to be sigmoid function.

In this assignment, I got a result of less than 1,000 epochs training the network with losses lower than 0.008. Now, let's dive into details.

How neural networks work



When the inputs are given to the network, some neurons are activated and that would pave an unique path to the output. In our scenario, the activated input neurons send signals to every connected hidden neurons in the next layer. This directly affects which neurons are activated in the hidden layer. And because of the activation function, the network can convert a linear function to unlinear function. After the information flows through all the neurons, it will output a result showing the possible number of this network.

Each neuron in the network transform data using a series of computations: a neuron multiplies an initial value by some weigh, sums results with other values comning into the same neuron, adjusts the resulting number by the neuron's bias and then normalizes the output with an sigmoid activation function. The **bias** is a neuron-specific number that adjusts the neuron's value once all the connections are processed, and the **activation function** ensures values that are passed on lie within a tunable, expected range. This process is repeated until the final output layer can provide *scores* or *predictions* related to the classification task at hand.

The reason why neural network can work so well mainly contribute to the **backpropagation** method. Technically it calculates the gradient of the loss function. It is commonly used in the gradient descent optimization algorithm. It is also called **backward propagation of errors**, because the error is calculated at the output and distributed back through the network layers. As the network is training, the parameters (weights and biases) can be adjusted step by step with backprapagation.

How to build the network in program

To implement the program, we need a Python library called Numpy, for doing fast algebra. Let me explain the core features of the neural networks code, below. The centerpiece is a `Network` class, which we use to represent a neural network. Here's the code we use to initialize a `Network` object:

```
class Network(object):

    def init(self, sizes):
        """The list "sizes" contains the number of neurons in respective layers."""
        self.sizes = sizes
        self.number_layers = len(sizes)
        self.biases = [np.random.randn(i, 1) for i in sizes[1:]]
        self.weights = [np.random.randn(i, j) for i,j in zip(sizes[1:],sizes[:-1])]
```

In this code, the list `sizes` contains the number of neurons in the respective layers. The biases and weights in the `Network` object are all initialized randomly, using the Numpy `np.random.randn` function to generate Gaussian distributions with mean 00 and standard deviation 11. This random initialization gives our gradient descent algorithm a place to start from.

With all this in mind, it's easy to write code computing the output from a `Network` instance. We begin by defining the sigmoid function and the derivative function of the sigmoid:

```
# Activation Function
def sigmoid(a):
    """Sigmoid function"""
    return 1 / (1 + np.exp(-a))

def sigmoid_deriv(a):
    """Derivative of the sigmoid function"""
    return sigmoid(a) * (1 - sigmoid(a))
```

We then add a `feedforward` method to the `Network` class, which, given an input `input_` for the network, returns the corresponding activation output.

```
def feedforward(self, input_):
    """Return the output of the neural network."""
    activation = input_
    for bias, weight in zip(self.biases, self.weights):
        output = np.dot(weight, activation) + bias
        activation = sigmoid(output)
    return activation
```

Of course, the main thing we want our `Network` objects to do is to learn. To that end we'll give them an `SGD` method which implements stochastic gradient descent.

```
def backprop(self, training_data, desired_output, lr_rate):
    """Update parameters"""
    # Store gradients of weights and biases for update
    grad_weights = np.zeros_like(self.weights)
    grad_biases = np.zeros_like(self.biases)

    # Store outputs and activations for backprop
    outputs = []
    activation = training_data
    activations = [activation]
    for b, w in zip(self.biases, self.weights):
    output = np.dot(w, activation) + b
    outputs.append(output)
    activation = sigmoid(output)
    activations.append(activation)

    # Compute the gradients of the last layer
    error = self.feedforward(training_data) - desired_output
    delta = error * sigmoid_deriv(outputs[-1])
    grad_biases[-1] = delta
    grad_weights[-1] = np.dot(delta, activations[-2].transpose())

    # Compute gradients of remaining layers
    for layer in range(2, self.number_layers):
    delta = np.dot(self.weights[-layer+1].transpose(), delta) * sigmoid_deriv(outputs[-
layer])
    grad_biases[-layer] = delta
    grad_weights[-layer] = np.dot(delta, activations[-layer-1].transpose())

    # Update weights and biases
    self.weights = [w-lr_rate*grad_w for w, grad_w in zip(self.weights, grad_weights)]
    self.biases = [b-lr_rate*grad_b for b, grad_b in zip(self.biases, grad_biases)]
```

This invokes something called the *backpropagation* algorithm, which is a fast way of computing the gradient of the cost function. So it works simply by computing these gradients for every training example in the , and then updating `self.weights` and `self.biases` appropriately. The `backprop` will update the weights and biases stored in lists.

Then I set the hyperparameters: training epoches and learning rate to be 50000 and 0.6 in this unique neural network structure.

After loading the data, we'll set up a `Network` with one hidden layer:

```
network = Network([2,2,1])
```

Finally, we will use gradient descent to learn from the data over 50000 epochs. And after training, we will get losses lower than 0.008.

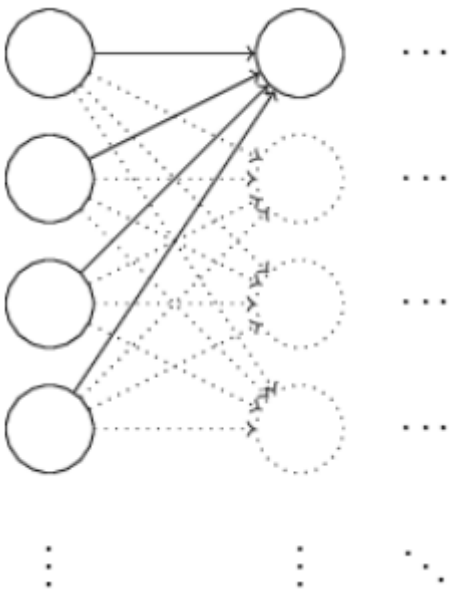
Different methods - Weight Initialization

I modified this part of code:

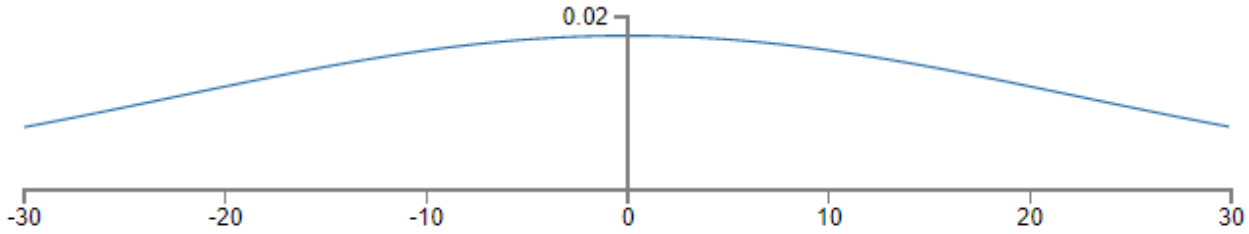
```
def __init__(self, sizes):
    """The list "sizes" contains the number of neurons in respective layers."""
    self.sizes = sizes
    self.number_layers = len(sizes)
    self.biases = [np.random.rand(i, 1) for i in sizes[1:]]
    self.weights = [np.random.rand(i, j) / j for i,j in zip(sizes[1:],sizes[:-1])] # Modify
this line to normalized Gaussian
```

When we create our neural networks, we have to make choices for the initial weights and biases. In the original codes, I just used the Gaussian normalization. It turns out that we can do quite a better than initializing with normalized Gaussian.

To see why, suppose we're working with a network with a large number - say $1,000$ - of input neurons. For now I'm going to concentrate specifically on the weights connecting the input neurons to the first neuron in the hidden layer, and ignore the rest of the network.



We'll suppose for simplicity that we're trying to train using a training input x in which half the input neurons are set to 1, and half the input neurons are set to 0. Let's consider the weighted sum $z = \sum w_j x_j + b$ of inputs to our hidden neuron. 500 terms in this sum vanish, because the corresponding input x_j is zero. And so z is a sum over a total of 501 normalized Gaussian random variables, accounting for the 500 weight terms and the 1 extra bias term. Thus z is itself distributed as a Gaussian with mean zero and standard deviation $\sqrt{501} \approx 22.4501 \approx 22.4$. That is, z has a very broad Gaussian distribution, not sharply peaked at all:



If that's the case then the output $\sigma(z)$ from the hidden neuron will be very close to either 1 or 0. That means our hidden neuron will have saturated. And when that happens, as we know, making small changes in the weights will make only absolutely miniscule changes in the activation of our hidden neuron. That miniscule change in the activation of the hidden neuron will, in turn, barely affect the rest of the neurons in the network at all, and we'll see a correspondingly miniscule change in the cost function. As a result, those weights will only learn very slowly when we use the gradient descent algorithm.

Suppose we have a neuron with n_{in} input weights. Then we shall initialize those weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{in}}$. That is, we'll squash the Gaussians down, making it less likely that our neuron will saturate.

The reason why the result with the normalized Gaussian method is not so improved is that the number of our neurons is too few.

Different methods - Cost Function

In this part, I only changed the codes by one line:

```
# Compute the gradients of the last layer
delta =  self.feedforward(training_data) - desired_output # I Modified this line
grad_biases[-1] = delta
grad_weights[-1] = np.dot(delta, activations[-2].transpose())
```

Ideally, we hope and expect that our neural networks will learn fast from their errors. Recall that we're using the quadratic cost function, which is given by

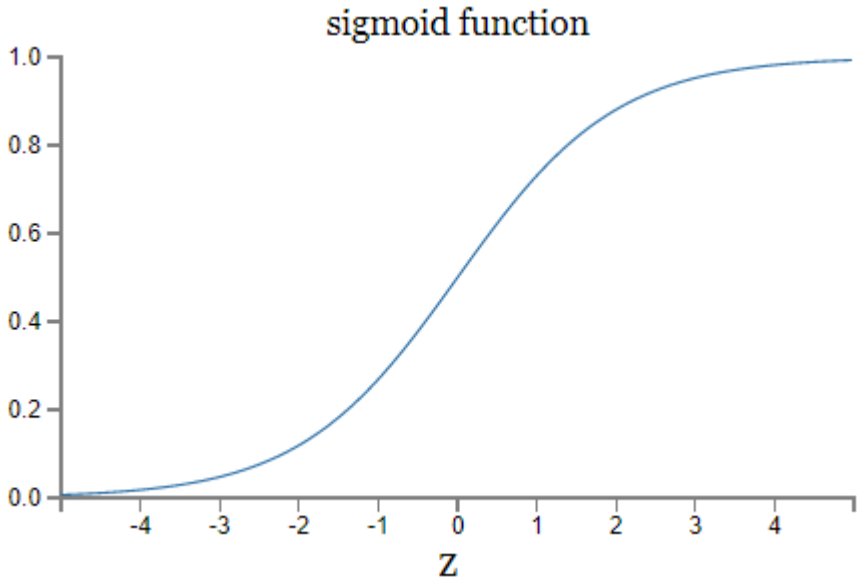
$$C = \frac{(y-a)^2}{2} \tag{1}$$

where a is the neuron's output and y is the corresponding desired output. To write this more explicitly in terms of the weight and bias, recall that $a = \sigma(z)$, where $z = wx + b$. Using the chain rule to differentiate with respect to the weight and bias we get

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x \tag{2}$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) \tag{3}$$

Recall the shape of the σ function.



We can see from this graph that when the neuron's output is close to 11, the curve gets very flat, and so $\sigma'(z)$ gets very small. This is the origin reason learning slowdown.

It turns out that we can solve the problem by replacing the quadratic cost function with a different cost function, known as the cross-entropy. We define the cross-entropy cost function for this scenario by

$$C = -\sum_x [y \ln a + (1-y) \ln(1-a)] \tag{4}$$

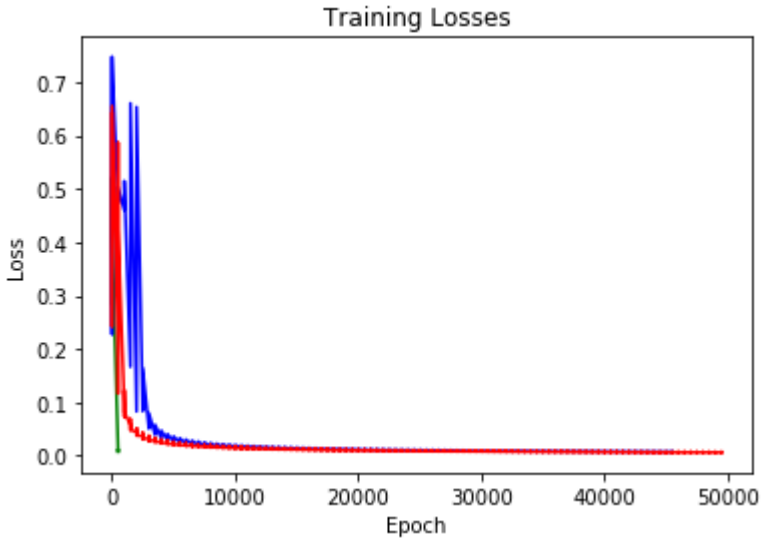
where a is the output of a neuron and y is the desired output. At last, when we compute the partial derivative of w we will obtain

$$\frac{\partial C}{\partial w_j} = \sum_x x(\sigma(z) - y). \tag{5}$$

It tells us that the rate at which the weight learns is controlled by $\sigma(z) - y$, i.e., by the error in the output. The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect. In particular, it avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation (2). When we use the cross-entropy, the $\sigma'(z)$ term gets canceled out, and we no longer need worry about it being small. This cancellation is the special miracle ensured by the cross-entropy cost function.

Now let's return to the codes and change the cost function to cross-entropy. Unsurprisingly, the neuron learns perfectly well in this instance, just as it did earlier. This time, the neuron learned quickly and it only cost less than $1,000$ epochs to train the network!

I have plotted all the loss from the three different methods. It is obvious that cross-entropy can get a more satisfied result.



Conclusion

In neural networks there are many hyperparameters need to be tuned. Though there are many methods to improve the outcomes we still don't know much of the mechanism.

In many parts of science - especially those parts that deal with simple phenomena - it's possible to obtain very solid, very reliable evidence for quite general hypotheses. But in neural networks there are large numbers of parameters and hyper-parameters, and extremely complex interactions between them. In such extraordinarily complex systems it's exceedingly difficult to establish reliable general statements. But in fact, we need such heuristics to inspire and guide our thinking. It's like the great age of exploration: the early explorers sometimes explored (and made new discoveries) on the basis of beliefs which were wrong in important ways.

In one word, we have a long way to go.