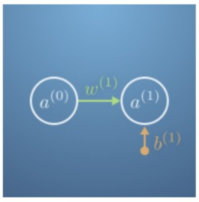
Congratulations! You passed!

Next Item

1 / 1 point

1. In this exercise we'll look in more detail about back-propagation, using the chain rule, in order to train our neural networks.

Let's look again at our two-node network.



Recall the activation equations are,

$a^{(1)} = \sigma(z^{(1)})$

$z^{(1)} = w^{(1)}a^{(0)} + b^{(1)}$.

Where we've introduced $z^{(1)}$ as the weighted sum of activation and bias.

We can formalise how good (or bad) our neural network is at getting the desired behaviour. For a particular input, x, and desired output y, we can define the *cost* of that specific *training example* as the square of the difference between the network's output and the desired output, that is,

$C_k = (a^{(1)} - y)^2$

Where k labels the training example and $a^{(1)}$ is assumed to be the activation of the output neuron when the input neuron $a^{(0)}$ is set to x

We'll go into detail about how to apply this to an entire set of training data later on. But for now, let's look at our toy example.

Recall our *NOT function* example from the previous quiz. For the input $x = 1$ we would like that the network outputs $y = 0$. For the starting weight and bias $w^{(1)} = 1.3$ and $b^{(1)} = -0.1$, the network actually outputs $a^{(1)} = 0.834$. If we work out the cost function for this example, we get

$C_1 = (0.834 - 0)^2 = 0.696$.

Do the same calculation for an input $x = 0$ and desired output $y = 1$. Use the code block to help you.

```
# First we set the state of the network
σ = np.tanh
w1 = 1.3
b1 = -0.1

# Then we define the neuron activation.
def a1(a0) :
  z = w1 * a0 + b1
  return σ(z)

# Experiment with different values of x below.
x = 0
print((a1(x)-1)**2)

```

Run

Reset

What is C_0 in this particular case? Give your result to 1 decimal place.

1.2

Correct Response
You have calculated $C_0 = (-0.100 - 1)^2 = 1.2$.

---

1 / 1 point

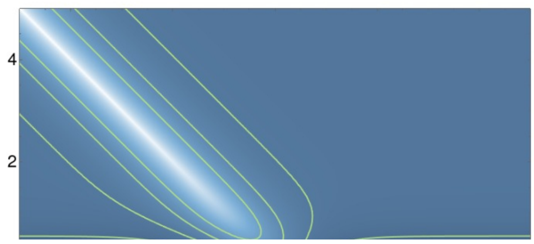
2. The cost function of a training set is the average of the individual cost functions of the data in the training set,

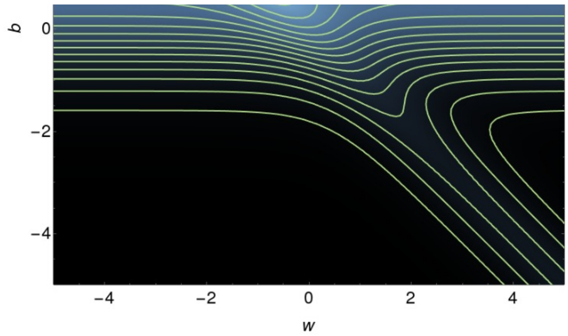
$C = \frac{1}{N}\sum_k C_k$,

where N is the number of examples in the training set.

For the NOT function we've been considering, where we have two examples in our training set, $(x = 0, y = 1)$ and $(x = 1, y = 0)$, the training set cost function is $C = \frac{1}{2}(C_0 + C_1)$.

Since our parameter space is 2D, ($w^{(1)}$ and $b^{(1)}$), we can draw the total cost function for this neural network as a contour map.





Here white represents low costs and black represents high costs.

Which of the following statements are true?

- [x] The optimal configuration lies somewhere along the line $b = -w$.

Correct
In this example the system asymptotically approaches a minimum along that line.

- [ ] The optimal configuration lies along the line $b = 0$.

Un-selected is correct

- [ ] None of the other statements are true.

Un-selected is correct

- [x] Descending perpendicular to the contours will improve the performance of the network.

Correct
Moving across the contours will get you closer to the minimum valley.

- [ ] There are many different local minima in this system.

Un-selected is correct

✔ 1 / 1 point

3. To improve the performance of the neural network on the training data, we can vary the weight and bias. We can calculate the derivative of the example cost with respect to these quantities using the chain rule.

$$\frac{\partial C_k}{\partial w^{(1)}} = \frac{\partial C_k}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial w^{(1)}}$$

$$\frac{\partial C_k}{\partial b^{(1)}} = \frac{\partial C_k}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial b^{(1)}}$$

.

Individually, these derivatives take fairly simple form. Go ahead and calculate them. We'll repeat the defining equations for convenience,

$a^{(1)} = \sigma(z^{(1)})$

$z^{(1)} = w^{(1)} a^{(0)} + b^{(n)}$

$C_k = (a^{(1)} - y)^2$

Select all true statements below.

- [ ] $$\frac{\partial z^{(1)}}{\partial b^{(1)}} = a^{(1)}$$

Un-selected is correct

- [x] $$\frac{\partial a^{(1)}}{\partial z^{(1)}} = \sigma'(z^{(1)})$$

Correct
The derivative of the activation $a^{(1)}$ with respect to the weighted sum $z^{(1)}$ is just the derivative of the sigmoid function, applied to the weighted sum.

- [ ] None of the other statements.

Un-selected is correct

- [ ] $$\frac{\partial z^{(1)}}{\partial w^{(1)}} = w^{(1)}$$

Un-selected is correct

- [x] $$\frac{\partial z^{(1)}}{\partial w^{(1)}} = a^{(0)}$$

Correct
Since $z^{(1)} = w^{(1)}a^{(0)} + b^{(1)}$ is a linear function, differentiating with respect to $w^{(1)}$ returns the coefficient $a^{(0)}$.

- [x] $$\frac{\partial z^{(1)}}{\partial b^{(1)}} = 1$$

Correct
The weighted sum changes exactly with the bias, if the bias is increased by some amount, then the weighted sum will increase by the same amount.

- [x] $$\frac{\partial C_k}{\partial a^{(1)}} = 2(a^{(1)} - y)$$

Correct
This is an application of the power rule and the chain rule.

- [ ] $$\frac{\partial C_k}{\partial a^{(1)}} = (1 - y)^2$$

Un-selected is correct

- [ ] $$\frac{\partial a^{(1)}}{\partial z^{(1)}} = \sigma$$

Un-selected is correct

---

1 / 1 point

4. Using your answer to the previous question, let's see it implemented in code.

The following code block has an example implementation of $\frac{\partial C_k}{\partial w^{(1)}}$. It is up to you to implement $\frac{\partial C_k}{\partial b^{(1)}}$.

Don't worry if you don't know exactly how the code works. It's more important that you get a feel for what is going on.

We will introduce the following derivative in the code,

$\frac{\mathrm{d}}{\mathrm{d}z} \tanh(z) = \frac{1}{\cosh^2 z}$.

Complete the function 'dCdb' below. Replace the ??? towards the bottom, with the expression you calculated in the previous question.

```
# First define our sigma function.
sigma = np.tanh

# Next define the feed-forward equation.
def a1 (w1, b1, a0) :
  z = w1 * a0 + b1
  return sigma(z)

# The individual cost function is the square of the difference between
# the network output and the training data output.
def C (w1, b1, x, y) :
  return (a1(w1, b1, x) - y)**2

# This function returns the derivative of the cost function with
# respect to the weight.
def dCdw (w1, b1, x, y) :
  z = w1 * x + b1
  dCda = 2 * (a1(w1, b1, x) - y) # Derivative of cost with activation
  dadz = 1/np.cosh(z)**2 # derivative of activation with weighted sum z
  dzdw = x # derivative of weighted sum z with weight
  return dCda * dadz * dzdw # Return the chain rule product.

# This function returns the derivative of the cost function with
# respect to the bias.
# It is very similar to the previous function.
# You should complete this function.
def dCdb (w1, b1, x, y) :
  z = w1 * x + b1
  dCda = 2 * (a1(w1, b1, x) - y)
  dadz = 1/np.cosh(z)**2
  """ Change the next line to give the derivative of
      the weighted sum, z, with respect to the bias, b. """
  dzdb = 1
  return dCda * dadz * dzdb

"""Test your code before submission:"""
# Let's start with an unfit weight and bias.
w1 = 2.3
b1 = -1.2
# We can test on a single data point pair of x and y.
x = 0
y = 1
# Output how the cost would change
# in proportion to a small change in the bias
print( dCdb(w1, b1, x, y) )

```

Run

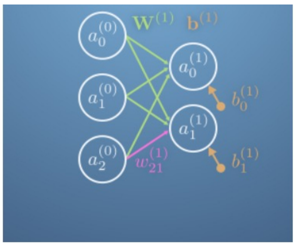
Reset

Correct Response
Well done. Feel free to examine your code to get a feel for what it is doing.

---

1 / 1 point

5. Recall that when we add more neurons to the network, our quantities are upgraded to vectors or matrices.



$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)})$,

$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}$

The individual cost functions remain scalars. Instead of becoming vectors, the components are summed over each output neuron.

$C_k = \sum_i (a_i^{(1)} - y_i)^2$

Note here that i labels the output neuron and is summed over, whereas k labels the training example.

The training data becomes a vector too,

$x \rightarrow \mathbf{x}$ and has the same number of elements as input neurons.

$y \rightarrow \mathbf{y}$ and has the same number of elements as output neurons.

This allows us to write the cost function in vector form using the modulus squared,

$C_k = |\mathbf{a}^{(1)} - \mathbf{y}|^2$.

Use the code block below to play with calculating the cost function for this network.

```
# Define the activation function.
sigma = np.tanh

# Let's use a random initial weight and bias.
W = np.array([[-0.94529712, -0.2667356 , -0.91219181],
              [ 2.05529992,  1.21797092,  0.22914497]])
b = np.array([ 0.61273249,  1.6422662 ])

# define our feed forward function
def a1 (a0) :
  # Notice the next line is almost the same as previously,
  # except we are using matrix multiplication rather than scalar multiplication
  # hence the '@' operator, and not the '*' operator.
  z = W @ a0 + b
  # Everything else is the same though,
  return sigma(z)

# Next, if a training example is,
x = np.array([0.7, 0.6, 0.2])
y = np.array([0.9, 0.6])

# Then the cost function is,
d = a1(x) - y # Vector difference between observed and expected activation
C = d @ d # Absolute value squared of the difference.
```

Run

Reset

For the initial weights and biases, what is the example cost function, C_k, when,

$x = \begin{bmatrix} 0.7 \\ 0.6 \\ 0.2 \end{bmatrix}$ and $y = \begin{bmatrix} 0.9 \\ 0.6 \end{bmatrix}$?

Give your answer to 1 decimal place.

1.8

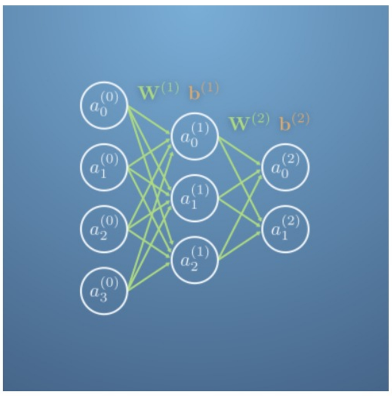
Correct Response

Well done. Feel free to continue to use the code block and experiment with varying other parameters.

✔

1 / 1 point

6. Let's now consider a neural network with hidden layers.



Training this network is done by *back-propagation* because we start at the output layer and calculate derivatives backwards towards the input layer with the chain rule.

Let's see how this works.

If we wanted to calculate the derivative of the cost with respect to the weight or bias of the final layer, then this is the same as previously (but now in vector form):

$$\frac{\partial C_k}{\partial \mathbf{W}^{(2)}} = \frac{\partial C_k}{\partial \mathbf{a}^{(2)}} \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}}$$

With a similar term for the bias. If we want to calculate the derivative with respects to weights of the previous layer, we use the expression,

$$\frac{\partial C_k}{\partial \mathbf{W}^{(1)}} = \frac{\partial C_k}{\partial \mathbf{a}^{(2)}} \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{a}^{(1)}} \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}}$$

Where $\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{a}^{(1)}}$ itself can be expanded to,

$$\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{a}^{(1)}} = \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}}$$

.

This can be generalised to any layer,

$$\frac{\partial C_k}{\partial \mathbf{W}^{(i)}} = \frac{\partial C_k}{\partial \mathbf{a}^{(N)}} \underbrace{\frac{\partial \mathbf{a}^{(N)}}{\partial \mathbf{a}^{(N-1)}} \frac{\partial \mathbf{a}^{(N-1)}}{\partial \mathbf{a}^{(N-2)}} \cdots \frac{\partial \mathbf{a}^{(i+1)}}{\partial \mathbf{a}^{(i)}}}_{\text{from layer } N \text{ to layer } i} \frac{\partial \mathbf{a}^{(i)}}{\partial \mathbf{z}^{(i)}} \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{W}^{(i)}}$$

By further application of the chain rule.

Choose the correct expression for the derivative,

$$\frac{\partial \mathbf{a}^{(j)}}{\partial \mathbf{a}^{(j-1)}}$$

Remembering the activation equations are,

$a^{(n)} = \sigma(z^{(n)})$

$z^{(n)} = w^{(n)} a^{(n-1)} + b^{(n)}$.

- ( ) $$\frac{\sigma'(\mathbf{z}^{(j)})}{\sigma'(\mathbf{z}^{(j-1)})}$$
- ( ) $\mathbf{W}^{(j)} \mathbf{a}^{(j)}$
- (x) $$\sigma'(\mathbf{z}^{(j)}) \mathbf{W}^{(j)}$$

Correct
Good application of the chain rule.

- ( ) $$\sigma'(\mathbf{z}^{(j)}) \mathbf{W}^{(j-1)}$$