The easiest way to understand the calculus behind backpropagation is to go through the layers of the neural network that the cost function and its components can be expressed as.

1 Cost Function, C

$$C = \frac{1}{L_n} \sum_{j}^{L_n} \frac{1}{2}(a_j^L - y_j)^2$$

- L_n is the number of neurons in the output layer, L
- a_j^L is the activation level of the jth neuron in layer L
- y_j is the expected activation level of the jth neuron in layer L

The cost function is a way to quantify how wrong the activation outputs of a neural network are upon completing a forward pass through the network, after being exposed to a training example. Having the ability to express cost as a continuous function of a_j^l, in this case $\frac{1}{2}$ mean squared error, allows us to understand the rate at which cost changes with respect to the activation levels of the output neurons. We express this as the derivative.

$$\frac{dC}{da_j^L} = \frac{1}{L_n}(a_j^L - y_j)$$

The derivative of cost with respect to the activation level of the jth neuron in layer L is the derivative of the cost function.

2 Activation Level of a neuron in the output layer, a_j^L

$$a_j^L = \sigma(z_j^L)$$

- z_j^L is the pre-activation level of the jth neuron at layer L

a_j^L can be expressed as the activation function of z_j^L; in this case, the activation function is sigmoidal. Similarly, we can take advantage of the derivative to understand the rate at which the activation level of the jth neuron changes with respect to its pre-activation level. Given that the derivative of the sigmoid function is $\sigma(x)(1-\sigma(x))$,

$$\frac{da_j^L}{dz_j^L} = \sigma(z_j^L)(1-\sigma(z_j^L)) = a_j^L(1-a_j^L)$$

3 Pre-activation level of a neuron in the output layer L, z_j^L

$$z_j^L = \sum_{k}^{L-1_n} w_{jk} a_k^{L-1} + b_j$$

- w_{jk} is the weight between the kth neuron in layer $L-1$ and the jth neuron in layer L
- a_k^{L-1} is the activation level of the kth neuron in layer $L-1$
- b_j the bias of the jth neuron in layer L
- $L-1_n$ is the number of neurons in layer $L-1$, the last hidden layer

The pre-activation levels of any layer are computed through dot product of the corresponding weight matrix and vector of activation levels from the previous layer, plus the bias vector. The pre-activation level of any specific neuron is the weighted activation level of a connected neuron, plus bias, summed over all neurons in the previous layer. Since z_j^L is a function defined by two variables, we must take the partial derivative to understand the respective rates of change.

$$\frac{\partial z_j^L}{\partial w_{jk}} = a_k^{L-1}$$

The partial derivative of the pre-activation level of the jth neuron with respect to the weight between neuron k and neuron j is the activation level of the kth neuron in layer $L-1$.

$$\frac{\partial z_j^L}{\partial b_j} = 1$$

The partial derivative of the pre-activation level of the jth neuron with respect to its bias is 1.

Cost Function, C, re-expressed

$$C = \frac{1}{L_n}\sum_j^{L_n} \frac{1}{2}(\sigma(\sum_k^{L-1_n} w_{jk}a_k^{L-1} + b_j) - y_j)^2$$

This equation simply represents the cost function in the terms defined in steps 2 and 3. Crucially, we recognize that cost is a composite function. Taking the derivative of cost with respect to a weight or bias therefore invokes the chain rule, where given $y = f(u)$, $u = g(x)$,

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Therefore, the derivative of cost with respect to a weight connected to a neuron in the output layer is

$$\frac{\partial C}{\partial w_{jk}} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}}$$

$$= \frac{1}{L_n}(a_j^L - y_j) \cdot a_j^L(1 - a_j^L) \cdot a_k^{L-1}$$

The derivative of cost with respect to the bias of a neuron in the output layer is

$$\frac{\partial C}{\partial b_j} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j}$$

$$= \frac{1}{L_n}(a_j^L - y_j) \cdot a_j^L(1 - a_j^L) \cdot 1$$

Error

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

Let us define δ_j^L as the derivative of cost with respect to the jth neuron's pre-activation level. Conceptually, this quantifies how wrong the pre-activation level is, and in what direction the value must shift. For a neuron in the output layer, this is easily calculated, as demonstrated above. In other layers, however, one cannot simply take the derivative of a cost function to derive $\frac{\partial C}{\partial a_j^L}$, as there is no expected activation level for these neurons. This is problematic, for if one expects to calculate gradients for all weights and biases, the error term as defined above fails to generalize beyond the output neurons.

The derivative of cost with respect to the activation level of a neuron can only be modeled by the contribution of the neuron to the error of the next layer's neurons. Luckily, this is straightforward. For an arbitrary neuron, z,

$$\delta_z = \frac{\partial C}{\partial a_z} = \sum_q^n w_{qz}\delta_q$$

- n is the number of neurons in the layer after neuron z's
- q is an arbitrary neuron in the layer after neuron z's
- w_{qz} is the weight from neuron z to q

- δ_q is the error term of q

This captures the proportional effect of neuron z on the subsequent layer. With this, it becomes evident why backpropagation is necessary. The error term for non-output layers is derived from the subsequent layer, which means that the calculation must begin with the error of neurons in the output layer and work backwards.

Therefore, the derivative of cost with respect to any weight from arbitrary neuron k to neuron j, is

$$\frac{\partial C}{\partial w_{jk}} = \delta_j \cdot a_k^{L-1}$$

And the derivative of cost with respect to any bias for an arbitrary neuron j is

$$\frac{\partial C}{\partial b_j} = \delta_j \cdot 1 = \delta_j$$

Gradients

Now that we understand how the individual derivatives are calculated, we can assemble them as a gradient. Vectorizing these operations allows for greater computational efficiency. Thus, we will model the process using linear algebra.

Since the error term is central to these calculations, we must first derive δ^l, the vector representing the error of the neurons in an arbitrary layer l. Recall that the process for calculating the errors of the output layer is unique. For the output layer L,

$$\delta^L = \frac{1}{L_n}(a^L - y) \odot \frac{\partial \sigma}{\partial z^L}$$

- a^L is the vector of activation levels for the output layer, calculated during the forward pass
- y is the vector of expected activation levels for the output layer
- z^L is the vector of pre-activation levels for the output layer

Multiplying the vector $a^L - y$ with a scalar $\frac{1}{L_n}$ yields a vector with elements corresponding to the derivative of cost with respect to the activation level of each output neuron. Taking the Hadamard product (element-wise multiplication) with the vector $\frac{\partial \sigma}{\partial z^L}$ multiplies each derivative in the cost vector with the derivative of the activation level with respect to the pre-activation level, yielding a vector of error terms for the output layer.

For all other layers l,

$$\delta^l = (W^{l+1})^T \cdot \delta^{l+1} \odot \frac{\partial \sigma}{\partial z^l}$$

- W^{l+1} is the matrix of weights connecting layer l with layer $l+1$
- δ^{l+1} is the error vector of the subsequent layer
- z^l is the vector of pre-activation levels of layer l

The resulting vector of the dot product $(W^{l+1})^T \cdot \delta^{l+1}$ consists of the derivative of cost with respect to the activation level of the neurons in l. Recall that the error of neurons in layer l are computed as the weighted sum of errors in $l+1$. Transposing W^{l+1} allows each weight from layer l to $l+1$ to be multiped by the corresponding error of the neuron in $l+1$, and then summed. Once again, taking the Hadamard product with $\frac{\partial \sigma}{\partial z^L}$ yields the error vector.

Now that we have established how to find the error vectors in a computationally efficient manner, we must use these vectors to calculate the gradients.

The weight gradient for W^l is a matrix yielded by vector multiplication

$$\nabla W^l = \delta^l \cdot (a^{l-1})^T$$

Since vectors are typically columnar by default, transposing a^{l-1} enables the dot product operation with δ^l. Note that this is not a typical dot product, which would be a row vector multipled by a column vector, yielding a scalar.

The reverse, as in this case, is called the outer product.

Also notice the similarity of the dot product to the previously derived derivative of cost with respect to w_{jk}. Indeed, the elements of the matrix are those derivatives, corresponding to a particular weight. Specifically, the element at position (j,k) in ∇W^l is:

$$(\nabla W^l)_{jk} = \delta^l_j \cdot a^{l-1}_k = \frac{\partial C}{\partial w^l_{jk}}$$

As the vectors δ^l and a^{l-1} have dimensionality equal to the number of neurons in their respective layers, the outer product yields a matrix with dimensionality equal to w^l. If layer $l-1$ has m neurons and layer l has n neurons:

$$\delta^l \text{ is size } n \times 1$$

$$(a^{l-1})^T \text{ is size } 1 \times m$$

$$(n \times 1) \cdot (1 \times m) = (n \times m)$$

All that remains is to compute the updated weight matrix using matrix subtraction.

$$W^l - \eta \cdot \nabla W^l$$

• η is the learning rate, a scalar used to avoid making the updates to W^l too large.

The gradient for bias is even simpler; for the bias vector of a layer l, b^l,

$$\nabla b^l = \delta^l$$

Since deriving the bias gradient requires no further operations on the error vector, it should be apparent how the element and size of the bias gradient correspond to that of the bias vector. Finally, updating the existing bias vector,

$$b^l - \eta \cdot \nabla b^l$$

Once these operations are applied at every layer, backpropagation is complete, using stochastic gradient descent. For batch gradient descent, find the average gradient after passing all training examples through the network, and then update the weights and biases accordingly. For mini-batch gradient descent, find the average gradient after passing in a randomly selected set of training examples, and then update the weights and biases. Computationally speaking, this is the most efficient method.