

The calculus behind backpropagation becomes clear when you view the cost function as a continuous, differentiable composite of the loss function, the activation function, and the pre-activation computation.

1 Cost function, C

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

- n_L is the number of neurons in the output layer, L
- a_j^L is the activation level of the j th neuron in layer L
- y_j is the expected activation level of the j th 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. Expressing cost as a continuous function of a_j^L , in this case half of the 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.

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

This derivative captures how sensitive the cost is to changes in each output neuron's activation.

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 j th 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 sigmoid. Similarly, we can use the derivative to understand the rate at which the activation level of the j th 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

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

- w_{jk}^L is the weight from the k th neuron in layer $L - 1$ to the j th neuron in layer L
- a_k^{L-1} is the activation level of the k th neuron in layer $L - 1$
- b_j^L is the bias of the j th neuron in layer L
- n_{L-1} is the number of neurons in layer $L - 1$, the last hidden layer

The pre-activation levels of any layer are computed through the matrix-vector product of the corresponding weight matrix and vector of activation levels from the previous layer, plus the bias vector. The pre-activation level of a neuron is the sum of weighted activations from all neurons in the previous layer, plus a bias term. Since z_j^L is a function of both the weight and bias, we must take the partial derivative to understand the respective rates of change.

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

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

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

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

Cost function, C , re-expressed

$$C = \frac{1}{n_L} \sum_{j=1}^{n_L} \frac{1}{2} (\sigma(\sum_{k=1}^{n_{L-1}} w_{jk}^L a_k^{L-1} + b_j^L) - y_j)^2$$

This equation represents the cost function in the terms defined in sections 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 $h = f(u)$, $u = g(x)$,

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

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

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^L} &= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} \\ &= \frac{1}{n_L} (a_j^L - y_j) \cdot a_j^L (1 - a_j^L) \cdot a_k^{L-1} \end{aligned}$$

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

$$\begin{aligned} \frac{\partial C}{\partial b_j^L} &= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial b_j^L} \\ &= \frac{1}{n_L} (a_j^L - y_j) \cdot a_j^L (1 - a_j^L) \cdot 1 \end{aligned}$$

Error, δ_j^L

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

Let us define δ_j^L as the derivative of cost with respect to the j th 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 directly compute $\frac{\partial C}{\partial a_j^L}$ from the cost function, as we have no ground-truth labels to derive it for non-output layers. 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 computed from the contribution of the neuron to the error of the next layer's neurons. Fortunately, this is straightforward. For an arbitrary neuron, j , in layer l ,

$$\frac{\partial C}{\partial a_j^l} = \sum_{q=1}^{n_{l+1}} w_{qj}^{l+1} \delta_q^{l+1}$$

- n_{l+1} is the number of neurons that receive input from neuron j
- q is an arbitrary neuron that receives input from neuron j
- w_{qj}^{l+1} is the weight from neuron j to q

- δ_q^{l+1} is the error term of q

This captures the proportional effect of neuron j 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

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

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

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

Gradients, ∇W^l and ∇b^l

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 ,

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

- 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 $a^L - y$ by a scalar $\frac{1}{n_L}$ 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 by **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 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 is 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 multiplied 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 column vectors by convention, transposing a^{l-1} enables multiplication with δ^l . Note that this is not a typical dot product, which would be a row vector multiplied by a column vector, yielding a scalar. The

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

Also notice the similarity of the outer 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:

$$(\nabla W^l)_{jk} = \delta_j^l \cdot a_k^{l-1} = \frac{\partial C}{\partial w_{jk}^l}$$

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 elements and size of the bias gradient correspond to those 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. This process describes stochastic gradient descent—updating after each example. 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. This often provides the best tradeoff between convergence stability and efficiency; batches are large enough to reduce variance of gradient estimates, but small enough to permit frequent parameter updates per epoch.

Perceptrons

Perceptrons represent the fundamental implementation of a neural network. The earliest of these networks, now referred to as single-layer perceptrons, consist solely of an input and output layer, with no hidden layers. Consequently, they could only learn linearly separable functions. The entirety of a single-layer perceptron can be articulated as:

$$\text{output} = \sigma(Wx + b)$$

Notably, the process involves only one linear transformation of the input vector. Therefore, updates to the matrix through learning can only attempt to better approximate a linear relationship. A nonlinear activation function does not alter the linearly transformed representation of x .

In order to learn nonlinear functions, the input vector must undergo nonlinear transformations. Multiple linear transformations in succession, e.g., $A(B(C(x)))$ amount to a single linear transformation, $(ABC)x$. However, nonlinear activation functions in-between each successive transformation mean each matrix operates on a nonlinearly transformed representation of the input vector, x . Thus, we are motivated to construct a multilayer perceptron, enabling the neural network to learn nonlinear patterns in the data. Backpropagation was initially introduced as a means for multilayer perceptrons to learn.

A multilayer perceptron with a single hidden layer of sufficient width is, in principle, enough to approximate

any continuous function. However, the number of neurons increases exponentially with the complexity of the function, making training and inference computationally prohibitive. Fewer neurons distributed across multiple hidden layers are simpler to train, and additionally improve generalization by avoiding overfitting.

Still, multilayer perceptrons face limitations. Tasks where the relevant semantic information of input vectors aren't encoded at fixed positions are intractable for MLPs, because the weights learned through backpropagation correspond to fixed positions in the input vector. This is especially problematic for language tasks, which often involve input vectors of varying size and context dependent semantic structure. For instance:

“The chicken refused to cross the road because it was tired”.

“The chicken refused to cross the road because it was wide.”

These sentences are nearly identical in structure, but “it” refers to “the chicken” in the first and “the road” in the second. Multilayer perceptrons lack a mechanism to dynamically distinguish the antecedent of “it” based on the content of the adjective, as the weights connected to “it” define a fixed transformation. In other words, a multilayer perceptron can only approximate context-dependent relationships through pattern matching on similar examples seen during training, and cannot generalize the underlying rule to novel configurations.

Recurrent Neural Networks

Context-dependent relationships, such as those seen in language, are a feature of sequential data, where the meaning or value at some position depends on prior information. In order to learn these dependencies, recurrent neural networks make significant structural changes to the perceptron model. First, they introduce a hidden state, encapsulating the prior context of any sequence. Much like how a multilayer perceptron transforms its input vector, the hidden state undergoes learned transformations, enabling the model to detect nonlinear patterns from prior context. However, ‘prior’ is only meaningful relative to some present position — the model therefore requires a structure that processes tokens sequentially, updating the hidden state as it progresses through the sequence. Within each iteration, known as a timestep t , the RNN passes forward two distinct vectors through their own set of transformations: the presently attended token x_t and the hidden state derived from the previous timestep, h_{t-1} . After each vector is transformed by its respective weight matrix, they are summed together along with a bias vector, then passed into an activation function (usually \tanh). This yields the new hidden state, representing both prior context and the present token. The transformation at the first layer of a timestep is,

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$$

where h_t is the new hidden state of this layer, stored for use in the subsequent timestep. Including a second layer, the transformation becomes,

$$\begin{aligned} h_t^{(1)} &= \tanh(W_h^{(1)} h_{t-1}^{(1)} + W_x^{(1)} x_t + b^{(1)}) \\ h_t^{(2)} &= \tanh(W_h^{(2)} h_{t-1}^{(2)} + W_x^{(2)} h_t^{(1)} + b^{(2)}) \end{aligned}$$

Notably, the hidden state from each layer is passed forward into the next. This stacking parallels depth in multilayer perceptrons: each successive layer operates not on raw tokens but on what the previous layer learned, enabling the model to represent patterns of greater complexity. Unlike MLPs, the focus is not on outputs at each step, but on the development of the hidden state; intermediate outputs are typically discarded, with only the final hidden state, having accumulated the full sequence context, driving the model's prediction.

Essentially, an RNN's hidden state flows along two axes: temporally, from one timestep to the next, and hierarchically, from lower layers to higher layers within a timestep. At each layer and timestep, the model combines information from both directions — the prior hidden state (temporal) and either the current token (at Layer 1) or the hidden state of the layer below (at deeper layers) — to produce an updated representation. When it comes to solving context-dependent relationships, such as the antecedent of ‘it’ from the previously discussed example: by the time the RNN processes ‘it’, the hidden states already carry forward representations of both ‘the chicken’ and ‘the road.’ When the model then processes the adjective, that token's representation interacts with the accumulated context, allowing the referent to be disambiguated.