brilliant.org /wiki/backpropagation/

Backpropagation | Brilliant Math & Science Wiki

25-31 minutes

Backpropagation was invented in the 1970s as a general optimization method for performing automatic differentiation of complex nested functions. However, it wasn't until 1986, with the publishing of a paper by Rumelhart, Hinton, and Williams, titled "Learning Representations by Back-Propagating Errors," that the importance of the algorithm was appreciated by the machine learning community at large.

Researchers had long been interested in finding a way to train multilayer artificial neural networks that could automatically discover good "internal representations," i.e. features that make learning easier and more accurate. Features can be thought of as the stereotypical input to a specific node that activates that node (i.e. causes it to output a positive value near 1). Since a node's activation is dependent on its incoming weights and bias, researchers say a node has learned a feature if its weights and bias cause that node to activate when the feature is present in its input.

By the 1980s, hand-engineering features had become the de facto standard in many fields, especially in computer vision, since experts knew from experiments which features (e.g. lines, circles, edges, blobs in computer vision) made learning simpler. However, hand-engineering successful features requires a lot of knowledge and practice. More importantly, since it is not automatic, it is usually very slow.

Backpropagation was one of the first methods able to demonstrate that artificial neural networks could learn good internal representations, i.e. their hidden layers learned nontrivial features. Experts examining multilayer feedforward networks trained using backpropagation actually found that many nodes learned features similar to those designed by human experts and those found by neuroscientists investigating biological neural networks in mammalian brains (e.g. certain nodes learned to detect edges, while others computed Gabor filters). Even more importantly, because of the efficiency of the algorithm and the fact that domain experts were no longer required to discover appropriate features, backpropagation allowed artificial neural networks to be applied to a much wider field of problems that were previously off-limits due to time and cost constraints.

Backpropagation is analogous to calculating the delta rule for a multilayer feedforward network. Thus, like the delta rule, backpropagation requires three things:

> 1) **Dataset** consisting of input-output pairs $\left(\vec{x}_i, \vec{y}_i\right)$, where $\vec{x}_i$ is the input and $\vec{y}_i$ is the desired output of the network on input $\vec{x}_i$. The set of input-output pairs of size N is denoted $X = \left\{\left(\vec{x}_1, \vec{y}_1\right), \dots, \left(\vec{x}_N, \vec{y}_N\right)\right\}$.
>
> 2) A **feedforward neural network**, as formally defined in the article concerning feedforward neural networks, whose parameters are collectively denoted θ. In backpropagation, the parameters of primary interest are w_{ij}^k, the weight between node j in layer l_k and node i in layer l_{k-1}, and b_i^k, the bias for node i in layer l_k. There are no connections between nodes in the same layer and layers are fully connected.
>
> 3) An **error function**, $E(X, \theta)$, which defines the error between the desired output $\vec{y}_i$ and the calculated output $\hat{\vec{y}}_i$ of the neural network on input $\vec{x}_i$ for a set of input-output pairs $\left(\vec{x}_i, \vec{y}_i\right) \in X$ and a particular value of the parameters θ.

Training a neural network with gradient descent requires the calculation of the gradient of the error function $E(X, \theta)$ with respect to the weights w_{ij}^k and biases b_i^k. Then, according to the learning rate α, each iteration of gradient descent updates the weights and biases (collectively denoted θ) according to

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta^t)}{\partial \theta},$$

where θ^t denotes the parameters of the neural network at iteration t in gradient descent.

What's the Target?

As mentioned in the previous section, one major problem in training multilayer feedforward neural networks is in deciding how to learn good internal representations, i.e. what the weights and biases for hidden layer nodes should be. Unlike the perceptron, which has the delta rule for approximating a well-defined target output, hidden layer nodes don't have a target output since they are used as intermediate steps in the computation.

Since hidden layer nodes have no target output, one can't simply define an error function that is specific to that node. Instead, any error function for that node will be dependent on the values of the parameters in the previous layers (since previous layers determine the input for that node) and following layers $\big($since the output of that node will affect the computation of the error function $E(X, \theta)\big)$. This coupling of parameters between layers can make the math quite messy (primarily as a result of using the product rule, discussed below), and if not implemented cleverly, can make the final gradient descent calculations slow. Backpropagation addresses both of these issues by simplifying the mathematics of gradient descent, while also facilitating its efficient calculation.

Formal Definition

The formulation below is for a neural network with one output, but the algorithm can be applied to a network with any number of outputs by consistent application of the chain rule and power rule. Thus, for all the following examples, input-output pairs will be of the form $(\vec{x}, y)$, i.e. the target value y is not a vector.

Remembering the general formulation for a feedforward neural network,

> w_{ij}^k: weight for node j in layer l_k for incoming node i
> b_i^k: bias for node i in layer l_k
> a_i^k: product sum plus bias (activation) for node i in layer l_k
> o_i^k: output for node i in layer l_k
> r_k: number of nodes in layer l_k
>
> g: activation function for the hidden layer nodes
> g_o: activation function for the output layer nodes

The error function in classic backpropagation is the mean squared error

$$E(X,\theta)=\frac{1}{2N}\sum_{i=1}^{N}\left(\hat{y}_i-y_i\right)^2,$$

where y_i is the target value for input-output pair $(\vec{x}_i, y_i)$ and $\hat{y}_i$ is the computed output of the network on input $\vec{x}_i$. Again, other error functions can be used, but the mean squared error's historical association with backpropagation and its convenient mathematical properties make it a good choice for learning the method.

The derivation of the backpropagation algorithm is fairly straightforward. It follows from the use of the chain rule and product rule in differential calculus. Application of these rules is dependent on the differentiation of the activation function, one of the reasons the heaviside step function is not used (being discontinuous and thus, non-differentiable).

Preliminaries

For the rest of this section, the derivative of a function $f(x)$ will be denoted $f^{'}(x)$, so that the sigmoid function's derivative is $\sigma^{'}(x)$.

To simplify the mathematics further, the bias b_i^k for node i in layer k will be incorporated into the weights as w_{0i}^k with a fixed output of $o_0^{k-1}=1$ for node 0 in layer $k-1$. Thus,

$$w_{0i}^k=b_i^k.$$

To see that this is equivalent to the original formulation, note that

$$a_i^k=b_i^k+\sum_{j=1}^{r_{k-1}}w_{ji}^k o_j^{k-1}=\sum_{j=0}^{r_{k-1}}w_{ji}^k o_j^{k-1},$$

where the left side is the original formulation and the right side is the new formulation.

Using the notation above, backpropagation attempts to minimize the following error function with respect to the neural network's weights:

$$E(X,\theta)=\frac{1}{2N}\sum_{i=1}^{N}\left(\hat{y}_i-y_i\right)^2$$

by calculating, for each weight w_{ij}^k, the value of $\frac{\partial E}{\partial w_{ij}^k}$. Since the error function can be decomposed into a sum over individual error terms for each individual input-output pair, the derivative can be calculated with respect to each input-output pair individually and then combined at the end (since the derivative of a sum of functions is the sum of the derivatives of each function):

$$\frac{\partial E(X, \theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial}{\partial w_{ij}^k} \left(\frac{1}{2} \left(\hat{y}_d - y_d \right)^2 \right) = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial E_d}{\partial w_{ij}^k}.$$

Thus, for the purposes of derivation, the backpropagation algorithm will concern itself with only one input-output pair. Once this is derived, the general form for all input-output pairs in X can be generated by combining the individual gradients. Thus, the error function in question for derivation is

$$E = \frac{1}{2} \left(\hat{y} - y \right)^2,$$

where the subscript d in E_d, $\hat{y}_d$, and y_d is omitted for simplification.

Error Function Derivatives

The derivation of the backpropagation algorithm begins by applying the chain rule to the error function partial derivative

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k},$$

where a_j^k is the activation (product-sum plus bias) of node j in layer k before it is passed to the nonlinear activation function (in this case, the sigmoid function) to generate the output. This decomposition of the partial derivative basically says that the change in the error function due to a weight is a product of the change in the error function E due to the activation a_j^k times the change in the activation a_j^k due to the weight w_{ij}^k.

The first term is usually called the **error**, for reasons discussed below. It is denoted

$$\delta_j^k \equiv \frac{\partial E}{\partial a_j^k}.$$

The second term can be calculated from the equation for a_j^k above:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}\left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1}\right) = o_i^{k-1}.$$

Thus, the partial derivative of the error function E with respect to a weight w_{ij}^k is

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}.$$

Thus, the partial derivative of a weight is a product of the error term δ_j^k at node j in layer k, and the output o_i^{k-1} of node i in layer $k-1$. This makes intuitive sense since the weight w_{ij}^k connects the output of node i in layer $k-1$ to the input of node j in layer k in the computation graph.

It is important to note that the above partial derivatives have all been calculated without any consideration of a particular error function or activation function. However, since the error term δ_j^k still needs to be calculated, and is dependent on the error function E, at this point it is necessary to introduce specific functions for both of these. As mentioned previously, classic backpropagation uses the mean squared error function (which is the squared error function for the single input-output pair case) and the sigmoid activation function.

The calculation of the error δ_j^k will be shown to be dependent on the values of error terms in the next layer. Thus, computation of the error terms will proceed backwards from the output layer down to the input layer. This is where backpropagation, or backwards propagation of errors, gets its name.

The Output Layer

Starting from the final layer, backpropagation attempts to define the value δ_1^m, where m is the final layer (the subscript is 1 and not j because this derivation concerns a one-output neural network, so there is only one output node $j = 1$). For example, a four-layer neural network will have

$m = 3$ for the final layer, $m = 2$ for the second to last layer, and so on. Expressing the error function E in terms of the value a_1^m $\Big($since δ_1^m is a partial derivative with respect to $a_1^m\Big)$ gives

$$E = \frac{1}{2}\Big(\hat{y} - y\Big)^2 = \frac{1}{2}\Big(g_o(a_1^m) - y\Big)^2,$$

where $g_o(x)$ is the activation function for the output layer.

Thus, applying the partial derivative and using the chain rule gives

$$\delta_1^m = \Big(g_0(a_1^m) - y\Big)g_o^{'}(a_1^m) = \Big(\hat{y} - y\Big)g_o^{'}(a_1^m).$$

Putting it all together, the partial derivative of the error function E with respect to a weight in the final layer w_{i1}^m is

$$\frac{\partial E}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} = \Big(\hat{y} - y\Big)g_o^{'}(a_1^m)\, o_i^{m-1}.$$

The Hidden Layers

Now the question arises of how to calculate the partial derivatives of layers other than the output layer. Luckily, the chain rule for multivariate functions comes to the rescue again. Observe the following equation for the error term δ_j^k in layer $1 \leq k < m$:

$$\delta_j^k = \frac{\partial E}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k},$$

where l ranges from 1 to r^{k+1} (the number of nodes in the next layer). Note that, because the bias input o_0^k corresponding to w_{0j}^{k+1} is fixed, its value is not dependent on the outputs of previous layers, and thus l does not take on the value 0.

Plugging in the error term δ_l^{k+1} gives the following equation:

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}.$$

Remembering the definition of a_l^{k+1}

$$a_l^{k+1} = \sum_{j=1}^{r^k} w_{jl}^{k+1} g\left(a_j^k\right),$$

where $g(x)$ is the activation function for the hidden layers,

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'\left(a_j^k\right).$$

Plugging this into the above equation yields a final equation for the error term δ_j^k in the hidden layers, called the **backpropagation formula**:

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'\left(a_j^k\right) = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

Putting it all together, the partial derivative of the error function E with respect to a weight in the hidden layers w_{ij}^k for $1 \le k < m$ is

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'\left(a_j^k\right) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

Backpropagation as Backwards Computation

This equation is where backpropagation gets its name. Namely, the error δ_j^k at layer k is dependent on the errors δ_k^{k+1} at the next layer $k+1$. Thus, errors flow backward, from the last layer to the first layer. All that is needed is to compute the first error terms based on the computed output $\hat{y} = g_o(a_1^m)$ and target output y. Then, the error terms for the previous layer are computed by performing a product sum $\left(\text{weighted by } w_{jl}^{k+1}\right)$ of the error terms for the next layer and scaling it by $g'\left(a_j^k\right)$, repeated until the input layer is reached.

This backwards propagation of errors is very similar to the forward computation that calculates the neural network's output. Thus, calculating the output is often called the **forward phase** while calculating the error terms and derivatives is often called the **backward phase**. While going in the forward direction, the inputs are repeatedly recombined from the *first layer to the last* by product sums dependent on the weights w_{ij}^k and transformed by nonlinear activation functions $g(x)$ and $g_o(x)$. In the backward direction, the "inputs" are the final layer's error terms, which are repeatedly recombined from the *last layer to the first* by product sums dependent on the weights w_{jl}^{k+1} and transformed by nonlinear scaling factors $g_o'\left(a_j^m\right)$ and $g'\left(a_j^k\right)$.

Furthermore, because the computations for backwards phase are dependent on the activations a_j^k and outputs o_j^k of the nodes in the previous (the non-error term for all layers) and next layer (the error term for hidden layers), all of these values must be computed before the backwards phase can commence. Thus, the forward phase precedes the backward phase for every iteration of gradient descent. In the forward phase, activations a_j^k and outputs o_j^k will be remembered for use in the backwards phase. Once the backwards phase is completed and the partial derivatives are known, the weights $\left(\text{and associated biases } b_j^k = w_{0j}^k\right)$ can be updated by gradient descent. This process is repeated until a local minimum is found or convergence criterion is met.

Using the terms defined in the section titled Formal Definition and the equations derived in the section titled Deriving the Gradients, the backpropagation algorithm is dependent on the following five equations:

> For the partial derivatives,
>
> $$\frac{\partial E_d}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}.$$

> For the final layer's error term,
>
> $$\delta_1^m = g_o'(a_1^m)\left(\hat{y}_d - y_d\right).$$

> For the hidden layers' error terms,
>
> $$\delta_j^k = g'\left(a_j^k\right)\sum_{l=1}^{r^{k+1}} w_{jl}^{k+1}\delta_l^{k+1}.$$

> For combining the partial derivatives for each input-output pair,
>
> $$\frac{\partial E(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial}{\partial w_{ij}^k}\left(\frac{1}{2}\left(\hat{y}_d - y_d\right)^2\right) = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial E_d}{\partial w_{ij}^k}.$$

> For updating the weights,
>
> $$\Delta w_{ij}^k = -\alpha \frac{\partial E(X, \theta)}{\partial w_{ij}^k}.$$

The General Algorithm

The backpropagation algorithm proceeds in the following steps, assuming a suitable learning rate α and random initialization of the parameters w_{ij}^k:

> 1) **Calculate the forward phase** for each input-output pair $(\vec{x}_d, y_d)$ and store the results $\hat{y}_d$, a_j^k, and o_j^k for each node j in layer k by proceeding from layer 0, the input layer, to layer m, the output layer.
>
> 2) **Calculate the backward phase** for each input-output pair $(\vec{x}_d, y_d)$ and store the results $\frac{\partial E_d}{\partial w_{ij}^k}$ for each weight w_{ij}^k connecting node i in layer $k-1$ to node j in layer k by proceeding from layer m, the output layer, to layer 1, the input layer.
>
> a) Evaluate the error term for the final layer δ_1^m by using the second equation.
> b) Backpropagate the error terms for the hidden layers δ_j^k, working backwards from the final hidden layer $k = m-1$, by repeatedly using the third equation.
> c) Evaluate the partial derivatives of the individual error E_d with respect to w_{ij}^k by using the first equation.
>
> 3) **Combine the individual gradients** for each input-output pair $\frac{\partial E_d}{\partial w_{ij}^k}$ to get the total gradient $\frac{\partial E(X, \theta)}{\partial w_{ij}^k}$ for the entire set of input-output pairs $X = \left\{(\vec{x}_1, y_1), \dots, (\vec{x}_N, y_N)\right\}$ by using the fourth equation (a simple average of the individual gradients).
>
> 4) **Update the weights** according to the learning rate α and total gradient $\frac{\partial E(X, \theta)}{\partial w_{ij}^k}$ by using the fifth equation (moving in the direction of the negative gradient).

Backpropagation In Sigmoidal Neural Networks

The classic backpropagation algorithm was designed for regression problems with sigmoidal activation units. While backpropagation can be applied to classification problems as well as networks with non-sigmoidal activation functions, the sigmoid function has convenient mathematical properties which, when combined with an appropriate output activation function, greatly simplify the algorithm's understanding. Thus, in the classic formulation, the activation function for hidden nodes is sigmoidal $\big(g(x) = \sigma(x)\big)$ and the output activation function is the identity function $\big(g_o(x) = x\big)$ (the network output is just a weighted sum of its hidden layer, i.e. the activation).

Backpropagation is actually a major motivating factor in the historical use of sigmoid activation functions due to its convenient derivative:

$$g'(x) = \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\big(1 - \sigma(x)\big).$$

Thus, calculating the derivative of the sigmoid function requires nothing more than remembering the output $\sigma(x)$ and plugging it into the equation above.

Furthermore, the derivative of the output activation function is also very simple:

$$g_o'(x) = \frac{\partial g_o(x)}{\partial x} = \frac{\partial x}{\partial x} = 1.$$

Thus, using these two activation functions removes the need to remember the activation values a_1^m and a_j^k in addition to the output values o_1^m and o_j^k, greatly reducing the memory footprint of the algorithm. This is because the derivative for the sigmoid activation function in the backwards phase only needs to recall the output of that function in the forward phase, and is not dependent on the actual activation value, which is the case in the more general formulation of backpropagation where $g'\left(a_j^k\right)$ must be calculated. Similarly, the derivative for the identity activation function doesn't depend on anything since it is a constant.

Thus, for a feedforward neural network with sigmoidal hidden units and an identity output unit, the error term equations are as follows:

> For the final layer's error term,

> $$\delta_1^m = \hat{y}_d - y_d.$$

> For the hidden layers' error terms,
>
> $$\delta_j^k = o_j^k\left(1 - o_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1}\delta_l^{k+1}.$$

Code Example

The following code example is for a sigmoidal neural network as described in the previous subsection. It has one hidden layer and one output node in the output layer. The code is written in Python3 and makes heavy use of the NumPy library for performing matrix math. Because the calculations of the gradient for individual input-output pairs $(\vec{x}_d, y_d)$ can be done in parallel, and many calculations are based on taking the dot product of two vectors, matrices are a natural way to represent the input data, output data, and layer weights. NumPy's efficient computation of matrix products and the ability to use modern GPUs (which are optimized for matrix operations) can give significant speedups in both the forward and backward phases of computation.

```
import numpy as np

# define the sigmoid function
def sigmoid(x, derivative=False):

    if (derivative == True):
        return sigmoid(x,derivative=False) * (1 -
sigmoid(x,derivative=False))
    else:
        return 1 / (1 + np.exp(-x))

# choose a random seed for reproducible results
np.random.seed(1)

# learning rate
alpha = .1

# number of nodes in the hidden layer

```

```
num_hidden = 3

# inputs
X = np.array([
    [0, 0, 1],
    [0, 1, 1],
    [1, 0, 0],
    [1, 1, 0],
    [1, 0, 1],
    [1, 1, 1],
])

# outputs
# x.T is the transpose of x, making this a column
vector
y = np.array([[0, 1, 0, 1, 1, 0]]).T

# initialize weights randomly with mean 0 and
range [-1, 1]
# the +1 in the 1st dimension of the weight
matrices is for the bias weight
hidden_weights = 2*np.random.random((X.shape[1] +
1, num_hidden)) - 1
output_weights = 2*np.random.random((num_hidden +
1, y.shape[1])) - 1

# number of iterations of gradient descent
num_iterations = 10000

# for each iteration of gradient descent
for i in range(num_iterations):

    # forward phase
    # np.hstack((np.ones(...), X) adds a fixed
input of 1 for the bias weight
    input_layer_outputs =
np.hstack((np.ones((X.shape[0], 1)), X))
    hidden_layer_outputs =
np.hstack((np.ones((X.shape[0], 1)),
sigmoid(np.dot(input_layer_outputs,
hidden_weights))))
```

```
    output_layer_outputs =
np.dot(hidden_layer_outputs, output_weights)

    # backward phase
    # output layer error term
    output_error = output_layer_outputs - y
    # hidden layer error term
    # [:, 1:] removes the bias term from the
backpropagation
    hidden_error = hidden_layer_outputs[:, 1:] *
(1 - hidden_layer_outputs[:, 1:]) *
np.dot(output_error, output_weights.T[:, 1:])

    # partial derivatives
    hidden_pd = input_layer_outputs[:, :,
np.newaxis] * hidden_error[: , np.newaxis, :]
    output_pd = hidden_layer_outputs[:, :,
np.newaxis] * output_error[:, np.newaxis, :]

    # average for total gradients
    total_hidden_gradient = np.average(hidden_pd,
axis=0)
    total_output_gradient = np.average(output_pd,
axis=0)

    # update weights
    hidden_weights += - alpha *
total_hidden_gradient
    output_weights += - alpha *
total_output_gradient

# print the final outputs of the neural network on
the inputs X
print("Output After Training:
\n{}".format(output_layer_outputs))
```

The matrix X is the set of inputs $\vec{x}$ and the matrix y is the set of outputs y. The number of nodes in the hidden layer can be customized by setting the value of the variable `num_hidden`. The learning rate α is controlled by the variable `alpha`. The number of iterations of gradient descent is controlled by the variable `num_iterations`.

By changing these variables and comparing the output of the program to the target values y, one can see how these variables control how well backpropagation can learn the dataset X and y. For example, more nodes in the hidden layer and more iterations of gradient descent will generally improve the fit to the training dataset. However, using too large or too small a learning rate can cause the model to diverge or converge too slowly, respectively.

Previous Chapter Next Chapter