WIKIPEDIA

Backpropagation

In machine learning, **backpropagation** (**backprop**,[1] **BP**) is a widely used algorithm for training feedforward neural networks. Generalizations of backpropagation exist for other artificial neural networks (ANNs), and for functions generally. These classes of algorithms are all referred to generically as "backpropagation".[2] In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes it feasible to use gradient methods for training multilayer networks, updating weights to minimize loss; gradient descent, or variants such as stochastic gradient descent, are commonly used. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this is an example of dynamic programming.[3]

The term *backpropagation* strictly refers only to the algorithm for computing the gradient, not how the gradient is used; however, the term is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as by stochastic gradient descent.[4] Backpropagation generalizes the gradient computation in the delta rule, which is the single-layer version of backpropagation, and is in turn generalized by automatic differentiation, where backpropagation is a special case of reverse accumulation (or "reverse mode").[5] The term *backpropagation* and its general use in neural networks was announced in Rumelhart, Hinton & Williams (1986a), then elaborated and popularized in Rumelhart, Hinton & Williams (1986b), but the technique was independently rediscovered many times, and had many predecessors dating to the 1960s; see § History.[6] A modern overview is given in the deep learning textbook by Goodfellow, Bengio & Courville (2016).[7]

Contents

Overview

Backpropagation computes the gradient in weight space of a feedforward neural network, with respect to a loss function. Denote:

- x: input (vector of features)
- y: target output

  For classification, output will be a vector of class probabilities (e.g., $(0.1, 0.7, 0.2)$, and target output is a specific class, encoded by the one-hot/dummy variable (e.g., $(0, 1, 0)$).

- C: loss function or "cost function"[a]

  For classification, this is usually cross entropy (XC, log loss), while for regression it is usually squared error loss (SEL).

- L: the number of layers
- $W^l = (w^l_{jk})$: the weights between layer $l-1$ and l, where w^l_{jk} is the weight between the k-th node in layer $l-1$ and the j-th node in layer l[b]
- f^l: activation functions at layer l

  For classification the last layer is usually the logistic function for binary classification, and softmax (softargmax) for multi-class classification, while for the hidden layers this was traditionally a sigmoid function (logistic function or others) on each node (coordinate), but today is more varied, with rectifier (ramp, ReLU) being common.

In the derivation of backpropagation, other intermediate quantities are used; they are introduced as needed below. Bias terms are not treated specially, as they correspond to a weight with a fixed input of 1. For the purpose of backpropagation, the specific loss function and activation functions do not matter, as long as they and their derivatives can be evaluated efficiently. Traditional activation functions include but are not limited to sigmoid, tanh, and ReLU. Since, swish,[8] mish,[9] and other activation functions were proposed as well.

The overall network is a combination of function composition and matrix multiplication:

$$g(x) := f^L(W^L f^{L-1}(W^{L-1} \cdots f^1(W^1 x) \cdots))$$

For a training set there will be a set of input–output pairs, $\left\{(x_i, y_i)\right\}$. For each input–output pair (x_i, y_i) in the training set, the loss of the model on that pair is the cost of the difference between the predicted output $g(x_i)$ and the target output y_i:

$$C(y_i, g(x_i))$$

Note the distinction: during model evaluation, the weights are fixed, while the inputs vary (and the target output may be unknown), and the network ends with the output layer (it does not include the loss function). During model training, the input–output pair is fixed, while the weights vary, and the network ends with the loss function.

Backpropagation computes the gradient for a *fixed* input–output pair (x_i, y_i), where the weights w_{jk}^l can vary. Each individual component of the gradient, $\partial C / \partial w_{jk}^l$, can be computed by the chain rule; however, doing this separately for each weight is inefficient. Backpropagation efficiently computes the gradient by avoiding duplicate calculations and not computing unnecessary intermediate values, by computing the gradient of each layer – specifically, the gradient of the weighted *input* of each layer, denoted by δ^l – from back to front.

Informally, the key point is that since the only way a weight in W^l affects the loss is through its effect on the *next* layer, and it does so *linearly*, δ^l are the only data you need to compute the gradients of the weights at layer l, and then you can compute the previous layer δ^{l-1} and repeat recursively. This avoids inefficiency in two ways. Firstly, it avoids duplication because when computing the gradient at layer l, you do not need to recompute all the derivatives on later layers $l+1, l+2, \ldots$ each time. Secondly, it avoids unnecessary intermediate calculations because at each stage it directly computes the gradient of the weights with respect to the ultimate output (the loss), rather than unnecessarily computing the derivatives of the values of hidden layers with respect to changes in weights $\partial a_{j'}^{l'} / \partial w_{jk}^l$.

Backpropagation can be expressed for simple feedforward networks in terms of matrix multiplication, or more generally in terms of the adjoint graph.

Matrix multiplication

For the basic case of a feedforward network, where nodes in each layer are connected only to nodes in the immediate next layer (without skipping any layers), and there is a loss function that computes a scalar loss for the final output, backpropagation can be understood simply by matrix multiplication.[c] Essentially, backpropagation evaluates the expression for the derivative of the cost function as a product of derivatives between each layer *from right to left* – "backwards" – with the gradient of the weights between each layer being a simple modification of the partial products (the "backwards propagated error").

Given an input–output pair (x, y), the loss is:

$$C(y, f^L(W^L f^{L-1}(W^{L-1} \cdots f^2(W^2 f^1(W^1 x)) \cdots)))$$

To compute this, one starts with the input x and works forward; denote the weighted input of each hidden layer as z^l and the output of hidden layer l as the activation a^l. For backpropagation, the activation a^l as well as the derivatives $(f^l)'$ (evaluated at z^l) must be cached for use during the backwards pass.

The derivative of the loss in terms of the inputs is given by the chain rule; note that each term is a total derivative, evaluated at the value of the network (at each node) on the input x:

$$\frac{dC}{da^L} \circ \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \circ \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdots \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x},$$

where $\circ$ is a Hadamard product, that is an element-wise product.

These terms are: the derivative of the loss function;[d] the derivatives of the activation functions;[e] and the matrices of weights:[f]

$$\frac{dC}{da^L} \circ (f^L)' \cdot W^L \circ (f^{L-1})' \cdot W^{L-1} \cdots (f^1)' \cdot W^1.$$

The gradient ∇ is the transpose of the derivative of the output in terms of the input, so the matrices are transposed and the order of multiplication is reversed, but the entries are the same:

$$\nabla_x C = (W^1)^T \cdot (f^1)' \ldots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C.$$

Backpropagation then consists essentially of evaluating this expression from right to left (equivalently, multiplying the previous expression for the derivative from left to right), computing the gradient at each layer on the way; there is an added step, because the gradient of the weights isn't just a subexpression: there's an extra multiplication.

Introducing the auxiliary quantity δ^l for the partial products (multiplying from right to left), interpreted as the "error at level l" and defined as the gradient of the input values at level l:

$$\delta^l := (f^l)' \circ (W^{l+1})^T \cdots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C.$$

Note that δ^l is a vector, of length equal to the number of nodes in level l; each component is interpreted as the "cost attributable to (the value of) that node".

The gradient of the weights in layer l is then:

$$\nabla_{W^l} C = \delta^l (a^{l-1})^T.$$

The factor of a^{l-1} is because the weights W^l between level $l-1$ and l affect level l proportionally to the inputs (activations): the inputs are fixed, the weights vary.

The δ^l can easily be computed recursively, going from right to left, as:

$$\delta^{l-1} := (f^{l-1})' \circ (W^l)^T \cdot \delta^l.$$

The gradients of the weights can thus be computed using a few matrix multiplications for each level; this is backpropagation.

Compared with naively computing forwards (using the δ^l for illustration):

$$\begin{aligned}
\delta^1 &= (f^1)' \circ (W^2)^T \cdot (f^2)' \cdots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C \\
\delta^2 &= (f^2)' \cdots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C \\
&\vdots \\
\delta^{L-1} &= (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C \\
\delta^L &= (f^L)' \circ \nabla_{a^L} C,
\end{aligned}$$

there are two key differences with backpropagation:

1. Computing δ^{l-1} in terms of δ^l avoids the obvious duplicate multiplication of layers l and beyond.
2. Multiplying starting from $\nabla_{a^L} C$ – propagating the error *backwards* – means that each step simply multiplies a vector (δ^l) by the matrices of weights $(W^l)^T$ and derivatives of activations $(f^{l-1})'$. By contrast, multiplying forwards, starting from the changes at an earlier layer, means that each multiplication multiplies a *matrix* by a *matrix*. This is much more expensive, and corresponds to tracking every possible path of a change in one layer l forward to changes in the layer $l+2$ (for multiplying W^{l+1} by W^{l+2}, with additional multiplications for the derivatives of the activations), which unnecessarily computes the intermediate quantities of how weight changes affect the values of hidden nodes.

Adjoint graph

For more general graphs, and other advanced variations, backpropagation can be understood in terms of automatic differentiation, where backpropagation is a special case of reverse accumulation (or "reverse mode").[5]

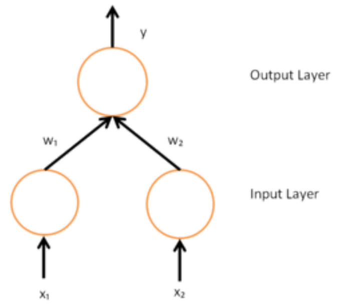
Intuition

Motivation

The goal of any supervised learning algorithm is to find a function that best maps a set of inputs to their correct output. The motivation for backpropagation is to train a multi-layered neural network such that it can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output.[10]

Learning as an optimization problem

To understand the mathematical derivation of the backpropagation algorithm, it helps to first develop some intuition about the relationship between the actual output of a neuron and the correct output for a particular training example. Consider a simple neural network with two input units, one output unit and no hidden units, and in which each neuron uses a linear output (unlike most work on neural networks, in which mapping from inputs to outputs is non-linear)[g] that is the weighted sum of its input.



A simple neural network with two input units (each with a single input) and one output unit (with two inputs)

Initially, before training, the weights will be set randomly. Then the neuron learns from training examples, which in this case consist of a set of tuples (x_1, x_2, t) where x_1 and x_2 are the inputs to the network and t is the correct output (the output the network should produce given those inputs, when it has been trained). The initial network, given x_1 and x_2, will compute an output y that likely differs from t (given random weights). A loss function $L(t, y)$ is used for measuring the discrepancy between the target output t and the computed output y. For regression analysis problems the squared error can be used as a loss function, for classification the categorical crossentropy can be used.

As an example consider a regression problem using the square error as a loss:

$$L(t, y) = (t - y)^2 = E,$$

where E is the discrepancy or error.

Consider the network on a single training case: $(1, 1, 0)$. Thus, the input x_1 and x_2 are 1 and 1 respectively and the correct output, t is 0. Now if the relation is plotted between the network's output y on the horizontal axis and the error E on the vertical axis, the result is a parabola. The minimum of the parabola corresponds to the output y which minimizes the error E. For a single training case, the minimum also touches the horizontal axis, which means the error will be zero and the network can produce an output y that exactly matches the target output t. Therefore, the problem of mapping inputs to outputs can be reduced to an optimization problem of finding a function that will produce the minimal error.

However, the output of a neuron depends on the weighted sum of all its inputs:

$$y = x_1 w_1 + x_2 w_2,$$

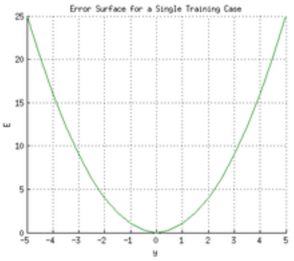
where w_1 and w_2 are the weights on the connection from the input units to the output unit. Therefore, the error also depends on the incoming weights to the neuron, which is ultimately what needs to be changed in the network to enable learning.

In this example, upon injecting the training data $(1, 1, 0)$, the loss function becomes

$$E = (t-y)^2 = y^2 = (x_1 w_1 + x_2 w_2)^2 = (w_1 + w_2)^2.$$

Then, the loss function E takes the form of a parabolic cylinder with its base directed along $w_1 = -w_2$. Since all sets of weights that satisfy $w_1 = -w_2$ minimize the loss function, in this case additional constraints are required to converge to a unique solution. Additional constraints could either be generated by setting specific conditions to the weights, or by injecting additional training data.

One commonly used algorithm to find the set of weights that minimizes the error is gradient descent. By backpropagation, the steepest descent direction is calculated of the loss function versus the present synaptic weights. Then, the weights can be modified along the steepest descent direction, and the error is minimized in an efficient way.



Error surface of a linear neuron for a single training case

Derivation

The gradient descent method involves calculating the derivative of the loss function with respect to the weights of the network. This is normally done using backpropagation. Assuming one output neuron,[h] the squared error function is

$$E = L(t, y)$$

where

L is the loss for the output y and target value t,
t is the target output for a training sample, and
y is the actual output of the output neuron.

For each neuron j, its output o_j is defined as

$$o_j = \varphi(\text{net}_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right),$$

where the activation function φ is non-linear and differentiable (even if the ReLU is not in one point). A historically used activation function is the logistic function:

$$\varphi(z) = \frac{1}{1+e^{-z}}$$

which has a convenient derivative of:

$$\frac{d\varphi(z)}{dz} = \varphi(z)(1-\varphi(z))$$

The input net_j to a neuron is the weighted sum of outputs o_k of previous neurons. If the neuron is in the first layer after the input layer, the o_k of the input layer are simply the inputs x_k to the network. The number of input units to the neuron is n. The variable w_{kj} denotes the weight between neuron k of the previous layer and neuron j of the current layer.

Finding the derivative of the error

Calculating the partial derivative of the error with respect to a weight w_{ij} is done using the chain rule twice:

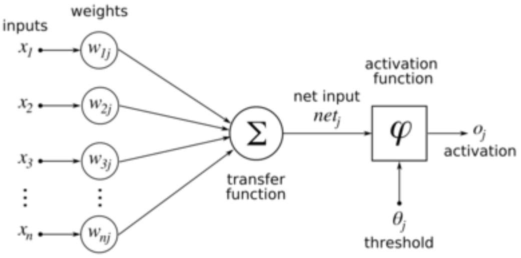


Diagram of an artificial neural network to illustrate the notation used here

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} \qquad \textbf{(Eq. 1)}$$

In the last factor of the right-hand side of the above, only one term in the sum net_j depends on w_{ij}, so that

$$\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij} o_i = o_i. \qquad \textbf{(Eq. 2)}$$

If the neuron is in the first layer after the input layer, o_i is just x_i.

The derivative of the output of neuron j with respect to its input is simply the partial derivative of the activation function:

$$\frac{\partial o_j}{\partial \text{net}_j} = \frac{\partial \varphi(\text{net}_j)}{\partial \text{net}_j} \qquad \textbf{(Eq. 3)}$$

which for the logistic activation function case is:

$$\frac{\partial o_j}{\partial \text{net}_j} = \frac{\partial}{\partial \text{net}_j}\varphi(\text{net}_j) = \varphi(\text{net}_j)(1-\varphi(\text{net}_j)) = o_j(1-o_j)$$

This is the reason why backpropagation requires the activation function to be differentiable. (Nevertheless, the ReLU activation function, which is non-differentiable at 0, has become quite popular, e.g. in AlexNet)

The first factor is straightforward to evaluate if the neuron is in the output layer, because then $o_j = y$ and

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} \qquad \textbf{(Eq. 4)}$$

If half of the square error is used as loss function we can rewrite it as

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(t-y)^2 = y-t$$

However, if j is in an arbitrary inner layer of the network, finding the derivative E with respect to o_j is less obvious.

Considering E as a function with the inputs being all neurons $L = \{u, v, \dots, w\}$ receiving input from neuron j,

$$\frac{\partial E(o_j)}{\partial o_j} = \frac{\partial E(\text{net}_u, \text{net}_v, \dots, \text{net}_w)}{\partial o_j}$$

and taking the total derivative with respect to o_j, a recursive expression for the derivative is obtained:

$$\frac{\partial E}{\partial o_j} = \sum_{\ell \in L}\left(\frac{\partial E}{\partial \text{net}_\ell}\frac{\partial \text{net}_\ell}{\partial o_j}\right) = \sum_{\ell \in L}\left(\frac{\partial E}{\partial o_\ell}\frac{\partial o_\ell}{\partial \text{net}_\ell}\frac{\partial \text{net}_\ell}{\partial o_j}\right) = \sum_{\ell \in L}\left(\frac{\partial E}{\partial o_\ell}\frac{\partial o_\ell}{\partial \text{net}_\ell}w_{j\ell}\right)$$

Therefore, the derivative with respect to o_j can be calculated if all the derivatives with respect to the outputs o_ℓ of the next layer – the ones closer to the output neuron – are known. [Note, if any of the neurons in set L were not connected to neuron j, they would be independent of w_{ij} and the corresponding partial derivative under the summation would vanish to 0.]

Substituting **Eq. 2**, **Eq. 3** **Eq.4** and **Eq. 5** in **Eq. 1** we obtain:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}o_i$$

$$\frac{\partial E}{\partial w_{ij}} = o_i\delta_j$$

with

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j} = \begin{cases} \frac{\partial L(o_j,t)}{\partial o_j}\frac{d\varphi(\text{net}_j)}{d\text{net}_j} & \text{if } j \text{ is an output neuron,} \\ (\sum_{\ell \in L} w_{j\ell}\delta_\ell)\frac{d\varphi(\text{net}_j)}{d\text{net}_j} & \text{if } j \text{ is an inner neuron.} \end{cases}$$

if φ is the logistic function, and the error is the square error:

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j} = \begin{cases} (o_j - t_j)o_j(1-o_j) & \text{if } j \text{ is an output neuron,} \\ (\sum_{\ell \in L} w_{j\ell}\delta_\ell)o_j(1-o_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$

To update the weight w_{ij} using gradient descent, one must choose a learning rate, $\eta > 0$. The change in weight needs to reflect the impact on E of an increase or decrease in w_{ij}. If $\frac{\partial E}{\partial w_{ij}} > 0$, an increase in w_{ij} increases E; conversely, if $\frac{\partial E}{\partial w_{ij}} < 0$, an increase in w_{ij} decreases E. The new Δw_{ij} is added to the old weight, and the product of the learning rate and the gradient, multiplied by -1 guarantees that w_{ij} changes in a way that always decreases E. In other words, in the equation immediately below, $-\eta\frac{\partial E}{\partial w_{ij}}$ always changes w_{ij} in such a way that E is decreased:

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} = -\eta o_i \delta_j$$

Second-order gradient descent

Using a Hessian matrix of second-order derivatives of the error function, the Levenberg-Marquardt algorithm often converges faster than first-order gradient descent, especially when the topology of the error function is complicated.[11][12] It may also find solutions in smaller node counts for which other methods might not converge.[12] The Hessian can be approximated by the Fisher information matrix.[13]

Loss function

The loss function is a function that maps values of one or more variables onto a real number intuitively representing some "cost" associated with those values. For backpropagation, the loss function calculates the difference between the network output and its expected output, after a training example has propagated through the network.

Assumptions

The mathematical expression of the loss function must fulfill two conditions in order for it to be possibly used in backpropagation.[14] The first is that it can be written as an average $E = \frac{1}{n}\sum_x E_x$ over error functions E_x, for n individual training examples, x. The reason for this assumption is that the backpropagation algorithm calculates the gradient of the error function for a single training example, which needs to be generalized to the overall error function. The second assumption is that it can be written as a function of the outputs from the neural network.

Example loss function

Let y, y' be vectors in $\mathbb{R}^n$.

Select an error function $E(y, y')$ measuring the difference between two outputs. The standard choice is the square of the Euclidean distance between the vectors y and y':

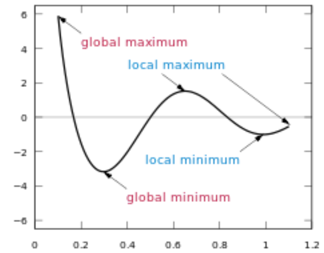
$$E(y, y') = \tfrac{1}{2}\lVert y - y'\rVert^2$$

The error function over n training examples can then be written as an average of losses over individual examples:

$$E = \frac{1}{2n}\sum_x \lVert (y(x) - y'(x)) \rVert^2$$

Limitations

- Gradient descent with backpropagation is not guaranteed to find the global minimum of the error function, but only a local minimum; also, it has trouble crossing plateaus in the error function landscape. This issue, caused by the non-convexity of error functions in neural networks, was long thought to be a major drawback, but Yann LeCun *et al.* argue that in many practical problems, it is not.[15]
- Backpropagation learning does not require normalization of input vectors; however, normalization could improve performance.[16]
- Backpropagation requires the derivatives of activation functions to be known at network design time.



Gradient descent may find a local minimum instead of the global minimum.

History

The term *backpropagation* and its general use in neural networks was announced in Rumelhart, Hinton & Williams (1986a), then elaborated and popularized in Rumelhart, Hinton & Williams (1986b), but the technique was independently rediscovered many times, and had many predecessors dating to the 1960s.[6][17]

The basics of continuous backpropagation were derived in the context of control theory by Henry J. Kelley in 1960,[18] and by Arthur E. Bryson in 1961.[19][20][21][22][23] They used principles of dynamic programming. In 1962, Stuart Dreyfus published a simpler derivation based only on the chain rule.[24] Bryson and Ho described it as a multi-stage dynamic system optimization method in 1969.[25][26] Backpropagation was derived by multiple researchers in the early 60's[22] and implemented to run on computers as early as 1970 by Seppo Linnainmaa.[27][28][29] Paul Werbos was first in the US to propose that it could be used for neural nets after analyzing it in depth in his 1974 dissertation.[30] While not applied to neural networks, in 1970 Linnainmaa published the general method for automatic differentiation (AD).[28][29] Although very controversial, some scientists believe this was actually the first step toward developing a back-propagation algorithm.[22][23][27][31] In 1973 Dreyfus adapts parameters of controllers in proportion to error gradients.[32] In 1974 Werbos mentioned the possibility of applying this principle to artificial neural networks,[30] and in 1982 he applied Linnainmaa's AD method to non-linear functions.[23][33]

Later the Werbos method was rediscovered and described in 1985 by Parker,[34][35] and in 1986 by Rumelhart, Hinton and Williams.[17][35][36] Rumelhart, Hinton and Williams showed experimentally that this method can generate useful internal representations of incoming data in hidden layers of neural networks.[10][37][38] Yann LeCun proposed the modern form of the back-propagation learning algorithm for neural networks in his PhD thesis in 1987. In 1993, Eric Wan won an international pattern recognition contest through backpropagation.[22][39]

During the 2000s it fell out of favour, but returned in the 2010s, benefitting from cheap, powerful GPU-based computing systems. This has been especially so in speech recognition, machine vision, natural language processing, and language structure learning research (in which it has been used to explain a variety of phenomena related to first[40] and second language learning.[41]).

Error backpropagation has been suggested to explain human brain ERP components like the N400 and P600.[42]

See also

- Artificial neural network
- Biological neural network
- Catastrophic interference
- Ensemble learning
- AdaBoost
- Overfitting
- Neural backpropagation
- Backpropagation through time

Notes

a. Use C for the loss function to allow L to be used for the number of layers

b. This follows Nielsen (2015), and means (left) multiplication by the matrix W^l corresponds to converting output values of layer $l-1$ to input values of layer l: columns correspond to input coordinates, rows correspond to output coordinates.

c. This section largely follows and summarizes Nielsen (2015).

d. The derivative of the loss function is a covector, since the loss function is a scalar-valued function of several variables.

e. The activation function is applied to each node separately, so the derivative is just the diagonal matrix of the derivative on each node. This is often represented as the Hadamard product with the vector of derivatives, denoted by $(f^l)'\odot$, which is mathematically identical but better matches the internal representation of the derivatives as a vector, rather than a diagonal matrix.

f. Since matrix multiplication is linear, the derivative of multiplying by a matrix is just the matrix: $(Wx)' = W$.

g. One may notice that multi-layer neural networks use non-linear activation functions, so an example with linear neurons seems obscure. However, even though the error surface of multi-layer networks are much more complicated, locally they can be approximated by a paraboloid. Therefore, linear neurons are used for simplicity and easier understanding.

h. There can be multiple output neurons, in which case the error is the squared norm of the difference vector.

References

1. Goodfellow, Bengio & Courville 2016, p. 200 (https://www.deeplearningbook.org/contents/mlp.html#pf25), "The **back-propagation** algorithm (Rumelhart *et al.*, 1986a), often simply called **backprop**, ..."
2. Goodfellow, Bengio & Courville 2016, p. 200 (https://www.deeplearningbook.org/contents/mlp.html#pf25), "Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function"
3. Goodfellow, Bengio & Courville 2016, p. 214 (https://www.deeplearningbook.org/contents/mlp.html#pf33), "This table-filling strategy is sometimes called *dynamic programming*."
4. Goodfellow, Bengio & Courville 2016, p. 200 (https://www.deeplearningbook.org/contents/mlp.html#pf25), "The term back-propagation is often misunderstood as meaning the whole learning algorithm for multilayer neural networks. Backpropagation refers only to the method for computing the gradient, while other algorithms, such as stochastic gradient descent, is used to perform learning using this gradient."
5. Goodfellow, Bengio & Courville (2016, p. 217 (https://www.deeplearningbook.org/contents/mlp.html#pf36)–218), "The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called *reverse mode accumulation*."
6. Goodfellow, Bengio & Courville (2016, p. 221 (https://www.deeplearningbook.org/contents/mlp.html#pf39)), "Efficient applications of the chain rule based on dynamic programming began to appear in the 1960s and 1970s, mostly for control applications (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Dreyfus, 1973) but also for sensitivity analysis (Linnainmaa, 1976). ... The idea was finally developed in practice after being independently rediscovered in different ways (LeCun, 1985; Parker, 1985; Rumelhart *et al.*, 1986a). The book *Parallel Distributed Processing* presented the results of some of the first successful experiments with back-propagation in a chapter (Rumelhart *et al.*, 1986b) that contributed greatly to the popularization of back-propagation and initiated a very active period of research in multilayer neural networks."
7. Goodfellow, Bengio & Courville (2016, 6.5 Back-Propagation and Other Differentiation Algorithms (https://www.deeplearningbook.org/contents/mlp.html#pf25), pp. 200–220)
8. Ramachandran, Prajit; Zoph, Barret; Le, Quoc V. (2017-10-27). "Searching for Activation Functions". arXiv:1710.05941 (https://arxiv.org/abs/1710.05941) [cs.NE (https://arxiv.org/archive/cs.NE)].
9. Misra, Diganta (2019-08-23). "Mish: A Self Regularized Non-Monotonic Activation Function". arXiv:1908.08681 (https://arxiv.org/abs/1908.08681) [cs.LG (https://arxiv.org/archive/cs.LG)].
10. Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986a). "Learning representations by back-propagating errors". *Nature*. **323** (6088): 533–536. Bibcode:1986Natur.323..533R (https://ui.adsabs.harvard.edu/abs/1986Natur.323..533R). doi:10.1038/323533a0 (https://doi.org/10.1038%2F323533a0). S2CID 205001834 (https://api.semanticscholar.org/CorpusID:205001834).
11. Tan, Hong Hui; Lim, King Han (2019). "Review of second-order optimization techniques in artificial neural networks backpropagation". *IOP Conference Series: Materials Science and Engineering*. **495** (1): 012003. Bibcode:2019MS&E..495a2003T (https://ui.adsabs.harvard.edu/abs/2019MS&E..495a2003T). doi:10.1088/1757-899X/495/1/012003 (https://doi.org/10.1088%2F1757-899X%2F495%2F1%2F012003). S2CID 208124487 (https://api.semanticscholar.org/CorpusID:208124487).
12. Wiliamowski, Bogdan; Yu, Hao (June 2010). "Improved Computation for Levenberg–Marquardt Training" (https://www.eng.auburn.edu/~wilambm/pap/2010/Improved%20Computation%20for%20LM%20Training.pdf) (PDF). *IEEE Transactions on Neural Networks and Learning Systems*. **21** (6).
13. Martens, James (August 2020). "New Insights and Perspectives on the Natural Gradient Method" (https://arxiv.org/pdf/1412.1193.pdf) (PDF). *Journal of Machine Learning Research* (21). arXiv:1412.1193 (https://arxiv.org/abs/1412.1193).
14. Nielsen (2015), "[W]hat assumptions do we need to make about our cost function ... in order that backpropagation can be applied? The first assumption we need is that the cost function can be written as an average ... over cost functions ... for individual training examples ... The second assumption we make about the cost is that it can be written as a function of the outputs from the neural network ..."
15. LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey (2015). "Deep learning". *Nature*. **521** (7553): 436–444. Bibcode:2015Natur.521..436L (https://ui.adsabs.harvard.edu/abs/2015Natur.521..436L). doi:10.1038/nature14539 (https://doi.org/10.1038%2Fnature14539). PMID 26017442 (https://pubmed.ncbi.nlm.nih.gov/26017442). S2CID 3074096 (https://api.semanticscholar.org/CorpusID:3074096).
16. Buckland, Matt; Collins, Mark (2002). *AI Techniques for Game Programming*. Boston: Premier Press. ISBN 1-931841-08-X.
17. Rumelhart; Hinton; Williams (1986). "Learning representations by back-propagating errors" (http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf) (PDF). *Nature*. **323** (6088): 533–536. Bibcode:1986Natur.323..533R (https://ui.adsabs.harvard.edu/abs/1986Natur.323..533R). doi:10.1038/323533a0 (https://doi.org/10.1038%2F323533a0). S2CID 205001834 (https://api.semanticscholar.org/CorpusID:205001834).
18. Kelley, Henry J. (1960). "Gradient theory of optimal flight paths". *ARS Journal*. **30** (10): 947–954. doi:10.2514/8.5282 (https://doi.org/10.2514%2F8.5282).
19. Bryson, Arthur E. (1962). "A gradient method for optimizing multi-stage allocation processes". *Proceedings of the Harvard Univ. Symposium on digital computers and their applications, 3–6 April 1961*. Cambridge: Harvard University Press. OCLC 498866871 (https://www.worldcat.org/oclc/498866871).
20. Dreyfus, Stuart E. (1990). "Artificial Neural Networks, Back Propagation, and the Kelley-Bryson Gradient Procedure". *Journal of Guidance, Control, and Dynamics*. **13** (5): 926–928. Bibcode:1990JGCD...13..926D (https://ui.adsabs.harvard.edu/abs/1990JGCD...13..926D). doi:10.2514/3.25422 (https://doi.org/10.2514%2F3.25422).

21. Mizutani, Eiji; Dreyfus, Stuart; Nishio, Kenichi (July 2000). "On derivation of MLP backpropagation from the Kelley-Bryson optimal-control gradient formula and its application" (https://coeieor.wpengine.com/wp-content/uploads/2019/03/ijcnn2k.pdf) (PDF). Proceedings of the IEEE International Joint Conference on Neural Networks.
22. Schmidhuber, Jürgen (2015). "Deep learning in neural networks: An overview". *Neural Networks*. **61**: 85–117. arXiv:1404.7828 (https://arxiv.org/abs/1404.7828). doi:10.1016/j.neunet.2014.09.003 (https://doi.org/10.1016%2Fj.neunet.2014.09.003). PMID 25462637 (https://pubmed.ncbi.nlm.nih.gov/25462637). S2CID 11715509 (https://api.semanticscholar.org/CorpusID:11715509).
23. Schmidhuber, Jürgen (2015). "Deep Learning" (https://doi.org/10.4249%2Fscholarpedia.32832). *Scholarpedia*. **10** (11): 32832. Bibcode:2015SchpJ..1032832S (https://ui.adsabs.harvard.edu/abs/2015SchpJ..1032832S). doi:10.4249/scholarpedia.32832 (https://doi.org/10.4249%2Fscholarpedia.32832).
24. Dreyfus, Stuart (1962). "The numerical solution of variational problems" (https://doi.org/10.1016%2F0022-247x%2862%2990004-5). *Journal of Mathematical Analysis and Applications*. **5** (1): 30–45. doi:10.1016/0022-247x(62)90004-5 (https://doi.org/10.1016%2F0022-247x%2862%2990004-5).
25. Russell, Stuart; Norvig, Peter (1995). *Artificial Intelligence : A Modern Approach*. Englewood Cliffs: Prentice Hall. p. 578. ISBN 0-13-103805-2. "The most popular method for learning in multilayer networks is called Back-propagation. It was first invented in 1969 by Bryson and Ho, but was more or less ignored until the mid-1980s."
26. Bryson, Arthur Earl; Ho, Yu-Chi (1969). *Applied optimal control: optimization, estimation, and control*. Waltham: Blaisdell. OCLC 3801 (https://www.worldcat.org/oclc/3801).
27. Griewank, Andreas (2012). "Who Invented the Reverse Mode of Differentiation?". *Optimization Stories*. Documenta Matematica, Extra Volume ISMP. pp. 389–400. S2CID 15568746 (https://api.semanticscholar.org/CorpusID:15568746).
28. Seppo Linnainmaa (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 6–7.
29. Linnainmaa, Seppo (1976). "Taylor expansion of the accumulated rounding error". *BIT Numerical Mathematics*. **16** (2): 146–160. doi:10.1007/bf01931367 (https://doi.org/10.1007%2Fbf01931367). S2CID 122357351 (https://api.semanticscholar.org/CorpusID:122357351).
30. The thesis, and some supplementary information, can be found in his book, Werbos, Paul J. (1994). *The Roots of Backpropagation : From Ordered Derivatives to Neural Networks and Political Forecasting*. New York: John Wiley & Sons. ISBN 0-471-59897-6.
31. Griewank, Andreas; Walther, Andrea (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition* (https://books.google.com/books?id=xoiiLaRxcbEC). SIAM. ISBN 978-0-89871-776-1.
32. Dreyfus, Stuart (1973). "The computational solution of optimal control problems with time lag". *IEEE Transactions on Automatic Control*. **18** (4): 383–385. doi:10.1109/tac.1973.1100330 (https://doi.org/10.1109%2Ftac.1973.1100330).
33. Werbos, Paul (1982). "Applications of advances in nonlinear sensitivity analysis" (http://werbos.com/Neural/SensitivityIFIPSeptember1981.pdf) (PDF). *System modeling and optimization*. Springer. pp. 762–770.
34. Parker, D.B. (1985). "Learning Logic". Center for Computational Research in Economics and Management Science. Cambridge MA: Massachusetts Institute of Technology.
35. Hertz, John (1991). *Introduction to the theory of neural computation*. Krogh, Anders., Palmer, Richard G. Redwood City, Calif.: Addison-Wesley. p. 8. ISBN 0-201-50395-6. OCLC 21522159 (https://www.worldcat.org/oclc/21522159).
36. Anderson, James Arthur; Rosenfeld, Edward, eds. (1988). *Neurocomputing Foundations of research*. MIT Press. ISBN 0-262-01097-6. OCLC 489622044 (https://www.worldcat.org/oclc/489622044).
37. Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986b). "8. Learning Internal Representations by Error Propagation" (https://archive.org/details/paralleldistribu00rume). In Rumelhart, David E.; McClelland, James L. (eds.). *Parallel Distributed Processing : Explorations in the Microstructure of Cognition*. Vol. 1 : Foundations. Cambridge: MIT Press. ISBN 0-262-18120-7.
38. Alpaydin, Ethem (2010). *Introduction to Machine Learning* (https://books.google.com/books?id=4j9GAQAAIAAJ). MIT Press. ISBN 978-0-262-01243-0.
39. Wan, Eric A. (1994). "Time Series Prediction by Using a Connectionist Network with Internal Delay Lines". In Weigend, Andreas S.; Gershenfeld, Neil A. (eds.). *Time Series Prediction : Forecasting the Future and Understanding the Past*. Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis. Vol. 15. Reading: Addison-Wesley. pp. 195–217. ISBN 0-201-62601-2. S2CID 12652643 (https://api.semanticscholar.org/CorpusID:12652643).
40. Chang, Franklin; Dell, Gary S.; Bock, Kathryn (2006). "Becoming syntactic". *Psychological Review*. **113** (2): 234–272. doi:10.1037/0033-295x.113.2.234 (https://doi.org/10.1037%2F0033-295x.113.2.234). PMID 16637761 (https://pubmed.ncbi.nlm.nih.gov/16637761).
41. Janciauskas, Marius; Chang, Franklin (2018). "Input and Age-Dependent Variation in Second Language Learning: A Connectionist Account" (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6001481). *Cognitive Science*. **42**: 519–554. doi:10.1111/cogs.12519 (https://doi.org/10.1111%2Fcogs.12519). PMC 6001481 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6001481). PMID 28744901 (https://pubmed.ncbi.nlm.nih.gov/28744901).
42. Fitz, Hartmut; Chang, Franklin (2019). "Language ERPs reflect learning through prediction error propagation". *Cognitive Psychology*. **111**: 15–52. doi:10.1016/j.cogpsych.2019.03.002 (https://doi.org/10.1016%2Fj.cogpsych.2019.03.002). hdl:21.11116/0000-0003-474D-8 (https://hdl.handle.net/21.11116%2F0000-0003-474D-8). PMID 30921626 (https://pubmed.ncbi.nlm.nih.gov/30921626). S2CID 85501792 (https://api.semanticscholar.org/CorpusID:85501792).

Further reading

- Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). "6.5 Back-Propagation and Other Differentiation Algorithms" (https://www.deeplearningbook.org/contents/mlp.html#pf25). *Deep Learning* (http://www.deeplearningbook.org). MIT Press. pp. 200–220. ISBN 9780262035613.
- Nielsen, Michael A. (2015). "How the backpropagation algorithm works" (http://neuralnetworksanddeeplearning.com/chap2.html). *Neural Networks and Deep Learning* (http://neuralnetworksanddeeplearning.com). Determination Press.
- McCaffrey, James (October 2012). "Neural Network Back-Propagation for Programmers" (https://docs.microsoft.com/en-us/archive/msdn-magazine/2012/october/test-run-neural-network-back-propagation-for-programmers). *MSDN Magazine*.
- Rojas, Raúl (1996). "The Backpropagation Algorithm" (https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf) (PDF). *Neural Networks : A Systematic Introduction*. Berlin: Springer. ISBN 3-540-60505-3.

External links

- Backpropagation neural network tutorial at the Wikiversity
- Bernacki, Mariusz; Włodarczyk, Przemysław (2004). "Principles of training multi-layer neural network using backpropagation" (http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html).
- Karpathy, Andrej (2016). "Lecture 4: Backpropagation, Neural Networks 1" (https://www.youtube.com/watch?v=i94OvYb6noo&list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC&index=4). *CS231n*. Stanford University. Archived (https://ghostarchive.org/varchive/youtube/20211212/i94OvYb6noo) from the original on 2021-12-12 – via YouTube.
- "What is Backpropagation Really Doing?" (https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3). *3Blue1Brown*. November 3, 2017. Archived (https://ghostarchive.org/varchive/youtube/20211212/Ilg3gGewQ5U) from the

original on 2021-12-12 – via YouTube.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=1104872812"

This page was last edited on 17 August 2022, at 09:26 (UTC).