Backpropagation

From Wikipedia, the free encyclopedia

Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

Backpropagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method, although it is also used in some unsupervised networks such as autoencoders. It is a generalization of the delta rule to multi-layered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Backpropagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.

Contents

Motivation

The goal of any supervised learning algorithm is to find a function that best maps a set of inputs to its correct output. An example would be a simple classification task, where the input is an image of an animal, and the correct output would be the name of the animal. Some input and output patterns can be easily learned by single-layer neural networks (i.e. perceptrons). However, these single-layer perceptrons cannot learn some relatively simple patterns, such as those that are not linearly separable. For example, a human may classify an image of an animal by recognizing certain features such as the number of limbs, the texture of the skin (whether it is furry, feathered, scaled, etc.), the size of the animal, and the list goes on. A single-layer neural network however, must learn a function that outputs a label solely using the intensity of the pixels in the image. There is no way for it to learn any abstract features of the input since it is limited to having only one layer. A multi-layered network overcomes this limitation as it can create internal representations and learn different features in each layer.[1] The first layer may be responsible for learning the orientations of lines using the inputs from the individual pixels in the image. The second layer may combine the features learned in

the first layer and learn to identify simple shapes such as circles. Each higher layer learns more and more abstract features such as those mentioned above that can be used to classify the image. Each layer finds patterns in the layer below it and it is this ability to create internal representations that are independent of outside input that gives multi-layered networks their power. The goal and motivation for developing the backpropagation algorithm was to find a way 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.[1]

Summary

The backpropagation learning algorithm can be divided into two phases: propagation and weight update.

Phase 1: Propagation

Each propagation involves the following steps:

1. Forward propagation of a training pattern's input through the neural network in order to generate the propagation's output activations.
2. Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas (the difference between the input and output values) of all output and hidden neurons.

Phase 2: Weight update

For each weight-synapse follow the following steps:

1. Multiply its output delta and input activation to get the gradient of the weight.
2. Subtract a ratio (percentage) of the gradient from the weight.

This ratio (percentage) influences the speed and quality of learning; it is called the *learning rate*. The greater the ratio, the faster the neuron trains; the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.

Repeat phase 1 and 2 until the performance of the network is satisfactory.

Algorithm

Algorithm for a 3-layer network (only one hidden layer):

```
initialize network weights (often small random values)
do
   forEach training example ex
      prediction = neural-net-output(network, ex)  // forward pass
      actual = teacher-output(ex)
      compute error (prediction - actual) at the output units
      compute Δw_h for all weights from hidden layer to output layer  // backward pass
      compute Δw_i for all weights from input layer to hidden layer   // backward pass continued
      update network weights // input layer not modified by error estimate
until all examples classified correctly or another stopping criterion satisfied
return the network
```

As the algorithm's name implies, the errors propagate backwards from the output nodes to the input nodes. Technically speaking, backpropagation calculates the gradient of the error of the network regarding the network's modifiable weights.[2] This gradient is almost always used in a simple stochastic gradient descent algorithm to find weights that minimize the error. Often the term "backpropagation" is used in a more general

sense, to refer to the entire procedure encompassing both the calculation of the gradient and its use in stochastic gradient descent. Backpropagation usually allows quick convergence on satisfactory local minima for error in the kind of networks to which it is suited.

Backpropagation networks are necessarily multilayer perceptrons (usually with one input, one hidden, and one output layer). In order for the hidden layer to serve any useful function, multilayer networks must have non-linear activation functions for the multiple layers: a multilayer network using only linear activation functions is equivalent to some single layer, linear network. Non-linear activation functions that are commonly used include the logistic function, the softmax function, and the gaussian function.

The backpropagation algorithm for calculating a gradient has been rediscovered a number of times, and is a special case of a more general technique called automatic differentiation in the reverse accumulation mode.

It is also closely related to the Gauss–Newton algorithm, and is also part of continuing research in neural backpropagation.

Intuition

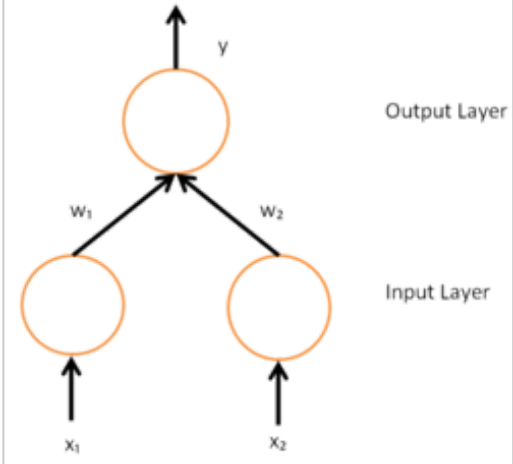
Learning as an optimization problem

Before showing the mathematical derivation of the backpropagation algorithm, it helps to develop some intuitions about the relationship between the actual output of a neuron and the correct output for a particular training case. Consider a simple neural network with two input units, one output unit and no hidden units. Each neuron uses a linear output[note 1] that is the weighted sum of its input.

Initially, before training, the weights will be set randomly. Then the neuron learns from training examples, which in this case consists 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 eventually produce given the identical inputs). The network given x_1 and x_2 will compute an output y which very likely differs from t (since the weights are initially random). A common method for measuring the discrepancy between the expected output t and the actual output y is using the squared error measure:

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

where E is the discrepancy or error.



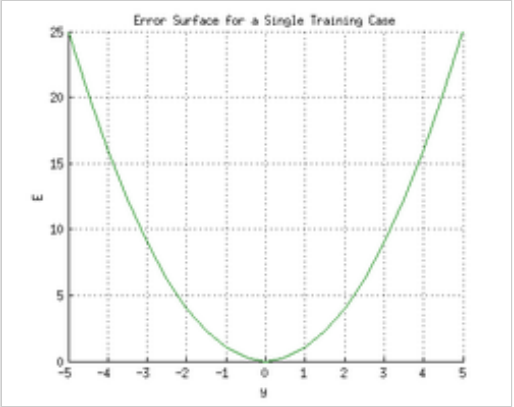
A simple neural network with two input units and one output unit

As an example, 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 actual output y is plotted on the x-axis against the error E on the y-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 x-axis, which means the error will be zero and the network can produce an output y that exactly matches the expected 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:

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

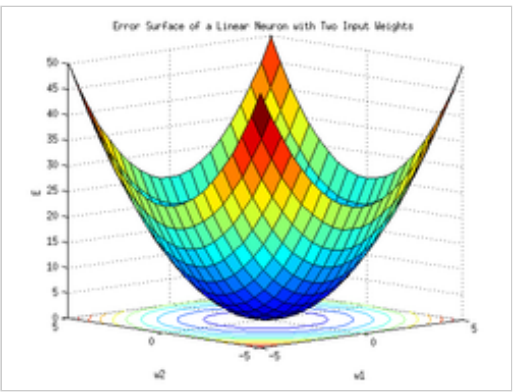
where w_1 and w_2 are the weights on the connection from the input units to the output unit. Therefore, the error also depends on the incoming weights to the neuron, which is ultimately what needs to be changed in the network to enable learning. If each weight is plotted on a separate horizontal axis and the error on the vertical axis, the result is a parabolic bowl (If a neuron has k weights, then the dimension of the error surface would be $k+1$, thus a $k+1$ dimensional equivalent of the 2D parabola).



Error surface of a linear neuron for a single training case.

The backpropagation algorithm aims to find the set of weights that minimizes the error. There are several methods for finding the minima of a parabola or any function in any dimension. One way is analytically by solving systems of equations, however this relies on the network being a linear system, and the goal is to be able to also train multi-layer, non-linear networks (since a multi-layered linear network is equivalent to a single-layer network). The method used in backpropagation is gradient descent.

An analogy for understanding gradient descent



Error surface of a linear neuron with two input weights

The basic intuition behind gradient descent can be illustrated by a hypothetical scenario. A person is stuck in the mountains is trying to get down (i.e. trying to find the minima). There is heavy fog such that visibility is extremely low. Therefore, the path down the mountain is not visible, so he must use local information to find the minima. He can use the method of gradient descent, which involves looking at the steepness of the hill at his current position, then proceeding in the direction with the most negative steepness (i.e. downhill). If he was trying to find the top of the mountain (i.e. the maxima), then he would proceed in the direction with most positive steepness (i.e. uphill). Using this method, he would eventually find his way down the mountain. However, assume also that the steepness of the hill is not immediately obvious with simple observation, but rather it requires a sophisticated instrument to measure, which the person happens to have at the moment. It takes quite some time to measure the steepness of the hill with the instrument, thus he should minimize his use of the instrument if he wanted to get down the mountain before sunset. The difficulty then is choosing the frequency at which he should measure the steepness of the hill so not to go off track.

In this analogy, the person represents the backpropagation algorithm, and the path down the mountain represents the set of weights that will minimize the error. The steepness of the hill represents the slope of the error surface at that point. The direction he must travel in corresponds to the gradient of the error surface at that point. The instrument used to measure steepness is differentiation (the slope of the error surface can be calculated by taking the derivative of the squared error function at that point). The distance he travels in between measurements (which is also proportional to the frequency as which he takes measurement) is the learning rate of the algorithm. See the limitation section for a discussion of the limitations of this type of "hill climbing" algorithm.

Derivation

Since backpropagation uses the gradient descent method, one needs to calculate the derivative of the squared error function with respect to the weights of the network. Assuming one output neuron,[note 2] the squared error function is:

$$E = \tfrac{1}{2}(t - y)^2,$$

where

E is the squared error,
t is the target output for a training sample, and
y is the actual output of the output neuron.

The factor of $\frac{1}{2}$ is included to cancel the exponent when differentiating. Later, the expression will be multiplied with an arbitrary learning rate, so that it doesn't matter if a constant coefficient is introduced now.

For each neuron j, its output o_j is defined as

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

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_{ij} denotes the weight between neurons i and j.

The activation function φ is in general non-linear and differentiable. A commonly used activation function is the logistic function:

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

which has a nice derivative of:

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

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:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_\text{j}} \frac{\partial \text{net}_\text{j}}{\partial w_{ij}}$$

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

$$\frac{\partial \text{net}_\text{j}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{k=1}^{n} w_{kj} x_k\right) = x_i.$$

The derivative of the output of neuron j with respect to its input is simply the partial derivative of the activation function (assuming here that the logistic function is used):

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

This is the reason why backpropagation requires the activation function to be differentiable.

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

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

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 of the inputs of all neurons $L = u, v, \ldots, w$ receiving input from neuron j,

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

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

$$\frac{\partial E}{\partial o_j} = \sum_{l \in L}\left(\frac{\partial E}{\partial \mathrm{net}_l}\frac{\partial \mathrm{net}_l}{\partial o_j}\right) = \sum_{l \in L}\left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial \mathrm{net}_l}w_{jl}\right)$$

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 one closer to the output neuron – are known.

Putting it all together:

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

with

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

To update the weight w_{ij} using gradient descent, one must choose a learning rate, α. The change in weight, which is added to the old weight, is equal to the product of the learning rate and the gradient, multiplied by -1:

$$\Delta w_{ij} = -\alpha\frac{\partial E}{\partial w_{ij}}$$

The -1 is required in order to update in the direction of a minimum, not a maximum, of the error function.

For a single-layer network, this expression becomes the Delta Rule. To better understand how backpropagation works, here is an example to illustrate it: The Back Propagation Algorithm (https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf), page 20.

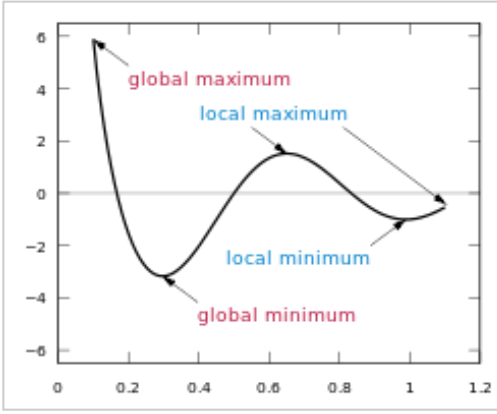
Modes of learning

There are three modes of learning to choose from: on-line, batch and stochastic. In on-line and stochastic learning, each propagation is followed immediately by a weight update. In batch learning, many propagations occur before updating the weights. On-line learning is used for dynamic environments that provide a continuous stream of new patterns. Stochastic learning and batch learning both make use of a training set of static patterns. Stochastic goes through the data set in a random order in order to reduce its chances of getting

stuck in local minima. Stochastic learning is also much faster than batch learning since weights are updated immediately after each propagation. Yet batch learning will yield a much more stable descent to a local minimum since each update is performed based on all patterns.

Limitations

- The result may converge to a local minimum. The "hill climbing" strategy of gradient descent is guaranteed to work if there is only one minimum. However, often the error surface has many local minima and maxima. If the starting point of the gradient descent happens to be somewhere between a local maximum and local minimum, then going down the direction with the most negative gradient will lead to the local minimum.
- The convergence obtained from backpropagation learning is very slow.
- The convergence in backpropagation learning is not guaranteed.
  - However, convergence to the global minimum is said to be guaranteed using the adaptive stopping criterion.[3]
- Backpropagation learning does not require normalization of input vectors; however, normalization could improve performance.[4]



Gradient descent can find the local minimum instead of the global minimum

History

Vapnik cites (Bryson, A.E.; W.F. Denham; S.E. Dreyfus. Optimal programming problems with inequality constraints. I: Necessary conditions for extremal solutions. AIAA J. 1, 11 (1963) 2544-2550) as the first publication of the backpropagation algorithm in his book "Support Vector Machines.". Arthur E. Bryson and Yu-Chi Ho described it as a multi-stage dynamic system optimization method in 1969.[5][6] It wasn't until 1974 and later, when applied in the context of neural networks and through the work of Paul Werbos,[7] David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams,[1][8] that it gained recognition, and it led to a “renaissance” in the field of artificial neural network research. During the 2000s it fell out of favour but has returned again in the 2010s, now able to train much larger networks using huge modern computing power such as GPUs. For example, in 2013 top speech recognisers now use backpropagation-trained neural networks.

Notes

1. 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.
2. There can be multiple output neurons, in which case the error is the squared norm of the difference vector.

See also

- Artificial neural network
- Biological neural network
- Catastrophic interference
- Ensemble learning
- AdaBoost
- Overfitting

References

1. Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". *Nature* **323** (6088): 533–536. doi:10.1038/323533a0 (https://dx.doi.org/10.1038%2F323533a0).
2. Paul J. Werbos (1994). The Roots of Backpropagation. From Ordered Derivatives to Neural Networks and Political Forecasting. New York, NY: John Wiley & Sons, Inc.
3. Lalis, Jeremias; Gerardo, Bobby; Byun, Yung-Cheol (2014). "An Adaptive Stopping Criterion for Backpropagation Learning in Feedforward Neural Network" (http://www.sersc.org/journals/IJMUE/vol9_no8_2014/13.pdf) (PDF). *International Journal of Multimedia and Ubiquitous Engineering* **9** (8): 149–156. doi:10.14257/ijmue.2014.9.8.13 (https://dx.doi.org/10.14257%2Fijmue.2014.9.8.13). Retrieved 17 March 2015.
4. ISBN 1-931841-08-X,
5. Stuart Russell; Peter Norvig. *Artificial Intelligence A Modern Approach*. p. 578. "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 largely ignored until the mid-1980s."
6. Arthur Earl Bryson, Yu-Chi Ho (1969). *Applied optimal control: optimization, estimation, and control*. Blaisdell Publishing Company or Xerox College Publishing. p. 481.
7. Paul J. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974
8. Alpaydın, Ethem (2010). *Introduction to machine learning* (2nd ed.). Cambridge, Mass.: MIT Press. p. 250. ISBN 978-0-262-01243-0. "...and hence the name *backpropagation* was coined (Rumelhart, Hinton, and Williams 1986a)."

External links

- A Gentle Introduction to Backpropagation - An intuitive tutorial by Shashi Sathyanarayana (http://numericinsight.com/uploads/A_Gentle_Introduction_to_Backpropagation.pdf) The article contains pseudocode ("Training Wheels for Training Neural Networks") for implementing the algorithm.
- Neural Network Back-Propagation for Programmers (a tutorial) (http://msdn.microsoft.com/en-us/magazine/jj658979.aspx)
- Backpropagation for mathematicians (http://www.matematica.ciens.ucv.ve/dcrespin/Pub/backprop.pdf)
- Chapter 7 The backpropagation algorithm (http://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf) of *Neural Networks - A Systematic Introduction* (http://page.mi.fu-berlin.de/rojas/neural/index.html.html) by Raúl Rojas (ISBN 978-3540605058)
- Implementation of BackPropagation in C++ (http://www.codeproject.com/KB/recipes/BP.aspx)
- Implementation of BackPropagation in C# (http://www.codeproject.com/KB/cs/BackPropagationNeuralNet.aspx)
- Implementation of BackPropagation in Java (https://github.com/guycole/BackProp1)
- Another Implementation of BackPropagation in Java (https://github.com/agibsonccc/java-deeplearning/blob/67ffee2c431f5fdbf3be9f393279c98caaa35f76/deeplearning4j-core/src/main/java/org/deeplearning4j/nn/BaseMultiLayerNetwork.java)
- Implementation of BackPropagation in Ruby (http://ai4r.org/neuralNetworks.html)
- Implementation of BackPropagation in Python (http://arctrix.com/nas/python/bpnn.py)
- Implementation of BackPropagation in PHP (http://freedelta.free.fr/r/php-code-samples/artificial-intelligence-neural-network-backpropagation/)
- Quick explanation of the backpropagation algorithm (http://www.tek271.com/documents/others/into-to-neural-networks)
- Graphical explanation of the backpropagation algorithm (http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html)
- Concise explanation of the backpropagation algorithm using math notation (http://pandamatak.com/people/anand/771/html/node37.html) by Anand Venkataraman
- Backpropagation neural network tutorial at the Wikiversity (http://en.wikiversity.org/wiki/Learning_and_Neural_Networks)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=674820920"

Categories: Machine learning algorithms | Artificial neural networks

---

- This page was last modified on 6 August 2015, at 10:35.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.