WIKIPEDIA

Backpropagation

Backpropagation algorithms are a family of methods used to efficiently train artificial neural networks (ANNs) following a gradient descent approach that exploits the chain rule. The main feature of backpropagation is its iterative, recursive and efficient method for calculating the weights updates to improve the network until it is able to perform the task for which it is being trained.[1] It is closely related to the Gauss–Newton algorithm.

Backpropagation requires the derivatives of activation functions to be known at network design time. Automatic differentiation is a technique that can automatically and analytically provide the derivatives to the training algorithm. In the context of learning, backpropagation is commonly used by the gradient descent optimization algorithm to adjust the weight of neurons by calculating the gradient of the loss function; backpropagation computes the gradient(s), whereas (stochastic) gradient descent uses the gradients for training the model (via optimization).

Contents

Motivation

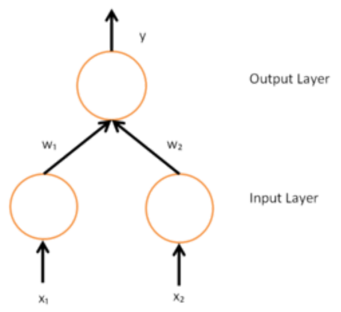
The goal of any supervised learning algorithm is to find a function that best maps a set of inputs to their correct output. The motivation for backpropagation is to train a multi-layered neural network such that it can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output.[2]

Intuition

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)[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 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 expected output t and the actual output y. For regression analysis problems the squared error can be used as a loss function, for classification the categorical crossentropy can be used.



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

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 actual output y is plotted on the horizontal axis against 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 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. For a neuron with k weights, the same plot would require an elliptic paraboloid of $k + 1$ dimensions.

One commonly used algorithm to find the set of weights that minimizes the error is gradient descent. Backpropagation is then used to calculate the steepest descent direction in an efficient way.

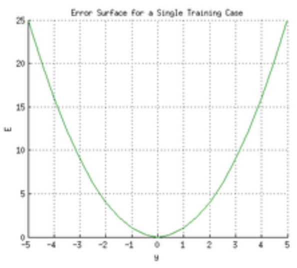
Derivation for a single-layered network

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,[note 2] the squared error function is

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

where

E 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.



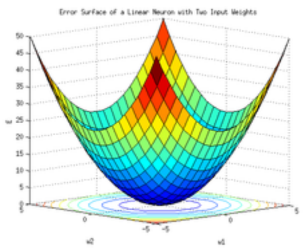
Error surface of a linear neuron for a single training case

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:

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



Error surface of a linear neuron with two input weights

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:

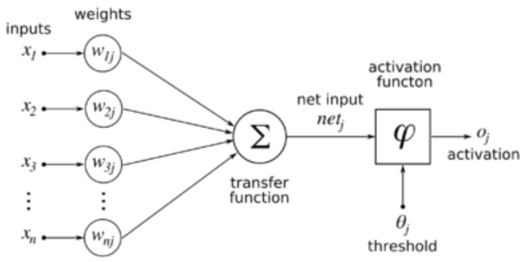


Diagram of an artificial neural network to illustrate the notation used here.

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

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

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

which for the logistic activation function case is:

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

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

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

If the logistic function is used as activation and square error 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,

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

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

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

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.

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

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

with

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

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

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

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:

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

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 back propagation.[3] 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:

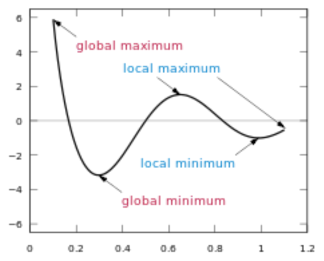
$$E = \frac{1}{2n} \sum_x \lVert (y(x) - y'(x)) \rVert^2$$

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.[4]
- Backpropagation learning does not require normalization of input vectors; however, normalization could improve performance.[5]

History

The basics of continuous backpropagation were derived in the context of control theory by Henry J. Kelley[6] in 1960 and by Arthur E. Bryson in 1961.[7][8][9][10][11][12] They used principles of dynamic programming. In 1962, Stuart Dreyfus published a simpler derivation based only on the chain rule.[13] Bryson and Ho described it as a multi-stage dynamic system optimization method in 1969.[14][15]



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

Backpropagation was derived by multiple researchers in the early 60's[16] and implemented to run on computers as early as 1970 by Seppo Linnainmaa[17][18][19] Examples of 1960s researchers include Arthur E. Bryson and Yu-Chi Ho in 1969.[20][21] 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 PhD Thesis.[22] In 1986, through the work of David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams,[23] and James McClelland,[24] backpropagation gained recognition.

While not applied to neural networks, in 1970 Linnainmaa published the general method for automatic differentiation (AD).[18][19] Although very controversial, some scientists believe this was actually the first step toward developing a back-propagation algorithm.[11][12][25][26]

In 1973 Dreyfus adapts parameters of controllers in proportion to error gradients.[27] In 1974 Werbos mentioned the possibility of applying this principle to artificial neural networks,[28] and in 1982 he applied Linnainmaa's AD method to non-linear functions.[12][29]

In 1986 Rumelhart, Hinton and Williams showed experimentally that this method can generate useful internal representations of incoming data in hidden layers of neural networks.[2][30] Yann LeCun, inventor of the Convolutional Neural Network architecture, proposed the modern form of the back-propagation learning algorithm for neural networks in his PhD thesis in 1987. But it is only much later, in 1993, that Wan was able to win[11] an international pattern recognition contest through backpropagation.[31]

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 language structure learning research, where the connectionist models using this algorithm have been able to explain a variety of phenomena related to first[32] and second language learning.[33]

See also

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

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.

References

1. Goodfellow, Ian; Bengio, Yoshua; Courville, Aaaron (2016) *Deep Learning*. MIT Press. p. 196. ISBN 9780262035613
2. Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". *Nature*. **323** (6088): 533–536. Bibcode:1986Natur.323..533R (http://adsabs.harvard.edu/abs/1986Natur.323..533R). doi:10.1038/323533a0 (https://doi.org/10.1038%2F323533a0).
3. Nielsen, Michael A. (2015-01-01). "Neural Networks and Deep Learning" (http://neuralnetworksanddeeplearning.com/chap2.html).
4. LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey (2015). "Deep learning". *Nature*. **521** (7553): 436–444. Bibcode:2015Natur.521..436L (http://adsabs.harvard.edu/abs/2015Natur.521..436L). doi:10.1038/nature14539 (https://doi.org/10.1038%2Fnature14539). PMID 26017442 (https://www.ncbi.nlm.nih.gov/pubmed/26017442).
5. Buckland, Matt; Collins, Mark. *AI Techniques for Game Programming*. ISBN 978-1-931841-08-5.
6. Kelley, Henry J. (1960). "Gradient theory of optimal flight paths". *ARS Journal*. **30** (10): 947–954. Bibcode:1960ARSJ...30.1127B (http://adsabs.harvard.edu/abs/1960ARSJ...30.1127B). doi:10.2514/8.5282 (https://doi.org/10.2514%2F8.5282).
7. Arthur E. Bryson (1961, April). A gradient method for optimizing multi-stage allocation processes. In Proceedings of the Harvard Univ. Symposium on digital computers and their applications.
8. 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 (http://adsabs.harvard.edu/abs/1990JGCD...13..926D). doi:10.2514/3.25422 (https://doi.org/10.2514%2F3.25422).
9. Stuart Dreyfus (1990). Artificial Neural Networks, Back Propagation and the Kelley-Bryson Gradient Procedure. J. Guidance, Control and Dynamics, 1990.
10. Mizutani, Eiji; Dreyfus, Stuart; Nishio, Kenichi (July 2000). "On derivation of MLP backpropagation from the Kelley-Bryson optimal-control gradient formula and its application" (http://queue.ieor.berkeley.edu/People/Faculty/dreyfus-pubs/ijcnn2k.pdf) (PDF). Proceedings of the IEEE International Joint Conference on Neural Networks.
11. 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://www.ncbi.nlm.nih.gov/pubmed/25462637).
12. Schmidhuber, Jürgen (2015). "Deep Learning" (http://www.scholarpedia.org/article/Deep_Learning#Backpropagation). *Scholarpedia*. **10** (11): 32832. Bibcode:2015SchpJ..1032832S (http://adsabs.harvard.edu/abs/2015SchpJ..1032832S). doi:10.4249/scholarpedia.32832 (https://doi.org/10.4249%2Fscholarpedia.32832).
13. Dreyfus, Stuart (1962). "The numerical solution of variational problems" (http://linkinghub.elsevier.com/retrieve/pii/0022247X62900045). *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).
14. Stuart Russell; Peter Norvig. *Artificial Intelligence A Modern Approach* (https://books.google.com/books?id=XS9CjwEACAAJ). p. 578. "The most popular method for learning in multilayer networks is called Back-propagation."
15. Bryson, A. E.; Yu-Chi, Ho (1 January 1975). *Applied Optimal Control: Optimization, Estimation and Control* (https://books.google.com/books?id=P4TKxn7qW5kC). CRC Press. ISBN 978-0-89116-228-5.
16. Schmidhuber, Jürgen (2015-01-01). "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://www.ncbi.nlm.nih.gov/pubmed/25462637).
17. Griewank, Andreas (2012). Who Invented the Reverse Mode of Differentiation?. Optimization Stories, Documenta Matematica, Extra Volume ISMP (2012), 389-400.
18. 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.
19. 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).
20. Stuart Russell and 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 more or less ignored until the mid-1980s."
21. Bryson, Arthur Earl; Ho, Yu-Chi (1969). *Applied optimal control: optimization, estimation, and control*. Blaisdell Publishing Company or Xerox College Publishing. p. 481.

22. Werbos, P. (1974). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University.
23. Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986-10-09). "Learning representations by back-propagating errors". *Nature*. **323** (6088): 533–536. Bibcode:1986Natur.323..533R (http://adsabs.harvard.edu/abs/1986Natur.323..533R). doi:10.1038/323533a0 (https://doi.org/10.1038%2F323533a0).
24. Rumelhart, David E; Mcclelland, James L (1986-01-01). *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations* (https://www.researchgate.net/publication/200033859).
25. "Who Invented the Reverse Mode of Differentiation? - Semantic Scholar" (https://www.semanticscholar.org/paper/Who-Invented-the-Reverse-Mode-of-Differentiation-Griewank-Trefethen/8ff0c546aff84566635f0a9a2e01feb3d6588c1c). *www.semanticscholar.org*. 2012. Retrieved 2017-08-04.
26. 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.
27. 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).
28. Werbos, Paul John (1975). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences* (https://books.google.com/books?id=z81XmgEACAAJ). Harvard University.
29. Werbos, Paul (1982). "Applications of advances in nonlinear sensitivity analysis". *System modeling and optimization* (http://werbos.com/Neural/SensitivityIFIPSeptember1981.pdf) (PDF). Springer. pp. 762–770.
30. Alpaydin, Ethem (2010). *Introduction to Machine Learning* (https://books.google.com/books?id=4j9GAQAAIAAJ). MIT Press. ISBN 978-0-262-01243-0.
31. Wan, Eric A. (1993). "Time series prediction by using a connectionist network with internal delay lines" (https://pdfs.semanticscholar.org/667c/2b372d3387011510a21cc7e0b267e36259dd.pdf) (PDF). SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY-PROCEEDINGS. p. 195.
32. 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://www.ncbi.nlm.nih.gov/pubmed/16637761).
33. Janciauskas, Marius; Chang, Franklin (2017-07-26). "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://www.ncbi.nlm.nih.gov/pubmed/28744901).

External links

- Neural Network Back-Propagation for Programmers (a tutorial) (http://msdn.microsoft.com/en-us/magazine/jj658979.aspx)
- Generalized Backpropagation (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)
- 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
- Visualization of a learning process using backpropagation algorithm (http://rrusin.blogspot.com/2016/03/learning-sinus-function-using-neural.html)
- Backpropagation neural network tutorial at the Wikiversity
- "What is Backpropagation Really Doing?" (https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3). *3Blue1Brown*. November 3, 2017 – via YouTube.