Matt Mazur

Home
About
Archives
Contact
Now
Projects

A Step by Step Backpropagation Example

Background

Backpropagation is a common method for training a neural network. There is no shortage of papers online that attempt to explain how backpropagation works, but few that include an example with actual numbers. This post is my attempt to explain how it works with a concrete example that folks can compare their own calculations to in order to ensure they understand backpropagation correctly.

If this kind of thing interests you, you should sign up for my newsletter where I post about AI-related projects that I'm working on.

Backpropagation in Python

You can play around with a Python script that I wrote that implements the backpropagation algorithm in this Github repo.

Backpropagation Visualization

For an interactive visualization showing a neural network as it learns, check out my Neural Network visualization.

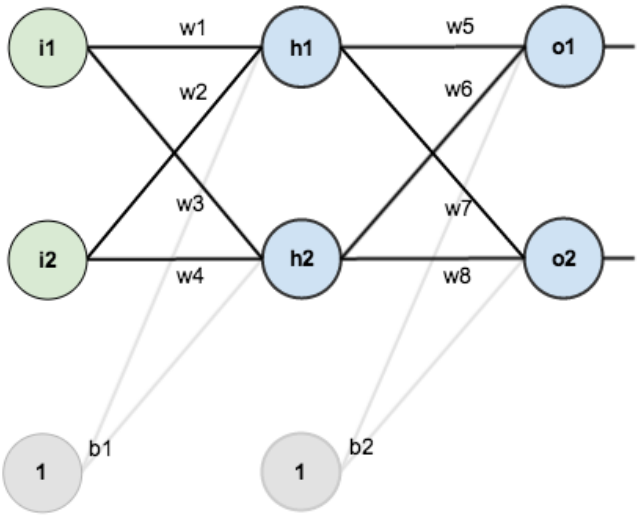
Additional Resources

If you find this tutorial useful and want to continue learning about neural networks and their applications, I highly recommend checking out Adrian Rosebrock's excellent tutorial on Getting Started with Deep Learning and Python.

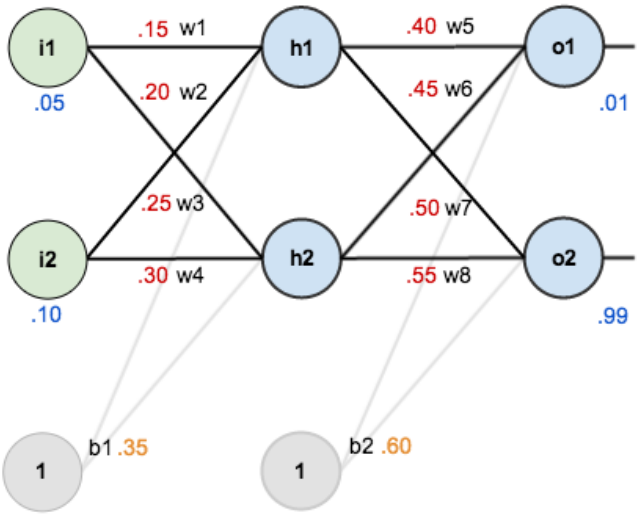
Overview

For this tutorial, we're going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.

Here's the basic structure:



In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs:



The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

For the rest of this tutorial we're going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

The Forward Pass

To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs

forward though the network.

We figure out the *total net input* to each hidden layer neuron, *squash* the total net input using an *activation function* (here we use the *logistic function*), then repeat the process with the output layer neurons.

> Total net input is also referred to as just *net input* by some sources.

Here's how we calculate the total net input for h_1:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here's the output for o_1:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

And carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$

Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

> Some sources refer to the target as the *ideal* and the output as the *actual*.

> The $\frac{1}{2}$ is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway so it doesn't matter that we introduce a constant here [1].

For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

Repeating this process for o_2 (remembering that the target is 0.99) we get:

$$E_{o2} = 0.023560026$$

The total error for the neural network is the sum of these errors:

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

The Backwards Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

Output Layer

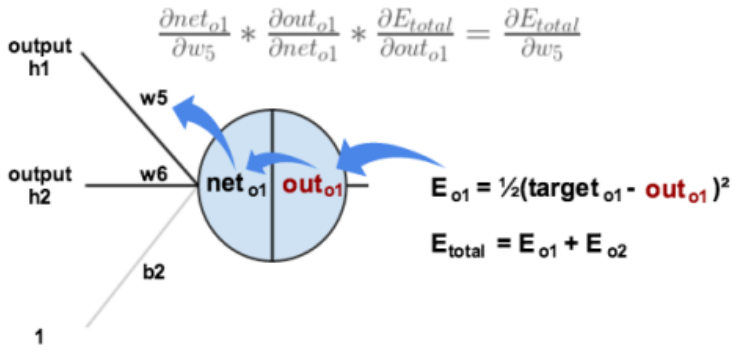
Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$.

> $\frac{\partial E_{total}}{\partial w_5}$ is read as "the partial derivative of E_{total} with respect to w_5". You can also say "the gradient with respect to w_5".

By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

Visually, here's what we're doing:



We need to figure out each piece in this equation.

First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$-(target - out)$ is sometimes expressed as $out - target$

When we take the partial derivative of the total error with respect to out_{o1}, the quantity $\frac{1}{2}(target_{o2} - out_{o2})^2$ becomes zero because out_{o1} does not affect it which means we're taking the derivative of a constant which is zero.

Next, how much does the output of o_1 change with respect to its total net input?

The partial derivative of the logistic function is the output multiplied by 1 minus the output:

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Finally, how much does the total net input of $o1$ change with respect to w_5?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

You'll often see this calculation combined in the form of the delta rule:

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

Alternatively, we have $\frac{\partial E_{total}}{\partial out_{o1}}$ and $\frac{\partial out_{o1}}{\partial net_{o1}}$ which can be written as $\frac{\partial E_{total}}{\partial net_{o1}}$, aka δ_{o1} (the Greek letter delta) aka the *node delta*. We can use this to rewrite the calculation above:

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

Therefore:

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

Some sources extract the negative sign from δ so it would be written as:

$$\frac{\partial E_{total}}{\partial w_5} = -\delta_{o1} out_{h1}$$

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

Some sources use α (alpha) to represent the learning rate, others use η (eta), and others even use ϵ (epsilon).

We can repeat this process to get the new weights w_6, w_7, and w_8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

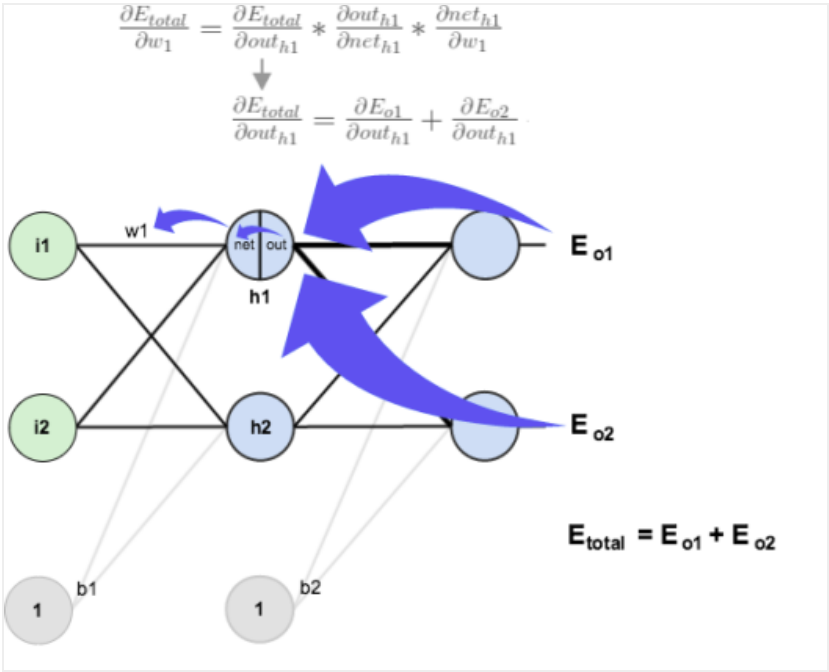
Hidden Layer

Next, we'll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

Big picture, here's what we need to figure out:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

Visually:



We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1} affects both out_{o1} and out_{o2} therefore the $\frac{\partial E_{total}}{\partial out_{h1}}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

We can calculate $\frac{\partial E_{o1}}{\partial net_{o1}}$ using values we calculated earlier:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

And $\frac{\partial net_{o1}}{\partial out_{h1}}$ is equal to w_5:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Plugging them in:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Now that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w}$ for each weight:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

You might also see this written as:

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \delta_o * w_{ho}) * out_{h1}(1 - out_{h1}) * i_1$$

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1$$

We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Repeating this for w_2, w_3, and w_4

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

If you've made it this far and found any errors in any of the above or can think of any ways to make it clearer for future readers, don't hesitate to drop me a note. Thanks!

Related

Experimenting with a Neural Network-based Poker Bot
In "Poker Bot"

The State of Emergent Mind
In "Emergent Mind"

I'm going to write more often. For real this time.
In "Writing"

Posted on March 17, 2015 by Mazur. This entry was posted in Machine Learning and tagged ai, backpropagation, machine learning, neural networks. Bookmark the permalink.

← Introducing ABTestCalculator.com, an Open Source A/B Test Significance Calculator

TetriNET Bot Source Code Published on Github →

557 thoughts on "A Step by Step Backpropagation Example"

← Older Comments

Pingback: My First Neural Net – Overthink

Pingback: BP神经网络 | Codeba

Atinesh

— September 25, 2017 at 8:50 am

Thanks for such a clear explanation of backprop

Reply

ZR

— September 30, 2017 at 9:24 am

Wonderful tutorial. I have a question with the initial calculation in the forward propagation. The calculation performed above states that the input for h1 is (0.15 * 0.05) + (0.2 * 0.1)+bias (weights 1 & 2). However, the graph, I think, shows (0.15 * 0.05) + (0.25 * 0.1)+bias (weights 1 & 3).

Have I looked at the graph incorrectly?

Thanks!

Reply

mrachid

— October 7, 2017 at 8:32 am

If you look at the first graph, the one without the weight values, you can see that w1 and w2 is weights for h1 and that w3 and w4 are weights for h2. The graph with the values may make us mix up w2 and w3, because of the labels disposition.

Reply

mrachid

— October 7, 2017 at 8:33 am

Great tutorial, btw :)

Reply

Laurent A.
— October 4, 2017 at 3:22 pm

I have found here an explanation on why the gradient for the output layer was simply “output – target”. Now, I got it. 1000 thanks for taking time to share :)

Reply

Vinay
— October 6, 2017 at 8:56 am

Very good explanation with nice example. Thank you very much!!!

Reply

Ping! A Deep Learning primer for all » Prithiviraj Damodaran

Anusha
— October 8, 2017 at 8:52 pm

Thnaks a ton for such a crystal clear explanation! Made my day :)

Reply

Asaf Zebulon
— October 9, 2017 at 10:39 pm

this is the best explanation about backpropagation that I’ve ever found. thanks!

Reply

christineshen
— October 10, 2017 at 3:29 am

Actually why is that? I don’t see the explanations here… to me the formula here is (output – target) * g'(z). And I still don’t know why I read in some other materials that the error term of the output layer is just (output – target).

Reply

Edu
— October 10, 2017 at 3:43 pm

Very nice explanation !
I hope you could clarify one thing. In your diagram, the same bias is applied to all perceptrons in a given layer (b1 is applied to h1 and h2 , b2 is applied to o1 and o2).
Is that the normal approach? Shouldn’t we have a bias per perceptron?

Thanks a lot !

Reply

Deadletter Grad

— October 18, 2017 at 3:16 am

It appears there are several ways of doing it. One way is a single bias setting, no weights, to all nodes.

Another is, as above, using one setting for each layer. Equal to that would be ONE bias (of say 1) but different weights to each layer.

Yet another is one setting, but unique weights to each node.

Since the only goal is to help the logistic or sigmoid function not pass through the origin, they all work – but I think the reason to have them be different is to avoid shifting the entire function to the right or left simultaneously at all nodes, forcing adjustments to be very miniscule. With different weights, it offers the chance to adjust different nodes in different ways.

This example doesn't include it, but some people also update the weights on the bias settings.

Reply

Kaushik

— October 10, 2017 at 10:05 pm

Awesome tutorial!

Reply

Dorin

— October 12, 2017 at 4:20 pm

best resource on understanding back propagation I have came across.

Reply

Priya

— October 14, 2017 at 6:34 am

Thank u for such a needful explanation

Reply

Waasala

— October 14, 2017 at 1:19 pm

This valuable explanation saved a lot of time...Thanks a lot for the very clear explanation of BP function.

Reply

henry

— October 14, 2017 at 7:21 pm

Thank you for the great tutorial!
When you do:
d(E_total) / d(out_o1) = 2 * 1 / 2 * (target_o1 – out_o1)^(2-1) * -1 + 0

Where does the part >> * -1<< come from ???

Reply

Alex

— October 18, 2017 at 2:22 am

@henry

(Fog)' = g' * (f'og)
With g being a-x g' is -1
F being (a-x)^2
F'og is 2 (a-x)

Reply

Mustafa Murat ARAT

— October 29, 2017 at 10:22 pm

You are getting the derivative with respect to output_o1. It is a variable with a negative sign, meaning (-1 * output_o1). If you take the derivative of a variable whose degree is 1 (a variable without an
exponent actually has an exponent of 1), you will have a constant which is -1 here.

Reply

Long le

— October 17, 2017 at 11:27 pm

Very clear tutorial. Thank you very much

Reply

Jakes

— October 21, 2017 at 3:58 pm

All examples I can find online shows an explanation of a single data set. How do you train an ANN with let's say two data sets, as such the weights for both data sets are the same, but it yields the correct results for each data set?

Reply

Ping! Long Short Term Memory (LSTM) – Vladislav EKT

Jonathan Woodbury
— October 22, 2017 at 9:50 pm

Thank you very much.

Reply

Emmanuel Tetteh
— October 24, 2017 at 6:46 am

Very insightful. Great work.

Reply

Nishant
— October 25, 2017 at 2:28 pm

Thank you so much ! Couldn't be better :))

Reply

Amit
— October 26, 2017 at 10:35 am

Awesome stuff. Exactly what beginners need.

Reply

Miguel MG
— October 27, 2017 at 6:41 pm

Very good explanation

Reply

Shashi Dhungel
— October 30, 2017 at 2:06 pm

This is a great tutorial. One thing that will help is to explain why the error rate only goes

down and why the error does not increase. What forces the error to go down in every iteration. Why should we expect it to go down.

I believe it has to do with the slope. When the slope is negative then it causes the learning rate to add and when the slope is negative it cases the learning rate to subtract from the original rate.

Reply

shashidhungel

— October 30, 2017 at 2:09 pm

This is a great tutorial. I think it will be clearer if you can explain why the error rate only goes down with each iteration (assuming there is only one equilibrium). How the slope directions contributes to ensuring that each iteration decreases the cost?

Reply

Mario A Vinasco

— October 31, 2017 at 1:27 pm

Matt, very good tutorial !!
Q: how do you apply back propagation on batches? lets say I feed forward 50 training examples from my data set, I calculate the errors for each, then do I take average Error and do the back prop? Thanks

Reply

mariovinasco

— October 31, 2017 at 1:30 pm

Matt, how does BackPropagation work with batches of training data? If I do 50 training samples per batch, do I take the average of the errors and do the back prop ? thanks

Reply

Yiting

— October 31, 2017 at 1:42 pm

Super clear explanation 👍

Reply

Sonya
— November 1, 2017 at 7:34 am

Thank you very much for this example, it has made the concept so much more clear!

Reply

adideshp
— November 2, 2017 at 8:46 am

Great Tutorial. Thanks a lot :)

Reply

Felix
— November 2, 2017 at 11:45 pm

I got a solution after 2 months learning.. :) Thank you very much..

Reply

Richard
— November 3, 2017 at 8:52 pm

Very Very good

Reply

Nasim Mahmud
— November 4, 2017 at 9:08 am

Hey there, I want to have an output value 2.5 but using the equations above I only get output value 1. What should I do now?? I have set the target value 2.5 but also get output value 1.

Reply

Burak Ipek
— November 4, 2017 at 6:49 pm

Thank you. I was studying for 2 weeks without understanding this. You saved me.

Reply

Sam

— November 5, 2017 at 6:29 pm

I'm stuck. How do I update the bias?

Reply

Kaezer

— November 5, 2017 at 9:13 pm

Thank you, Professor.

Reply

julianthefrank

— November 7, 2017 at 4:11 am

Reblogged this on Julian Frank's Blog and commented:
A Really Good Step by step Explanation of Back Propagation in Neurons

Reply

D-Frame

— November 9, 2017 at 4:03 am

Ignoring the whole backpropagation thing, I don't even understand how we could ever wind up with 0.01 as final output. Using the "squash" logistics function above, even when all inputs and weights are zero, we get 0.5 as output. You can't ever go below 0.5 this way. What am I missing?

Reply

← Older Comments

Leave a Reply

Enter your comment here...