Matt Mazur

About

Hey there! I'm the founder of Preceden, a web-based timeline maker, and a data analyst consultant. I also built Lean Domain Search and many other software products over the years.

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.

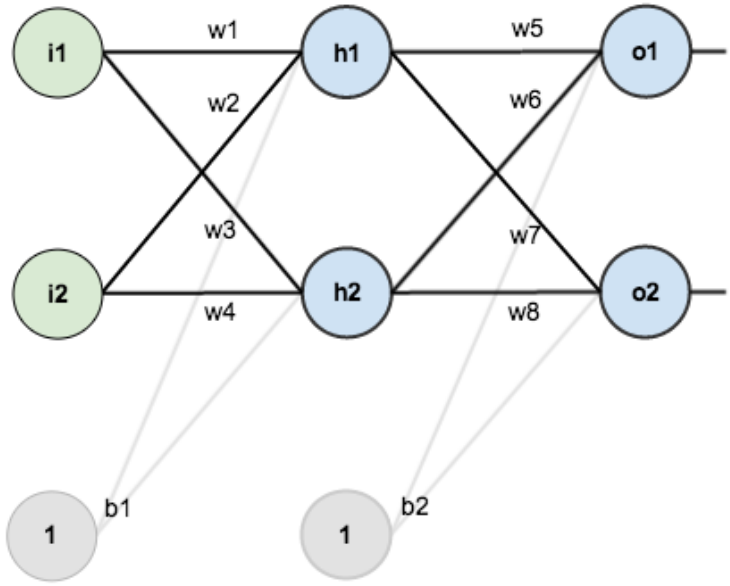
Additional Resources

If you find this tutorial useful and want to continue learning about neural networks, machine learning, and deep learning, I highly recommend checking out Adrian Rosebrock's new book, Deep Learning for Computer Vision with Python. I really enjoyed the book and will have a full review up soon.

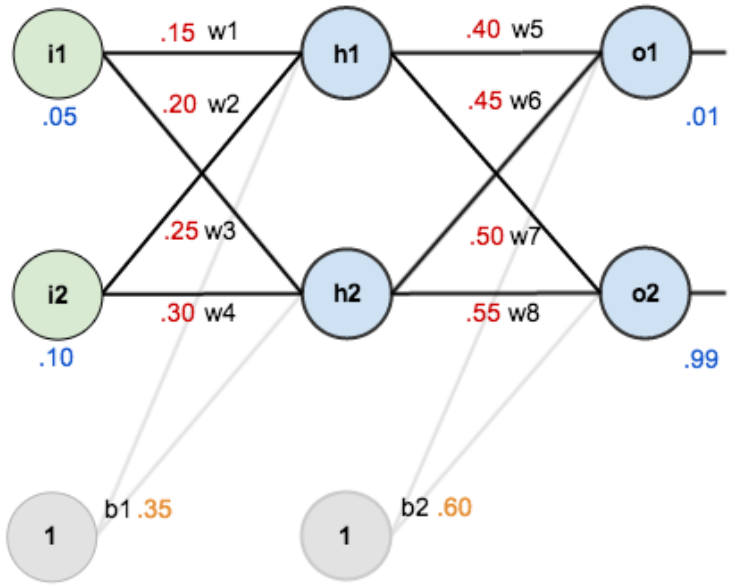
Overview

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

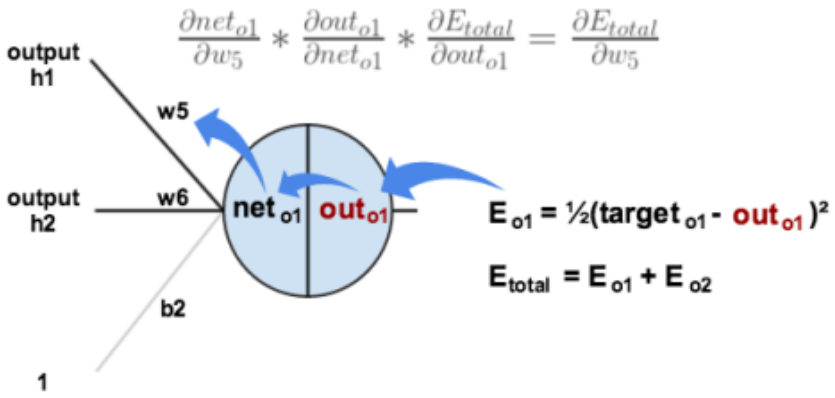
Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$.

$\frac{\partial E_{total}}{\partial w_5}$ 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).

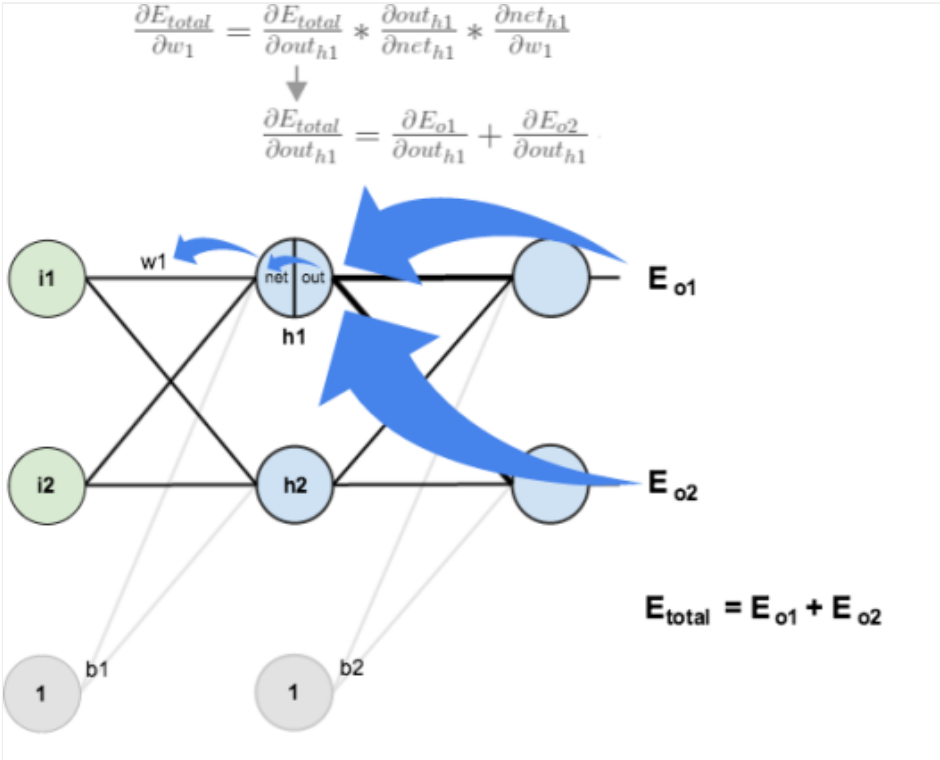
Hidden Layer

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:

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

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

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

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 →

839 thoughts on “A Step by Step Backpropagation Example”

← Older Comments

bharati
— December 5, 2018 at 2:04 am

nice explanation.

Reply

H.Taner Unal
— December 10, 2018 at 1:31 pm

Hey Matt. I’m stuck here: I got dEo2/douth1=-0.017144288. you got -0.019049. Can you check again?

Reply

Arindam
— December 13, 2018 at 3:43 am

Nicely explained. Thanks :)

Reply

Florian
— December 16, 2018 at 6:10 am

Very helpful. Thanks

Reply

Tansuluu
— December 19, 2018 at 7:26 am

Thank you a lot! Didn’t get on lecture.Here is the best explanation!

Reply

Loftur
— December 19, 2018 at 8:57 am

To get the net input of h1 you need to calculate w1*i1 + w3*i2 + b1*1. The weight w2 does not connect to h1

Reply

Braydon Burkhardt
— January 10, 2019 at 10:13 pm

It can look like that in the pictures, however h1 is in fact from w1 and w2. The weight # is created based off of the hidden layer. For example h1 has w1 and w2 while h2 has w3 and w4.

Reply

Mike
— December 20, 2018 at 6:15 am

If we want to extend this to a batch of training samples, do we calculate the output layer errors simply by averaging the error at each output over the training samples, then obtain the total by summing the averages?

e.g. E[O][1] = sum between 1 and N of E[O][1][n], where n is the training sample index and {1 <= n <= N} (Is that correct?)

Reply

Mike
— December 20, 2018 at 6:25 am

EDIT – forgot to divide sum by N

Reply

John
— December 20, 2018 at 4:35 pm

I want to thank you for this write up. Best one on the net! Your breakdown of the math is incredibly helpful

Reply

Oleg
— December 23, 2018 at 5:22 am

Don't we need to update b1 and b2?

Reply

Braydon Burkhardt

— January 10, 2019 at 10:07 pm

For really basic NN's, it is not needed and can be left alone. Think of the biases as shifting your graph while the weights change the slope. If you wish to update the biases, simply do the same thing as the weights but don't add the previous node as it is not attached to it. For example b2 is the same as w5, but without the hiddens (w5 would be h1) because it is not connected to any hiddens. b1 is the same as w1 but without the hidden stuff like h1(1-h1). Hope this helps, if not just leave another question.

Reply

Braydon Burkhardt

— January 10, 2019 at 10:09 pm

Sorry but I messed this up, the h1(1-h1) would be removed for b2 NOT b1, as b1 connects to hidden layer and the b2 does not. For b1 you would remove the input (i1) stuff.

Reply

Raghavender rao

— December 26, 2018 at 2:04 pm

This is really amazing.thankyou for the clear explanation.

Reply

Pingl Neuronal networking in VBA+Excel |

Raymond Zhang

— December 29, 2018 at 7:50 am

Excuse me. If I have 1000 training data, In bckpropagation, the loss function would be E=sum((target1-out1)^2+(target2-out2)^2+.....(target1000-out1000)^2), should partial derivative of the weights to E be divided by 1000(the number of data) when update the weights

Reply

Ghulamuddin Ansari

— December 29, 2018 at 10:50 pm

Wonderful explaination!, btw how to reduce biases at input and hidden layer?

Reply

Ping! Deep Learning Resources – Machine Learning

Fabio

— December 31, 2018 at 7:38 am

Bravo Matt, la migliore spiegazione!

Reply

Ping! A quick introduction to derivatives for machine learning people – Data Science Austria

prometheusgr

— January 6, 2019 at 5:49 pm

Hey Matt, I've been trying to duplicate your results in a network I am building, and running into an issue. Using your examples as my test data, I have passed all tests except for the weight values in w6 and w7. My values are not matching.

From a sniff test perspective, w6 appears it should be increasing since the output is lower than the expected (0.7 -> ~0.9) , but your calculation has the weight being reduced (0.45 -> ~0.41). With w7 your example has the weight increasing (0.5 -> ~0.51), but it appears it should be decreasing since the output is higher than expected (~0.7 -> 0.1).

I know this is an older post, but if you could double check those values and let me know, that would help me a lot.

:)

Reply

Maria

— January 7, 2019 at 7:11 am

Thank you Matt for your very good explanation! How do you create the figures of the basic structre, … etc. ?

Reply

amin

— January 10, 2019 at 4:13 am

Hi I need help, please have someone help me

Reply

Braydon Burkhardt

— January 10, 2019 at 10:01 pm

What is your problem?

Reply

sadikul haque

— January 15, 2019 at 8:24 am

that was very nice

Reply

Johannes G. Mooyman

— January 15, 2019 at 10:08 am

could the problem be solved linear programming ?

Reply

Ping! Introduction to Neural Networks – Glass Box Medicine

signalspa

— January 23, 2019 at 9:41 am

Wonderful explanation, sir!

Reply

Walter

— January 24, 2019 at 3:18 pm

I'm stuck at calculating w1. The procedure says to calculate dEtotal/dw1 we need dEo1/douth1, proceeds to use the chain rule, and substitutes 0.74136507 for dEo1/douto1. 0.74136507 was earlier given as the value of dEtotal/douto1. What am I missing? Why are the values the same? How is dEo2/douto1 calculated? Thanks in advance!

Reply

Walter

— January 26, 2019 at 7:46 am

I actually got it after a few repeat attempts. The key is, of course, in using the chain rule more to go deeper from d/dout to d/dnet. It's then possible to calculate everything.

Reply

Furkan

— January 31, 2019 at 10:23 am

why the w1 partial derivative formula is not,

i.hizliresim.com/nQlkpa.jpg

Am I wrong?

Reply

← Older Comments

Leave a Reply

Enter your comment here...