Matt Mazur

Home
About
Archives
Contact
Now
Projects

Follow via Email

Enter your email address to follow this blog and receive notifications of new posts by email.

Join 1,863 other followers

Enter your email address

Follow

About

I'm a developer at Automattic where I work on growth and analytics for WordPress.com. I also built Lean Domain Search, Preceden and a number of other software products over the years.

I love solving problems and helping others do the same. Drop me a note if I can help with anything.

Search …

Follow me on Twitter

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

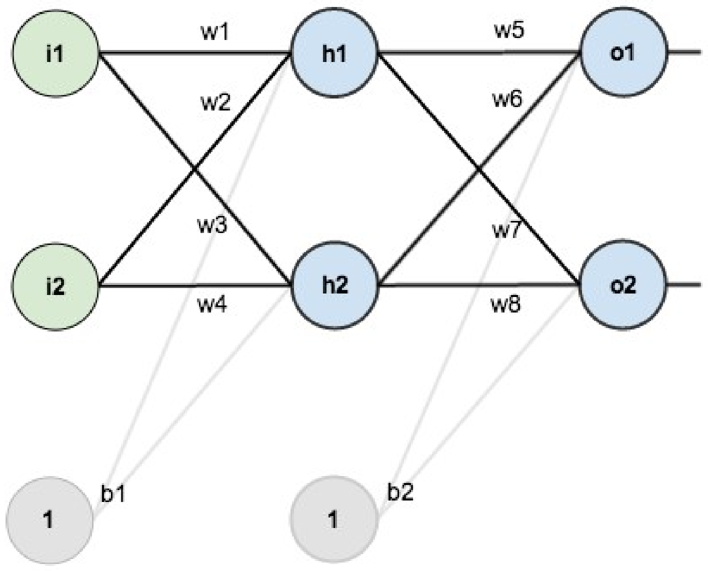
If you find this tutorial useful and want to continue learning about neural networks and their applications, I highly recommend checking out Adrian Rosebrock's excellent tutorial on Getting Started with Deep Learning and Python.

Overview

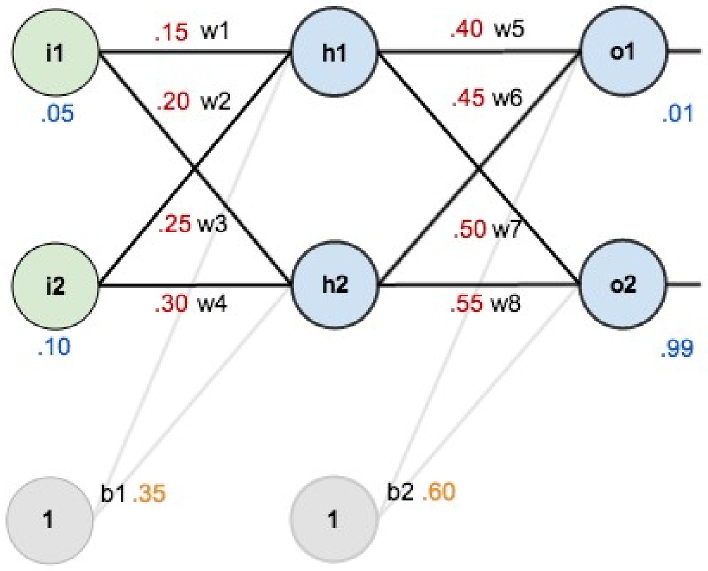
For this tutorial, we're going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.

Here's the basic structure:

Follow



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

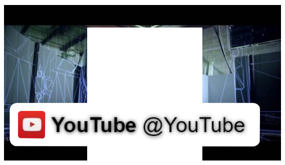


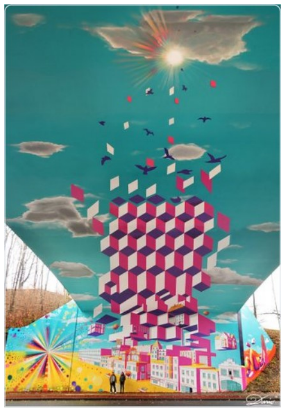
The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

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*.

Follow

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

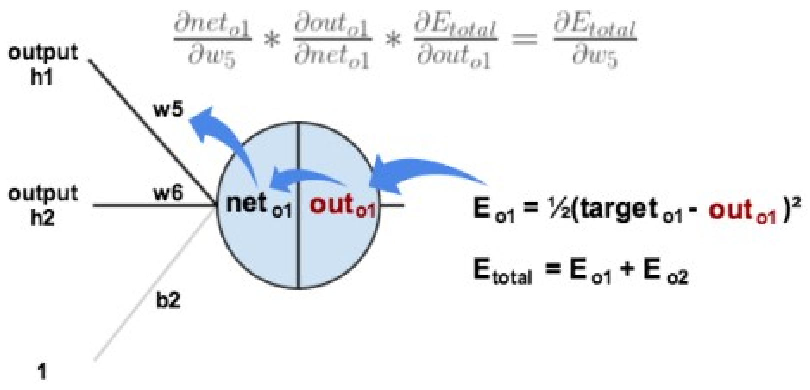
> $\frac{\partial E_{total}}{\partial w_5}$ is read as "the partial derivative of E_{total} with respect to w_5". You can also say "the gradient with respect to w_5".

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:

Follow


$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$
output h1
output h2
w5
w6
b2
1
net o1
out o1
$E_{o1} = ½(target_{o1} - out_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$

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:

Follow

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

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

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

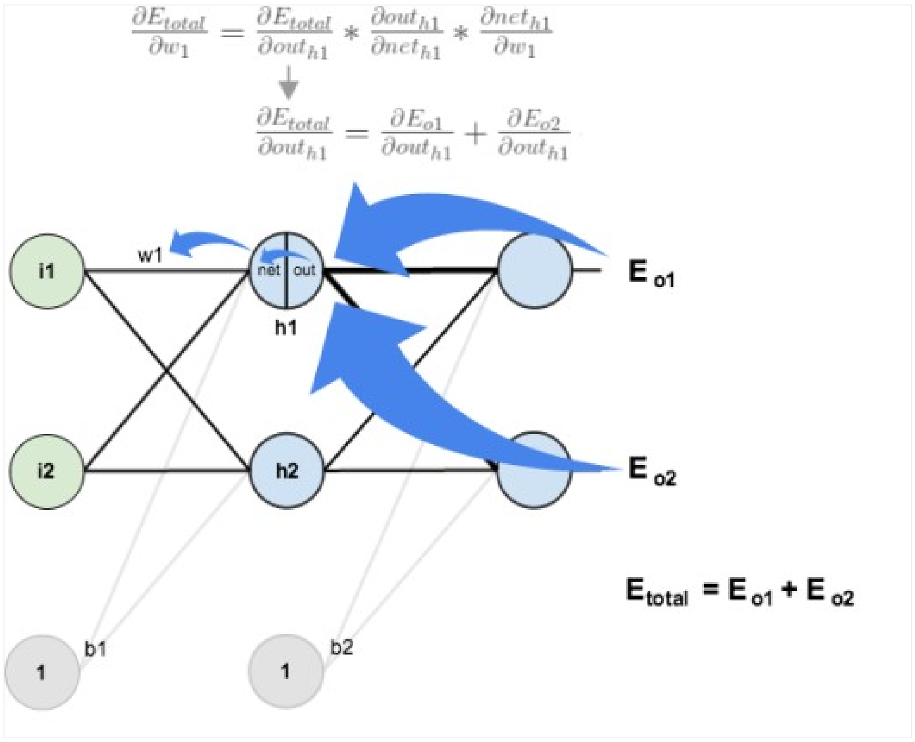
Follow

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:



$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$
$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$
i1
w1
net
out
h1
E o1
i2
h2
E o2
$E_{total} = E_{o1} + E_{o2}$
1
b1
1
b2

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:

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

Follow

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

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 $\frac{\partial E_{o2}}{\partial out_{o1}}$, 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:

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

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

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:
>
> $$\frac{\partial E_{total}}{\partial w_1} = \left(\sum\limits_{o}\frac{\partial E_{total}}{\partial out_{o}} * \frac{\partial out_{o}}{\partial net_{o}} * \frac{\partial net_{o}}{\partial out_{h1}}\right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$
>
> $$\frac{\partial E_{total}}{\partial w_1} = \left(\sum\limits_{o}\delta_{o} * w_{ho}\right) * out_{h1}(1 - out_{h1}) * i_1$$
>
> $$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1}i_1$$

We can now update w_1:

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

Follow

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 0.000035085. 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

The State of Emergent Mind
In "Emergent Mind"

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

Emergent Mind #10
In "Emergent Mind"

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 →

146 thoughts on "A Step by Step Backpropagation Example"

← Older Comments

Tarun

— March 25, 2016 at 2:37 pm

Follow

awesome article !! I have just one doubt…while calculating delta o1 , shouldn't we multiply the given equation by weight matrix of layer 2 ? I saw that in Andrew N.Gs videos he does this and am hence confused

Reply

Ping! Machine Learning : Resources | New Technology Information

Diego Domenech
— March 27, 2016 at 8:44 pm

Thanks a lot, best explanation I've seen about backpropagation

Reply

Lavanya
— March 28, 2016 at 11:47 am

Hey Matt, that was a very neat explanation! I am workin on BPA for SIMO system and tryin to implement using Simulink.

Is there any easy way to do in simulink?

Reply

111
— March 29, 2016 at 3:15 am

Thanks a ton!!!!!!

Reply

Trương Ngọc Quyền
— March 29, 2016 at 11:28 pm

thanks a lot for this lesson!!

Reply

Follow

Tariq

— March 30, 2016 at 5:00 pm

my guide is now published – it aims to make understanding how neural networks work as accessible as possible – with lots of diagrams, examples and discussion – there are even fun experiments to "see inside the mind of a neural network" and getting it all working on a Raspberry Pi Zero (£4 or $5)

Reply

Tariq

— March 30, 2016 at 5:01 pm

forgot the link!

http://www.amazon.co.uk/Make-Your-Own-Neural-Network-ebook/dp/B01DLHCW72/

Reply

Flipperty

— March 30, 2016 at 6:44 pm

I'll be buying your book–it's a great idea and well executed; but, I was hoping you could walk me through the differentiation of the total error function on pages 94-95 in the draft pdf version online. A minus that pops up that I can't account for. The same happens in the above post, and I'm just not sure why. Any hep you can provide is much appreciated.

Reply

Tariq

— March 30, 2016 at 8:58 pm

The minus sign pops up because d/dx (a-x) is -1.

In the example we have d/dx (a-x)^2 which is done in two steps. First the power rule to get 2*(a-x) but then the inside of the brackets too which gives you the extra linux sign.

This is actually the chain rule: df/dx = df/dy * dy/dx
so if f=(a-x)^2 and y=(a-x) then df/dx = df/dy * dy/dx = 2y * (-1) = (a-x) * (-1)

=================

Some people also get caught out by the weight updates. The weights are changed in proportion to – dE/dw .. that minus is there so that the weights go in the opposite direction of the gradient. My bc

Follow

trates this too with pictures.

Reply

Flipperty

— March 30, 2016 at 9:04 pm

Thank you! That's what I thought, but I just needed confirmation. A lot of people don't mention the chain rule, so that's what was throwing me.

Your book is excellent, and I bought a copy.

Everybody should buy a copy!

I hope it does really well.

Tariq

— March 30, 2016 at 9:07 pm

Thanks Flipperty – if you like it, please do leave a review on amazon. And if you have suggestions for improvement, do let me know too!

Amitesh

— April 3, 2016 at 11:11 am

Hi Matt! Great article,probably the best I've come across on the internet for Backpropagation. Just noticed a small error – When you calculate outo1 for the first time in the forward pass, the formula should be 1/1-(e^-neto1) whereas you have written it to be 1/1-(e^neth1). That's it! Amazing article once again. You've got me hooked on to NNets now and earned yourself a worshipper :)

Reply

Mazur

— April 23, 2016 at 7:44 pm

Thank you (and the others) for pointing this out. Fixed!

Reply

katysei

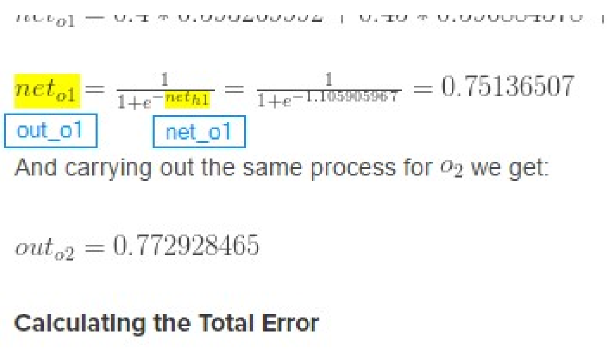
— April 18, 2016 at 2:40 am

Follow

Hi Mat, why aren't you putting weights on biases and update them during back prop?
thanks
joseph

Reply

Mengü Nazlı
— April 27, 2016 at 2:03 pm



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

$net_{o1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$

out_o1 net_o1

And carrying out the same process for o_2 we get:

$out_{o2} = 0.772928465$

Calculating the Total Error

I still see the typo here, is there a revised version of this page or you just missed it?

Reply

Mazur
— April 29, 2016 at 9:45 pm

Thanks, that was very helpful. Fixed!

Reply

Dara
— May 6, 2016 at 12:48 pm

Hi Mazur, this is great. I have question about this.
Doesn't it map the training output before calculate the error with target value?. Because "logistic function" outputs are always between 0 and 1. Then what happen if my target is really high (eg:300)? error would be really high?. After finish training, my target is 300 and final output would be somewhere between 0 and 1, since transfer function at the output node is "logistic". That is the part which I still cannot understand. even after training, doesn't it map the output to be large (should be taken near to 300 from 0-1)?

Reply

@myoneuralnet
— May 6, 2016 at 1:40 pm

You should map your desired outputs to match the activation function – otherwise you will drive the network to saturation. The same is also true of the inputs – you should try to optimise them to that they match the region of greatest variation of the activation function .. some people call this normalisation.

Follow

intro to neural networks has a section on this .. http://www.amazon.com/dp/B01EER4Z4G

Reply

John

— May 7, 2016 at 11:05 am

Hey Mazur, great tutorial, as said earlier, the best on Internet. Just have a question, when you have more hidden layers, the concept for the feedforward is the same, but the derivatives surely will modify, my question is how to calculate it? It will be something like Dtotal = DTotal/Dout * Dout/Dnet * Dnet/Dweight * Dweight/Douthidden * Douthidden/Dnethidden * Dnethidden/Dweight? Don't know if you will understand the way i wrote it xD Thanks in advance.

Reply

roberto brunialti

— May 9, 2016 at 2:38 am

Thanks Matt for the great article. Even me, a math dummy, was able to implement the back prop algorithm in C++…
I'm worring abaout generalization of this algorithm: is it viable for networks with more than an hidden layer? It seems to me that the hidden part of the algorithm
is specific for jus one layer.

Reply

mehmet

— May 11, 2016 at 5:46 pm

This is wonderful. Thks a lot.

Reply

← Older Comments

Leave a Reply

Blog at WordPress.com. The Publish Theme.

Follow

Follow