Grzegorz Gwardys

Experience with Machine/Deep Learning and more …

Convolutional Neural Networks backpropagation: from intuition to derivation

ON APRIL 22, 2016JANUARY 14, 2017 / BY GRZEGORZGWARDYS / IN EXPLANATION

Disclaimer: It is assumed that the reader is familiar with terms such as Multilayer Perceptron, delta errors or backpropagation. If not, it is recommended to read for example a chapter 2 (http://neuralnetworksanddeeplearning.com/chap2.html) of free online book 'Neural Networks and Deep Learning' by Michael Nielsen (http://michaelnielsen.org/).

Convolutional Neural Networks (CNN) are now a standard way of image classification – there are publicly accessible deep learning frameworks, trained models and services. It's more time consuming to install stuff like caffe (http://caffe.berkeleyvision.org/) than to perform state-of-the-art object classification or detection. We also have many methods of getting knowledge -there is a large number of deep learning courses (http://cs224d.stanford.edu/)/MOOCs (https://www.udacity.com/course/deep-learning--ud730), free e-books (http://www.deeplearningbook.org/)or even direct ways of accessing to the strongest Deep/Machine Learning minds such as Yoshua Bengio (https://plus.google.com/+YoshuaBengio/posts), Andrew NG (https://www.quora.com/session/Andrew-Ng/1)or Yann Lecun (https://www.facebook.com/yann.lecun?fref=ts) by Quora, Facebook or G+.

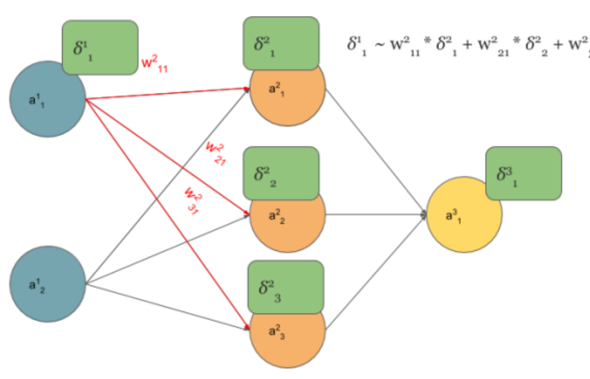
Nevertheless, when I wanted to get deeper insight in CNN, I could not find a "CNN backpropagation for dummies". Notoriously I met with statements like: "If you understand backpropagation in standard neural networks, there should not be a problem with understanding it in CNN" or "All things are nearly the same, except matrix multiplications are replaced by convolutions". And of course I saw tons of ready equations.

It was a little consoling, when I found out that I am not alone, for example: Hello, when computing the gradients CNN, the weights need to be rotated, Why ? (https://plus.google.com/111541909707081118542/posts/P8bZBNpg84Z)

$$\delta_j^\ell = f'(\mathbf{u}_j^\ell) \circ \mathrm{conv2}(\delta_j^{\ell+1},\ \mathrm{rot180}(\mathbf{k}_j^{\ell+1}),\ \text{'full'}).$$

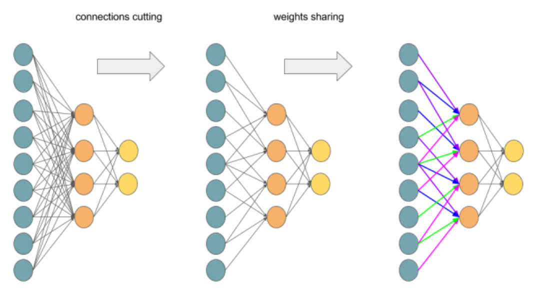
The answer on above question, that concerns the need of rotation on weights in gradient computing, will be a result of this long post.

We start from multilayer perceptron and counting delta errors on fingers:



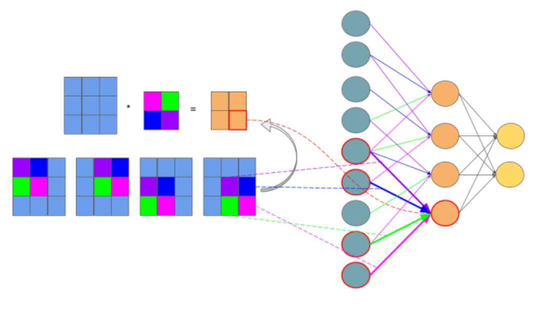
We see on above picture that δ_1^1 is proportional to deltas from next layer that are scaled by weights.

But how do we connect concept of MLP with Convolutional Neural Network? Let's play with MLP:



Transforming Multilayer Perceptron to Convolutional Neural Network

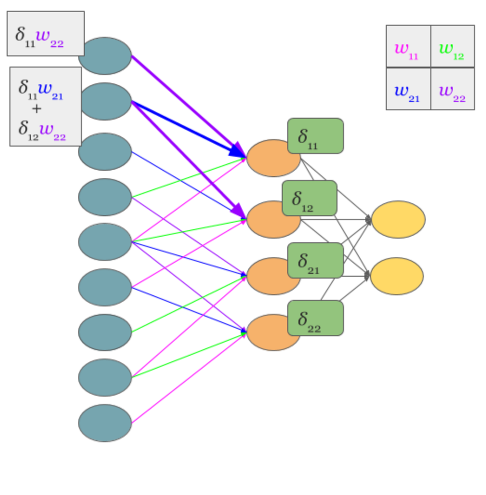
If you are not sure that after connections cutting and weights sharing we get one layer Convolutional Neural Network, I hope that below picture will convince you:



Feedforward in CNN is identical with convolution operation

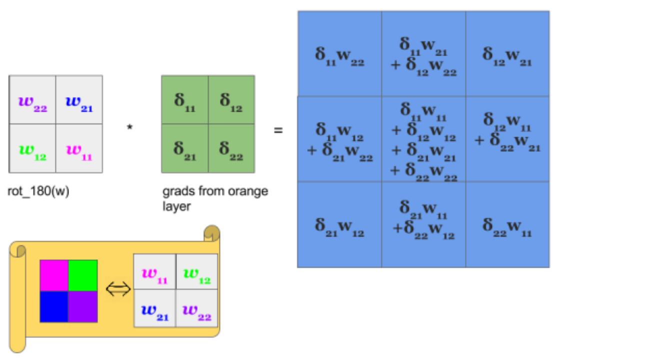
The idea behind this figure is to show, that such neural network configuration is identical with a 2D convolution operation and weights are just filters (also called kernels, convolution matrices, or masks).

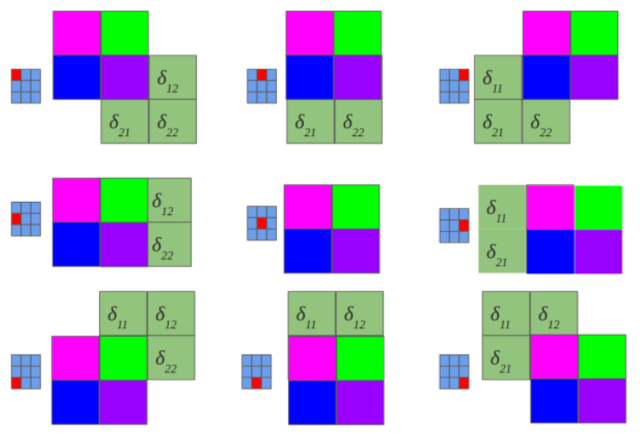
Now we can come back to gradient computing by counting on fingers, but from now we will be only focused on CNN. Let's begin:



Backpropagation also results with convolution

No magic here, we have just summed in “blue layer” scaled by weights gradients from “orange” layer. Same process as in MLP’s backpropagation. However, in the standard approach we talk about dot products and here we have … yup, again convolution:





Yeah, it is a bit different convolution than in previous (forward) case. There we did so called valid convolution, while here we do a full convolution (more about nomenclature here (http://www.johnloomis.org/ece563/notes/filter/conv/convolution.html)). What is more, we rotate our kernel by 180 degrees. But still, we are talking about convolution!

Now, I have some good news and some bad news:

1. you see (BTW, sorry for pictures aesthetics 🙂), that matrix dot products are replaced by convolution operations both in feed forward and backpropagation.
2. you know that seeing something and understanding something … yup, we are going now to get our hands dirty and prove above statement 🙂 before getting next, I recommend to read, mentioned already in the disclaimer, *chapter 2 (http://neuralnetworksanddeeplearning.com/chap2.html) of M. Nielsen book. I tried to make all quantities to be consistent with work of Michael.*

In the standard MLP, we can define an error of neuron j as:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

where z_j^l is just:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

and for clarity, $a_j^l = \sigma(z_j^l)$, where σ is an activation function such as sigmoid, hyperbolic tangent or relu (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)).

But here, we do not have MLP but CNN and matrix multiplications are replaced by convolutions as we discussed before. So instead of z_j we do have a $z_{x,y}$:

$$z_{x,y}^{l+1} = w^{l+1} * \sigma(z_{x,y}^l) + b_{x,y}^{l+1} = \sum_a \sum_b w_{a,b}^{l+1} \sigma(z_{x-a,y-b}^l) + b_{x,y}^{l+1}$$

Above equation is just a convolution operation during feedforward phase illustrated in the above picture titled ‘Feedforward in CNN is identical with convolution operation’

Now we can get to the point and answer the question Hello, when computing the gradients CNN, the weights need to be rotated, Why ? (https://plus.google.com/111541909707081118542/posts/P8bZBNpg84Z)

We start from statement:

$$\delta_{x,y}^l = \frac{\partial C}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \frac{\partial C}{\partial z_{x',y'}^{l+1}} \frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l}$$

We know that $z_{x,y}^l$ is in relation to $z_{x',y'}^{l+1}$ which is indirectly showed in the above picture titled ‘Backpropagation also results with convolution’. So sums are the result of chain rule. Let’s move on:

$$\frac{\partial C}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \frac{\partial C}{\partial z_{x',y'}^{l+1}} \frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} \frac{\partial(\sum_a \sum_b w_{a,b}^{l+1} \sigma(z_{x'-a,y'-b}^l) + b_{x',y'}^{l+1})}{\partial z_{x,y}^l}$$

First term is replaced by definition of error, while second has become large because we put it here expression on $z_{x',y'}^{l+1}$. However, we do not have to fear of this big monster – all components of sums equal 0, except these ones that are indexed: $x = x' - a$ and $y = y' - b$. So:

$$\sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} \frac{\partial(\sum_a \sum_b w_{a,b}^{l+1} \sigma(z_{x'-a,y'-b}^l) + b_{x',y'}^{l+1})}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} w_{a,b}^{l+1} \sigma'(z_{x,y}^l)$$

If $x = x' - a$ and $y = y' - b$ then it is obvious that $a = x' - x$ and $b = y' - y$ so we can reformulate above equation to:

$$\sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} w_{a,b}^{l+1} \sigma'(z_{x,y}^l) = \sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} w_{x'-x,y'-y}^{l+1} \sigma'(z_{x,y}^l)$$

OK, our last equation is just …

$$\sum_{x'}\sum_{y'}\delta^{l+1}_{x',y'}w^{l+1}_{x'-x,y'-y}\sigma'(z^l_{x,y}) = \delta^{l+1} * w^{l+1}_{-x,-y}\sigma'(z^l_{x,y})$$

Where is the rotation of weights? Actually $ROT180(w^{l+1}_{x,y}) = w^{l+1}_{-x,-y}$.

So **the answer** on question Hello, when computing the gradients CNN, the weights need to be rotated, Why ? (https://plus.google.com/111541909707081118542/posts/P8bZBNpg84Z) is simple: **the rotation of the weights just results from derivation of delta error in Convolution Neural Network**.

OK, we are really close to the end. One more ingredient of backpropagation algorithm is update of weights $\frac{\partial C}{\partial w^l_{a,b}}$:

$$\frac{\partial C}{\partial w^l_{a,b}} = \sum_x\sum_y \frac{\partial C}{\partial z^l_{x,y}}\frac{\partial z^l_{x,y}}{\partial w^l_{a,b}} = \sum_x\sum_y \delta^l_{x,y}\frac{\partial(\sum_{a'}\sum_{b'} w^l_{a',b'}\sigma(z^l_{x-a',y-b'})+b^l_{x,y})}{\partial w^l_{a,b}} = \sum_x\sum_y \delta^l_{x,y}\sigma(z^{l-1}_{x-a,y-b}) = \delta^l_{a,b}*\sigma(z^{l-1}_{-a,-b}) = \delta^l_{a,b}*\sigma(ROT180(z^{l-1}_{a,b}))$$

So paraphrasing the backpropagation algorithm (http://neuralnetworksanddeeplearning.com/chap2.html#the_backpropagation_algorithm) for CNN:

1. Input x: set the corresponding activation a^1 for the input layer.
2. Feedforward: for each l = 2,3, …,L compute $z^l_{x,y} = w^l * \sigma(z^{l-1}_{x,y}) + b^l_{x,y}$ and $a^l_{x,y} = \sigma(z^l_{x,y})$
3. Output error δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. Backpropagate the error: For each l=L-1,L-2,…,2 compute $\delta^l_{x,y} = \delta^{l+1} * ROT180(w^{l+1}_{x,y})\sigma'(z^l_{x,y})$
5. Output: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{a,b}} = \delta^l_{a,b} * \sigma(ROT180(z^{l-1}_{a,b}))$

The end 🙂

CNN CONVOLUTIONAL NEURAL NETWORKS DEEP LEARNING DERIVATION INTUITION

29 thoughts on “Convolutional Neural Networks backpropagation: from intuition to derivation”

1. **knmuged**
Reblogged this on mugedblog and commented:
Kolejny post Grześka, tym razem o Konwolucyjnych Sieciach Neuronowych!

   APRIL 25, 2016 AT 5:32 AM ↪ REPLY
2. **easton**
It is awesome!
Weeks ago, I met the same puzzle like yours. It is true that few explanations are made on it during the online resources,yet there is still some, like:
http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/
This blog let me figure out how to use the BP in Conv(though the notations used in it are totally different with which in Michael Nielsen’s book, which is awkward…).

   What’s more, using the toolkits designed for ML like Theano makes the backpropgation of delta no more a question. Theano use the method called ‘auto differentiation’, which can get the partial derivatiations from the Cost with respect to w&b just in one line code. I think that’s why most tutorials jump over BP in non-fully-connected-NN: there is no need to calculate derivatiation manually.

   MAY 6, 2016 AT 1:56 AM ↪ REPLY
   grzegorzgwardys
   I completely agree with you! Automatic differentiation is a big switch, not only in Deep Learning (for example PyMC 3 for Bayesian statistical modeling, that is based on Theano). However, I like to fully understand the matter and I hope that this blog post would be helpful for learners 🙂

   MAY 6, 2016 AT 1:47 PM ↪ REPLY
3. **nellaivijay**
Reblogged this on My Blog.

   MAY 8, 2016 AT 11:32 AM ↪ REPLY
4. **Francky**
thank you! there is not much well explained backpropagation example on the net.

   JUNE 9, 2016 AT 7:25 AM ↪ REPLY
5. Pingback: Convolutional Autoencoder for Dummies – Grzegorz Gwardys

6. **Vamsi Parasa**
Amazing! Your's is the best explanation so far i found on the internet!!
Would it be please possible to upload a youtube video of your derivation, please?
Thanks!

 AUGUST 8, 2016 AT 10:08 AM ↪ REPLY
grzegorzgwardys
Thank you for kind words 🙂 Youtube video … interesting concept, maybe some day 😉

 NOVEMBER 4, 2016 AT 6:02 PM ↪ REPLY

7. **Andy**
It's a really helpful explanation, but I still have some question:
1. Why would we ROT180() when calculating the gradient?
2. The size of the gradient? I mean let's say l-th layer has like 5 x 5 delta, and the value of (l-1)-th is 9 x 9, how would we solve that?

 AUGUST 31, 2016 AT 5:10 AM ↪ REPLY

8. **yaza**
Good Job 🙂
And, what about update of biases?

\frac{\partial C}{\partial b^l_{i,j}}=\frac{\partial C}{\partial z^l_{i,j}}\frac{\partial z^l_{i,j}}{\partial b^l_{i,j}}=\delta^l_{i,j}\rightarrow \triangle b^l=\delta^l

 OCTOBER 14, 2016 AT 4:24 PM ↪ REPLY

9. **Venkatanaresh**
Thanks a lot. Its the best tutorial on CNN. It is useful for all the people in neural networks community.

 OCTOBER 26, 2016 AT 8:40 AM ↪ REPLY
grzegorzgwardys
Thank you for kind words 🙂

 NOVEMBER 4, 2016 AT 6:00 PM ↪ REPLY

10. **Marcin**
In one of the pictures (the one under "yup, again convolution:", with rot_180(w) * grads from orange layer), are the values correct? The picture shows convolution of the rotated weight kernel (w22, w21, w12, w11) by deltas, but the output is as if deltas were convoluted by the normal (not rotated) weight kernel (w11, w12, w21, w22). So the derived delta at col=1 and row=0 (blue matrix) is shown as (d11 * w21 + d12 * w22), where it actually should be (d11 * w12 + d12 * w11), if convolved by the rotated weights matrix.
Am I missing something? Thanks!

 NOVEMBER 4, 2016 AT 5:30 PM ↪ REPLY
grzegorzgwardys
Yup, it's OK. We are talking about convolutions not correlations and that's why we need another rotation. That's why next image is also OK 😉

 NOVEMBER 4, 2016 AT 5:59 PM ↪ REPLY

 1. **Marcin**
 Thank you for answering. I realized my "convolution" was in fact incorrect, I was not reversing the order and as a result I was getting incorrect output. Wikipedia helped with that: https://en.wikipedia.org/wiki/Kernel_(image_processing)
 Thanks a lot!

  NOVEMBER 4, 2016 AT 9:02 PM

 2. **Stefan**
 But does it matter? It just learns a set of weights that in the end it uses to classify images. Do frameworks like TensorFlow, Theano use correlations or actual convolutions in their CNN layers?

  NOVEMBER 19, 2016 AT 10:45 PM

grzegorzgwardys
It does matter to be precise in such kind of 'math posts'. If we talk about "FFT versions' (look at: https://arxiv.org/abs/1312.5851) then we have surely convolutions.

 NOVEMBER 20, 2016 AT 12:19 PM ↪ REPLY

11. **Marcin**
It looks like the final operation can be made a little simpler by introducing a "full correlation" operation. Full correlation would work like the regular correlation (dot product) only with the addition of zero padding, like in the case of full convolution.
Since convolution works like an inverted correlation, and the final operation uses convolution by a rotated kernel (also an inverse of sorts), both can be replaced by a single "full correlation" operation by the original (not rotated) kernel.
Below is a Scala function implementing the "full correlation" operation. Matrices are stored as arrays of floats in [row0, row1, …] format.

```
def fullCorr2d(a: Array[Float], aCols: Int, aRows: Int,
k: Array[Float], kCols: Int, kRows: Int)(x: Int, y: Int): Float = {
assert(kCols == kRows) // expect a square kernel matrix
val k2 = kCols / 2
var sum = 0f
for (j <- 0 until kRows) {
for (i = 0 && aCol >= 0 && aRow < aRows && aCol < aCols) {
val w = k(j * kCols + i) // kernel weight
sum += w * a(aRow * aCols + aCol)
} // input values outside of the valid range are zero and not needed
}
}
sum
}
```

 DECEMBER 1, 2016 AT 8:41 PM ↪ REPLY

12. **Marcin**
I have another question, regarding the reverse (by rotated kernel) convolution.
If the original convolution (during forward pass) was applied with a stride > 1, does that mean that during the back propagation phase

the reverse convolution (by the rotated kernel) also needs to be applied with the same stride? It seems that that should be true, given this equation (shown above using math symbols): delta(layer,x,y) = [delta(layer+1,x,y) conv ROT180(kernel(layer +1))] * activationDerivative(output(layer,x,y)) <<< output matrix size depends on stride
Thanks a lot!

DECEMBER 2, 2016 AT 9:41 PM ↪ REPLY

13. **Adam Mendenhall**
First: I think you made a typo in the overview; step 2, Feedforward. You've got "z_{x,y}^l=w_{x,y}^l*\sigma(z_{x,y}^l)+b_{x,y}^l" written, but z_{x,y}^l appears twice. The z_{x,y}^l inside the sigmoid should be z_{x,y}^(l-1) or something like that, because convolution operates on the previous layer (or the input), right?

    Second: This is literally the best article on conv nets I have ever read. Thank you.

    JANUARY 1, 2017 AT 7:46 AM ↪ REPLY

14. **Adam Mendenhall**
Sorry to submit another post, but:

    You use the notation a_{x,y}^l for the activation of some filter at some (x,y) in an image. Why would you use the same notation for bias and weight terms? They don't change based on positions in an input image; they 'belong' to the filter. Also, I assume that in the feedforward step, compute z_{x,y}^l for each layer means compute z_{x,y}^l for each pixel in each layer. Is that correct?

    JANUARY 1, 2017 AT 8:23 AM ↪ REPLY

    grzegorzgwardys
    Adam, trying to answer your questions:

    1. I do not see any "z_{x,y}^l=w_{x,y}^l*\sigma(z_{x,y}^l)+b_{x,y}^l". However, there is an eq. "z_{x,y}^(l+1)=w_{x,y}^l*\sigma(z_{x,y}^l)+b_{x,y}^(l+1)" so it seems legit.
    2. I tried to be consistent with Michael Nielsen as much as possible. While he has z_j and b_j I have two-dimensional case so I have z_{x,y} and b_{x,y}. Of course, you can use matrix notation without any x,y positions.
    3. In this blog post I simplified a problem, because we do not have here many feature maps at the same level – only one. ^l defines index of layer.

    JANUARY 1, 2017 AT 5:37 PM ↪ REPLY

15. **Adam Mendenhall**
That's odd; I see an incorrect feedforward equation in my browser. Anyway, thanks for the notation cleanup.

    JANUARY 2, 2017 AT 5:54 PM ↪ REPLY

16. **Adam Mendenhall**
In the example you give, you have a filter of size 2×2, and an input of size 3×3, with an output of size 2×2. If both the filter and input were size 3×3, the output would be size 1×1, and there would be no weight sharing; the errors for each neuron correspond only to the output by a single weight (no sums of deltas). Is that true? If so, why is it said that convolutional neural nets share weights always (since with 3×3 filters there is no weight sharing)?

    JANUARY 7, 2017 AT 3:08 AM ↪ REPLY

17. **Dobry**
The best tutorial on CNN I have ever seen. It's time for YouTube channel for sure!

    JANUARY 10, 2017 AT 3:49 AM ↪ REPLY

18. **Neil**
why w{x,y}*a ,w need subscript when use the symbol of convolution? its each element would be use no matter what the position is.

    JANUARY 14, 2017 AT 6:59 AM ↪ REPLY

    grzegorzgwardys
    Yup, you're right, thx. I see also that Adam Mendenhall was right. Lesson for me – if you are tired, do not review such things (even own work).

    JANUARY 14, 2017 AT 10:54 AM ↪ REPLY

19. **Brandon Beiler**
hi grzegorzgwardys,

    Thanks for the great explanation. I am working with my professor to apply a CNN to a gesture recognition problem. In this post, for simplicity I assume, you mentioned doing the backpropogation with an input that was only 2 dimensions (1 layer deep instead of multiple) as well as a filter that was only 2 dimensions (1 layer deep). However, in our code, as forward propagate, your inputs to the convolution become increasingly deep, and your filters likewise grow in the z (depth) dimension. I understand the whole idea of backpropogating with the flipped filter, however, how do you work with differing depths during backprop? If I have an input that is 10 deep, and I convolve it with a filter that is 10 deep, I get out a 1 deep feature map. So when I backprop the error to that 1 deep feature map, I can't just convolve that 1 deep feature map delta with the 10 deep filter that I used in the forward propogation. Or can I? Any help in this area or pointing towards help would be greatly appreciated. Thank you.

    FEBRUARY 20, 2017 AT 6:25 PM ↪ REPLY

20. Pingback: 反向传播算法 | Sandezi