# Ex 4 Tutorial

## ex4 Tutorial for forward propagation and cost

Tom MosherMentorAssignment: Neural Network Learning · 2 years ago · Edited

Note: this thread is closed to comments. If you have a question, please post it in the Week 5 Discussion Forum area.

This tutorial uses the vectorized method. If you're using a for-loop over the training examples, you're doing it the hard way, and you're on your own.

A note on Errata: The cost and gradient equations in the ex4.pdf file are correct. There may be some errata in the video lectures. Check the Course Wiki to be sure.

I'll use the less-than-helpful greek letters and math notation from the video lectures in this tutorial, though I'll start off with a glossary so we can agree on what they are. I will also suggest some common variable names, so students can more easily get help on the Forum.

It is left to the reader to convert these descriptions into program statements. You will need to determine the correct order and transpositions for each matrix multiplication, so that the result has the correct size.

**Glossary:**

Each of these variables will have a subscript, noting which NN layer it is associated with.

**Θ**

: A Theta matrix of weights to compute the inner values of the neural network. When we used a vector theta, it was noted with the lower-case theta character

***θ***

.

***z***

: is the result of multiplying a data vector with a

**Θ**

matrix. A typical variable name would be "z2".

***a***

: The "activation" output from a neural layer. This is always generated using a sigmoid function

***g*()**

on a

***z***

value. A typical variable name would be "a2".

***δ***

: lower-case delta is used for the "error" term in each layer. A typical variable name would be "d2".

**Δ**

: upper-case delta is used to hold the sum of the product of a
***δ***
value with the previous layer's
***a***
value. In the vectorized solution, these sums are calculated automatically though the magic of matrix algebra. A typical variable name would be "Delta2".
**Θ**
_gradient : This is the thing we're solving for, the partial derivative of theta. There is one of these variables associated with each
**Δ**
. These values are returned by nnCostFunction(), so the variable names must be "Theta1_grad" and "Theta2_grad".
***g*()**
is the sigmoid function.
***g***
′
**()**
is the sigmoid gradient function.
Tip: One handy method for excluding a column of bias units is to use the notation SomeMatrix(:,2:end). This selects all of the rows of a matrix, and omits the entire first column.
See the Appendix at the bottom of the tutorial for information on the sizes of the data objects.
A note regarding bias units, regularization, and back-propagation:
There are two methods for handing exclusion of the bias units in the Theta matrices in the back-propagation and gradient calculations. I've described only one of them here, it's the one that I understood the best. Both methods work, choose the one that makes sense to you and avoids dimension errors. It matters not a whit whether the bias unit is excluded before or after it is calculated - both methods give the same results, though the order of operations and transpositions required may be different. Those with contrary opinions are welcome to write their own tutorial.
**Forward Propagation:**
We'll start by outlining the forward propagation process. Though this was already accomplished once during Exercise 3, you'll need to duplicate some of that work because computing the gradients requires some of the intermediate results from forward propagation. Also, the y values in ex4 are a matrix, instead of a vector. This changes the method for computing the cost J.
1 - Expand the 'y' output values into a matrix of single values (see ex4.pdf Page 5). This is most easily done using an eye() matrix of size num_labels, with vectorized indexing by 'y'. A useful variable name would be "y_matrix", as this...

y_matrix = eye(num_labels)(y,:)

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Note: For MATLAB users, this expression must be split into two lines, such as...

1
2

eye_matrix = eye(num_labels)
y_matrix = eye_matrix(y,:)

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

2 - Perform the forward propagation:
$a_1$
equals the X input matrix with a column of 1's added (bias units) as the first column.
$z_2$
equals the product of
$a_1$
and
$\Theta_1$
$a_2$
is the result of passing
$z_2$
through
$g()$
Then add a column of bias units to
$a_2$
(as the first column).
NOTE: Be sure you DON'T add the bias units as a new row of Theta.

$z^{(3)}$ equals the product of $a^{(2)}$ and $\Theta^{(2)}$ $a^{(3)}$ is the result of passing $z^{(3)}$ through $g()$

**Cost Function, non-regularized:**

3 - Compute the unregularized cost according to ex4.pdf (top of Page 5), using $a^{(3)}$, your y_matrix, and $m$ (the number of training examples). Note that the 'h' argument inside the log() function is exactly a3. Cost should be a scalar value. Since y_matrix and a3 are both matrices, you need to compute the double-sum.

Remember to use element-wise multiplication with the log() function. For a discussion of why you can't (easily) use matrix multiplication here, see this thread:

https://www.coursera.org/learn/machine-learning/discussions/weeks/5/threads/ag_zHUGDEeaXnBKVQIdqyw

Also, we're using the natural log, not log10().

Now you can run ex4.m to check the unregularized cost is correct, then you can submit this portion to the grader.

**Cost Regularization:**

4 - Compute the regularized component of the cost according to ex4.pdf Page 6, using $\Theta^{(1)}$ and $\Theta^{(2)}$ (excluding the Theta columns for the bias units), along with $\lambda$, and $m$

. The easiest method to do this is to compute the regularization terms separately, then add them to the unregularized cost from Step 3.
You can run ex4.m to check the regularized cost, then you can submit this portion to the grader.
---
**Appendix:**
Here are the sizes for the Ex4 character recognition example, using the method described in this tutorial.
NOTE: The submit grader (and the gradient checking process) uses a different test case; these sizes are NOT for the submit grader or for gradient checking.
a1: 5000x401
z2: 5000x25
a2: 5000x26
a3: 5000x10
d3: 5000x10
d2: 5000x25
Theta1, Delta1 and Theta1_grad: 25x401
Theta2, Delta2 and Theta2_grad: 10x26
=========
Here is a link to the test cases, so you can check your work:
https://www.coursera.org/learn/machine-learning/discussions/iyd75Nz_EeWBhgpcuSIffw
The test cases for ex4 include the values of the internal variables discussed in the tutorial.
=========

## ex4 tutorial for nnCostFunction and backpropagation

Tom MosherMentorWeek 5 · 2 years ago · Edited
Keywords: ex4 tutorial backpropagation nnCostFunction
(note: if you have a question about this tutorial, please start a new thread. This one is full and is closed to additional replies)
==========
You can design your code for backpropagation based on analysis of the dimensions of all of the data objects. This tutorial uses the vectorized method, for easy comprehension and speed of execution.
Reference the four steps outlined on Page 9 of ex4.pdf.
---
Let:
m = the number of training examples
n = the number of training features, including the initial bias unit.
h = the number of units in the hidden layer - NOT including the bias unit
r = the number of output classifications
---
1: Perform forward propagation, see the separate tutorial if necessary.
2:
$\delta$

3
or d3 is the difference between a3 and the y_matrix. The dimensions are the same as both, (m x r).
3: z2 came from the forward propagation process - it's the product of a1 and Theta1, prior to applying the sigmoid() function. Dimensions are (m x n)
·
(n x h) --> (m x h)
4:
$\delta$
2
or d2 is tricky. It uses the (:,2:end) columns of Theta2. d2 is the product of d3 and Theta2(no bias), then element-wise scaled by sigmoid gradient of z2. The size is (m x r)
·
(r x h) --> (m x h). The size is the same as z2, as must be.
5:
Δ
1
or Delta1 is the product of d2 and a1. The size is (h x m)
·
(m x n) --> (h x n)
6:
Δ
2
or Delta2 is the product of d3 and a2. The size is (r x m)
·
(m x [h+1]) --> (r x [h+1])
7: Theta1_grad and Theta2_grad are the same size as their respective Deltas, just scaled by 1/m.
Now you have the unregularized gradients. Check your results using ex4.m, and submit this portion to the grader.
===== Regularization of the gradient ==========
Since Theta1 and Theta2 are local copies, and we've already computed our hypothesis value during forward-propagation, we're free to modify them to make the gradient regularization easy to compute.
8: So, set the first column of Theta1 and Theta2 to all-zeros. Here's a method you can try in your workspace console:

1
2

```
Q = rand(3,4)       % create a test matrix
Q(:,1) = 0          % set the 1st column of all rows to 0
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

9: Scale each Theta matrix by
$\lambda/m$
. Use enough parenthesis so the operation is correct.
10: Add each of these modified-and-scaled Theta matrices to the un-regularized Theta gradients that you computed earlier.
---
You're done. Use the test case (from the Resources menu) to test your code, and the ex4 script, then run the submit script.
The test case for ex4 include the values of the internal variables discussed in the tutorial.
---

## Computing the NN cost J using the matrix product

Tom MosherMentorWeek 5 · 7 months ago · Edited
Students often ask why they can't use matrix multiplication to compute the cost value J in the Neural Network cost function. This post explains why.
Short answer: You **can** use matrix multiplication, but it is tricky.
Here is the equation for the unregularized cost J:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right],$$

Notice the double-sum. 'i' ranges over the training examples 'm', and 'k' ranges over the output labels 'K'. The cost has two parts - the first involves the product of 'y' and log(h), and the second involves the product of (1-y) and log(1-h).
Note that 'y' and 'h' are both matrices of size (m x K), and the multiplication in the cost equation is a scalar product for each element in the matrices.
Recall that for linear and logistic regression, 'y' and 'h' were both vectors, so we could compute the sum of their products easily using vector multiplication. After transposing one of the vectors, we get a result of size (1 x m) * (m x 1). That's a scalar value. So that worked fine, as long as 'y' and 'h' are vectors.
But the when 'h' and 'y' are matrices, the same trick does not work as easily. Here's why.
Let's first show the math using the element-wise product of two matrices A and B. For simplicity, let's use m= 3 and K=2.
***A=***
⎡
⎣
|
|

***a***
***c***
***e***
***b***

***d***
***f***
⎤
⎦
⎥
⎥
**,*B*=**
⎡
⎣
⎥
⎥
***m***
***o***
***q***
***n***
***p***
***r***
⎤
⎦
⎥
⎥
The sum over the rows and columns of the element-wise product is:
$$\sum\sum A.*B=am+bn+co+dp+eq+fr$$
Now let's detail the math for this using a matrix product. Since A and B are the same size, but the number of rows and columns are not the same, we must transpose one of the matrices before we compute the product. Let's transpose the 'A' matrix, so the product matrix will be size (K x K). We could of course invert the 'B' matrix, but then the product matrix would be size (m x m). The (m x m) matrix is probably a lot larger than (K x K).
It turns out (and is left for the reader to prove) that both the (m x m) and (K x K) matrices will give the same results for the cost J.
***A***
′
**∗*B*=[**
***a***
***b***
***c***
***d***
***e***

***f***
**]**∗
⎡
⎣
|
|
***m***
***o***
***q***
***n***
***p***
***r***
⎤
⎦
|
|
After the matrix product, we get:
***A***
′
∗***B*=[**
**(*am*+*co*+*eq*)**
**(*bm*+*do*+*fq*)**
**(*an*+*cp*+*er*)**
**(*bn*+*dp*+*fr*)**
**]**
So this is a size (K x K) result, as expected. Note that the terms which lie on the main diagonal are the same terms that result from the double-sum of the element-wise product. The next step is to compute the sum of the diagonal elements using the "trace()" command, or by sum(sum(...)) after element-wise multiplying by an identity matrix of size (K x K).
The sum-of-product terms that are NOT on the main diagonal are unwanted - they are not part of the cost calculation. So simply using sum(sum(...)) over the matrix product will include these terms, and you will get an incorrect cost value.
The performance of each of these methods - double-sum of the element-wise product, or the matrix product with either trace() or the sum of the diagonal elements - should be evaluated, and the best one used for a given data set.

END