Coursera Beta

- ▪ Harmandeep Singh
- Contact Us
- My Contributions
- Log Out

ML:Neural Networks: Learning

From Coursera

Contents

Cost Function

Let's first define a few variables that we will need to use:

L = total number of layers in the network

s_l = number of units (not counting bias unit) in layer l

K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the kth output.

Our cost function for neural networks is going to be a generalization of the one we used for logistic regression.

Recall that the cost function for regularized logistic regression was:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For neural networks, it is going to be slightly more complicated:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log((h_\Theta(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, between the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer; and

- the triple sum simply adds up the squares of all the individual Θs in the entire network.

- the i in the triple sum does **not** refer to training example i

Backpropagation Algorithm

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

Our goal is to compute:

$\min_\Theta J(\Theta)$

Processing math: 63%

That is, we want to minimize our cost function J using an optimal set of parameters in theta.

In this section we'll look at the equations we use to compute the partial derivative of $J(\Theta)$:

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$

In backpropagation we're going to compute for every node:

$\delta_j^{(l)}$ = "error" of node j in layer l

Recall that $a_j^{(l)}$ is activation node j in layer l.

For the **last layer**, we can compute the vector of delta values with:

$$\delta^{(L)} = a^{(L)} - y$$

Where L is our total number of layers and $a^{(L)}$ is the vector of activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y.

To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \ .* \ g'(z^{(l)})$$

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$.

The g-prime derivative terms can also be written out as:

$$g'(z^{(l)}) = a^{(l)} \ .* \ (1 - a^{(l)})$$

This can be shown and proved in calculus.

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\frac{\partial g(z)}{\partial z} = -\left(\frac{1}{1+e^{-z}}\right)^2 \frac{\partial}{\partial z}(1+e^{-z}) = -\left(\frac{1}{1+e^{-z}}\right)^2 e^{-z}(-1) = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{1}{1+e^{-z}}\right)(e^{-z}) = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right) = g(z)(1-g(z))$$

The full backpropagation equation for the inner nodes is then:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \ .* \ a^{(l)} \ .* \ (1 - a^{(l)})$$

A. Ng states that the derivation and proofs are complicated and involved, but you can still implement the above equations to do back propagation without knowing the details.

We can compute our partial derivative terms by multiplying our activation values and our error values for each training example t:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = \frac{1}{m} \sum_{t=1}^{m} a_j^{(t)(l)} \delta_i^{(t)(l+1)}$$

This however ignores regularization, which we'll deal with later.

Note: δ^{l+1} and a^{l+1} are vectors with s_{l+1} elements. Similarly, $a^{(l)}$ is a vector with s_l elements. Multiplying them produces a matrix that is s_{l+1} by s_l which is the same dimension as $\Theta^{(l)}$. That is, the process produces a gradient term for every element in $\Theta^{(l)}$. (Actually, $\Theta^{(l)}$ has $s_{l+1} + 1$ rows, so the dimensionality is not exactly the same).

We can now take all these equations and put them together into a backpropagation algorithm:

Backpropagation Algorithm

- Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$
- Set $\Delta_{i,j}^{(l)} := 0$ for all (l, i, j)
- For training example $t = 1$ to m:
    - Set $a^{(1)} := x^{(t)}$
    - Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
    - Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
    - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \ .* \ a^{(l)} \ .* \ (1 - a^{(l)})$
    - $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
- $D_{i,j}^{(l)} := \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)}\right)$ If $j \neq 0$ NOTE: Typo in lecture slide omits outside parentheses. This version is correct.
- $D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)}$ If $j = 0$

The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative.

The actual proof is quite involved, but, the $D_{i,j}^{(l)}$ terms are the partial derivatives and the results we are looking for:

$$D_{i,j}^{(l)} = \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}}.$$

Backpropagation Intuition

The cost function is:

$$J(\theta) = -\frac{1}{m}\left[\sum_{t=1}^{m}\sum_{k=1}^{K} y_k^{(t)} \log(h_\theta(x^{(t)}))_k + (1 - y_k^{(t)}) \log(1 - h_\theta(x^{(t)})_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\theta_{j,i}^{(l)})^2$$

If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

$$cost(t) = y^{(t)} \log(h_\theta(x^{(t)})) + (1 - y^{(t)}) \log(1 - h_\theta(x^{(t)}))$$

More intuitively you can think of that equation roughly as:

$$cost(t) \approx (h_\theta(x^{(t)}) - y^{(t)})^2$$

Intuitively, $\delta_j^{(l)}$ is the "error" for $a_j^{(l)}$ (unit j in layer l)

More formally, the delta values are actually the derivative of the cost function:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t)$$

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are.

Note: In lecture, sometimes i is used to index a training example. Sometimes it is used to index a unit in a layer. In the Back Propagation Algorithm described here, t is used to index a training example rather than overloading the use of i.

Implementation Note: Unrolling Parameters

With neural networks, we are working with sets of matrices:

$$\Theta_1, \Theta_2, \Theta_3, \ldots$$
$$D_1, D_2, D_2, \ldots$$

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector:

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:

```
Theta1 = reshape(thetaVector(1:110),10,11)
Theta2 = reshape(thetaVector(111:220),10,11)
Theta3 = reshape(thetaVector(221:231),1,11)
```

NOTE: The lecture slides show an example neural network with 3 layers. However, *3* theta matrices are defined: Theta1, Theta2, Theta3. There should be only 2 theta matrices: Theta1 (10 x 11), Theta2 (1 x 11).

Gradient Checking

Gradient checking will assure that our backpropagation works as intended.

We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial\Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative **with respect to** Θ_j as follows:

$$\frac{\partial}{\partial\Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

A good small value for ϵ (epsilon), guarantees the math above to become true. If the value be much smaller, may we will end up with numerical problems. The professor Andrew usually uses the value $\epsilon = 10^{-4}$.

We are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

We then want to check that gradApprox $\approx$ deltaVector.

Once you've verified **once** that your backpropagation algorithm is correct, then you don't need to compute gradApprox again. The code to compute gradApprox is very slow.

Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly.

Instead we can randomly initialize our weights:

Initialize each $\Theta^{(l)}_{ij}$ to a random value between $[-\epsilon, \epsilon]$:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{\mathrm{Loutput} + \mathrm{Linput}}}$$

\Theta^{(l)} = 2 \epsilon \; \mathrm{rand}(\mathrm{Loutput}, \mathrm{Linput} + 1) - \epsilon

```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

rand(x,y) will initialize a matrix of random real numbers between 0 and 1. (Note: this epsilon is unrelated to the epsilon from Gradient Checking)

Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers total.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If more than 1 hidden layer, then the same number of units in every hidden layer.

Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get $h_\theta(x^{(i)})$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

When we perform forward and back propagation, we loop on every training example:

```
for i = 1:m,
  Perform forward propagation and backpropagation using example (x(i),y(i))
  (Get activations a(l) and delta terms d(l) for l = 2,...,L
```

Bonus: Explanation of Derivatives Used in Backpropagation

Start!

- We know that for a logistic regression classifier (which is what all of the output neurons in a neural network are), we use the cost function, J(\theta) = -ylog(h_{\theta}(x)) - (1-y)log(1-h_{\theta}(x)), and apply this over the K output neurons, and for all m examples.
- The equation to compute the partial derivatives of the theta terms in the output neurons:

\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}

- And the equation to compute partial derivatives of the theta terms in the [last] hidden layer neurons (layer L-1):

\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}

- Clearly they share some pieces in common, so a delta term (\delta^{(L)}) can be used for the common pieces between the output layer and the hidden layer immediately before it (with the possibility that there could be many hidden layers if we wanted):

\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}

- And we can go ahead and use another delta term ($\delta^{(L-1)}$) for the pieces that would be shared by the final hidden layer and a hidden layer before that, if we had one. Regardless, this delta term will still serve to make the math and implementation more concise.

$\delta^{(L-1)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$
$\delta^{(L-1)} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$

- With these delta terms, our equations become:

$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$
$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$

Now, time to evaluate these derivatives:

- Let's start with the output layer:

$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$

- Using $\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$, we need to evaluate both partial derivatives.
- Given $J(\theta) = -y\log(a^{(L)}) - (1-y)\log(1-a^{(L)})$, where $a^{(L)} = h_{\theta}(x)$), the partial derivative is:

$\frac{\partial J(\theta)}{\partial a^{(L)}} = \frac{y-1}{a^{(L)}-1} - \frac{y}{a^{(L)}}$

- And given $a = g(z)$, where $g = \frac{1}{1+e^{-z}}$, the partial derivative is:

$\frac{\partial a^{(L)}}{\partial z^{(L)}} = a^{(L)}(1-a^{(L)})$

- So, let's substitute these in for $\delta^{(L)}$:

$\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$
$\delta^{(L)} = (\frac{y-1}{a^{(L)}-1} - \frac{y}{a^{(L)}}) (a^{(L)}(1-a^{(L)}))$
$\delta^{(L)} = a^{(L)} - y$

- So, for a 3-layer network (L=3),

$\delta^{(3)} = a^{(3)} - y$

- Note that this is the correct equation, as given in our notes.
- Now, given $z = \theta * input$, and in layer L the input is $a^{(L-1)}$, the partial derivative is:

$\frac{\partial z^{(L)}}{\partial \theta^{(L-1)}} = a^{(L-1)}$

- **Put it together for the output layer**:

$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$
$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = (a^{(L)} - y) (a^{(L-1)})$

- Let's continue on for the hidden layer (let's assume we only have 1 hidden layer):
  $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$
- Let's figure out $\delta^{(L-1)}$.
- Once again, given $z = \theta * input$, the partial derivative is:
  $\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = \theta^{(L-1)}$
- And: $\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = a^{(L-1)}(1-a^{(L-1)})$
- So, let's substitute these in for $\delta^{(L-1)}$:
  $\delta^{(L-1)} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$
  $\delta^{(L-1)} = \delta^{(L)} (\theta^{(L-1)}) (a^{(L-1)}(1-a^{(L-1)}))$
  $\delta^{(L-1)} = \delta^{(L)} \theta^{(L-1)} a^{(L-1)}(1-a^{(L-1)})$
- So, for a 3-layer network,
  $\delta^{(2)} = \delta^{(3)} \theta^{(2)} a^{(2)}(1-a^{(2)})$
- **Put it together for the [last] hidden layer**:
  $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$
  $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = (\delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}) (a^{(L-2)})$
  $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = ((a^{(L)} - y) (\theta^{(L-1)})(a^{(L-1)}(1-a^{(L-1)}))) (a^{(L-2)})$

Bonus: NN for linear systems

Introduction

The NN we created for classification can easily be modified to have a linear output. First solve the 4th programming exercise. You can create a new function script, nnCostFunctionLinear.m, with the following characteristics

- There is only one output node, so you do not need the 'num_labels' parameter.
- Since there is one linear output, you do not need to convert y into a logical matrix.
- You still need a non-linear function in the hidden layer.
- The non-linear function is often the tanh() function - it has an output range from -1 to +1, and its gradient is easily implemented. Let $g(z) = \tanh(z)$.
- The gradient of tanh is $g'(z) = 1 - g(z)^2$. Use this in backpropagation in place of the sigmoid gradient.
- Use linear regression for the NN output (do not use a sigmoid function on the output layer).
- Cost computation: Use the linear cost function for J (from ex1 and ex5) for the unregularized portion. For the regularized portion, use the same method as ex4.
- Where reshape() is used to form the Theta matrices, replace 'num_labels' with '1'.

You will also need to create a predictLinear() function, using the tanh() function in the hidden layer, and a linear output.

Testing your linear NN

Here is a test case for your nnCostFunctionLinear()

```
% inputs
nn_params = [31 16 15 -29 -13 -8 -7 13 54 -17 -11 -9 16]'/ 10;
il = 1;
hl = 4;
X = [1 ; 2 ; 3];
y = [1 ; 4 ; 9];
lambda = 0.01;

% command
[j g] = nnCostFunctionLinear(nn_params, il, hl, X, y, lambda)

% results
j =  0.020815
g =
  -0.0131002
  -0.0110085
  -0.0070569
   0.0189212
  -0.0189639
  -0.0192539
  -0.0102291
   0.0344732
   0.0024947
   0.0080624
   0.0021964
   0.0031675
  -0.0064244
```

Now create a script that uses the 'ex5data1.mat' from ex5, but without creating the polynomial terms. With 8 units in the hidden layer and MaxIter set to 200, you should be able to get a final cost value of 0.3 to 0.4. The results will vary a bit due to the random Theta initialization. If you plot the training set and the predicted values for the training set (using your predictLinear() function), you should have a good match.

Next: Advice for Applying Machine Learning Back to Index: Main

Retrieved from "https://share.coursera.org/wiki/index.php?title=ML:Neural_Networks:_Learning&oldid=33181"
Category: ML:Lecture Notes

- This page was last modified on 29 January 2016, at 23:47.
- This page has been accessed 47,289 times.
- Privacy policy
- About Coursera
- Disclaimers