machine learning_week5

Created:	18/11/13 22:10	**Updated:**	18/11/20 09:19

Neural Network： Learning

CostFunction and Backpropagation

- **denotation:**
    -
        - 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
- **CostFunction:**
    - $$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$
    - note
        - at week 4 we realize that neural network works like linking many logistic regression. For a logistic regression we use J(θ), and here the double sum simply **adds up** J(θ) calculated for each cell in **the output layer**.(that's why the loop k=1:K)
            - $\sum\sum_{k=1}^{K}\left[y_k\right.$
        - the triple sum simply adds up the squares of all the individual Θs in the **entire** network.
            - $\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$
        - why do we add up all the theta in the entire network, but in the double sum there don't loop from i=1:L-1?
            - remember, we are focusing on build up cost function which is used to judge the accuracy of prediction. Further, the cost function will also use to generate thetas.
            - so the double loop is to count all the predictions (hypothesis). But in regularization we would to generate all theta with better performance, so we should include computing all Θs.

Backpropagation Algorithm

- goal: min J(θ) 👉 base problem: Seeking partial derivative of all thetas. $\frac{\partial}{\partial\Theta_{i,j}^{(l)}}J(\Theta)$
    - method: use Backpropagation
- algorithm here: (**unregularized**)
    - Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

Backpropagation algorithm

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j). (used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

For $i = 1$ to m ← $(x^{(i)}, y^{(i)})$.

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

- crucial step:
  - 4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$

    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:

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

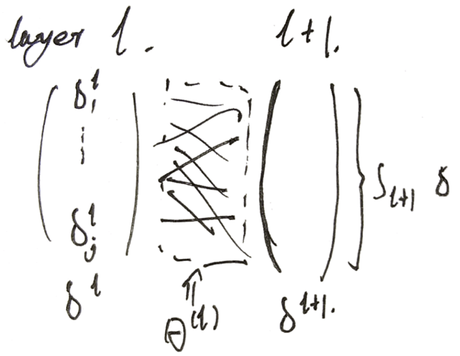
    5. $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

    Hence we update our new Δ matrix.

    - $D_{i,j}^{(l)} := \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)}\right)$, if j≠0.
    - $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}$ If j=0

    The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

- Personal understanding:
  - compare to forward propagation, which compute each layer's units from left to right, backward propagation compute each layer's δ from right to left.
  - **what δ actually is?**
    1. in video, pro Andrew Ng explain it as Intuitively, $\delta_j^{(l)}$ is the "error" for $a_j^{(l)}$ (unit j in layer l).
    2. review our goal: find **partial derivative of theta** to minimize J(θ). Therefore, the δ formally define the derivative.
       - $$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t)$$
    3. so our work still the same: find derivative, which equals to solve each δ.
  - how to **solve δ** ?
    - by calculus, we actually working on chain rule
    - If we consider simple non-multiclass classification (k = 1) and disregard regularization,
      - so here each **δ**, we compute it by:



$\Rightarrow$ for each δ_i^l in δ^l.

we have.

$$\delta_i^l = \theta_{11}^l \times \delta_1^{l+1} + \theta_{12}^l \times \delta_2^{l+1} + \cdots \theta_{1,S_{l+1}}^l \delta_{S_{l+1}}^{l+1}$$
$$= (\theta_1^l)^T \delta^{l+1}$$

to vectorlize:

$\Rightarrow$ compute δ^l

have. $\delta^l = (\theta^l)^T \cdot \delta^{l+1}$

- more detail explanation: https://www.zhihu.com/question/27239198/answer/89853077

Implement BP:

- skills:
  - unroll/reshape parameters
    - vector ↔ matrix
    - why?
      - matrix: better understanding when coding
      - vector: some advanced algorithms assume that you have all of your parameters unrolled into a big long vector
    - how? example here:
      - ```
        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)
        ```
      - **Learning Algorithm**

        → Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

        → Unroll to get `initialTheta` to pass to

        → `fminunc(@costFunction, initialTheta, options)`

        `function [jval, gradientVec] = costFunction(thetaVec)`

        From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

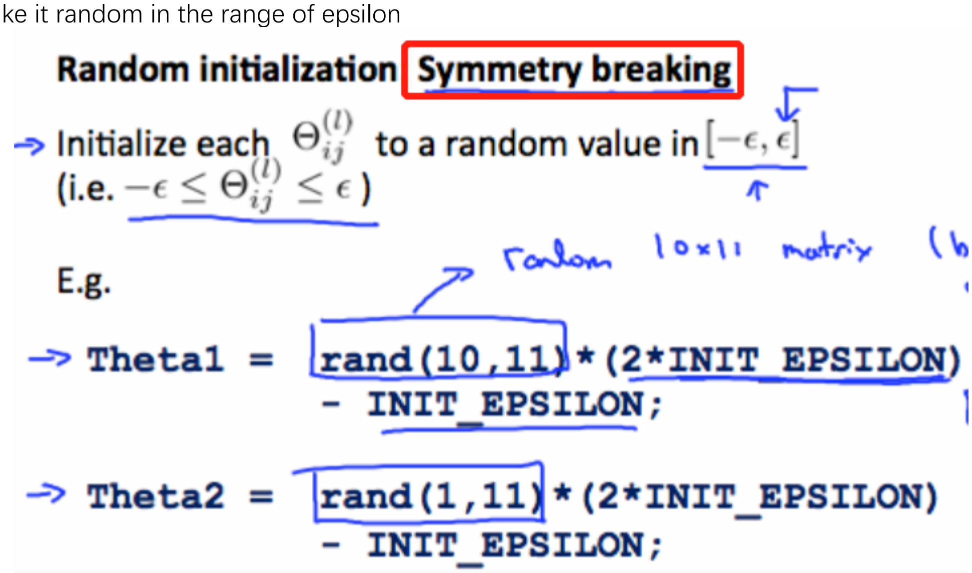
        Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

        Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.
  - Gradient Checking
    - there may be some subtle bugs in code which hard to find, at the same time program still can give a theta which let j(θ) decreased but actually wrong.
    - to find the bug in advance, we should monitor our theta from backprop : **compare it with mathematically approximate theta**
      - $$\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta+\epsilon)-J(\Theta-\epsilon)}{2\epsilon}$$
        ```

            - in vector or matrix : $$\frac{\partial}{\partial\Theta_j}J(\Theta) \approx \frac{J(\Theta_1,\ldots,\Theta_j+\epsilon,\ldots,\Theta_n) - J(\Theta_1,\ldots,\Theta_j-\epsilon,\ldots,\Theta_n)}{2\epsilon}$$
        - code example here (let epsilon = 10^{-4}

            ```
            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;
            ```

        - so before actually start to learning, use above **gradApprox** to verified backpropagation algorithm is correct.
        - Once done the test, turn off the testing loop(which cost many computing resource)
- **Random Initialize Theta:**
    - if assign initial theta with a same value, our neural network would degenerate to a logistic regression.
    - so make it random in the range of epsilon

    - how to choose epsilon?

        [2]One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of ϵ_{init} is $\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in}+L_{out}}}$, where $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the number of units in the layers adjacent to $\Theta^{(l)}$.

        - **but why ? _wait to solve.**

Assembling!

- before: pick a NN **Architecture.**
    - default: one hidden layer
    - if layer > 1, each hidden layer has the same amount of units.
    - input layer: dimension of features
    - output layer: number of classes
- Training:
    1. randomly initialize theta
    2. forward propagation
    3. cost function J(θ)
    4. **back propagation**
    5. gradient checking. Confirm BP works properly.
    6. Turn off gradient checking. Begin gradient descent or other function to minimize J(θ)