NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ3005
Artificial Intelligence

Neural Networks Learning

Asst/P Mahardhika Pratama

Email: mpratama@ntu.edu.sg

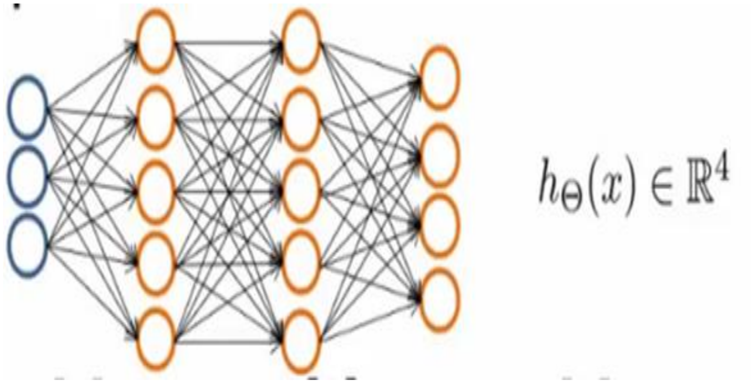
Office: N4-02a-08

Outline

- ❑Cost Function
- ❑Summary
- ❑Back Propagation Algorithm
- ❑Intuition of Back Propagation Algorithm
- ❑Implementation Notes

Cost Function

- Let's consider classification problem
  - Training set is $\{(x^1, y^1), (x^2, y^2), (x^3, y^3) \ldots (x^n, y^m)\}$
  - L = number of layers in the network
    - In our example below L = 4
  - s_l = number of units (not counting bias unit) in layer l
- So here
  - L = 4
  - $s_1 = 3$
  - $s_2 = 5$
  - $s_3 = 5$
  - $s_4 = 4$



Cost Function

- Two types of classification
- **Binary classification**
  - 1 output (0 or 1)
  - So single output node - value is going to be a real number
  - k = 1
  - $s_L = 1$
- **Multi-class classification**
  - k distinct classifications
  - Typically k is greater than or equal to three
  - If only two just go for binary
  - $s_L = k$
  - So y is a k-dimensional vector of real numbers

$$y \in \mathbb{R}^K \quad \text{E.g.} \begin{array}{cccc} \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, & \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, & \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, & \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \\ \text{pedestrian} & \text{car} & \text{motorcycle} & \text{truck} \end{array}$$

Cost Function

- Logistic Regression cost function is as follows:

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

- We generalize this cost function for *k* output case

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

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

  - this cost function outputs a *k* dimensional vector
  - $h_\Theta(x)_i$ refers to the ith value in that vector

Cost Function

- **First Half**

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

- This is just saying
  - For each training data example (i.e. 1 to m - the first summation)
    - Sum for each position in the output vector
- This is an average sum of logistic regression

Second Half

$$\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

- This is also called a **weight decay** term
- As before, the lambda value determines the importance of regularization term
- The regularization term is similar to that in logistic regression

Summary

- Forward Propagation
  - ✓ takes your neural network and the initial input into that network and pushes the input through the network
    - ❖ leads to the generation of an output hypothesis
- Back Propagation
  - ✓ takes the output you got from your network, compares it to the real value (y) and calculates how wrong the network was
  - ✓ using the error you've just calculated, back-calculates the error associated with each unit from the preceding layer
  - ✓ These "error" measurements for each unit can be used to calculate the partial derivatives
  - ✓ We use the partial derivatives with gradient descent to try minimize the cost function and update all the ϴ values
  - ✓ This repeats until gradient descent reports convergence

Back Propagation

- is used to minimize the cost function

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

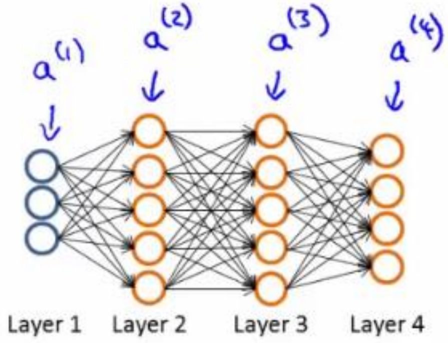
- Find parameters Θthat minimize $J(\Theta)$
- To minimize the cost function, we can write codes as follows:
  - $J(\Theta)$
  - Partial derivative terms
    - This is not trivial! ϴ is indexed in three dimensions because we have separate parameter values for each node in each layer going to each node in the following layer

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

      - i here represents the unit in layer l+1 you're mapping to (destination node)
      - j is the unit in layer l you're mapping from (origin node)
      - l is the layer your mapping from (to layer l+1) (origin layer)

Back Propagation

- Gradient Computation
  - Forward Computation
    - **Layer 1**
      - $a^1 = x$
      - $z^2 = \Theta^1 a^1$
    - **Layer 2**
      - $a^2 = g(z^2)$ (add a_0^2)
      - $z^3 = \Theta^2 a^2$
    - **Layer 3**
      - $a^3 = g(z^3)$ (add a_0^3)
      - $z^4 = \Theta^3 a^3$
    - **Output**
      - $a^4 = h_\Theta(x) = g(z^4)$



$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

Back Propagation

- For each node we can calculate (δ_j^l) - this is **the error of node j in layer l**
  - ✓ a_j^l is the activation of node j in layer l
  - ✓ $\delta_j^4 = a_j^4 - y_j$
    - [Activation of the unit] - [the actual value observed in the training example]
  - ✓ Instead of focussing on each node, let's think about this as a vectorized problem
    $\delta^4 = a^4 - y$
    - So here δ^4 is the vector of errors for the 4th layer
    - a^4 is the vector of activation values for the 4th layer
    - Θ^3 is the vector of parameters for the 3->4 layer mapping
    - δ^4 is (as calculated) the error vector for the 4th layer
    - $g'(z^3)$ is the first derivative of the activation function g evaluated by the input values given by z^3

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

  - ➢ You can do the calculus if you want (...), but when you calculate this derivative you get
  - ➢ $g'(z^3) = a^3 .* (1 - a^3)$
    - So, more easily
  - ➢ $\delta^3 = (\Theta^3)^T \delta^4 .*(a^3 .* (1 - a^3))$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^l \delta_i^{(l+1)}$$

- you can use δ to get the partial derivative of Θ with respect to individual parameters (if you ignore regularization, or regularization is 0

Back Propagation

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

First, set the delta values

- Set equal to 0 for all values Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j)
- Will be used as accumulators for computing the partial derivatives

Next, loop through the training set

For i=1 to m

- Set a^1 (activation of input layer) = x^i
- **Perform forward propagation** to compute a^l for each layer (l = 1,2, ... L)
- **Then**, use the output label for the specific example we're looking at to calculate δ^L where $\delta^L = a^L - y^i$
    - using **back propagation** we move back through the network from layer L-1 down
- Finally, use Δ to accumulate the partial derivative terms

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

Note here
l = layer
j = node in that layer
i = the error of the affected node in the target layer

- You can vectorize the Δ expression too

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T.$$

Back Propagation

Finally,

- After executing the body of the loop, exit the for loop and compute

$$D_{ij}^{(l)} := \frac{1}{m}\triangle_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} \text{ if } j \neq 0$$

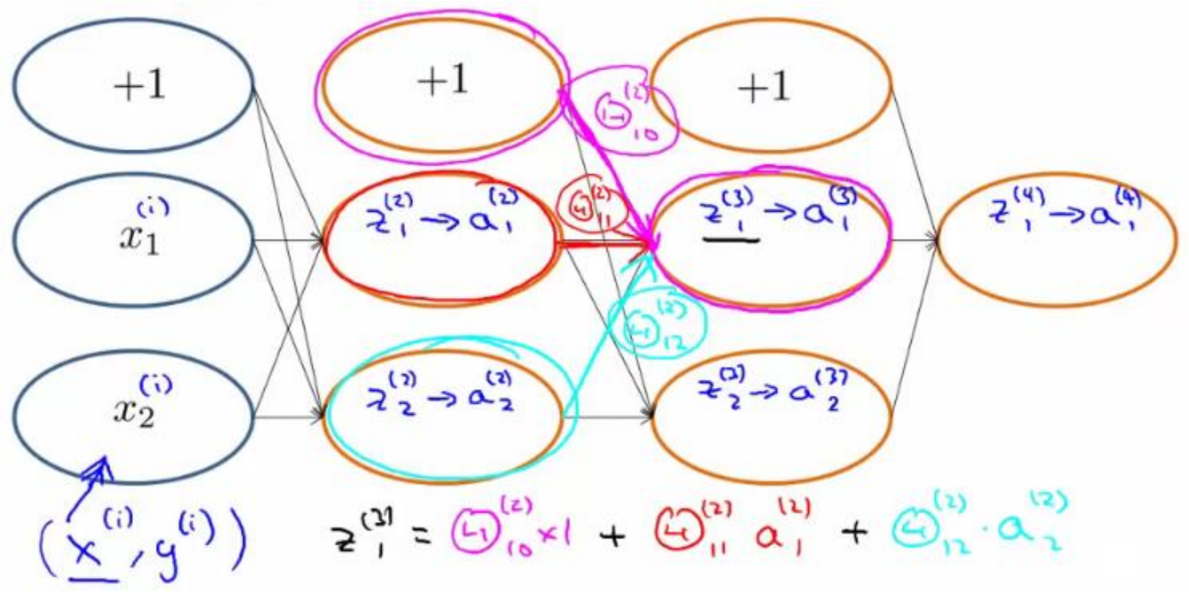
$$D_{ij}^{(l)} := \frac{1}{m}\triangle_{ij}^{(l)} \qquad\qquad \text{if } j = 0$$

- We can show that each D is equal to the following

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

- We have calculated the partial derivative for each parameter

Intuition of Back Propagation



Intuition of Back Propagation

- Back Propagation is as with forward propagation but done backward
- Cost Function:

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

- Cost function for a single sample:

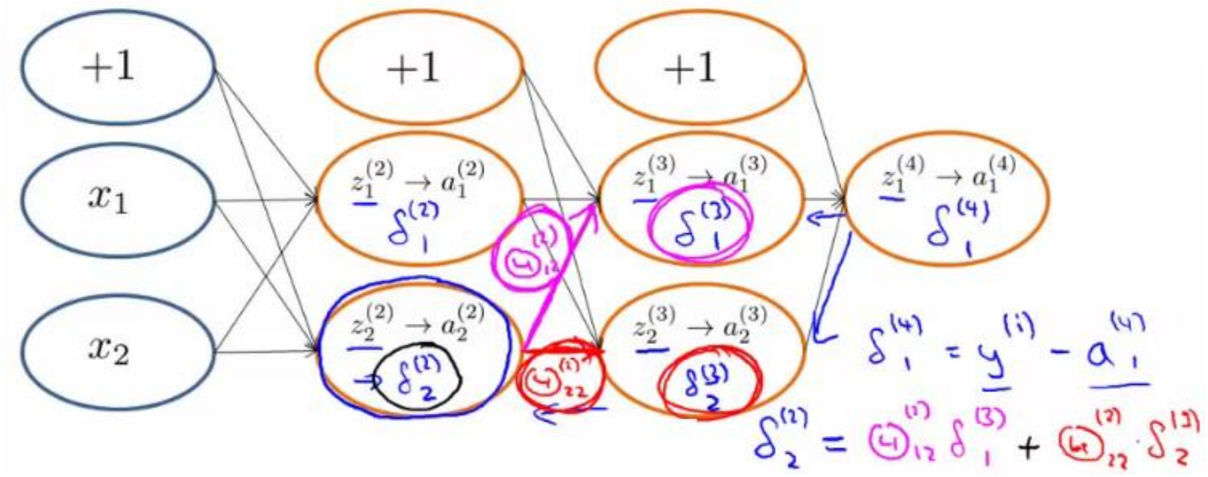
$$\mathrm{cost(i)} = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)})$$

- δ term on a unit as the "error" of cost for the activation value associated with a unit:

- Back propagation calculates the δ, and those δ values are the weighted sum of the next layer's delta values, weighted by the parameter associated with the links

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

Intuition of Back Propagation



Implementation Notes – Gradient Checking

- it looks like J(ϴ) is decreasing, but in reality it may not be decreasing by as much as it should
- Gradient checking helps make sure an implementation is working correctly

Example

Have an function J(ϴ)

Estimate derivative of function

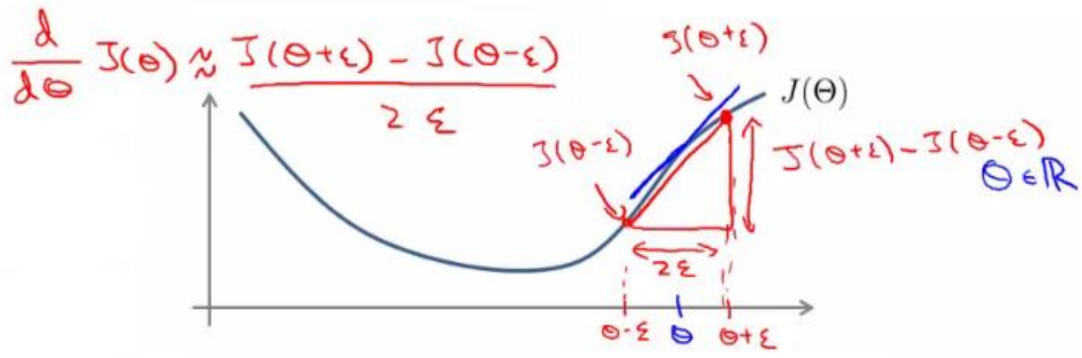
at point ϴ (where ϴ is a real number)

How?

- Numerically
  - Compute ϴ + ε
  - Compute ϴ - ε
  - Join them by a straight line
  - Use the slope of that line as an approximation to the derivative

Usually, epsilon is pretty small (0.0001)

- If epsilon becomes REALLY small then the term BECOMES the slopes derivative



Implementation Notes – Gradient Checking

- Implementation note
  - Implement back propagation to compute DVec
  - Implement numerical gradient checking to compute gradApprox
  - Check they're basically the same (up to a few decimal places)

Implementation Notes – Random Initialization

- Pick random small initial values for all the theta values
  - If you start them on zero (which does work for linear regression) then the algorithm fails - all activation values for each layer are the same
- So chose random values!
  - Between 0 and 1, then scale by epsilon (where epsilon is a constant)

Putting Things Together

1) - pick a network architecture

- Number of
  - **Input units** - number of dimensions x (dimensions of feature vector)
  - **Output units** - number of classes in classification problem
  - **Hidden units**
    - Default might be
      - 1 hidden layer
    - Should probably have
      - Same number of units in each layer
      - Or 1.5-2 x number of input features
    - Normally
      - More hidden units is better
      - But more is more computational expensive
      - We'll discuss architecture more later

2) - Training a neural network

2.1) Randomly initialize the weights

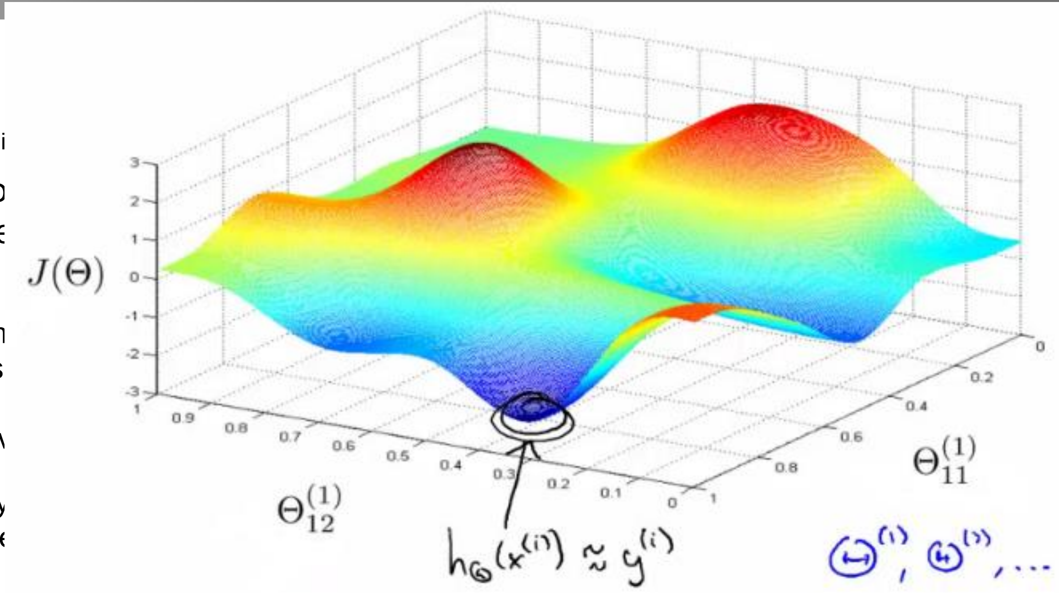
Small values near 0

2.2) Implement forward propagation to get $h_\Theta(x)^i$

2.3) Implement code to compute the cost functio

2.4) Implement back propagation to compute the

- Notes on implementation
  - Usually done with a for loop over trainin
  - *Can* be done without a for loop, but this
  - Be careful



2.5) Use gradient checking to compare the partial derivativ gradient of J(Ɵ)

- Disable the gradient checking code for when y

2.6) Use gradient descent or an advanced optimization me parameters Ɵ

- Here J(Ɵ) is non-convex
  - Can be susceptible to local minimum
  - In practice this is not usually a huge problem
  - Can't guarantee programs with find global optimum should find good local optimum at least