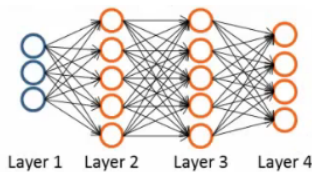


## 09: Neural Networks - Learning

[Previous](#) [Next](#) [Index](#)

### Neural network cost function

- NNs - one of the most powerful learning algorithms
  - Is a learning algorithm for fitting the derived parameters given a training set
  - Let's have a first look at a neural network cost function
- Focus on application of NNs for classification problems
- Here's the set up
  - Training set is  $\{(x^1, y^1), (x^2, y^2), (x^3, y^3) \dots (x^n, y^n)\}$
  - $L$  = 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

- $l = 4$
- $s_1 = 3$
- $s_2 = 5$
- $s_3 = 5$
- $s_4 = 4$

### Types of classification problems with NNs

- Two types of classification, as we've previously seen
- **Binary classification**
  - 1 output (0 or 1)
  - So single output node - value is going to be a real number
  - $k = 1$ 
    - NB  $k$  is number of units in output layer
  - $s_L = 1$
- **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

$$y \in \mathbb{R}^K \quad \text{e.g.,} \quad \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

pedestrian   car   motorcycle   truck

### Cost function for neural networks

- The (regularized) logistic regression cost function is as follows;

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

- For neural networks our cost function is a generalization of this equation above, so instead of one output we generate  $k$  outputs

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

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

- Our cost function now outputs a  $k$  dimensional vector
  - $h_{\Theta}(x)$  is a  $k$  dimensional vector, so  $h_{\Theta}(x)_i$  refers to the  $i$ th value in that vector

- Costfunction  $J(\Theta)$  is
  - $[-1/m]$  times a sum of a similar term to which we had for logic regression
  - But now this is also a sum from  $k = 1$  through to  $K$  ( $K$  is number of output nodes)
    - Summation is a sum over the  $k$  output units - i.e. for each of the possible classes
    - So if we had 4 output units then the sum is  $k = 1$  to 4 of the logistic regression over each of the four output units in turn
  - This looks really complicated, but it's not so difficult
    - We don't sum over the bias terms (hence starting at 1 for the summation)
      - Even if you do and end up regularizing the bias term this is not a big problem
    - Is just summation over the terms

**Woah there - lets take a second to try and understand this!**

- There are basically two halves to the neural network logistic regression 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 a massive regularization summation term, which I'm not going to walk through, but it's a fairly straightforward triple nested summation
- This is also called a **weight decay** term
- As before, the lambda value determines the importance of the two halves
- The regularization term is similar to that in logistic regression
- So, we have a cost function, but *how* do we minimize this bad boy?!

## Summary of what's about to go down

*The following section is, I think, the most complicated thing in the course, so I'm going to take a second to explain the general idea of what we're going to do;*

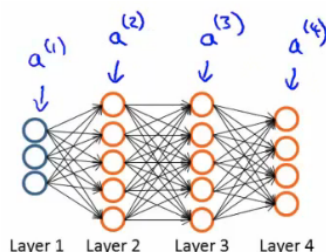
- We've already described **forward propagation**
  - This is the algorithm which takes your neural network and the initial input into that network and pushes the input through the network
    - It leads to the generation of an output hypothesis, which may be a single real number, but can also be a vector
- We're now going to describe **back propagation**
  - Back propagation basically takes the output you got from your network, compares it to the real value ( $y$ ) and calculates how wrong the network was (i.e. how wrong the parameters were)
  - It then, using the error you've just calculated, back-calculates the error associated with each unit from the preceding layer (i.e. layer  $L - 1$ )
  - This goes on until you reach the input layer (where obviously there is no error, as the activation is the input)
  - These "error" measurements for each unit can be used to calculate the **partial derivatives**
    - Partial derivatives are the bomb, because gradient descent needs them to minimize the cost function
  - We use the partial derivatives with gradient descent to try minimize the cost function and update all the  $\Theta$  values
  - This repeats until gradient descent reports convergence
- A few things which are good to realize from the get go
  - There is a  $\Theta$  matrix for each layer in the network
    - This has each node in layer  $l$  as one dimension and each node in  $l+1$  as the other dimension
  - Similarly, there is going to be a  $\Delta$  matrix for each layer
    - This has each node as one dimension and each training data example as the other

## Back propagation algorithm

- We previously spoke about the neural network cost function
- Now we're going to deal with **back propagation**
  - Algorithm used to minimize the cost function, as it **allows us to calculate partial derivatives!**

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

- The cost function used is shown above
  - We want to find parameters  $\Theta$  which minimize  $J(\Theta)$
  - To do so we can use one of the algorithms already described such as
    - Gradient descent
    - Advanced optimization algorithms
- To minimize a cost function we just write code which computes the following
  - **$J(\Theta)$** 
    - i.e. the cost function itself!
    - Use the formula above to calculate this value, so we've done that
  - **Partial derivative terms**
    - So now we need some way to do that
      - This is not trivial!  $\Theta$  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
      - i.e. each layer has a  $\Theta$  matrix associated with it!
        - We want to calculate the partial derivative  $\Theta$  with respect to a single parameter
$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$
  - Remember that the partial derivative term we calculate above is a REAL number (not a vector or a matrix)
    - $\Theta$  is the input parameters
      - $\Theta^1$  is the matrix of weights which define the function mapping from layer 1 to layer 2
      - $\Theta_{10}^1$  is the real number parameter which you multiply the bias unit (i.e. 1) with for the bias unit input into the first unit in the second layer
      - $\Theta_{11}^1$  is the real number parameter which you multiply the first (real) unit with for the first input into the first unit in the second layer
      - $\Theta_{21}^1$  is the real number parameter which you multiply the first (real) unit with for the first input into the second unit in the second layer
      - As discussed,  $\Theta_{ij}^l$ 
        - 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)
      - NB
        - *The terms destination node, origin node and origin layer are terms I've made up!*
  - So - this partial derivative term is
    - The partial derivative of a 3-way indexed dataset with respect to a real number (which is one of the values in that dataset)
  - **Gradient computation**
    - One training example
    - Imagine we just have a single pair (x,y) - entire training set
    - How would we deal with this example?
    - The forward propagation algorithm operates as follows
      - **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**
        - $a^4 = h_{\Theta}(x) = g(z^4)$



- This is the vectorized implementation of forward propagation
  - Lets compute activation values sequentially (below just re-iterates what we had above!)

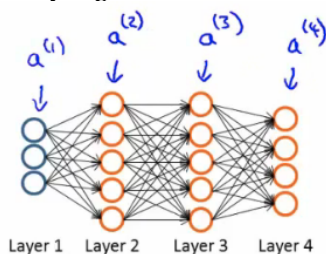
$$\begin{aligned}
 a^{(1)} &= x \\
 z^{(2)} &= \Theta^{(1)} a^{(1)} \\
 a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\
 z^{(3)} &= \Theta^{(2)} a^{(2)} \\
 a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\
 z^{(4)} &= \Theta^{(3)} a^{(3)} \\
 a^{(4)} &= h_{\Theta}(x) = g(z^{(4)})
 \end{aligned}$$

### What is back propagation?

- Use it to compute the partial derivatives
- Before we dive into the mechanics, let's get an idea regarding the intuition of the algorithm
  - For each node we can calculate  $\delta_j^{(l)}$  - this is **the error of node j in layer l**
    - If we remember,  $a_j^{(l)}$  is the activation of node j in layer l
    - Remember the activation is a totally calculated value, so we'd expect there to be some error compared to the "real" value
      - The delta term captures this error
      - But the problem here is, "what is this 'real' value, and how do we calculate it?!"
        - The NN is a totally artificial construct
        - The only "real" value we have is our actual classification (our y value) - so that's where we start
- If we use our example and look at the fourth (output) layer, we can first calculate
  - $\delta_j^{(4)} = a_j^{(4)} - y_j$ 
    - [Activation of the unit] - [the actual value observed in the training example]
    - We could also write  $a_j^{(4)}$  as  $h_{\Theta}(x)_j$ 
      - Although I'm not sure why we would?
  - This is an individual example implementation
- Instead of focussing on each node, let's think about this as a vectorized problem
  - $\delta^4 = a^4 - y$ 
    - So here  $\delta^4$  is the vector of errors for the 4th layer
    - $a^4$  is the vector of activation values for the 4th layer
- With  $\delta^4$  calculated, we can determine the error terms for the other layers as follows;
 
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \cdot g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \cdot g'(z^{(2)})$$
- Taking a second to break this down
  - $\Theta^3$  is the vector of parameters for the 3->4 layer mapping
  - $\delta^4$  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  $z^3$ 
    - You can do the calculus if you want (...), but when you calculate this derivative you get
    - $g'(z^3) = a^3 \cdot (1 - a^3)$
  - So, more easily
    - $\delta^3 = (\Theta^3)^T \delta^4 \cdot (a^3 \cdot (1 - a^3))$
  - $\cdot$  is the element wise multiplication between the two vectors
    - Why element wise? Because this is essentially an extension of individual values in a vectorized implementation, so element wise multiplication gives that effect
    - We highlighted it just in case you think it's a typo!

### Analyzing the mathematics



- And if we take a second to consider the vector dimensionality (with our example above [3-5-5-4])
  - $\Theta^3$  is a matrix which is [4 X 5] (if we don't include the bias term, 4 X 6 if we do)

- $(\Theta^3)^T$  = therefore, is a  $[5 \times 4]$  matrix
  - $\delta^4$  = is a  $4 \times 1$  vector
  - So when we multiply a  $[5 \times 4]$  matrix with a  $[4 \times 1]$  vector we get a  $[5 \times 1]$  vector
  - Which, low and behold, is the same dimensionality as the  $a^3$  vector, meaning we can run our pairwise multiplication
- For  $\delta^3$  when you calculate the derivative terms you get  $a^3 \cdot (1 - a^3)$
- Similarly For  $\delta^2$  when you calculate the derivative terms you get  $a^2 \cdot (1 - a^2)$ 
  - So to calculate  $\delta^2$  we do  $\delta^2 = (\Theta^2)^T \delta^3 \cdot (a^2 \cdot (1 - a^2))$
- There's no  $\delta^1$  term
  - Because that was the input!

### Why do we do this?

- We do all this to get all the  $\delta$  terms, and we want the  $\delta$  terms because through a very complicated derivation you can use  $\delta$  to get the partial derivative of  $\Theta$  with respect to individual parameters (if you ignore regularization, or regularization is 0, which we deal with later)
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^l \delta_i^{(l+1)}$
- By doing back propagation and computing the delta terms you can then compute the **partial derivative terms**
  - We need the partial derivatives to minimize the cost function!

### Putting it all together to get the partial derivatives!

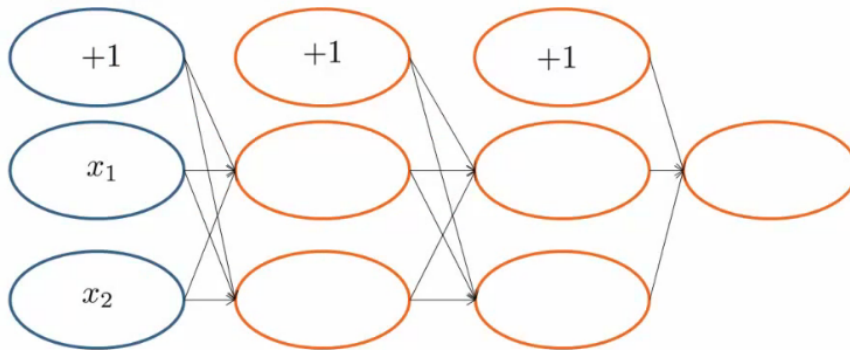
- What is really happening - lets look at a more complex example
- Training set of  $m$  examples  
Training set  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
- **First**, set the delta values  
Set  $\Delta_{ij}^{(l)} = 0$  (for all  $l, i, j$ )
  - Set equal to 0 for all values
  - Eventually these  $\Delta$  values will be used to compute the partial derivative
    - Will be used as accumulators for computing the partial derivatives
- **Next**, loop through the training set  
For  $i = 1$  to  $m$ 
  - i.e. for each example in the training set (dealing with each example as  $(x, y)$ )
  - Set  $a^1$  (activation of input layer) =  $x^1$
  - **Perform forward propagation** to compute  $a^l$  for each layer ( $l = 1, 2, \dots, L$ )
    - i.e. run forward propagation
  - **Then**, use the output label for the specific example we're looking at to calculate  $\delta^L$  where  $\delta^L = a^L - y^i$ 
    - So we initially calculate the delta value for the output layer
    - Then, using **back propagation** we move back through the network from layer  $L-1$  down to layer
  - Finally, use  $\Delta$  to accumulate the partial derivative terms  
 $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
  - Note here
    - $l$  = layer
    - $j$  = node in that layer
    - $i$  = the error of the affected node in the target layer
  - You can vectorize the  $\Delta$  expression too, as  
 $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
- **Finally**
  - After executing the body of the loop, exit the for loop and compute  
 $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$  if  $j \neq 0$   
 $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$  if  $j = 0$ 
    - When  $j = 0$  we have no regularization term
- At the end of ALL this
  - You've calculated all the  $D$  terms above using  $\Delta$ 
    - NB - each  $D$  term above is a real number!
  - 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

- We can then use these in gradient descent or one of the advanced optimization algorithms
- Phew!
  - What a load of hassle!

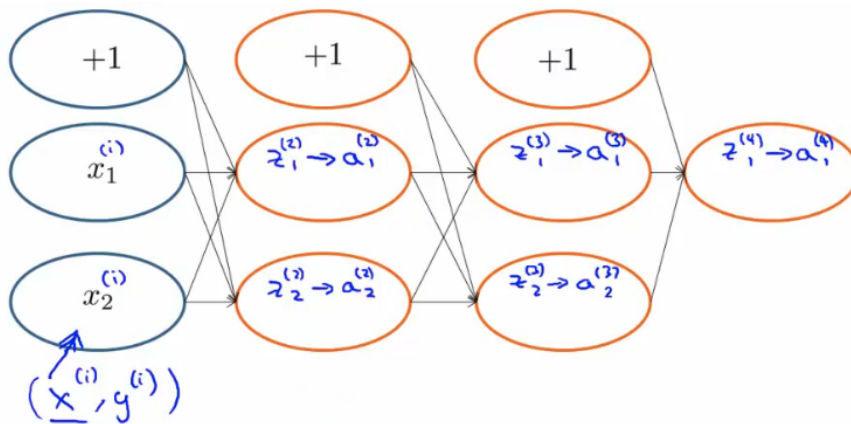
## Back propagation intuition

- Some additional back propagation notes
  - In case you found the preceding unclear, which it shouldn't be as it's fairly heavily modified with my own explanatory notes
- Back propagation is hard(ish...)
  - But don't let that discourage you
  - It's hard in as much as it's confusing - it's not difficult, just complex
- Looking at mechanical steps of back propagation

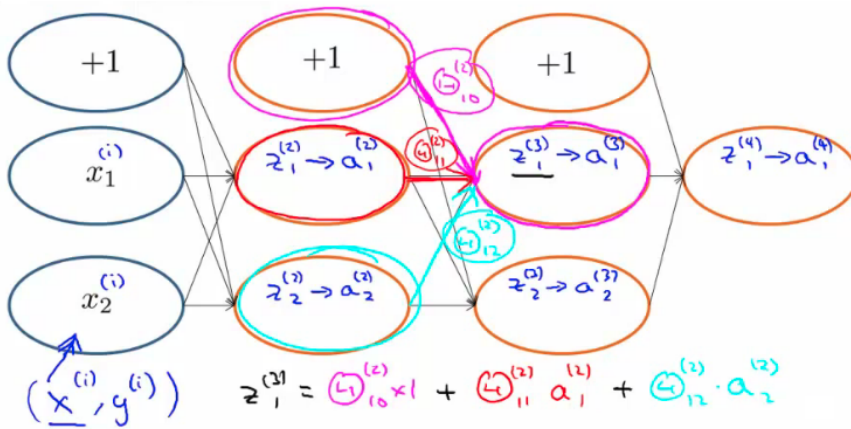
### Forward propagation with pictures!



- Feeding input into the input layer ( $x^i, y^i$ )
  - Note that  $x$  and  $y$  here are vectors from 1 to  $n$  where  $n$  is the number of features
    - So above, our data has two features (hence  $x_1$  and  $x_2$ )
- With out input data present we use **forward propagation**



- The sigmoid function applied to the  $z$  values gives the activation values
  - Below we show exactly how the  $z$  value is calculated for an example



### Back propagation

- With forwardprop done we move on to do back propagation
- Back propagation is doing something very similar to forward propagation, but backwards
  - Very similar though
- Let's look at the cost function again...
  - Below we have the cost function if there is a single output (i.e. binary classification)

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

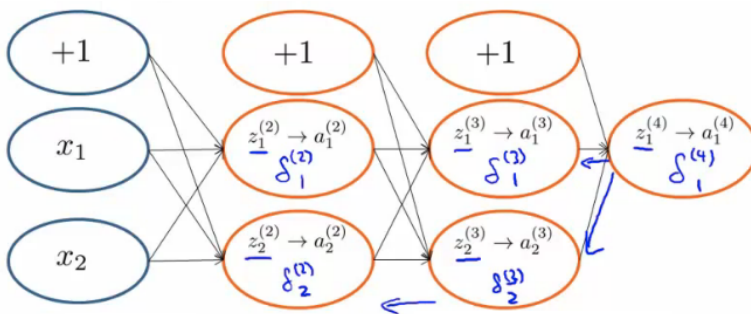
- This function cycles over each example, so the cost for one example really boils down to this

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

- Which, we can think of as a sigmoidal version of the squared difference (check out the derivation if you don't believe me)
  - So, basically saying, "how well is the network doing on example  $i$ "?
- We can think about a  $\delta$  term on a unit as the "error" of cost for the activation value associated with a unit
  - More formally (*don't worry about this...*),  $\delta$  is

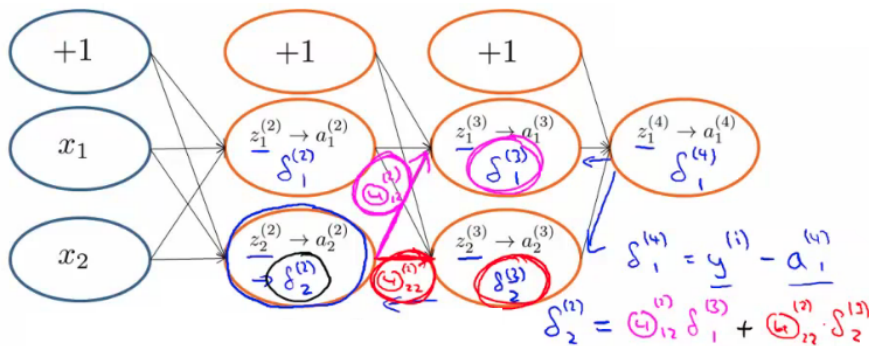
$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$$

- Where cost is as defined above
- Cost function is a function of  $y$  value and the hypothesis function
- So - for the output layer, back propagation sets the  $\delta$  value as  $[a - y]$ 
  - Difference between activation and actual value
- We then propagate these values backwards;



- Looking at another example to see *how* we actually calculate the delta value;





- So, in effect,
  - Back propagation calculates the  $\delta$ , and those  $\delta$  values are the weighted sum of the next layer's delta values, weighted by the parameter associated with the links
  - Forward propagation calculates the activation ( $a$ ) values, which
- Depending on how you implement you may compute the delta values of the bias values
  - However, these aren't actually used, so it's a bit inefficient, but not a lot more!

## Implementation notes - unrolling parameters (matrices)

- Needed for using advanced optimization routines

```
function [jVal, gradient] = costFunction(theta)
...
optTheta = fminunc(@costFunction, initialTheta, options)
```

- Is the MATLAB/octave code
  - But theta is going to be matrices
- fminunc takes the costfunction and initial theta values
  - These routines assume theta is a parameter vector
  - Also assumes the gradient created by costFunction is a vector
- For NNs, our parameters are matrices
  - e.g.

Neural Network (L=4):

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$  - matrices (**Theta1, Theta2, Theta3**)

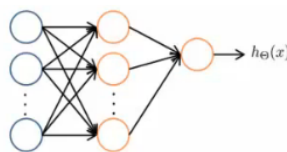
$D^{(1)}, D^{(2)}, D^{(3)}$  - matrices (**D1, D2, D3**)

**Example**

$$s_1 = 10, s_2 = 10, s_3 = 1$$

$$\Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$$

$$D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$$



- Use the `thetaVec = [ Theta1(:); Theta2(:); Theta3(:) ]`; notation to unroll the matrices into a long vector
- To go back you use
  - `Theta1 = reshape(thetaVec(1:110), 10, 11)`

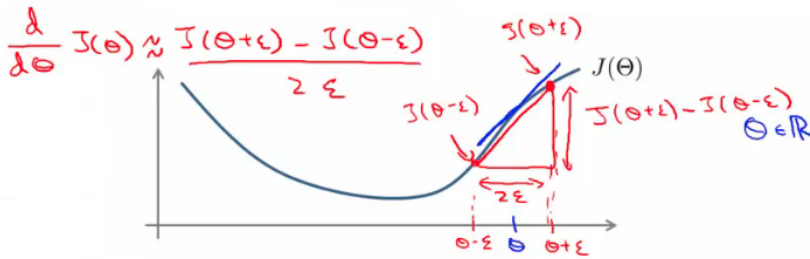
## Gradient checking

- Backpropagation has a lot of details, small bugs can be present and ruin it :- (
  - This may mean it looks like  $J(\Theta)$  is decreasing, but in reality it may not be decreasing by as much as it should
- So using a numeric method to check the gradient can help diagnose a bug
  - Gradient checking helps make sure an implementation is working correctly
- **Example**
  - Have an function  $J(\Theta)$
  - Estimate derivative of function at point  $\Theta$  (where  $\Theta$  is a real number)
  - How?



- Numerically

- Compute  $\Theta + \epsilon$
- Compute  $\Theta - \epsilon$
- 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
- The is the two sided difference (as opposed to one sided difference, which would be  $J(\Theta + \epsilon) - J(\Theta) / \epsilon$ )
- If  $\Theta$  is a vector with  $n$  elements we can use a similar approach to look at the partial derivatives

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon}$$

⋮

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon}$$

- So, in octave we use the following code the numerically compute the derivatives

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

- So on each loop `thetaPlus = theta` except for `thetaPlus(i)`
  - Resets `thetaPlus` on each loop
- Create a vector of partial derivative approximations
- Using the vector of gradients from backprop (DVec)
  - Check that `gradApprox` is basically equal to `DVec`
  - Gives confidence that the Backprop implementation is correct
- 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)
  - Before using the code for learning turn off gradient checking

- Why?

- `GradApprox` stuff is very computationally expensive
- In contrast backprop is much more efficient (just more fiddly)

## Random initialization

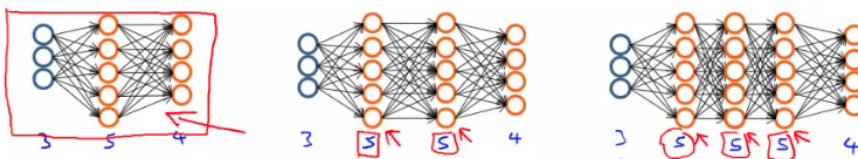
- Pick random small initial values for all the theta values
  - If you start them on zero (which does work for linear regression) then the algorithm fails - all activation values for each layer are the same
- So chose random values!

- Between 0 and 1, then scale by epsilon (where epsilon is a constant)

## Putting it all 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 computationally expensive
- We'll discuss architecture more later



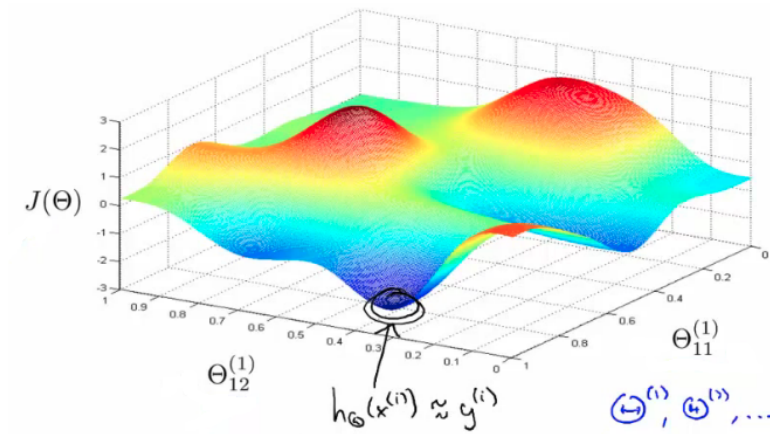
### • 2) - Training a neural network

- **2.1)** Randomly initialize the weights
  - Small values near 0
- **2.2)** Implement forward propagation to get  $h_{\Theta}(x)^i$  for any  $x^i$
- **2.3)** Implement code to compute the cost function  $J(\Theta)$
- **2.4)** Implement back propagation to compute the partial derivatives
- General implementation below

```
for i = 1:m {
    Forward propagation on  $(x^i, y^i)$  --> get activation (a) terms
    Back propagation on  $(x^i, y^i)$  --> get delta ( $\delta$ ) terms
    Compute  $\Delta := \Delta^1 + \delta^{1+1}(a^1)^T$ 
}
```

With this done compute the partial derivative terms

- Notes on implementation
  - Usually done with a for loop over training examples (for forward and back propagation)
  - Can be done without a for loop, but this is a much more complicated way of doing things
  - Be careful
- **2.5)** Use gradient checking to compare the partial derivatives computed using the above algorithm and numerical estimation of gradient of  $J(\Theta)$ 
  - Disable the gradient checking code for when you actually run it
- **2.6)** Use gradient descent or an advanced optimization method with back propagation to try to minimize  $J(\Theta)$  as a function of parameters  $\Theta$ 
  - Here  $J(\Theta)$  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



- e.g. above pretending data only has two features to easily display what's going on
  - Our minimum here represents a hypothesis output which is pretty close to  $y$
  - If you took one of the peaks hypothesis is far from  $y$
- Gradient descent will start from some random point and move downhill
  - Back propagation calculates gradient down that hill