**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

**BITS C464 – MACHINE LEARNING**

**I Semester 2014-2015**

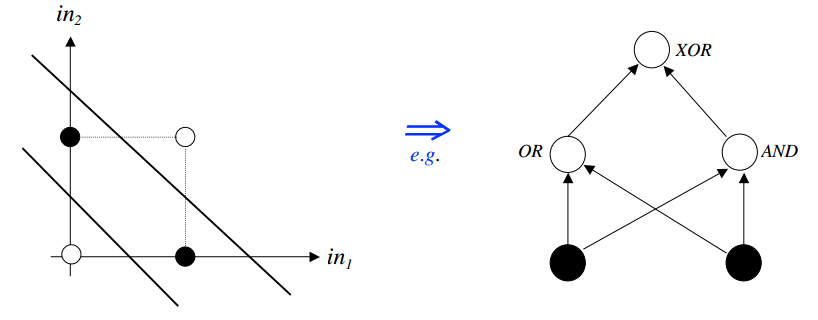
**WORKSHEET #5**

**MultiLayer Perceptron Model using Backpropogation**

**OBJECTIVE:-**

* **Multilayer Layer Perceptron model**
* **Learning a digit recognizer using Artificial Neural Network**

Remember that it is not possible to find weights which enable Single Layer Perceptrons to deal with non-linearly separable problems like XOR:

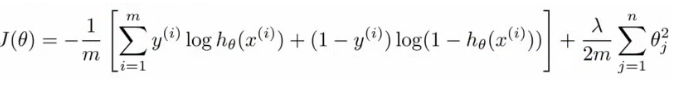


However, Multi-Layer Perceptrons (MLPs) are able to cope with non-linearly separable problems.

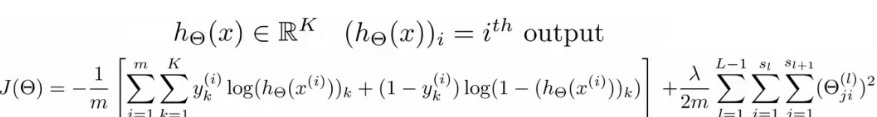
An MLP is a network of simple *neurons* called *perceptrons*. The basic concept of a single perceptron was introduced by Rosenblatt in 1958. The perceptron computes a single *output* from multiple real-valued *inputs* by forming a linear combination according to its input *weights* and then possibly putting the output through some nonlinear activation function. Mathematically this can be written as

|  |
| --- |
| where w denotes the vector of weights, x is the vector of inputs, b is the bias and is the activation function. A signal-flow graph of this operation is shown in Figure. in multilayer networks, the activation function is often chosen to be the logistic sigmoid  or the hyperbolic tangent     A single perceptron is not very useful because of its limited mapping ability. No matter what activation function is used, the perceptron is only able to represent an oriented ridge-like function. The perceptrons can, however, be used as building blocks of a larger, much more practical structure. A typical *multilayer* perceptron (MLP) network consists of a set of source nodes forming the *input layer*, one or more *hidden layers* of computation nodes, and an *output layer* of nodes. The input signal propagates through the network layer-by-layer. The signal-flow of such a network with one hidden layer is shown in Figure |

* 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**
  + We use the partial derivatives with gradient descent to try minimize the cost function and update all the Ɵ values
  + This repeats until gradient descent reports convergence
* The (regularized) logistic regression cost function is as follows;



* For multilayer perceptron our cost function is a generalization of this equation above, so instead of one output we generate *k* outputs

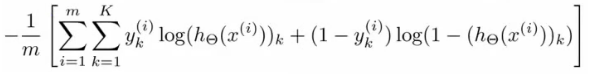


* Our cost function now outputs a *k* dimensional vector
  + hƟ(x) is a k dimensional vector, so hƟ(x)*i* refers to the ith value in that vector
* Cost function J(Ɵ) is
  + [-1/m] times a sum of a similar term to which we had for logic regression
  + But now this is also a sum from k = 1 through to K (K is number of output nodes)

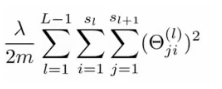
There are basically two halves to the neural network logistic regression cost function

**First half**

* 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



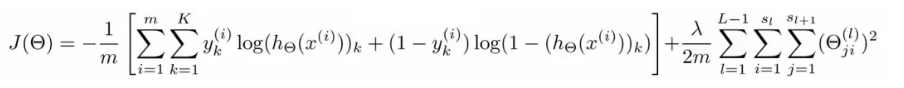
**Second half**



* This is also called a **weight decay** term

**Back propagation algorithm**

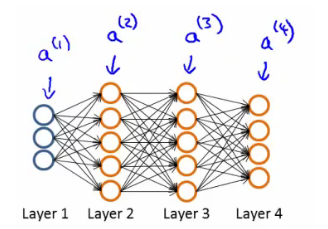
* Now we're going to deal with **back propagation**
  + Algorithm used to minimize the cost function, as it **allows us to calculate partial derivatives**!



* The cost function used is shown above
  + We want to find parameters Ɵ which minimize J(Ɵ)
  + To do so we can use one of the algorithms already described such as
    - Gradient descent
* To minimize a cost function we just write code which computes the following
  + **J(Ɵ)**
    - 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! Ɵ 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 Ɵ matrix associated with it!
        + We want to calculate the partial derivative Ɵ with respect to a single parameter



* + - Remember that the partial derivative term we calculate above is a REAL number (not a vector or a matrix)
  + **Gradient computation**
* **Layer 1**
  + a1 = x
  + z2 = Ɵ1a1
* **Layer 2**
  + a2 = g(z2) (add a02)
  + z3 = Ɵ2a2
* **Layer 3**
  + a3 = g(z3) (add a03)
  + z4 = Ɵ3a3
* **Output**
  + a4 = hƟ(x) = g(z4)



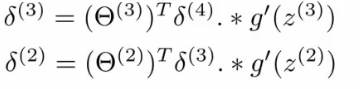
What is really happening - lets look at a more complex example

* Training set of m examples



* **First**, set the delta values



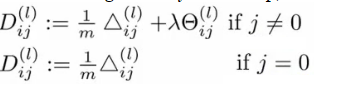
* + Set equal to 0 for all values
  + Eventually these Δ values will be used to compute the partial derivative
    - Will be used as accumulators for computing the partial derivatives
* **Next**, loop through the training set  
   
  + i.e. for each example in the training set (dealing with each example as (x,y)
  + Set a1(activation of input layer) = xi
  + **Perform** **forward propagation** to compute alfor each layer (l = 1,2, ... L)
    - i.e. run forward propagation
  + **Then**, use the output label for the specific example we're looking at to calculate δL where δL= aL- yi
    - 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
    - For error in hidden layer
    - 
  + Finally, use Δ to accumulate the partial derivative terms



* + 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, as



* **Finally**
  + After executing the body of the loop, exit the for loop and compute



* + - When j = 0 we have no regularization term
  + After calculating gradient vectors pass cost function and gradient to an optimization library provided and it will give optimal parameters of ANN and use it to predict the label of given dataset

**Exercise:**

1. Load the character recognition dataset given
2. Learn the underlying hypothesis using backpropagation algorithm(using the optimization library provided.