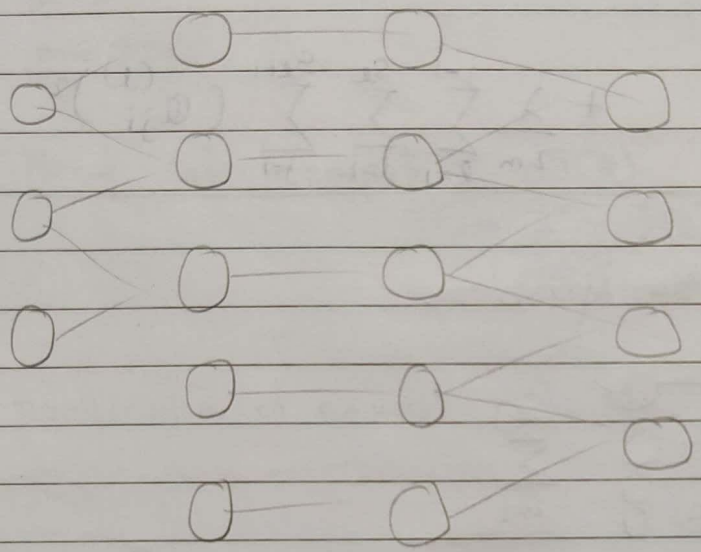
Week 5

-- @ Cost function — — —

Focus → Application of Neural networks on classification problems.

Neural Network (classification)



Training data
$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, x^{(m)}, y^{(m)})\}$

L = total no of layers in network.

$L = 4$

S_l = no of units in layer l (not counting bias unit)

layer 1 layer 2 layer 3 layer 4

$S_1 = 3,\ S_2 = 5,\ S_3 = 5,\ S_4 = S_L = 4$

- Binary classification:

  $y = 0$ / $y = 1$

  1 output unit

  $h_\Theta(x) \in \mathbb{R}$

  $S_L = 1$

  $K = 1$

- Multiclass classification (K classes) ($K \geq 3$)

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

  K output units.

  $h_\Theta(x) \in \mathbb{R}^K$

  $S_L = K$

logistic regn cost fun ~~cost~~ for generalization → neural network

Sadguru
Page No: 55
Date: / /

Cost function

- logistic regn.

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

- neural network.

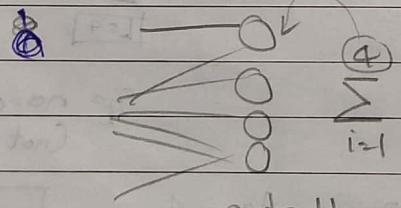
$h_\Theta(x) \in \mathbb{R}^K$ → K dimensional vector

$(h_\Theta(x))_i = i^{th}$ output → selects i^{th} element of vector

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

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

($K=1$) if we have binary classification problem.



- Basically the logistic regression algorithm's cost function but summing that cost function over each of my four output units in turn.

- In regularization term, basically we are summing Θ_{ij} except we don't sum over terms corresponding bias values ($i=0$)

Σ Σ

Doesn't make much difference

Que Suppose we try to minimize $J(\Theta)$, we need to supply code to compute ?

→ $J(\Theta)$ and partial derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$

for every i, j, l.

--@ Back propagation Algorithm — — —

☆ Back propogation algorithm

Gradient computation

- $J(\Theta)$
- $\min_{\Theta} J(\Theta)$

Need code to compute → $J(\Theta)$

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

→ parameter of neural network $\Theta_{ij}^{(l)} \in \mathbb{R}$

The flow of algorithm

- Given one training example (x, y):
- Forward propagation: → what will output

$a^{(1)} = x$

$z^{(2)} = \Theta^{(1)} a^{(1)}$

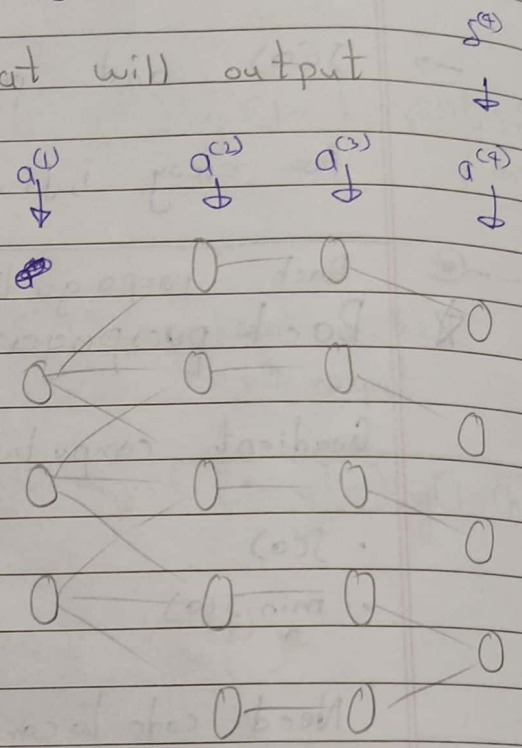
$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$) add bias term

$z^{(3)} = \Theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)

$z^{(4)} = \Theta^{(3)} a^{(3)}$

$a^{(4)} = h_\Theta(x) = g(z^{(4)})$



- Gradient computation: Backpropagation algorithm
  → To compute partial derivatives

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

$a_j^{(l)}$ = "activation"

For each output unit (layer $L = 4$)

$\delta_j^{(4)} = a_j^{(4)} - y_j$

$(h_\Theta(x))_j$

label value from training example.

$\delta^{(4)} = a^{(4)} - y$ → (vectorize Implementation)

→ Dimension = # output units (K)

.* → Element wise multiplication of these two vectors.

$g'(z^{(3)})$ → Dealvation of activation function g, elevaated at the input values given by ~~z(3)~~ $z^{(3)}$

(2 PTO for detail)

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

$$\rightarrow a^{(3)} .* (1 - a^{(3)})$$

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

$$\rightarrow a^{(2)} .* (1 - a^{(2)})$$

No $\delta^{(1)}$ term → no errors, we don't want to change input.

Back propagating errors from output layers to layer 3.

$\delta^{(4)} \leftarrow \delta^{(3)} \leftarrow \delta^{(4)}$

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

(Ignoring λ / if $\lambda = 0$ / Ignoring regularization term
↳ Fix latter)

* • Back propagation algorithm.

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

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

Δ's are going to be used as accumulators that will slowly add things in order to compute $\frac{\partial}{\partial}$ partial derivatives.

loop:

For $i = 1$ to m --- i^{th} iteration corresponds to $(x^{(i)}, y^{(i)})^{th}$ training set.

Input layer ← $((x^{(i)}, y^{(i)}))$

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

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

vectorization →

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

$D_{ij}^{(i)} := \frac{1}{m} \Delta_{ij}^{(i)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$ — Two

$D_{ij}^{(i)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$ — cases

corresponds to bias term

$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$ ← This will be used in gradient descent

que $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$ FP = forward prop / BP = Backward propagation

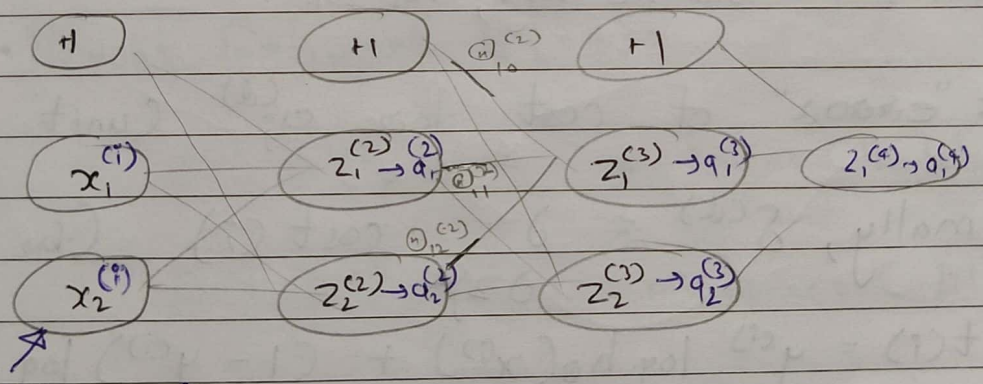
Sequence of operation.

→ FP using $x^{(1)}$ followed by BP using $y^{(1)}$.
Then
FP using $x^{(2)}$ ——————→ $y^{(2)}$

~~Forward Pr~~ In this vid :- How to use back propag. to compute derivative of your cost fun.

Back Propagation intuition — — —

• Foaward propagation.



$(x^{(i)}, y^{(i)})$

$$z_1^{(3)} = \Theta_{10}^{(2)} x_1 + \Theta_{11}^{(2)} a^{(2)} + \Theta_{12}^{(2)} a_1^{(2)}$$

At a time we use only one example, FP it · BP it.

Page No: 61 Sadguru
Date: / /

- What is back propagation doing?

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

Focussing on a single example $x^{(i)}, y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$)

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

(Think of $\text{cost}(i) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$

↳ Measures i.e.: How well is neural network doing on example?

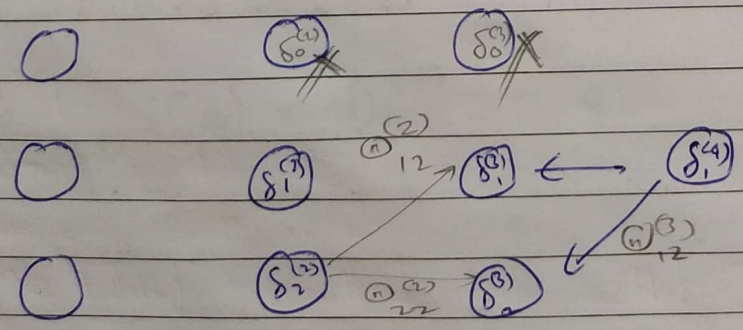
→ Just to clarify intuition.
Works in similar way, not same.

$\delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit j in layer l)

Formally, $\delta^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$ (for $j \geq 0$), where

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

- Backpropagation is calculation $\delta_j^{(l)}$
↳ quote error of activation value.



Implementation detail of unrolling pa... from mat... into vect... which we need in order ... use adva... optimization rou...

$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$

$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \delta_2^{(3)}$

$\delta_2^{(3)} = \Theta_{12}^{(3)} \delta_1^{(4)}$

-- @ Implementation Note Unrolling — — —

Implementation detail of unrolling parameters from matrices into vectors, which we need in order to use advanced optimization routine.

Unrolling Parameters :- matrices → Vectors.

Implementation note:-

- Advanced optimization for to → logistic regn.

```
function [jVal, gradient] = costFunction(theta)
```

output = cost fn & derivatives; gradient ∈ $\mathbb{R}^{n+1}$; input ∈ $\mathbb{R}^{n+1}$ vectors

```
...
optTheta = fminunc(@costFunction, initialTheta, options)
```

Advanced optimization algorithm

- Neural Network (L=4):

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

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

"unroll" into vectors so they end up being in format suitable for passing in to as theta off for getting out for gradient there

→ so, we could pass into fminunc (vectors)

+1 +1
10 10 1

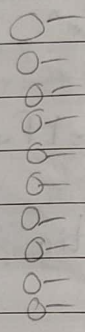
e.g. input layer, hidden layer, output layer.

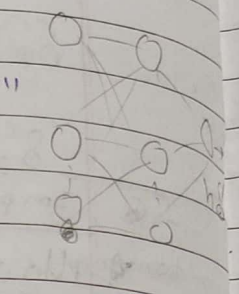
$S_1 = 10$, $S_2 = 10$, $S_3 = 1$

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

$\mathbb{R}^{(S_{j+1}) \times (S_j)}$

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





thetaVec = [Theta1(:) ; Theta2(:) ; Theta3(:)];

Vector — Matrix's

DVec = [D1(:) ; D2(:) ; D3(:)];

Theta1 = reshape (thetaVec (1:110) , 10, 11);

Theta2 = reshape (thetaVec (111:220) , 10 , 11);

Theta3 = reshape (thetaVec (221:231) , 1, 11);

We Just converted theta [Matrix] to theta [Vector]

Que: D_1 is 10×6 matrix & D_2 is a 1×11 matrix.
You set DVec = [D1(:); D2(:)];
How to get D2 back from DVec.
→ reshape(DVec(61:71), 1, 11)

Page No: 64 Sadguru
Date: / /

• Learning Algorithm.

• Have initial parameter $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$

• Unroll to get long vector initialTheta to pass to

fminunc(@costFunction, initialTheta, options)

Demo:→→→

>> Theta1 = ones(10, 11)

>> Theta2 = 2 * ones(10, 11)

>> Theta3 = 3 * ones(10, 11)

>> thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
>> size(thetaVec) = 231, 1
>> thetaVec =

2
2
3
3

>> reshape(thetaVec(1:110), 10, 11)

>> reshape(thetaVec(111:220), 10, 11)

>> reshape(thetaVec(221:231), 1, 11)

Advantage of Matrix representation: easier to do forward prop & back prop.

Advantage of Vector representation: Easier to use advanced optimization algorithms (CAOA) because they tend to think our parameters are unrolled in big vector

Implementing cost function.

(2) function [jVal, gradientVec] = costFunction (thetaVec)

$\Theta_1, \Theta_2, \Theta_3$ will be in matrices form

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

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

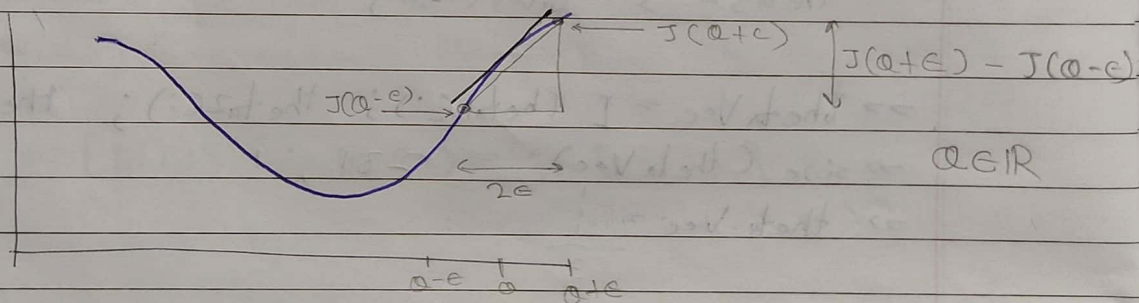
I think its calculating DVec

Unroll $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ to get gradientVec.

(3) Gradient Checking

→ Gradient Checking → Checking whether it is running right or not.

Numerical estimation of Gradients :-



Here we want slope of /

→ we estimate slope of both lines to be same.

double sided difference

one sided difference

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

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

→ If we take Θ very small, it is actually defination of derivative.

$\epsilon = 10^{-4}$ (In range of 10^{-4})

Que $J(\Theta) = \Theta^3$, let $\Theta = 1$, $\epsilon = 0.01$

$\frac{\partial}{\partial \Theta} J(\Theta) = 3$, what you get by approximation formula $\Rightarrow 3.0001$

Previously we considered Θ as rolled parameter

Moving on more general case.

* Parameter Vector Θ.

Maths

- $\Theta \in \mathbb{R}^n$ — (e.g. Θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)
- $\Theta = \Theta_1, \Theta_2, \ldots, \Theta_n$

$$\Rightarrow \frac{\partial}{\partial \Theta_1} J(\Theta) \approx \frac{J(\Theta_1 + \epsilon, \Theta_2, \Theta_3, \ldots \Theta_n) - J(\Theta_1 - \epsilon, \Theta_2, \ldots \Theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \Theta_2} J(\Theta) \approx \frac{J(\Theta_1, \Theta_2 + \epsilon, \Theta_3, \ldots, \Theta_n) - J(\Theta_1, \Theta_2 - \epsilon, \Theta_3, \ldots \Theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \Theta_n} J(\Theta) \approx \frac{J(\Theta_1, \Theta_2, \ldots, \Theta_n + \epsilon) - J(\Theta_1, \Theta_2, \ldots, \Theta_n - \epsilon)}{2\epsilon}$$

Program

```
for i = 1:n  ⟶ dimension of Θ
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus)) / (2*EPSILON)
                    ↳ ∂J(Θ)/∂Θi
end;
```

i We implement for loop to calculate top partial derivatives of the cost function for respect to every parameter in that network. and

P.T.O.

Gradient = slope.

Page No: 67 Sadguru
Date: / /

computes slowly — numerically computed derivative — computes fast

Check that gradApprox ≈ DVec

Backpoop text

This must be equal, at least for few decimals. To make sure everything is ✓

back poop

ii) We can then take the gradient that we got from back poop.

- DVec was derivative we get from back poop.
- Back poop was relatively efficient way to compute a derivative or a partial derivative of a cost fun with respect to all our parameters.

Now we can use DVec in advanced optimization algorithm

Implementation Note :-

1) Implement back poop to compute DVec (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
2) Implement numerical gradient check to compute gradApprox.
3) Make sure they give similar values. (1 & 2)
4) Turn off gradient checking. Using back poop code for learning.

Just more on, everything is ok, go on using back poop to train model.

Disable gradient checking code before training classifier

↓ computes slow

Que → We use back prop over numerical gradient computation during learning.
because
numerical gradient algo is slow.

--@ Random initialization

Before running any algo, gradient descent or any other advanced optimization
We need to pick some initial value for parameter theta.
Even advanced optimization algo, assumes we will pass initialTheta
After initializing, we will go to gradient descent to minimize J(θ)

We initialize theta = [0,0] in regression (even in logistic)
This will not work in neural network.

> optTheta = fminunc(@costFunction, initialTheta, options)

★ Zero Initialization

Random youtube video:—

Bias → initialize all to zero (0)

Weights →
- Can't initialize to zero → then all gradients will be zero.

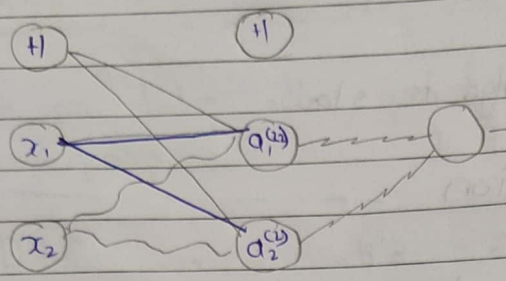
↳ Can't initialize all weights to same value.
↳ all hidden units will behave same.

Soln → $W_{ij}^{(k)}$ from $U[-b, b]$, where $b = \frac{\sqrt{6}}{\sqrt{H_x + H_{x-1}}}$

H_x = no of units in layer x.

★ Zero initialization

same weights →



If we initialize $\Theta_{ij}^{(l)} = 0$, for all i, j, l.

$$a_1^{(2)} = a_2^{(2)}$$

Because outgoing weights are same.

$$\delta_1^{(2)} = \delta_2^{(2)}$$

This means

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

→ Even after one gradient descent update

i.e. $\Theta_{01}^{(1)} := \alpha \frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta)$

$$\Theta_{02}^{(1)} = \alpha \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

It will remain same.

i.e. $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$ --- (non zero value, If changes from 0 to non zero, → but it will be equal)

Even after one another update, ~~both value will be same~~

similarly $\Theta_{011}^{(1)} = \Theta_{12}^{(1)}$ & $\Theta_{31}^{(1)} = \Theta_{32}^{(1)}$

So, even after one iteration, $a_1^{(2)} = a_2^{(2)}$

Hidden layer will have same functions as input (headed units are same)

So, this will highly redundant representation. because, it will consider only one feature, as all $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} \ldots$

- Therefore we go with random initialization.

✱ Random initialization: Symmetry breaking.

- problem of ~~symmetric~~ symmetric ways → The problem we faced in zero initialization. (Ways all being same)

Random initilization breaks symmetry

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\varepsilon, \varepsilon]$
(i.e. $-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$)

e.g. Theta1 = rand(10, 11) * (2 * INIT_EPSILON) - INIT_EPSILON;
→ (random 10×11 matrix) (between 0 & 1)

Theta2 = rand(1, 11) * (2 * INIT_EPSILON) - INIT_EPSILON;

Q a = rand(1,1) * (2 * INIT_EPSILON) - INIT_EPSILON; → a will only one number.
$\Theta_{ij}^{(l)} = a$ for all i, j, l

This will not work, & will be fail to break symmetry.

- The idea is to randomly initialize the waves to small values close to zero. b/w $-\epsilon$ & ϵ

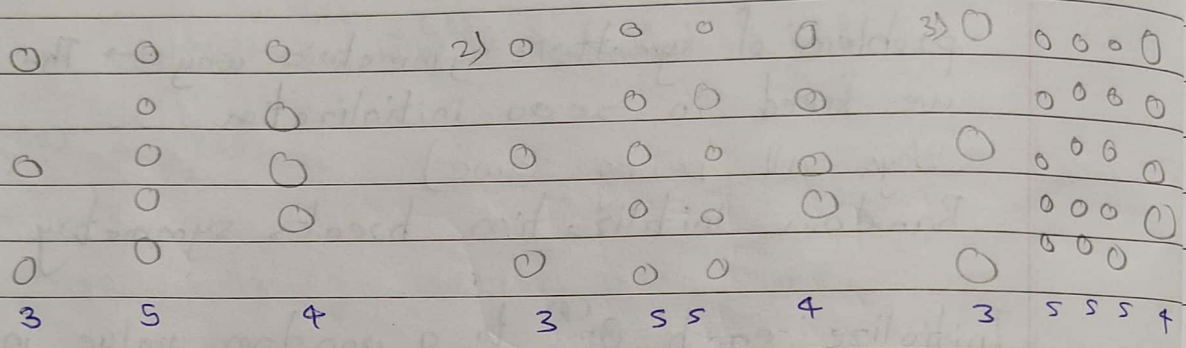
Then use gradient descent/any other advanced optimization video. to minimize $J(\Theta)$ as a function of parameters theta starting from randomly chosen initial value for parameters.

@ Putting it together

☆ Training a neural network.

To train neural network

↓ First choose architecture.



No. of input units: Dimension of features $x^{(i)}$

No. of output units: Number of classes no of classes in our classification problem.

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ or $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

$y \in \{1, 2, 3, \dots, 10\}$

$y = 5$ ✗

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$

$y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ✓

No of hidden layers

- Reasonable default :-
  - 1 hidden layer
    or
  - >1 hidden layer, having same no of hidden units in every layer (usually more hidden layer, the better) → but becomes computationally expensive.

1) is most common architecture.

No. of hidden must be same what comparable to input layers. (3 to 4 times no of inputs)

Training a neural network.

1) Randomly initialize weights (near to zero)
2) Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3) Implement ~~forward propagation to~~ code to compute cost function $J(\Theta)$
4) Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

P.T.O. to compute

The idea is, we take (x^1, y^1) (first training ex) FP & BP,
then we take (x^2, y^2) FP, BP
⋮
m times.

no of training examples.

for i = 1 : m
{
Perform forward propagation and back propagation using $(x^{(i)}, y^{(i)})$

(Get activations $a^{(l)}$) and delta term ($\delta^{(l)}$) for $l = 2, \dots, L$

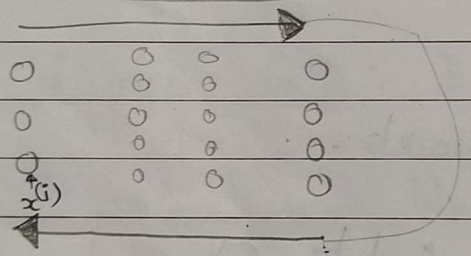
$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$$

}

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

$$D_{jk}^{(l)} := \frac{1}{m} \Delta_{jk}^{(l)} + \lambda \Theta_{jk}^{(l)} \quad \text{--- (If } j \neq 0)$$

$$D_{jk}^{(l)} := \frac{1}{m} \Delta_{jk}^{(l)} \quad \text{--- (If } j = 0)$$



5) Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using back propagation vs using mathematical numerical estimate of gradient of $J(\Theta)$

$$\text{gradApprox} \approx DVec$$

If found right, disable gradient checking code.

6) Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters (Θ)

BP computes

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

$J(\Theta)$ for neural network is non-convex

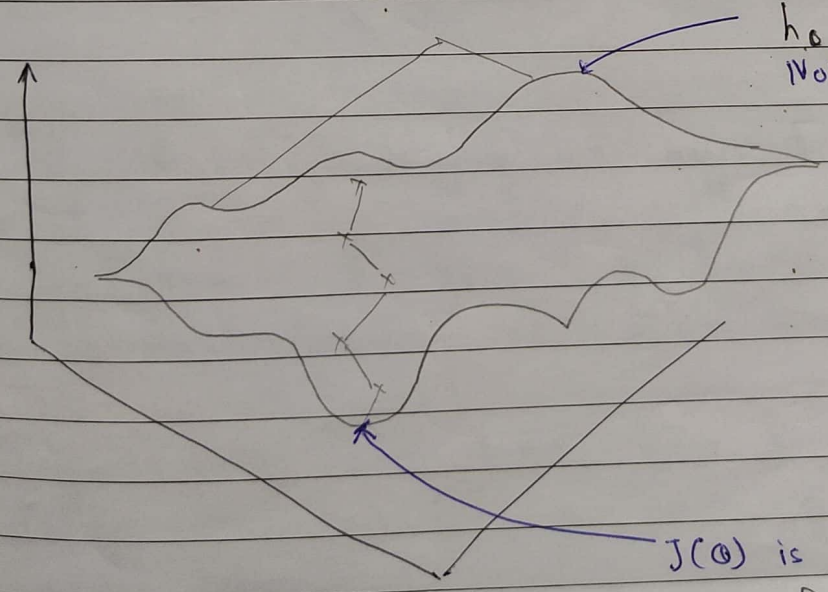
In theory algo like Gradient descent / AO method can stuck in local optimum.

but

In practice Gradient descent / AO finds very good local minimum.

It does not ~~gua~~ guarantee ~~to~~ global minimum.

→ In practice this is not a huge problem.



Gradient descent – start from random point
And it will repeatedly go downhill

Back propogation - Computes direction of Gradient

Gradient descent - Taking little steps downhill.
Until it gets global optimum
not necessary

Q If we use gradient descent with BP.
to minimize $J(\Theta)$ as a fn of Θ.
How to verify the learning algo is correct.

→

✓ ⇒ Plot $J(\Theta)$ as a function of number of iterations and make sure it is decreasing with every iteration.
(At least non-increasing)

⇒ Plot $J(\Theta)$ as a function of the number of iterations to make sure the parameter values are improving in classification accuracy