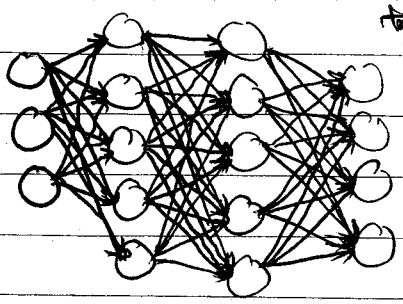
Neural Network:

Learning algorithm → To fit parameters of a neural network given a training set

Cost function

(application of neural network to classification problem)



training set → $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

L = total no. of layers in network ($L=4$ here)

s_l = no. of units (not counting bias unit) in layer l. $s_1=3$, $s_2=5$, $s_4=4=s_L$

Binary classification

$y = 0$ or 1

1 output unit

$h_\Theta(x)$

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

To simplify, $K=1$

$s_L = 1$ (L: index of final layer; s_L: num of layer we have in network; K: num of units in output layer)

Multi-class classification (K classes)

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

K output units

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

$s_L = K$ (num of output unit) $(K \geq 3)$

one-vs-all method when $K \geq 3$

Cost function we use for neural network is generalization of the one we use for logistic regression.

Cost Function

Logistic regression:

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

Note: Double sum adds up the logistic regression costs calculated for each cell in the output layer

Neural Network: $h_\Theta(x) \in \mathbb{R}^K$ $(h_\Theta(x))_i$ = i^{th} output (select out ith element of vector that is output by neural network)

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

Apply to y_k and $(h_\Theta(x^{(i)}))_k$

Sum K output, if have 4 output units:

example: $y_k \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

Then this is sum from $k=1$ to 4 of basically the logistic regression algorithm's cost function but summing that cost function over each of 4 output units in turn.

Because we take K output unit and compare that to the value of y_k which is that one of those vectors saying what cost it should be.

Regularization term

⇒ Summing all terms $\Theta_{ji}^{(l)}$ for all values of i, j, l except we don't sum over the terms corresponding to bias values (which is $i=0$) because we don't regularize and shrink their values as zero.

Note: Triple sum adds up the square of all the individual Θs in entire

No. _____

Date _____

Quiz:

Suppose we want To minimize $J(\Theta)$ as a function of Θ, using one of the advanced optimization methods (fminunc, conjugate gradient, BFGS, L-BFGS etc). What do we need to supply code to compute (as a function of Θ)?

Answer: $J(\Theta)$ and the (partial) derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ for every i, j, l

Neural Networks:

Learning

Backpropagation algorithm (minimize cost function)

Gradient computation

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

try to find parameter theta to

$$\min_\Theta J(\Theta)$$

Want to use gradient descent / advance optimization algorithms,

Need code to compute:

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

parameter in neural network: $\Theta_{ij}^{(l)} \in \mathbb{R}$ (real number)

Gradient computation

Given one training example (x, y):

Forward propagation:

$a^{(1)} = x$

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

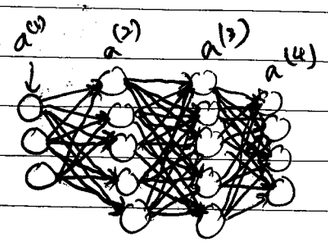
$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$) ← add bias unit

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

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

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

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



No. _____

Date _____

In order to compute the derivatives, use algorithm called backpropagation

Gradient computation: Backpropagation algorithm

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

capture our error in activation of that neural node

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

$$\delta_j^{(4)} = a_j^{(4)} - y_j \quad \left(= (h_\Theta(x))_j\right)$$

vector: $\delta^{(4)} = a^{(4)} - y$

vector whose dimension = num. of output units in network

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

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

($\delta^{(1)}$ No)

↑ because the first layer corresponds to input layer, which are just features in training set, so that doesn't have any error associated with that.

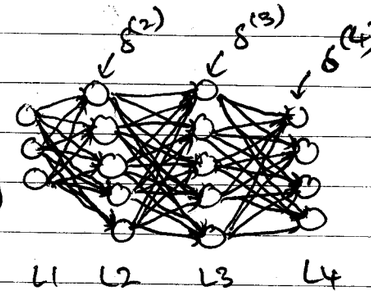
$g'(z^{(3)}) = a^{(3)} .* (1-a^{(3)})$

$g'(z^{(2)}) = a^{(2)} .* (1-a^{(2)})$

$a^{(3)}$: vector of activation values for the layer

$(1-a^{(3)})$: vector of ones

$a^{(2)}$: vector of activation



Backpropagation ⇒ come from the fact we start by computing δ for output layer and then go back a layer to compute δ, so back propagating errors from output layer to layer 3 to layer 2

Just few steps of computation, M is possible to prove

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$ (ignoring λ; if $\lambda = 0$) ← regularization

Neural Networks:

Learning

Backpropagation intuition

Implement backpropagation to compute derivatives with respect to your parameters, when we have large training set.

Backpropagation algorithm

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

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

capital Greek alphabet delta(δ) → These delta going to be used as accumulators that add things in order to compute

Next, loop through training set

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

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

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

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

backpropagation algorithm → Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ ~~$\delta^{(1)}$~~

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

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

After For loop, we 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 $j = 0$ (for bias term, that's why no extra regularization term)

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

Partial derivative of cost function with respect to each of the parameter

exactly equal to

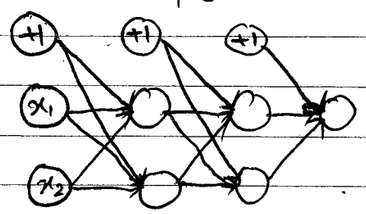
So can use those in gradient descent / advanced optimization algorithm

Quiz: Suppose you have 2 training examples $(x^{(1)}, y^{(1)})$ and $(x^{(2)}, y^{(2)})$. Which of the following is a correct sequence of operations for computing gradient (FP = forward propagation, BP = back propagation),

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

Backpropagation intuition

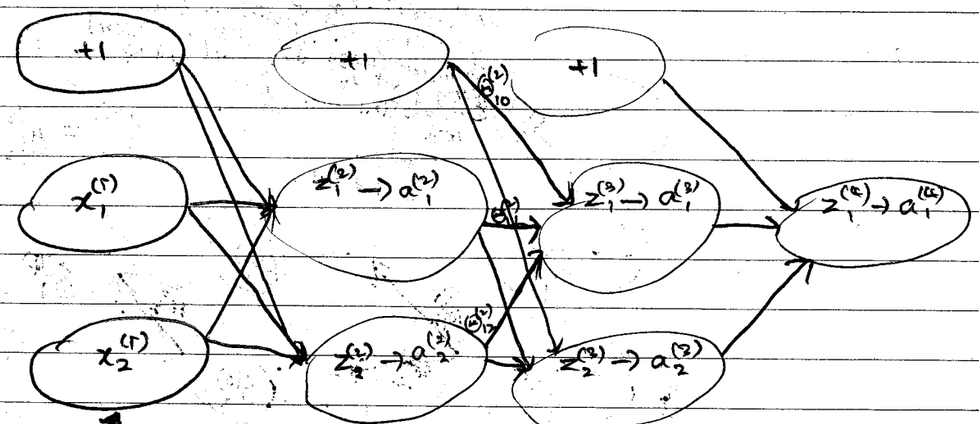
Forward Propagation



2 2 2 1

L_1 L_2 L_3 L_4

To illustrate it,



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

$z_1^{(3)} = \Theta_{10}^{(2)} x_1 + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)}$

if more than 1 output unit: $\sum_{k=1}^{K}$

What is backpropagation doing?

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

Only one output unit cost function

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

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

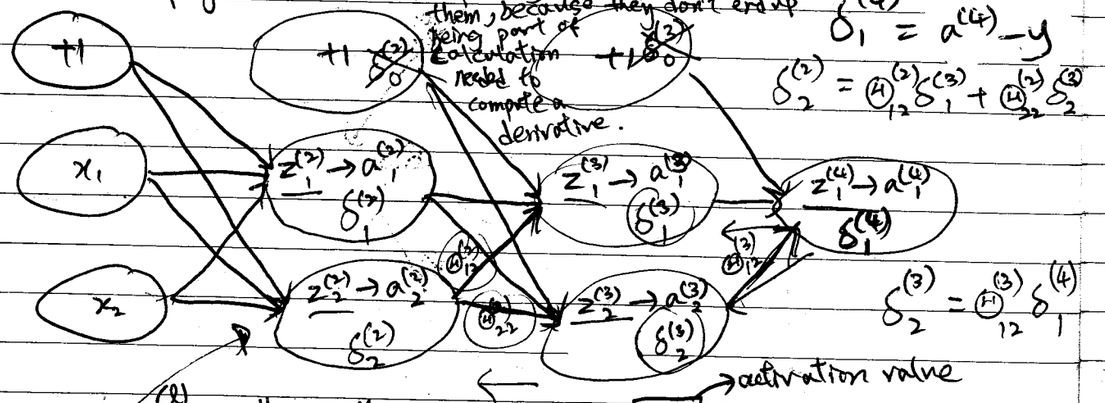
$$\text{cost}(i) = y^{(i)} \log h_\Theta(x^{(i)}) + (1-y^{(i)}) \log h_\Theta(x^{(i)})$$

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

i.e. (how well is the network doing on example i?) ← predicting

How close is the output to the actual observed label $y^{(i)}$?

Forward Propagation



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

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

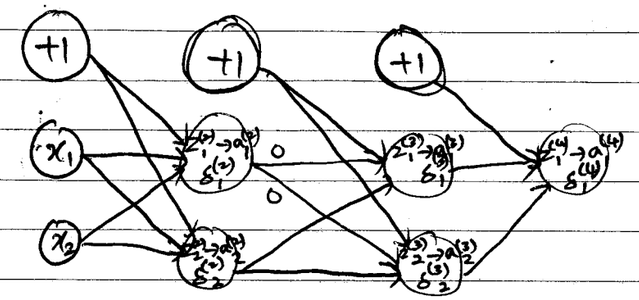
$$\delta_2^{(3)} = \Theta_{12}^{(3)}\delta_1^{(4)}$$

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

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

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

Measure of how much would we like to change the neural network's weight in order to affect these intermediate values of computation and affect final output of neural network $h_\Theta(x)$ and affect overall cost



Suppose both of the weights shown in red ($\Theta_{11}^{(2)}$ and $\Theta_{21}^{(2)}$) are equal to 0. After running BP, what can we say about value of $\delta_1^{(3)}$?

Answer: This is insufficient information to tell.

Neural Networks:

Learning

Implementation note:

Unrolling parameters (from matrices into vectors)

Advanced optimization ← return cost function and derivatives

```
function [jVal, gradient] = costFunction(theta)   <- input parameters theta
...                          ↳ R^{n+1}            ↳ R^{n+1} (vectors)
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural Network ($L=4$): ← not like logistic regression, use vector but use matrices

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

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

"Unroll" into vectors (unroll matrices into vector)

↳ so the format could suit and pass into advanced optimization algorithm.

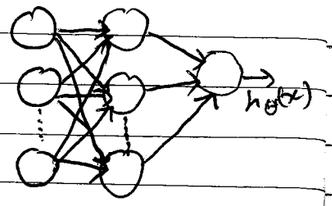
Vector ⟷ (Unroll) Matrices

Example

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

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

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



In octave, if want to convert btwn matrices and vectors

Unroll into big long vector

```
thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
DVec     = [D1(:); D2(:); D3(:)];
```

To reshape: back to matrices

```
Theta1 = reshape(thetaVec(1:110), 10, 11);
Theta2 = reshape(thetaVec(111:220), 10, 11);
Theta3 = reshape(thetaVec(221:231), 1, 11);
```

([row × column] matrices)

Quiz: Suppose $D^{(1)}$ is a 10×6 matrix and $D^{(2)}$ is a 1×11 matrix,

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

Which of the following would get D2 back from DVec?

Answer: reshape(DVec(61:71), 1, 11)

Learning algorithm (How we use unrolling idea to implement learning algorithm)

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$. (unroll into long vector)

Unroll to get initialTheta to pass to

fminunc(@costFunction, initialTheta, options)

function [jval, gradientVec] = costFunction(thetaVec)

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

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.

Advantage of matrices representation:
- more convenient in forward/back propagation
- easier for vectorized implementations.

Advantage of vectors representation:
- in advanced optimization algorithms

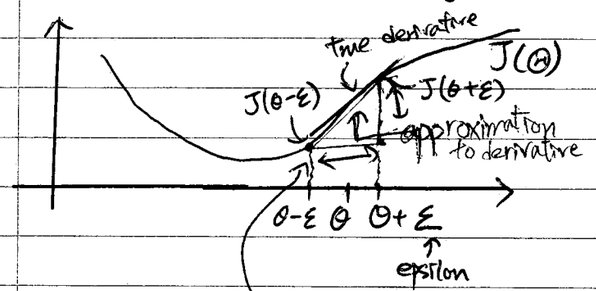
Neural Networks:

Learning

Gradient checking

May get bug in the implementation of backpropagation, use gradient checking to eliminate almost all the problems.
↳ can give implementation of forward/back prop 100% correct.

Numerical estimation of gradients



$\Theta \in \mathbb{R}$ (real number)

Vertical height = $J(\theta+\varepsilon) - J(\theta-\varepsilon)$

Horizontal = 2ε

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$ ← two-sided difference (more accurate)

if really small, $\varepsilon = 10^{-4}$ (usually make epsilon small)
(in mathematically it becomes derivatives)
↳ if not, may run into numerical problem.

OR formula like this

$$\frac{J(\theta+\varepsilon) - J(\theta)}{\varepsilon}$$ ← call one-sided difference (less accurate)

Implement in Octave:

approximating derivative

```
gradApprox = (J(theta + EPSILON) - J(theta - EPSILON))
             /(2*EPSILON)
```

Give a numerical estimate of gradient at that point.

Quiz: Let $J(\theta) = \theta^3$. Furthermore, let $\theta = 1$ and $\varepsilon = 0.01$. You use formula $\frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$ to approximate derivative. What value do you get using this approximation? (When $\theta = 1$, the true, exact derivative is $\frac{d}{d\theta} J(\theta) = 3$).

Answer: 3.0001

Parameter vector θ

$\theta \in \mathbb{R}^n$ (Eg. θ is "unrolled version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)

$\theta = [\theta_1, \theta_2, \theta_3, \dots \theta_n]$

Give you a way to numerically approximate the partial derivative of J respect to θ_i

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

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

$$\vdots$$

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

In Octave (n = dimension of parameter or vector theta.)

```
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) = thetaPlus(i) + EPSILON;   % [θ1, θ2, ..., θi+ε, ..., θn]
  thetaMinus = theta;
  thetaMinus(i) = thetaMinus(i) - EPSILON; % [θ1, θ2, ..., θi-ε, ..., θn]
  gradApprox(i) = (J(thetaplus) - J(thetaMinus))
                  /(2*EPSILON);
end;
```

gradApprox(i) $\approx \frac{\partial}{\partial \theta_i} J(\theta)$

Use this in neural network implementation

Check that gradApprox $\approx$ DVec

- DVec: derivative get from backprop
- To check implementation of backprop is correct
- Use this into gradient descent / advanced optimization algorithm
  - ⇒ make us more confident that we're computing the derivatives correctly

Implementation Note:

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

after checking, turn off g-checking and do BP

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of costFunction(...)) your code will be very slow.

Quiz:

Main reason we use BP rather than numerical gradient computation method during learning.

- The numerical gradient algorithm is very slow.

Neural Networks:

Learning

Random initialization

Initial value of Θ

For gradient descent and advanced optimization method, need initial value for Θ

optTheta = fminunc(@costFunction, initialTheta, options)

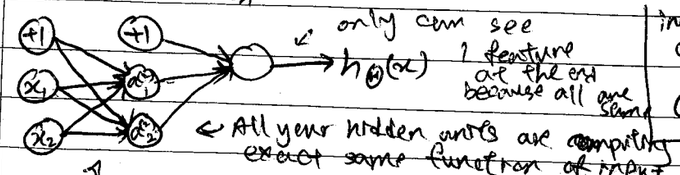
Consider gradient descent

Set initialTheta = zeros(n,1)?

initialize Θ as all vector zeros only work ok in logistic regression but not in training the network

Consider training following neural network,

Zero initialization



← only can see 1 feature at the end because all are same

← All your hidden units are computing exact same function of input (Highly redundant representation)

because weight = 0

end up $a_1^{(2)}$, $a_2^{(2)}$ compute same function of input

$$a_1^{(2)} = a_2^{(2)}, \text{ also } \delta_1^{(2)} = \delta_2^{(2)}$$

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

Problem is problem of symmetric ways, the ways are being the same

after 1 gradient descent update, will still same as each other

$$\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$$

After each update, parameters corresponding to inputs going into each of two hidden units are identical

To get around this problem, the way we initialize parameter of a neural network is with random initialization.

Random initialization: Symmetry breaking

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

↑ nth to do with ε that we use in gradient checking

End up $[-\varepsilon, \varepsilon]$

If dimension of Theta1 is 10x11, Theta2 is 1x11

E.g. (Octave)

```
Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

random 10x11 matrix (btwn 0 and 1)

rand(x,y) is a function in Octave random real numbers between 0 and 1.

Quiz: Consider this procedure for initializing parameters of a neural network.

1- Pick a random number r = rand(1,1) * (2 * INIT_EPSILON) - INIT_EPSILON;

2- Set $\Theta_{ij}^{(l)} = r$ for all i, j, l.

Does this work?

Answer: No, this fails to break symmetry.

Summary: To create a neural network, have to randomly initialize the weights to small values close to 0 ($\in [-\varepsilon, \varepsilon]$), then implement BP, do gradient checking and use gradient descent / advanced optimization algorithm to minimize $J(\Theta)$

Neural Networks:

Learning

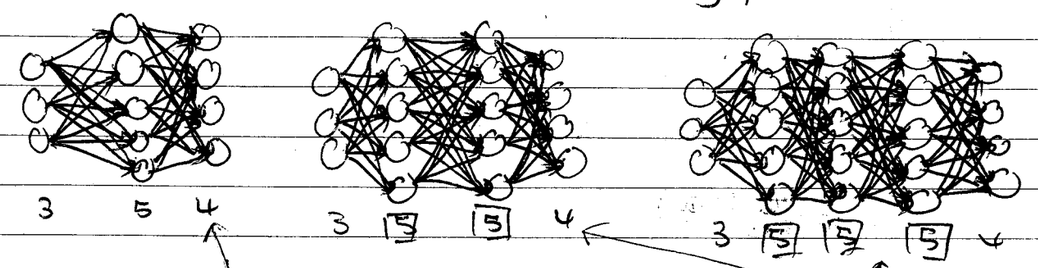
Putting it together

Overall summary

First —

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



Choices of how many hidden layer and how many hidden units in each layer, are architecture choices.

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

No. of output units: Number of classes

Reasonable default: 1 hidden layer, or if > 1 hidden layer, have same # of hidden units in every layer (usually the more the better)

↳ more computationally expensive but having more hidden units is a good thing

Let say multi-class (e) => $y \in \{1, 2, 3, \dots, 10\}$, don't forget to rewrite output y as these vector. $y = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$

If $y = 5$ → then $y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ → then to neural network

Second — Training a neural network

1- Randomly initialize weights (small values near zero)

2- Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$ (output) (compute)

3- Implement code to compute cost function $J(\Theta)$

4- Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

for i = 1:m {

Perform FP and BP using example $(x^{(i)}, y^{(i)})$

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

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

}

After FP and BP on $(x^{(1)}, y^{(1)})$ then FP, BP on $(x^{(2)}, y^{(2)})$ until $(x^{(m)}, y^{(m)})$

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

No.

Date

Third —

Training a neural network

5- Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(i)}} J(\Theta)$ computed using BP vs. using numerical estimate of gradient of $J(\Theta)$. Then disable gradient checking code.

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

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

For neural network,

$J(\Theta)$ — non-convex

Gradient descent very good in minimizing $J(\Theta)$ and get a very good local minimum, even it doesn't get to the global optimum

Contour plot of ~~Cost~~ using Gradient descent on $J(\Theta)$ by using 2 parameters
— BP is computing direction to the minimum point

Quiz Suppose you are using gradient descent together with BP to try to minimize $J(\Theta)$ as a function of Θ. Which of the following would be a useful step for ~~way~~ verifying that the learning algorithm is running correctly?

Answer: Plot $J(\Theta)$ as a function of the number of iterations and make sure it is decreasing (or at least non-increasing) with every iteration.

No.

Date

Quiz:

Correct statement:

1- Suppose we have a correct implementation of BP, training a neural network using g-descent. Suppose we plot $J(\Theta)$ as a function of the number of iterations, and find that it is increasing rather than decreasing. One possible cause of this is learning rate α is too large.

2- If we are training a neural network using gradient descent, one reasonable "debugging" step ~~is~~ to make sure it is ~~not~~ working is to plot $J(\Theta)$ as a function of the number of iterations, and make sure it is decreasing (or at least non-increasing) after each iteration.

3. If our neural network overfits the training set, one reasonable step is to take to increase the regularization parameter λ.

4. Using gradient checking can help verify if one's implementation of BP is bug-free.