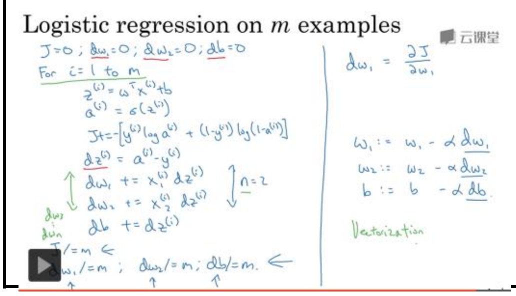
1. Training set size used in gradient descent: Batch, mini-bactch, tochastic gradient descenet

W:=W- a * dw
dw= 1/m * sum dw (under x(i)):

Cost function J: is the sum of every training example error-->
Gradient descent is to make this J on every example small-->minimize the sum of evey extraining training error

dw(i): parameter gradient descent direction, for making only J(i)-example x(i) cost smaller. but could not make sure sum of J(i) -all examples' error , is smaller.
--->dw(i) average over m size training example, make sure J(i) over m, is becoming smaller.



Martin Tan
CUHK

20 人赞同了该回答

因为优化是一门独立的学科，梯度下降算法本身就不是为了机器学习的问题而设计（梯度下降最早可追溯到200多年前的柯西），在梯度下降的大部分应用中，并没有“样本”的概念存在。

梯度下降的应用问题一般就是抽象为就是简单的最小化一个函数可导函数：$\min_x f(x)$，对于这个问题，梯度下降就是很简单的迭代：$x_{t+1} = x_t - \eta \nabla f(x_t)$。

只不过在机器学习中，我们有了很多个样本，而目标函数往往就是在每个样本上的loss求平均，也就是说此时问题有了额外的结构：$f(x) = \sum_{i=1}^{n} \ell_i(x)$。此时梯度下降算法就自然变成了：

$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla \ell_i(x_t)$，于是就有了你所说的“梯度求和”的形式。

当然，虽然梯度下降在求解机器学习问题中是一个有效的算法，但因为机器学习问题有更加特殊的结果，那么自然而然的大家就开始思考能否利用这种结构设计出更加高效的算法。于是就有了随机梯度下降算法（SGD）以及它的众多变种，其核心思想与你的想法有些类似，每次只算一个 $\ell_i(x)$ 的梯度，SGD中这个 i 是随机抽样得到的。

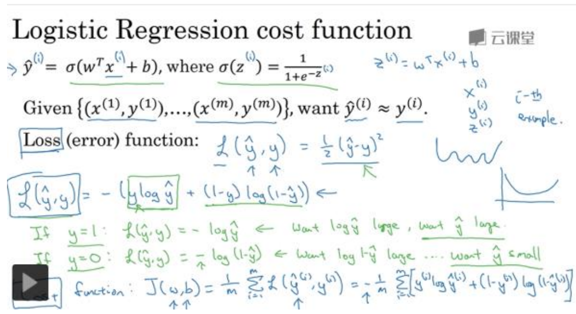
至于每轮迭代用最大的那一个样本梯度显然是不可行的。一个很容易可以想到的情况就是，如果有一个最大的样本梯度是朝一个方向的，但剩下所有的样本梯度都是朝另一个方向的，你觉得此时应该按最大的那个，还是按剩下所有的？一个更具体的例子，假设 $\ell_1(x) = (x+2)^2$，但 $\ell_2(x) = \cdots = \ell_{10}(x) = (x-1)^2$，很容易算出整个 $f(x)$ 的最小值在 $x = 0.7$ 的位置取到。那么假设我现在在 $x_0 = 0$ 出发想迭代求解这个问题，我应不应该按 $-\nabla \ell_1(x_0)$ 所指示的往负数方向走呢？

发布于 2017-11-18

2. Logistic cost function:

Linear cost function: distance : (h-y) ^2;
Logistic regression cost function:
distance of predicted value and labele value is good way to measure prediction performance/error.
Yet only for using gradient descent to minimize training error, need to make sure cost function J is convex.
Logistic regression distance error is not convex , that's why choose log type-convex.
(squared error for sigmoid function-->could not gurantte updating paremater big when error/cost is big -->converge slow:
dw = (y^-y) * (y^ (1-y^) *X.T

-->cross cost function: dw = (y^ - y) *X.T
)



3. training data -pre-processing

for color image, have 3 channel, -->feature vector: unrolled image into (w*l*3, 1) vector.

Example: Cat vs Non-Cat

The goal is to train a classifier that the input is an image represented by a feature vector, x, and predicts whether the corresponding label y is 1 or 0. In this case, whether this is a cat image (1) or a non-cat image (0).

An image is store in the computer in three separate matrices corresponding to the Red, Green, and Blue color channels of the image. The three matrices have the same size as the image, for example, the resolution of the cat image is 64 pixels X 64 pixels, the three matrices (RGB) are 64 X 64 each.

The value in a cell represents the pixel intensity which will be used to create a feature vector of n-dimension. In pattern recognition and machine learning, a feature vector represents an object, in this case, a cat or no cat.

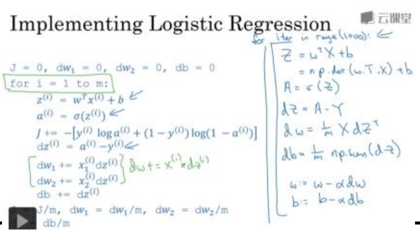
To create a feature vector, x, the pixel intensity values will be "unroll" or "reshape" for each color. The dimension of the input feature vector x is $n_x = 64 \times 64 \times 3 = 12\,288$.

$$x = \begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 255 \\ 134 \\ 202 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix} \begin{matrix} \text{red} \\ \\ \text{green} \\ \\ \text{blue} \end{matrix}$$

Forward: Z= W^T* X +b; A = g(Z) (1, m)

Backward: dZ= A-Y, (1,m)
db= 1/m * np. Sum (dZ) (1,1)
dw (n,1) = 1/m * X * dZ^T.

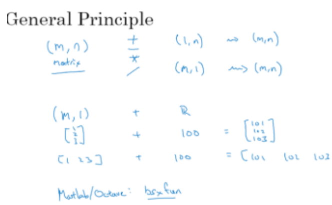
Note: if need many iterations, for loops still needed:
for 1: n_l:
Forward;
Backward;
Gradient descent.

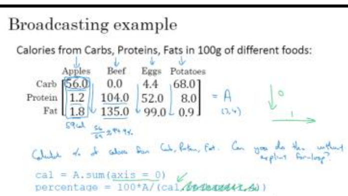


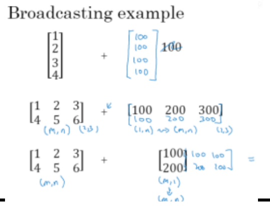
5. Broadcasting:

making algorithm faster in phython
sum in matrix column: cal = A.sum(axis=0): axis=1: vertical computation
sum in matrix row: percentatage = A / cal.reshape(1,4): each A colume divided by corresponding cal colume value

adding: A (m, n) + B (1, n): B will be mulitplied row to (m, n) matrix, then add with A.
adding A(m, n) + B(m, 1): B will multipley columes to (m,n)matrix, then add A.



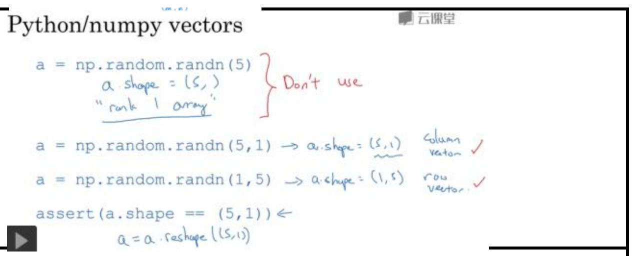




6. Techi: matrix in Python

>1. List colume and row number, instead of only one -in this case python will take this value as row and colume both in later computation.
Esp. when creating vector: do not use rank 1 array.

>could use 'assert' to make sure vector dimension;
>Cold use 'reshape' to adjust matrix dimenson.



7. Logistic regression cost function

>1. Pripnle: based on the label training data (x, y), find model parameter θ that given x, have the max probability for making this y value happen: max p(y=1 |x)

最大似然估计，通俗理解来说，就是已经有样本信息（数据），得到最具有可能导致这些样本结果出现的模型参数值!

y^=p(y|x): for given x, the probability of ouput is y=1; since y=1 or 0-->
1-y^ = p(y|x): for given x, the probability of output is y=0

>2. output probability: for y=0 and y=1, given signal input x,

$$p(y|x) = \hat{y}^{y} (1-\hat{y})^{(1-y)} \leftarrow$$

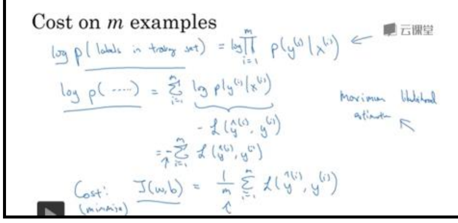
>3.: log (p(y|x)) = - J

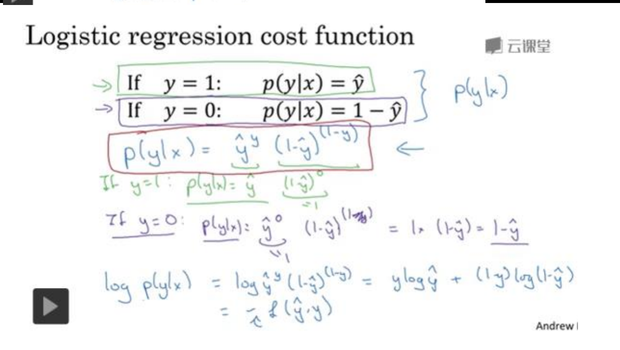
-->minilize J--> maximize log p , p
(note: use log for numerial underfloor problem)

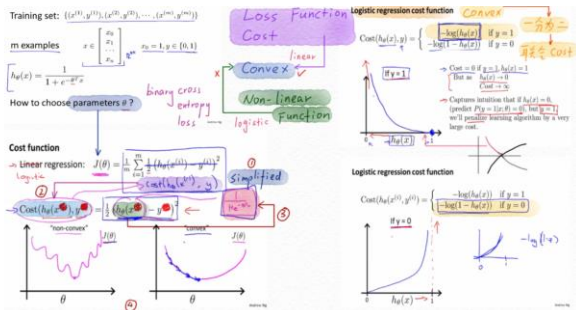
>4. Overal cost on m example: assumption: all examples distrubute independently.

Log p(...)= -J ,
log p: use maximum likelihood estimate , find model parameter θ
J: minimize

by minimize cost fucntion, **is actually carrrying out maximum likelihood with the logistic regression model, with assumption each training examples are identically independently distributed.**





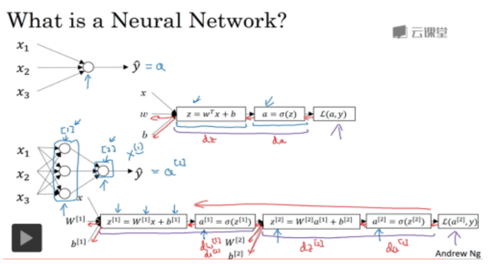


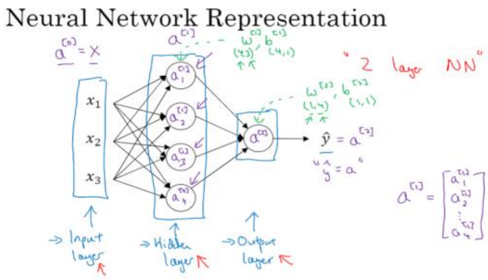
1. Neural network & logistic regression:

>1. Logistic regression: just one node of nerual network: one node in each layer
>2. Neural network: logistic nodes stack together to become one layer, -->multiple layer -->neural network.

Nerual network forward propagation and backpropagation: similar with logistic regression.

hidden layer: values here could not be obsevved; and have parameters associated.
Layer number: input layer not involved in accounting.



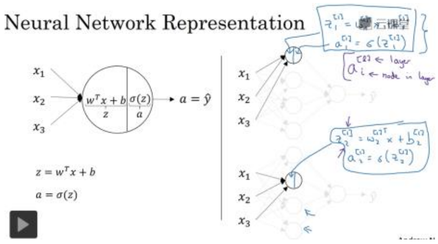


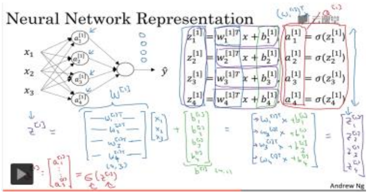
2. Neural network computation detail_multi nodes, signle training example:

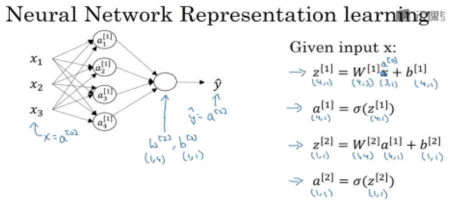
>1. every node in each layer: have two step computation:
Z= WT* X + b
a= g(z)

>2. Vectorization for each layer
Z[l]= layer l all units stack together vertically;
A=g(Z)

Z[l] =W[l] T*X +b [l]
A[l] = g (Z[l])
Note: b[l]: (n_l x 1), have one value for each node in layer: b[l]_i: is real value, may different value for different nodes in the same layer.



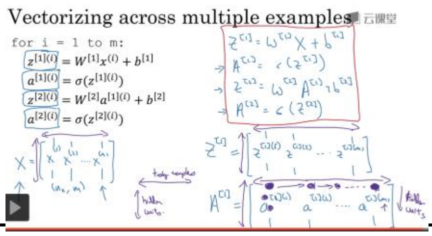




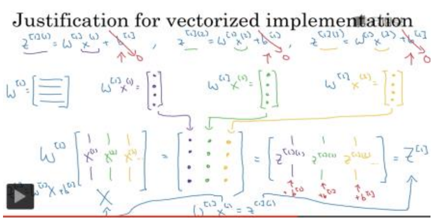
3. Neural network on m training example_vectorization_multi nodes, multi example

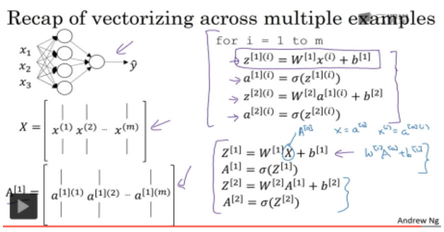
similar with logistic regression:
X: stacks in column for different example;

Z, A matrix:
>1. stacks in colume for different example
>2. Matrix row/ horizental direction: go through different examples;
>3. matrix column / vertical dierection: go through diffrent nodes in one layer.
Note:
b[l]: broadcasting columnly (total m column = example number): same value for one node in layer, broadcasting in columns when have multiple examples.



4. Neural network vectrorization justification_multi node, multi example_two layer



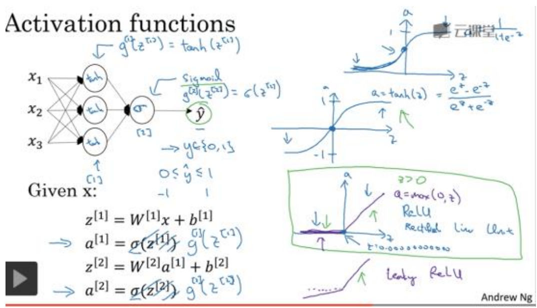


5. Activation function: tanh

>1. sigmoid function: 0-1;
>2. tanh function: -1 ~1: shift function of sigmoid: go through (0,0) and then rescale to -1~1.
tanh works better than sigmoid, as have 0 mean / normalization effect: just incase might need normalize the data/weight.???
-->
Hidden layer: use tanh function more popular than sigmoid
Output layer: will still use sigmoid for classification : output probability p: 0-1. while tanh is range from -1 to 1.

Note:
>>1. activation function used in different (hidden) layer, maybe different
>2. disadvantage of sigmoid, and tanh functino: when z is large, gradient is close to 0, --->gradient descent is very slow.--->

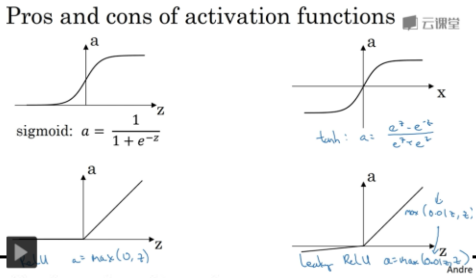
(note: use batch norm could help improving sigmoid , tanh disadvange: z large , gradient close to 0; batch norm-->z ~=0)



>3. Relu function: max (0, z)
no gradient when z=0, ok as in computation the chance of z =0 is too small;
or could also define gradient value when z=0.
disadvantage: derivative is equal to 0 when z <0 , in practice still work fine (have enough units to make z >0) , but have another version Relu: leaky Relu

>4.leaky Relu: have slight slope when z < 0: this slop could be a parameter to learn, based on algorithm performance.
works better than Relu, yet not used so much in practice

Activation function choosing rule:
outlayer: if output is 0 or 1/binas classificaiton, use sigmoid function.
all other units: default choice is Relu: rectified linear unite



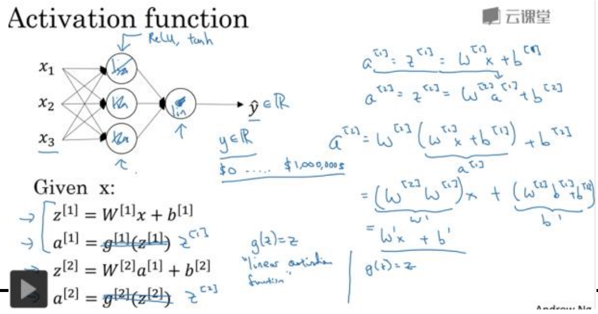
6. Why need non-linear activation function

>1. If activation function g[l] is linear or no activatioin function in hidden layer & output layer-->

neural network is linear function no matter how many layers used/how many deeper, could not learn interesting property. In this case could remove hidden layers.

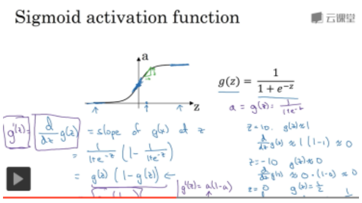
>2. if hidden layers activation function is linear, output layer is non-linear-->hidden layers could also removed as now hidden layer is more or less useless: two linear function combination of: input and hidden layer linear function, still linear.]

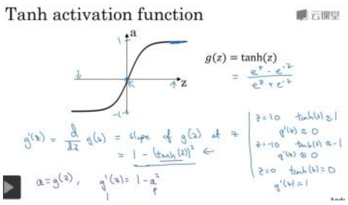
>. just one place might use linear activation function: doing machine learning on a regression problem.
: e.g house pirce predict: output layer may use line activation function , but rarly happen to hidden layer. in this case output layer could also use non-linear activation function : Relu (price >0).

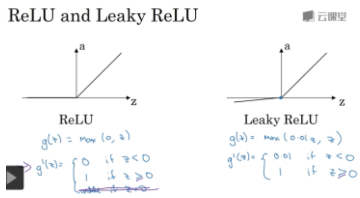


7. Derivation of activaiton function

Relu: derivation at z=0, ok to be any value set manually or could no define
y= sigmoid --->y' = y(1-y)
y=tanh --->y' = 1-y^2;



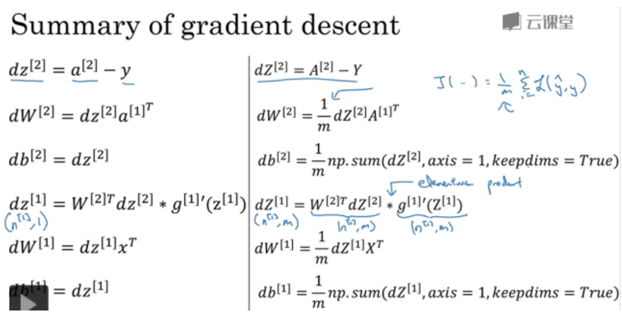




8. Neural network gradient descent
similar to logistic regression , cost function depending on activation function used.
>1. Forward propagation:
Z[l]=W[L] ^{T} A[l-1] +b[l];
A[l] = g (Z[l])

>2. Back propagation: on multi examples
(
dA[l]= W[l+1]T . dZ[l+1]
dZ[l] = dA[l]*g' (Z[l])
)
dZ[l] =W[l+1]^{T} dZ[l+1]. * g' (Z[l])-combine above two step together
W[l+1]T . dZ[l] is (n_l, m)
g' (Z[l]) also is (n_l, m). .* here means element wise product of two matrix.
dW[l] = 1/m* dZ[l] . A[l-1];
db[l] = 1/m * np. sum (dZ[l], axis=1 keepdims=True)

Note:
1. keepdims=True: incase outputing funny rank 1 arrays, db[l] dimension is (n_l, 1)
or could use reshape to make sure db[l] dimension.
2. In backpropagtion: one tech. tip is to check matrix dimension mapping in computation.



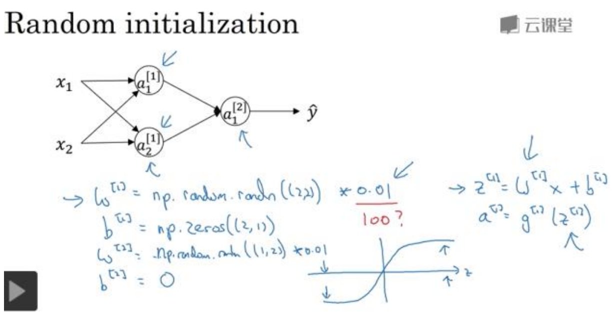
9. Initializaiton

method1: Initialize weight W initializaiton: all to 0
>1. ok for logistic regression (as only have total one node)
>2. not work for neural network: hidden units in each layer is symmetry.
    -->all units/nodes in each layer will compute same function: same as only have one node in each layer: all rows in layer weight W[l], W[l]_i are same.

method 2: random intialize weight:
W[l] = np.random.randn((2,2) * 0.01: randn-Gaussin distributioin , range is 0-1;
factor 0.01 make sure W[l] is very small-close to 0: because if initial weight is large, then Z[l] may also very large--->when use tanh/sigmoid activation function, inital weight will be at the flat parts (activation function =1/-1,) of activatin function, where g'(Z) also to 0 -->will slow backpropagation, slow gradient descent, slow learning.
(note: batch norm willl solve this problem-->BN make weight initialize not so important)

method3: b[l] could intialize all to 0: b[l]=np.zero((2,1))
will not cause hidden units in each layer symmetry problem, which only caused when weight is set to 0.

Summary:
if initial weight is large, Z is large ,**will cause activation function tanh/sigmoid to be satured, slowing down learning**. But if have no tanh/sigmoid function, should not be a problem.



what happens if intialize only some layers weight to 0, not all layersW[l]=0. should not cause symmatric issue in layer units?

e.g:
two layer: layer 1: 2 nodes; layer 2: one node; training on one example

X-->layer 1 (2 nodes)---->layer 2 (one nodes)-->y^
note: layer 2 : like logistic regression.

Forward:
Z[1] = W[1]T*X +b[1];
A[1] = g1 (Z[1])

Z[2] = W[2]T*A[1] +b[2];
y^=A[2] = g2 (Z[2])

Backward:
dA[2] <--loss function
dZ[2] = dA[2] .* g2'

dA[1] = W[2]*dA[2] .* g2';
dW[2] = 1/m* dZ[2] * A[1] = 1/m * dA[2] .* g2' * A[1]
db[2] = 1/m np.sum (dZ[2], dim=1) = 1/m np. sum (dA[2] .* g2', dim=1]

dZ[2] = dA[2] .* g2'

dZ[1] = dA[1] .* g1' = W[2]*dA[2] .* g2' .* g1' ;
dW[1] = 1/m* dZ[1] * X = 1/m * W[2]*dA[2] .* g2' .* g1' .* X
db[1] = 1/m np.sum (dZ[1], dim=1) = 1/m np. sum (W[2]*dA[2] .* g2' .* g1' , dim=1]

E.G: W1=W2= b1 = b2 =0
forward:

Z[1] = [0; 0]
A[1] = [a;a] (a !=0)

Z[2] = W[2]T*A[1] +b[2] =[0]
y^=A[2] = g2 (Z[2]) =[a2] != 0

backprop;
dA[2] = k (1x1) <--loss function

-->Second layer:
dZ[2] = dA[2] .* g2' = [k];
dW[2] = 1/m* dZ[2] * A[1] = 1/m * dA[2] .* g2' * A[1] = 1/m* k*g2 * [a;a]T = [dw2, dw2] != [0,0]
db[2] = 1/m np.sum (dZ[2], dim=1) = 1/m np. sum (dA[2] .* g2', dim=1] = [db2] !=0

-->second layer weight updated; yet symmetric dW[2] is same for all units.
note: dw[2]: parameter same befor and after udpating;

-->first layer:

dA[1] = W[2]*dA[2] .* g2';=[0;0]
dZ[1] = dA[1] .* g1' = W[2]*dA[2] .* g2' .* g1' ; = [0;0]
dW[1] = 1/m* dZ[1] * X = 1/m * W[2]*dA[2] .* g2' .* g1' .* X=0
db[1] = 1/m np.sum (dZ[1], dim=1) = 1/m np. sum (W[2]*dA[2] .* g2' .* g1' , dim=1]=0 (Note: db also influence by W[l+1], yet ok to initialize b to 0.)

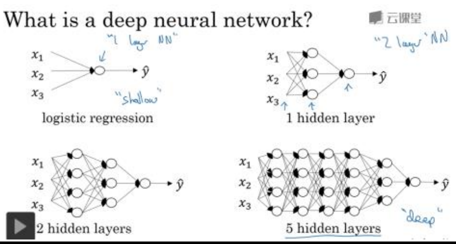
-->first layer:
no update as dw1, db1=0
error transfer from second layer to first layer by W[2], while W[2] initialize as all 0,-->error could not tranfer to previous layer, previous layer parameter could not be updated.

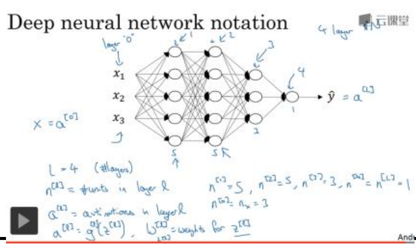
1. Deep neural network

deep : have many hidden layers

as hard to prdict in advacne exactly how deep a neural network needed, better try from logistic regression, then 1 hidden layer, 2 hidden layers, etc. and evaluate on dev set.

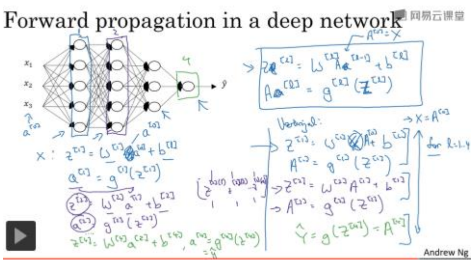
L= layer number: hidden layer + output layer
n[l]= units number in layer l.
a[l]: activation function a[0]=x, a[L]= y^





2. Deep neural network forward and back propagation on multi example
same as shallow neural network

2.1. Forward propagation: in each layer: input A[l-1], output A[l];
Z[l]=W[L] . A[l-1] +b[l]; (n_l, m); A[0]=x;
A[l] = g (Z[l])
need to repeat above for L layers, using 'for loop', which could not vectorized.



2.2: forward and backward block: in each layer

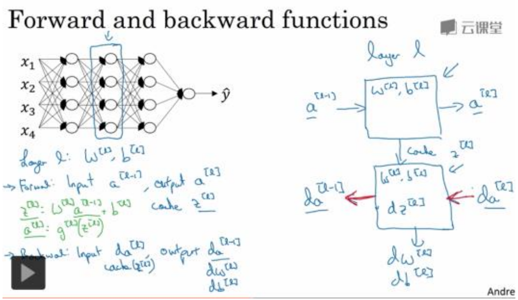
>1. Forward propagation: input A[l-1], output A[l];
Z[l]=W[l]. A[l-1] + b[l];
A[l]= g (Z[l])
store cache A[l], Z[l] , W[l], b[l] computed here , as will be used in backpropagation.

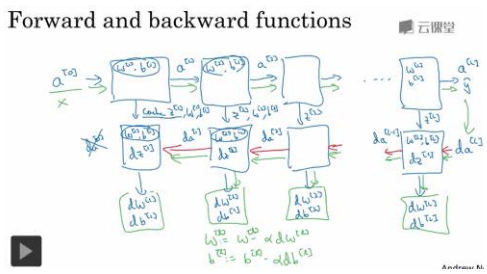
>2. Back propagation block: on multi examples: input dA[l], output dA[l-1]
means give a[l] change, how much will it affect a[l-1]
inpute: dA[l]
output:
> 1. dZ[l] = dA[l]*g' (Z[l])
>2. dA[l-1] =W[l]T . dZ[l] = W[l]T .dA[l]*g' (Z[l])

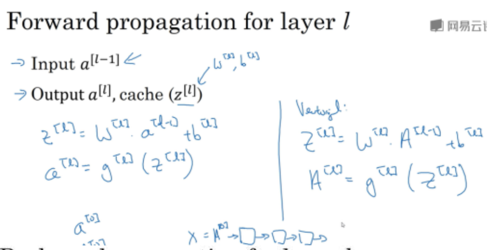
>3. output: also
dW[l] = 1/m* dZ[l] . A[l-1];
db[l] = 1/m * np. sum (dZ[l], axis=1 keepdims=True)

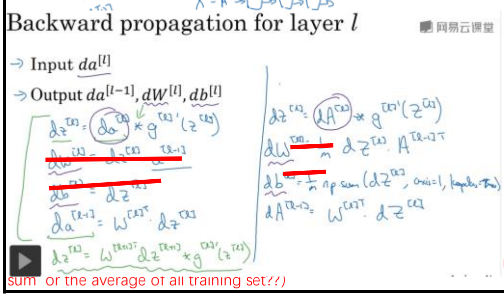
>3. One iteration:
forward: A[0]------->A[L]
Backward: y^=A[L]--> cost function J------>dA[L] ------------------->dW[1]
update W, b using gradient desecnt one time.

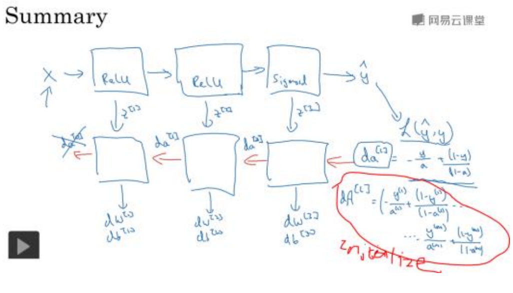
Note:
1. store cach Z[l] , W[l], b[l] computed in forward , as will be used in backpropagation.
2. Forward , initialize by input x; while backward initialize by dA[L], derivative of cost function over A[L]: (n_l, m)











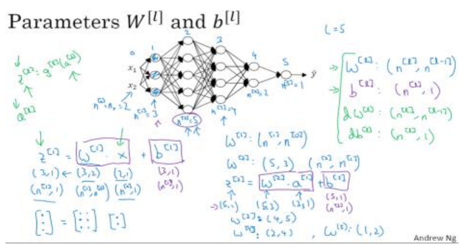
no matter loss function is the
sum or the average of all training set??)

3. Matrix dimension check in neural network_signal example

a[l-1]: input from layer l-1, dimension is (n_l-1, 1)
a[l]: have n_l output , (n_l, 1)
z[l]: (n_l, 1): number of units in layer l; same dimension as a[l]

b[l]: (n_l,1): number of units in layer l;
W[l]: (n_l, n_l-1)

dw /db/ dz /da have same dimension as W, b, Z, a
a[l] have same dimenson with hidden status z[l] (apply activaiton function on all hidden unit)

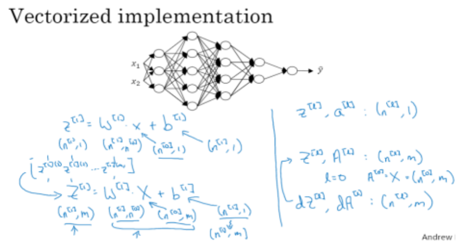


4. Matrix dimension check in neural network_multi example

W[l] dimension no change: (n_l, n_l-1);-dimension no influence by example size m
b[l] dimenson no change: (n_l, 1), yet phython broadcasting will used change to (n_l,m) in matrix computation.
others a, z, changed, stack m example computation result in colums

A[l-1]: input from layer l-1, dimension is (n_l-1, m)
A[l]: have n_l output , (n_l, m)
Z[l]: (n_l, m): number of hidden state in layer l; same dimension as A[l]

dw /db/ dZ/dA same dimension as W, b, Z,A
A[l] have same dimenson with Z[l]

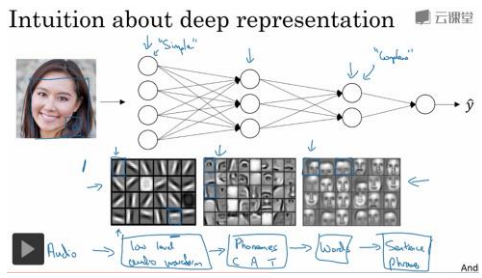


5. Deep neural network work well
5.1: intuition
e.g: face detection:

layer 1: may have 20 units to detect image edge.
Layer 2: 40 units, group edges togethers to form parts of faces.
Layer3: puting different face part to form face
layer4: detect face

--->could assume ealier layers learning simple features, and later layer put learned feature together so nueral netowrk could learn on more complex functions.

eg: audio:
ealier layers may learn simple sound wave features ,then later layer put together to form complex sound wavements-->learn basic units of sound



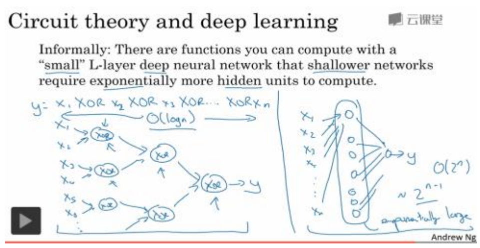
5.2: circuit theory for deep learning

there are functions compute with a 'small '(small hidden units number in layers) L-layer deep neural network (more layers, but small units in each layer), that shall lower networks require exponentially more hidden units to computes (small layer, large hidden units in each layer).

e.g: compute XOR function: of x1, ...xn
-->use deep neuro network:
neural network deepth is log(n), no much hidden units in each layer.
(note:
layer number L : n/(2^L) = 1-->L = log(n) / log (2)
hidden units number in layer l: n/(2^l) , total hidden units number: 2n = 1+2+...+2^t (2t=n)
)

>use 1 hidden layer neural network
hidden units is exponential large: 2^(n-1) (have this number possible configurations/combination).
(note:
each feature have 2 value 0, 1-->input cases is : 2^n; hidden units could be 2^(n-1)
)

there are many mathmatic functions that are much easier to compute with deep networks than with shallow networks.



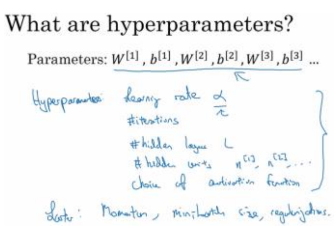
6. Hyperparameters

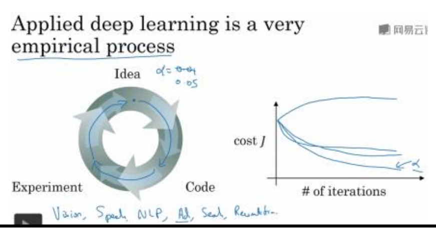
paramer: W, b

6.1 Hyperparameter defination: control the ultimate parameter W and b that end up with.
>1. learning rate a: determin how parameters evolve
>2. Iteration number for gradient descent: batch size
>3. Hidden layers number: L,
>4. hidden units n_l,
>5. activation function
>6. momentum, minibatch size, regulations
etc.

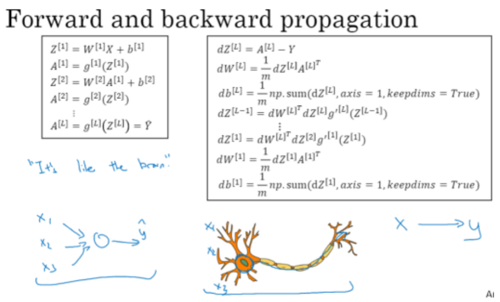
6.2Hyperparameter choose: based on empirical:
idea-->code-->experiment judge -->try idea...
evaluation different ideas based on cost function on dev set. many hyperparameters need to choose for new program, just try a range of values for them and see cost function./evaluation on dev set: based on pefromance: cost J/accuracy/signular evaluation F1 score-skrewed negative and positve examples, ..etc.
(note: choice hyperameters to try: no grid choise-random sampling, carse to fine and sampling 'uniformly' on good hyperameter 'range'-rescaled is necessary -balance sentive to hyperamter change)

Note: best hyperparameter may change with time: computer infrastructure- CPU/GPU change .--->if working on a problem for an extended period of time for many years, then after a few months, try a few values for the hyper parameters and double check if there is a better value .





7. Neural network and human brain



no big relation between the two: may more easier to explian these equations's function when use 'brain'
And take logistic regerssion as a single neruon, is also very loose / simplistic analogy.
No one actually know what a single neuron does, even some of it's function similar to logistic regression.: no idea: how neuro learns, if similar with backpropagation, gradient descent.

8. Neural network decison boundary??
-->decision boundary: only for classification problem, right?
-->classfication problem: output use sigmoid or softmax function.

With learned paramter W, if use sigmoid activation function:
y^=A[L]>= c -->W[1] .X =K (depending on c, W[2],..b) vector
-->decision boundary line number is row of W[1],

if overfitting (hidden unists too many), --> hidden units > input feature: decsion boundary line number: too many decison boundary line

when use other activaiton functions, decison boundary still be combination of different straight lines?-->depending on activaiton function, 'decison boundary' is more complex