

## Course 2 Week2

### Mini-batch gradient descent: (recap)

When you try to train on a large dataset, especially in the big data era, is just slow and memory consuming. Vectorization as said before allows you efficiently compute on  $m$  examples without the need of for loops but the problem arises when data are so large which in turns make the memory can't have all this amount of data so here comes the role of mini-batch gradient descent to solve what batch gradient descent can't. Mini-batch gradient descent depends on splitting your data into portions (mini-batches) and then apply the gradient descent consequently on the mini-batches.

Notations: Assume having 100,000 data in the training set ( $x^{(1)} \rightarrow x^{(100000)}$ ) so we are going to split this data into 100 mini-batch such that each mini batch has 1000 training example so the notation of 1st mini batch for example equals  $x^{\{1\}} \equiv [x^{(1)}, x^{(2)}, \dots, x^{(1000)}]$

generally  $\rightarrow$

$$x^{\{t\}} \equiv [x^{(t \times \frac{\text{total data}}{\text{number of mini-batches}) - (\frac{\text{total data}}{\text{number of mini-batches}) + 1}), \dots, x^{(t \times \frac{\text{total data}}{\text{number of mini-batches})}]$$

#### Notes:

1-Mini-batch size= total data / number of mini-batches

2-Mini-batch gradient descent needs for loops to loop over the mini-batches so vectorization is applied in each loop not on whole data.

#### Algorithm

### Mini-batch gradient descent

for  $t = 1, \dots, 5000$  {

Forward prop on  $x^{\{t\}}$ .

$$z^{\{t\}} = W^{\{t\}} x^{\{t\}} + b^{\{t\}}$$

$$A^{\{t\}} = \sigma(z^{\{t\}})$$

$$A^{\{t\}} = \sigma(z^{\{t\}})$$

$$\text{Compute cost } J = \frac{1}{1000} \sum_{i=1}^n \ell(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum \|W^{\{t\}}\|_F^2$$

Backprop to compute gradients w.r.t  $J^{\{t\}}$  (using  $(x^{\{t\}}, y^{\{t\}})$ )

$$W^{\{t+1\}} = W^{\{t\}} - \alpha \frac{\partial J}{\partial W}, \quad b^{\{t+1\}} = b^{\{t\}} - \alpha \frac{\partial J}{\partial b}$$

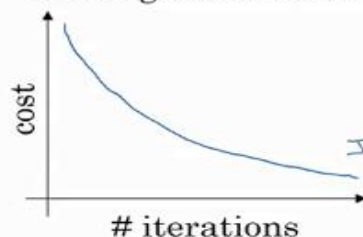
}

1 step of grad. desc.  
using  $x^{\{t+1\}}, y^{\{t+1\}}$ .  
(as if  $m=1000$ )  
 $x, y$

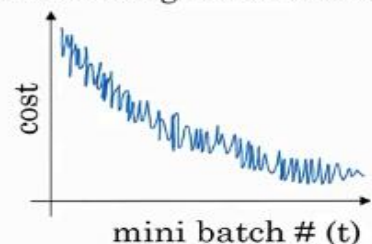
#### Cost along iterations

### Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent

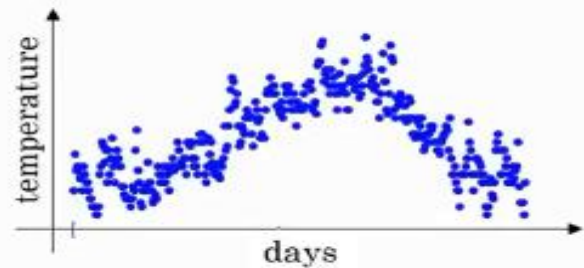


## Exponentially weighted averages:

Assume temperatures in London along the year was distributed as follows:

### Temperature in London

$$\begin{aligned}\theta_1 &= 40^\circ\text{F} & 4^\circ\text{C} \\ \theta_2 &= 49^\circ\text{F} & 9^\circ\text{C} \\ \theta_3 &= 45^\circ\text{F} & \\ \vdots & & \\ \theta_{180} &= 60^\circ\text{F} & 15^\circ\text{C} \\ \theta_{181} &= 56^\circ\text{F} & \\ \vdots & & \end{aligned}$$



This data looks a little bit noisy so we want now to compute local average or moving average of this data so this can done as follows:

$$v_0=0, v_1=0.9v_0 + 0.1\theta_1, v_2=0.9v_1 + 0.1\theta_2, \dots$$

So what is happening here that we compute the temperature of the day based on past several days' temperature to smoothen our data and reduce noise.

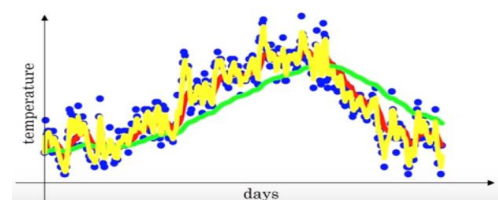
Equation:  $v_t = \beta v_{t-1} + (1-\beta) \theta_t$

$v_t$  can now be think as approximately averaging over  $\frac{1}{1-\beta}$  days' temperature (generally over last  $\frac{1}{1-\beta}$  data ). So we can say that  $v_{100}$  for example is actually a mix between real data at day 100 and data of all preceding 99 days where as day become closer it will have a higher effect so day 99 temperature has higher effect than day 98 temperature than day 97 temperature and so on which means by default higher coefficient. (exponential decrease in effect)

$$\begin{aligned}v_{100} &= 0.9v_{99} + 0.1\theta_{100} \\ v_{99} &= 0.9v_{98} + 0.1\theta_{99} \\ v_{98} &= 0.9v_{97} + 0.1\theta_{98} \\ &\dots \\ \rightarrow v_{100} &= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98}) \\ &= 0.1\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + \dots \end{aligned}$$

For our given examples if  $\beta$  is 0.9 then we are computing average over last 10 days (red), if  $\beta$  is 0.98 then we are computing over last 50 days (green) and if  $\beta$  is 0.5 then we are averaging over last 2 days(yellow). Take a look at the figure →

As  $\beta \downarrow$  so you are having much shorter averaging window so much more noisy data and much more susceptible to outliers but adapts more quickly to temperature changes (data change)



Note:

1-It is called exponential as if  $1-\beta=\epsilon$ , then  $(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$  so exponential decrease over  $\frac{1}{\epsilon}$ .

2-We use  $v_\theta$  sometimes to denote that this  $v$  is averaging over  $\theta$ .

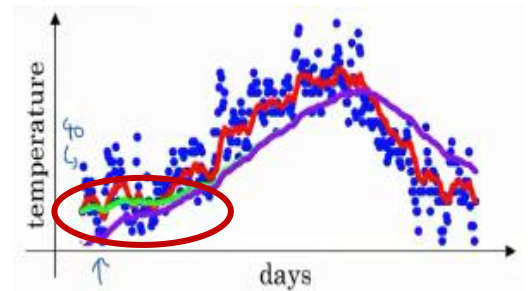
### Algorithm

$v_0 = 0$   
Repeat {  
  Get next  $\theta_t$   
   $v_\theta := \beta v_\theta + (1-\beta)\theta_t \leftarrow$   
}

This weighted averages is not so accurate so we will use a technique called bias correction to give more accurate computation of these averages.

### Bias correction

The problem is that at the first as we initialize  $v_0=0$  so the first steps of  $v$  will have very low values as shown in the figure (the difference between green (expected) and purple (actual) curves) as for  $v_1=0.98 v_0 + 0.02 \theta_1$  but as  $v_0=0$  so  $v_1=0.02 \theta_1$  so if  $\theta_1$  is 40 degrees so  $v_1$  will only equal 8 degrees !!! Here comes the role of bias correction.



The correction is that instead of using  $v_t = \beta v_{t-1} + (1-\beta) \theta_t$ , use  $\Rightarrow \frac{v_t}{1-\beta^t} = \beta v_{t-1} + (1-\beta) \theta_t$

So now for the first steps in the averages we have  $t$  small, so  $1-\beta^t$  has a reasonable small value that increases value of  $v_t$  and as we go up for higher  $t$  so  $1-\beta^t$  becomes close to 1 as we don't need this correction after the initial phase.

**Note:** Exponentially weighted average is effective however, it is not very easy to do as it involves holding the past values in memory buffer and constantly updating the buffer whenever a new observation is read.

[<http://www.ashukumar27.io/exponentially-weighted-average/>]

The previous link provides the same topic with the same example, so in case you don't understand the example by my style of writing, you can try this link for a different way of explanation of the same example.

## Optimization algorithms:

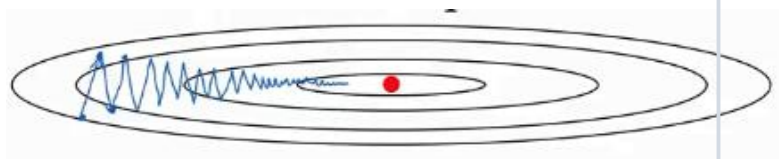
Gradient descent is not the best optimization algorithm for all problems so there are several optimization algorithms that gives better results than normal gradient descent.

### Gradient descent with momentum

This algorithm almost always works faster than the standard gradient descent algorithm. The basic idea here is to compute an exponentially weighted average of gradients then use them in updating weights and biases.

Assume having the contour plot shown and having our optima point at this red point and when starting gradient descent we had this oscillation shown so we can't have higher learning rate for faster learning in order to avoid divergence.

Another point of view is that we want to prevent overshooting in vertical axis ( $\updownarrow$ ) and at same time we want bigger steps on the horizontal axis ( $\leftrightarrow$ ) for faster learning. To apply this idea we gonna use grad descent with momentum which relies mainly on the idea of exponentially weighted averages.



### Algorithm

$v_{dw}=0$  ,  $v_{db}=0$

On iterations:

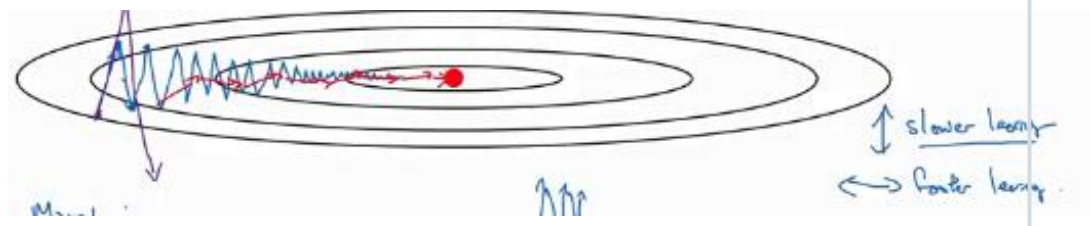
compute  $dw$ ,  $db$

$v_{dw} = \beta v_{dw} + (1-\beta) dw$

$v_{db} = \beta v_{db} + (1-\beta) db$

$w := w - \alpha v_{dw}$

$b := b - \alpha v_{db}$



This algorithm helps in damping out oscillations and smoothing the steps towards the minima as shown with red steps so this will lead to faster movement in the horizontal direction and more straight forward path towards the minima.

The most common value for hyper-parameter beta is 0.9 which average over last previous 10 iterations. In practice, bias correction is not implemented.

Some references use  $v_{dw} = \beta v_{dw} + dw$  instead of  $\beta v_{dw} + (1-\beta) dw$  by omitting  $(1-\beta)$  but in fact both are similar and that will only affect the  $\alpha$  chosen but the one with  $(1-\beta)$  is more preferred intuitively.

### Root mean squared propagation (RMSprop)

This algorithm is also used to speed up optimization process. Similarly we want to provide oscillation in vertical direction and speed up movement in the horizontal direction.

#### Algorithm

On iteration:

compute  $dw$ ,  $db$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2 \text{ (squaring here is element-wise)}$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw} + \epsilon}} \text{ (we add } \epsilon \text{ to ensure that no division by zero occurs)}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$$

### ADaptive Moment estimation (ADAM) optimization

This is a mix between momentum and RMS which is turned out to perform better than each of them separately.

Note: In ADAM we need bias correction for momentum.

#### Algorithm

$$v_{dw}=0, v_{db}=0, S_{dw}=0, S_{db}=0$$

On iteration:

Compute  $dw$ ,  $db$

$$v_{dw} = \beta v_{dw} + (1-\beta) dw \text{ (sometimes we use notation } \beta_1 \text{ instead of } \beta)$$

$$v_{db} = \beta v_{db} + (1-\beta) db$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

$$v_{dw}^{\text{corrected}} = \frac{v_{dw}}{(1-\beta^t)}$$

$$v_{db}^{\text{corrected}} = \frac{v_{db}}{(1-\beta^t)}$$

$$S_{dw}^{\text{corrected}} = \frac{S_{dw}}{(1-\beta_2^t)}$$

$$S_{db}^{\text{corrected}} = \frac{S_{db}}{(1-\beta_2^t)}$$

$$w := w - \alpha \frac{v_{dw}^{\text{corrected}}}{\sqrt{S_{dw}^{\text{corrected}} + \epsilon}}$$

$$b := b - \alpha \frac{v_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}} + \epsilon}}$$

Note:  $\beta$  usually equal 0.9,  $\beta_2$  usually equals 0.999 and  $\epsilon$  equals  $(10^{-8})$

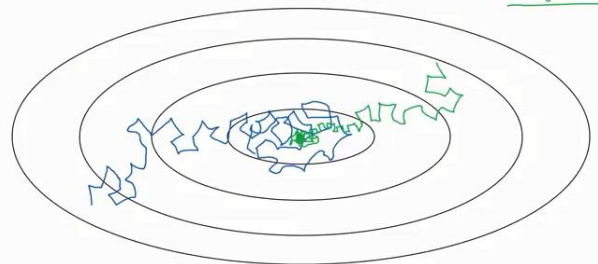


## Learning rate decay: (recap)

### Learning rate decay

We have talked about this idea of having a large learning rate ( $\alpha$ ) at the beginning of learning process (at the first epochs) then start to decay it with each new epoch so that we start with big steps towards local minima and then by time we decrease those steps as we go closer to convergence.

### Learning rate decay



The new info to add here is what relation to follow with reducing  $\alpha$ ?

$$\alpha = \frac{1}{1 + \text{decayrate} \times \text{epoch number}} \cdot \alpha_0$$

where  $\alpha_0$  is an initial learning rate to start with while decay rate is hyper-parameter to tune.

other methods of decaying:

1-exponential decay:  $\alpha = n^{\text{epoch number}}$  where  $n$  is a number less than 1.

2-

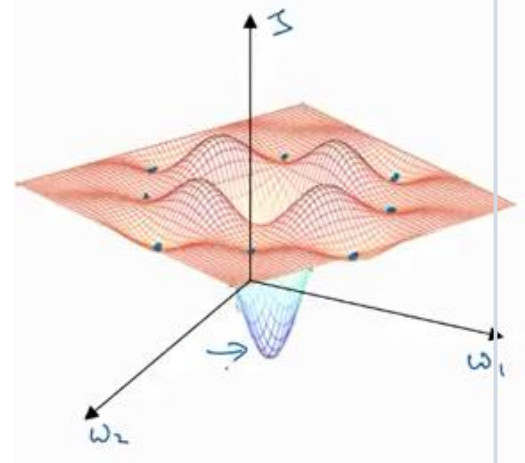
$$\alpha = \frac{k}{\sqrt{\text{epoch number}}} \cdot \alpha_0$$

3-Discrete steps as explained before.

## Problem of local optima

This problem simply means that when you are trying to optimize some weights to find the optima of the curve so it can get stuck in local optimas rather than global one as shown in the figure:

Note: optimas generally are points where the derivative equals 0.



## Course 2 Week3

### Batch normalization

We have always talked about feature scaling using different methods as normalization, standardization, mean normalization, ... and how it affects the training by making the data more centralized and rounded rather than elongated but what if we want to generalize this idea to the whole neural network not just the input features? here comes the role of batch normalization.

Batch normalization applies normalization to some or all neurons before applying activation function immediately which helped a lot in making learning parameters ( $w$ ,  $b$ ) of next layer learn faster. It makes  $Z$  controlled by a specific  $\mu$ ,  $\sigma$  instead of being so random noting that  $\mu$  &  $\sigma$  could be 0 & 1 or any other value (What controls their value is  $\gamma$  &  $\beta$ ).

#### Algorithm

### Implementing Batch Norm

Given some intermediate values in NN  $z^{(1)}, \dots, z^{(n)}$

$$\mu = \frac{1}{n} \sum_i z^{(i)}$$
$$\sigma^2 = \frac{1}{n} \sum_i (z^{(i)} - \mu)^2$$
$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
$$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$$

learnable parameters of model.

As shown in figure

applying this algorithm and using  $\tilde{Z}^{(i)}$  instead of  $Z^{(i)}$  makes learning process faster.

#### Notes:

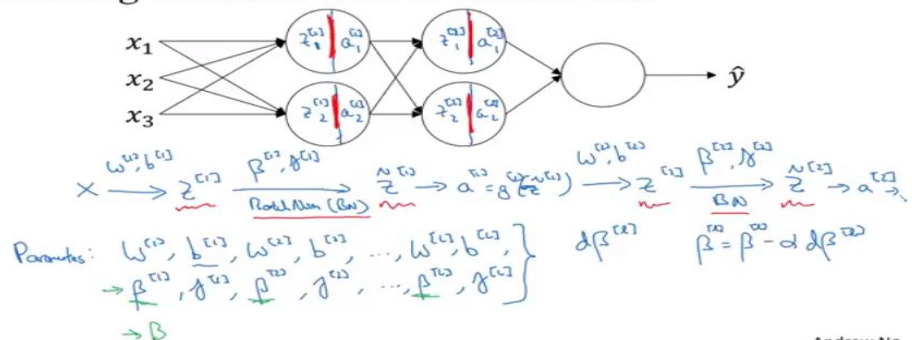
1-  $Z^{(i)}$  here is equivalent to  $Z^{[l](i)}$  but used as  $Z^{(i)}$  for simplicity not more.

2-  $\epsilon$  is added to variance to prevent dividing by zero.

3-  $\gamma$  &  $\beta$  here are learning parameters that will learn with the optimization algorithm used and they are used to let  $\mu$  &  $\sigma$  have values other than 0 and 1 to give some freedom for the outputs.

4- if  $\gamma = \sqrt{\sigma^2 + \epsilon}$  &  $\beta = \mu$  so this will lead to that  $\tilde{Z}^{(i)} \equiv Z^{(i)}$  (Act if there is no batch norm applied)

### Adding Batch Norm to a network



### Why batch norm works?

**Reason1:** Similarly as feature scaling, this helps in making all data centered and deviated closely rather than having some data ranging from 0 to 1 and others from 0 to 1000.

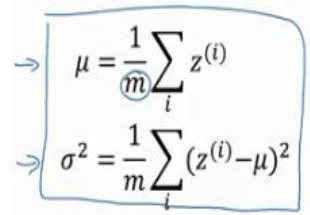
**Reason2:** Batch norm helps in reducing covariance shift done by the previous layers.

Assume having 4 hidden layers in network, if we cover the first 2 layers and start study the 3<sup>rd</sup> layer so we will find that the inputs  $a_1^{[2]}, \dots, a_4^{[2]}$  to layer 3 are the main reason for finding  $w^{[3]}, b^{[3]}, w^{[4]}, b^{[4]}, \dots$  using gradient descent process but if we uncover the first 2 layers and look again at the 3<sup>rd</sup> layer we can see that  $a_1^{[2]}, \dots, a_4^{[2]}$  are themselves affected by previous weights and suffering from covariance shift so batch norm controls this shift using its learning parameters ( $\gamma$  &  $\beta$ ). In other words we can say that this reduces the coupling between layers so that each layer of network can learn by itself and more independently from other layers.

Batch norm in fact has a slight regularization effect due to the noise added by computing mean and variance on mini-batches like dropout. But mainly batch norm is not used for regularization.

**Note:** As mini-batch size increases, the regularization effect decreases.

**TAKE CARE:** Batch normalization handles data one mini-batch at a time which means that mean and variance computed are computed on mini-batches not the whole data at once. This will have an effect on the test phase as in test phase we don't apply on a mini-batch but applies prediction on a single example every time so as  $\mu$  and  $\sigma^2$  are computed over a mini-batch ( $m$  here is number of examples in batch not whole training set) so when applying on a single training example on test time the equations will be meaningless so what to do now?


$$\begin{aligned}\mu &= \frac{1}{m} \sum_i z^{(i)} \\ \sigma^2 &= \frac{1}{m} \sum_i (z^{(i)} - \mu)^2\end{aligned}$$

As we have mini-batches and each mini batch has a specific  $\mu$  &  $\sigma^2$  so for layer  $\ell$  for example we have  $\mu^{\{1\}[\ell]}, \mu^{\{2\}[\ell]}, \mu^{\{3\}[\ell]}, \dots$  so to compute  $\mu^{[\ell]}$  used in test phase we will use exponential weighted average as follows :

$\frac{\nu_1}{1-\beta^t} = \beta \nu_0 + (1-\beta) \mu^{\{1\}[\ell]}, \frac{\nu_2}{1-\beta^t} = \beta \nu_1 + (1-\beta) \mu^{\{2\}[\ell]}, \dots$  until reach  $\nu_{\text{last mini-batch}}$  and then used this as my  $\mu$  for test phase. Similarly we do same steps for  $\sigma^2$ .

This is not the only way to compute  $\mu$  and  $\sigma^2$  but you can use any reasonable method to have a single  $\mu$  and  $\sigma^2$  for each layer using the ones of each mini-batch.



## Softmax regression

We are always talking about logistic regression for binary classes (2 classes) so positive or negative classes so softmax regression is the generalization of logistic regression for c

Recognizing cats, dogs, and baby chicks, other



3

1

2

0

3

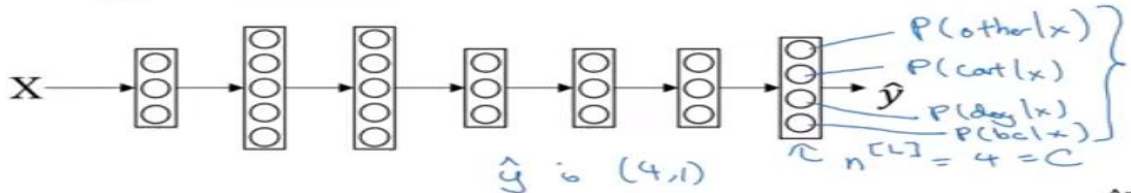
2

0

1

$C = \# \text{classes} = 4$

$(0, \dots, 3)$



classes.

If you have for example a network that classifies cats, dogs, baby chicks or others so now we have 4 classes (class0, class1, class2, class3). In this case the output layer will be of 4 neurons (c neurons). So we can say  $n^{[L]} = 4$  or c generally. Each neuron of the output layer provides the probability of a class of the c classes to be the output given input features x. So for our example, the 4 neuron outputs are  $(P(\text{cats} | X), P(\text{dogs} | X), P(\text{baby chicks} | X), P(\text{others} | X))$ .  $\hat{y}$  here is an output vector of dimension  $(4 \times 1)$ . Note that the summation of the 4 numbers or probabilities of the output layer should equal to 1.

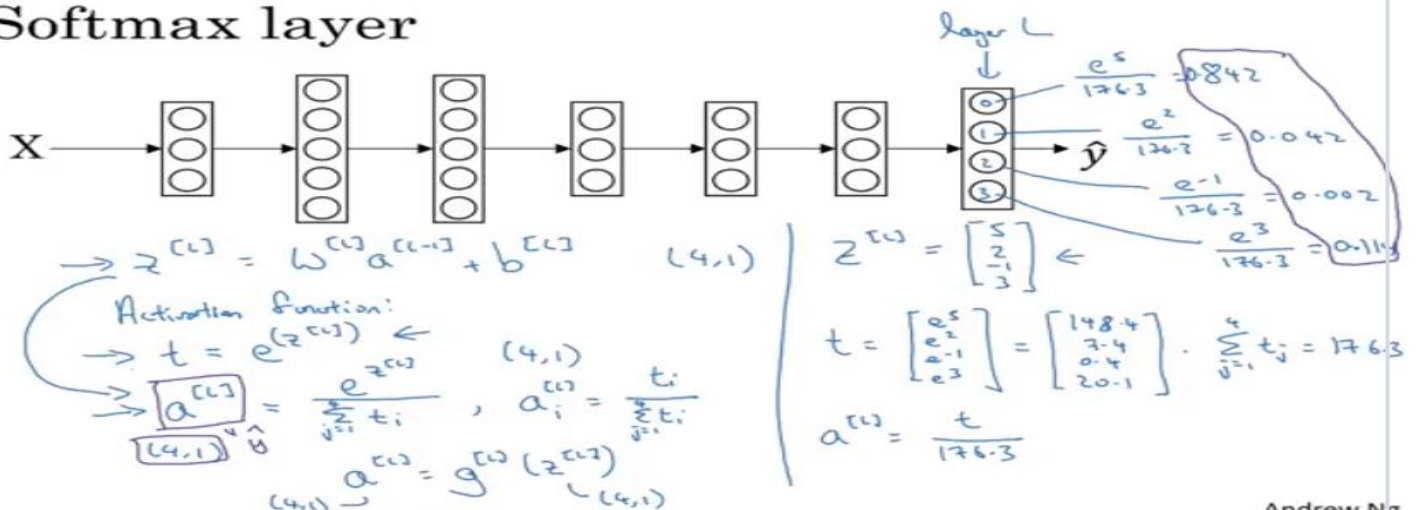
Softmax regression is done using softmax layer put in the output layer that differs from usual layers that it has softmax activation function so the flow is as follows:

$$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]} \rightarrow \text{activation function is } t = e^{(z^{[L]})} \rightarrow a^{[L]} = \frac{e^{(z^{[L]})}}{\sum_{j=1}^c t_j} \text{ so } a_i^{[L]} = \frac{t_i}{\sum_{j=1}^c t_j}$$

**Note:** dimensions of  $z^{[L]}$ , t &  $a^{[L]}$  is  $c \times 1$ .

The unusual thing about softmax activation function that it takes  $c \times 1$  input and outputs a  $c \times 1$  vector also unlike sigmoid and ReLU function which take a single real value and outputs a single real one.

## Softmax layer



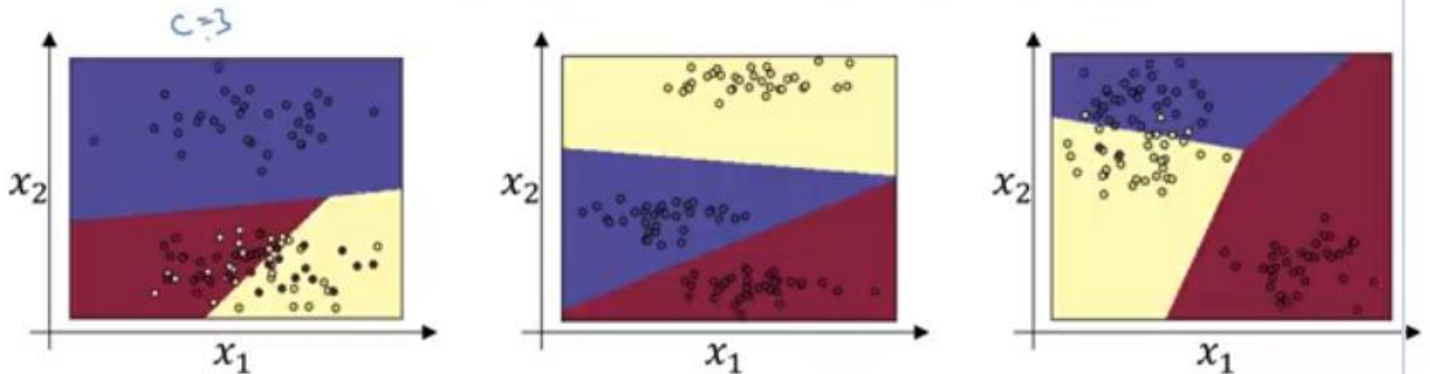
Assume we have only a softmax layer with no hidden layers so just take  $x_1$  &  $x_2$  and outputs 3 classes so what would a softmax layer do to this input data with 2 features  $x_1$  &  $x_2$  ?  
It will act as linear classifier to classify the data into 3 linearly separated regions as shown:

## Softmax examples

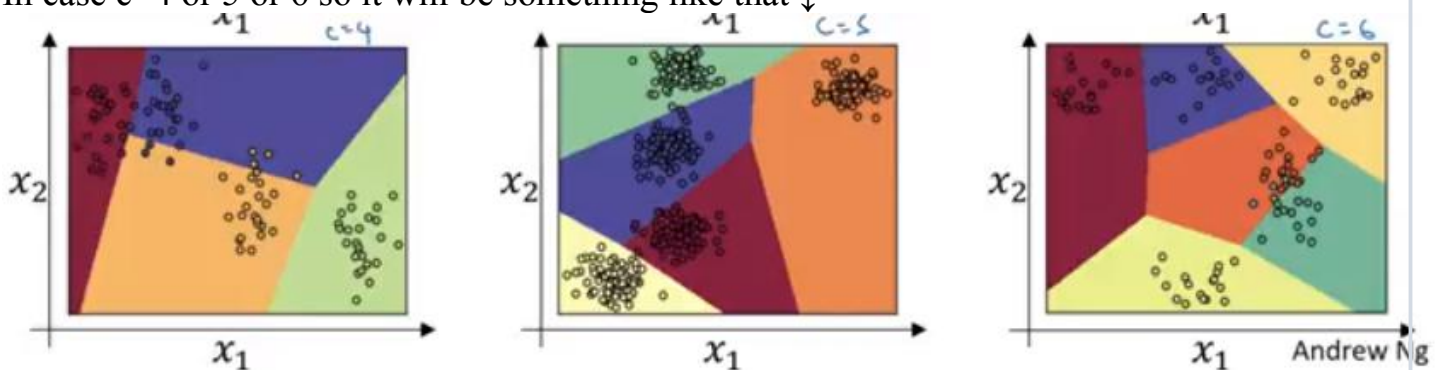
$$\begin{matrix} x_1 \\ x_2 \end{matrix} \rightarrow \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \rightarrow \hat{y}$$

$$z^{(1)} = w^{(1)}x + b^{(1)}$$

$$a^{(1)} = \hat{y} = g(z^{(1)})$$



In case  $c=4$  or 5 or 6 so it will be something like that ↓



It is called softmax classifier because hardmax ones will look at values of  $z^{[L]}$  and put 1 on the highest value and zeros for other  $c-1$  classes while softmax produces probability of each class as shown: →

## Understanding softmax

$$\begin{matrix} (4,1) \\ z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \end{matrix}$$

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5 / (e^5 + e^2 + e^{-1} + e^3) \\ e^2 / (e^5 + e^2 + e^{-1} + e^3) \\ e^{-1} / (e^5 + e^2 + e^{-1} + e^3) \\ e^3 / (e^5 + e^2 + e^{-1} + e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

“Soft max”

“hard max”

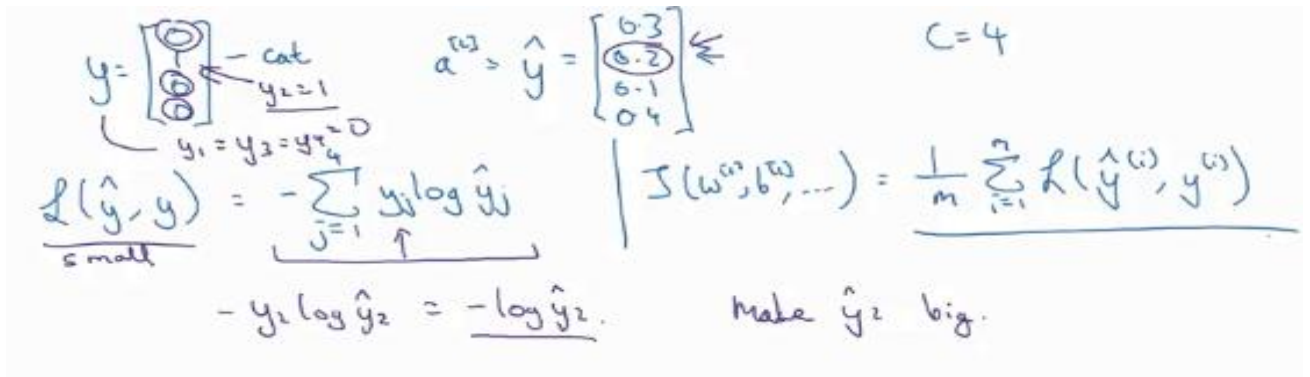
Softmax regression generalizes logistic regression to  $C$  classes.

If  $C=2$ , softmax reduces to logistic regress

### Loss function of softmax regression

$$\ell(\hat{y}, y) = - \sum_{j=1}^C y_j \log \hat{y}_j$$

Since  $y$  or  $y_{\text{actual}}$  is a vector having 0's for all classes except for the right class which has 1 so the loss function is actually equal to  $-(1) \log \hat{y}$  so the way to reduce the cost is to make the prediction of the right class as high as possible (1 is the best) to make the least cost possible



Handwritten notes illustrating the loss function for softmax regression. It shows a vector  $y$  with a 1 in the second position and 0s elsewhere, labeled as 'cat' and ' $y_2=1$ '. A vector  $\hat{y}$  is shown with values [0.3, 0.2, 0.1, 0.4], where 0.2 is circled and labeled ' $\hat{y}_2$ '. The loss function is written as  $\ell(\hat{y}, y) = - \sum_{j=1}^C y_j \log \hat{y}_j$ , with a note ' $\underbrace{\quad}_{\text{small}}$ ' under the loss and an arrow pointing to the term  $y_2 \log \hat{y}_2$ . Below this, it shows  $-y_2 \log \hat{y}_2 = -\log \hat{y}_2$  and says 'make  $\hat{y}_2$  big'. To the right, the total cost  $J$  is given as  $J(w^{(1)}, b^{(1)}, \dots) = \frac{1}{n} \sum_{i=1}^n \ell(\hat{y}^{(i)}, y^{(i)})$  and  $C=4$  is noted.

### Derivative with softmax

$dz^{[L]} = \hat{y} - y$  where  $dz^{[L]}$  is equivalent to  $\frac{\partial J}{\partial z^{[L]}}$  and its dimension is  $(C, 1)$