Metidji Sid Ahmed Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization DeepLearning.ai Course Summary

I. A lot of decisions to make while training your NNs : 2

II. There isn't a perfect decision that can be applied in every ML domain: 2

III. Splitting the data into training/dev/set 3

IV. Bias/Variance: 4

V. Basic Recipe for Machine Learning: 6

VI. Regularization (L2) and how it reduces the overfitting : 7

VII. Dropout regularization: another technique to avoid the overfitting 13

VIII. Good practices with Dropout regularization 15

IX. Other techniques to avoid the overfitting: 17

X. Normalizing the inputs in the training set and its importance: 18

XI. Vanishing / Exploding Gradient 22

XII. Gradient checking 24

XIII. The role of the optimization algorithms: 27

XIV. Mini batch Gradient 27

XV. Choosing the mini batch size : 30

XVI. Exponentially Weighted Average (moving average) : 32

XVII. Gradient descent with momentum: for quicker learning 37

XVIII. RMSPROP (Root Mean Square Prop): another way to speed up the learning rate efficiently 38

XIX. Adam optimizer (Adaption moment optimizer) : a combination of momentum and RMSprop: 40

XX. Learning rate decay; because we need huge steps only in the first iterations ... 42

XXI. Local optima problem : 44

XXII. Hyperparameters tuning process: 45

XXIII. Choosing the right scale to pick random hypermeters values matters ! 48

XXIV. Why having a linear scale for some hyperparameters is a bad idea, why would we need to log scale ? : 50

XXV. Hyperparametres in practice : between focusing on single model or training multiple models at the same time 51

XXVI. Batch normalization for quicker training: 52

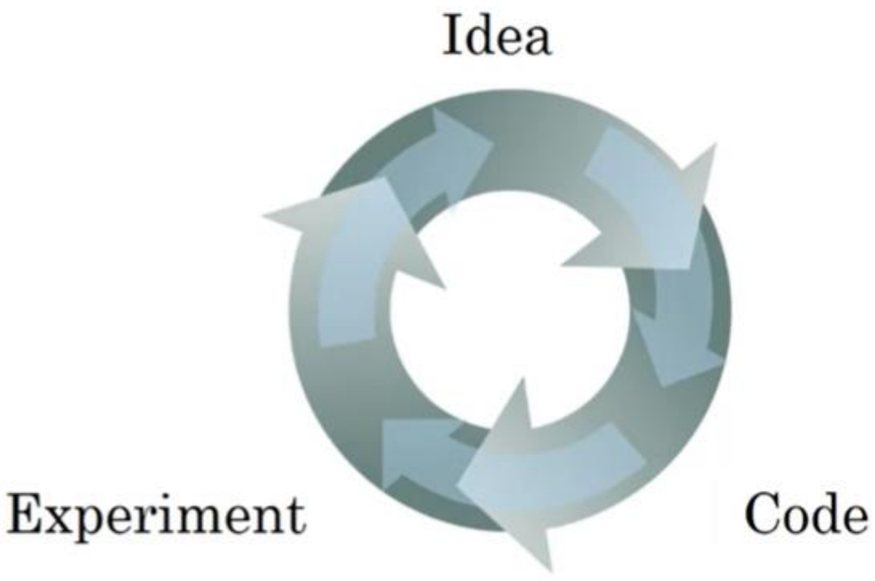
XXVII. Implementing Batch Normalization in the NN: 55

XXVIII. Multi classification with Softmax : the mathematical theory behind 63

I. A lot of decisions to make while training your NNs :

- Number of layers to apply
- For each layer, how many hidden units to use
- The learning rates
- The activation functions to use
- Other hyperparameters

➢ <u>Machine Learning is an iterative process:</u>



- In each iteration, you have an idea about a NN architecture to use , Then after evaluating it , you keep refining your idea each iteration until you get an optimal architecture

II. There isn't a perfect decision that can be applied in every ML domain:

- When a research jump from a ML domain (like NLP) to another one (like Computer Vision or Speech recognition) : he cannot transfer his experience from domain area to another domain by using the same decision he had made in the first domain due to encountering different variations (different inputs and different number of inputs , the amount of data , ...)

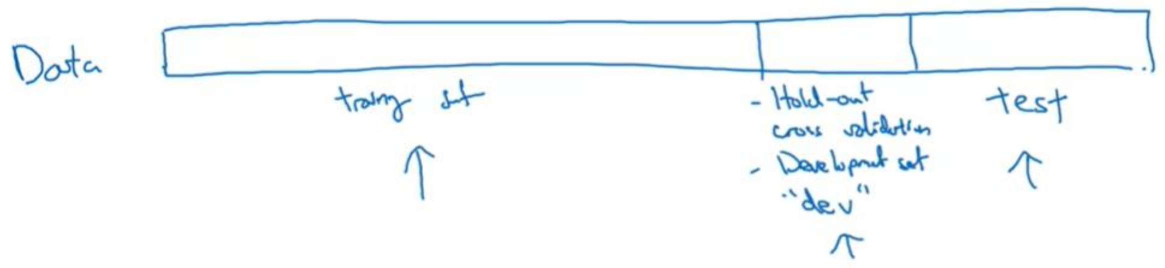
➢ <u>Conclusion :</u>

It's impossible even for the very experiment DL researches to guess the perfect hyperparameters from the first attempt, they all obliged to go in the ML development cycle, the experiment one does that efficiently!

The first thing to think about is splitting correctly your data for train/dev/test sets

III. Splitting the data into training/dev/set

Each data should be typically spitted into three portions :



- You train you DL model in the training set , You use the Dev set to do the cross validation and choose the best architecture based from the metrics you will get from the dev set , and then finally, Evaluate your final model in the test dataset to get the final metrics without any bias (since the architecture was selected according to their metrics with the dev set)
- For a short dataset (below 10K) :
  - the recommended split is 60%/20%/20%
- For modern big data (1M row) :
  - The dev and test sets should become in smaller percentage , since their utility is to only give us an overview about the model performance so it's wasteful to give them huge amount of rows , you might just use 10K for dev to just know which algorithms gives the best performance and 10K for test is enough too to just eb a confident estimator of your model performance , so the final ratio is 98%/1%/1%

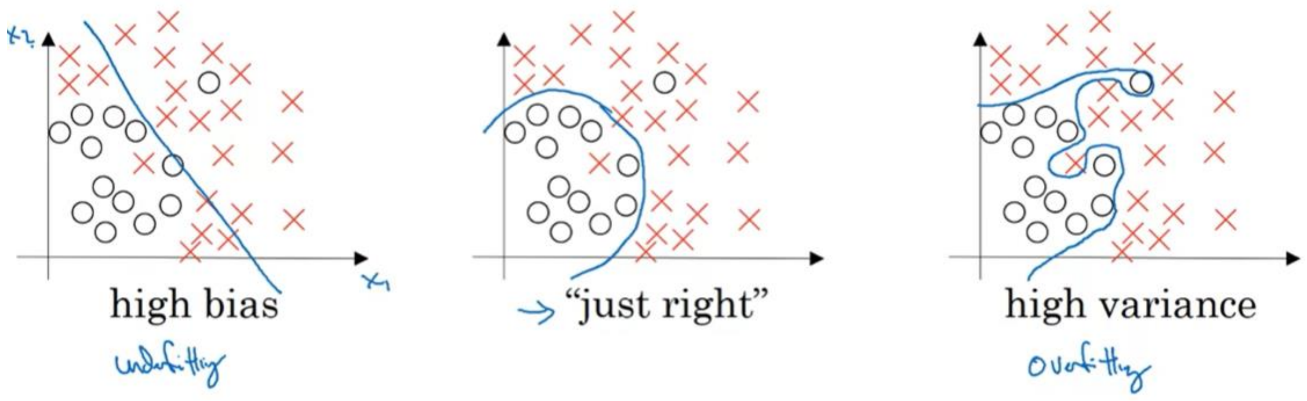
➢ <u>Mismatched train/dev data distribution</u>

- Example: let's say you are building a model that do some computer vision task on the pictures given by the user. It would be a bad practice to train your model only on images gathered from wallpapers websites, because these ones are in a high resolution , taken in perfect conditions …etc , while the pics that the user would take : they will be taken using his camera phone (little resolution) , they may even be blurry or taken in bad conditions . So, the train/dev datasets must have the same distribution between these two kinds of pictures so he can perform well in both kinds.

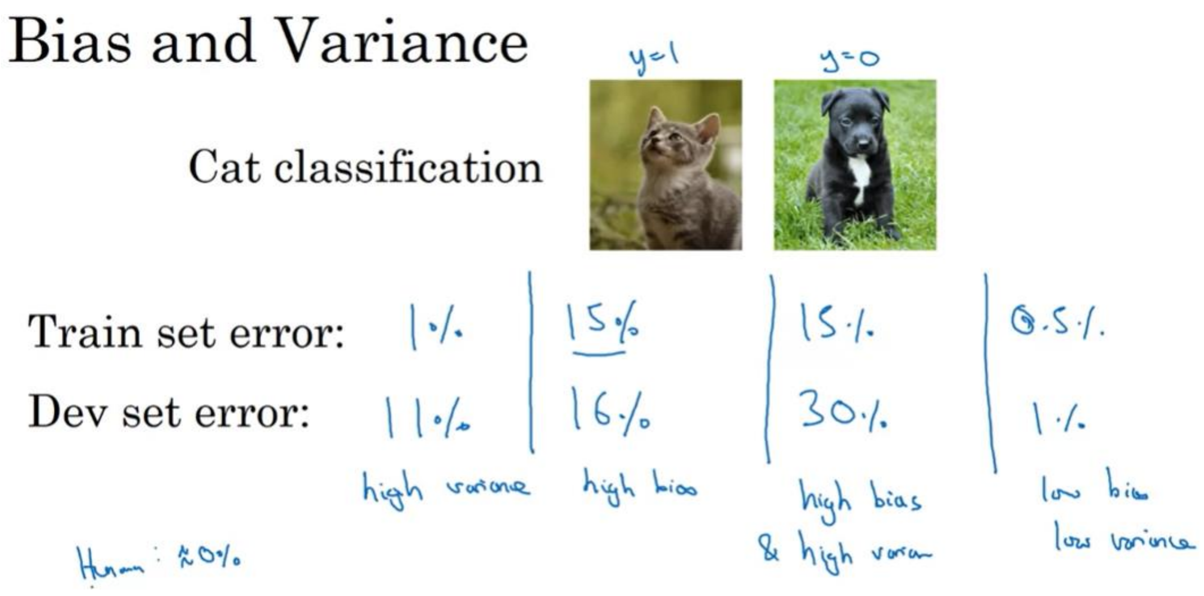
➢ Sometimes, it's okay to use only train/dev datasets:

- Sometimes, it's okay to not have a test set but only train/dev sets with 70%/30%
  - But there would be a risk to your model to get overfitted to the dev dataset , since there isn't the test dataset that remove this bias to the test dataset

IV. Bias/Variance:



- High bias means "Underfitting": It means that the model doesn't put the boundaries in a good manner and doesn't fit well to the data, it's a classifier with a high bias !
- High variance means "Overfitting": It means that the model is fitting perfectly to the data but doesn't look like a good fit, it's a classifier with huge variance
- The good model is the model who has a perfect bias with a perfect variance



Bias and Variance

Cat classification

y=1 y=0

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

Human: ≈0%

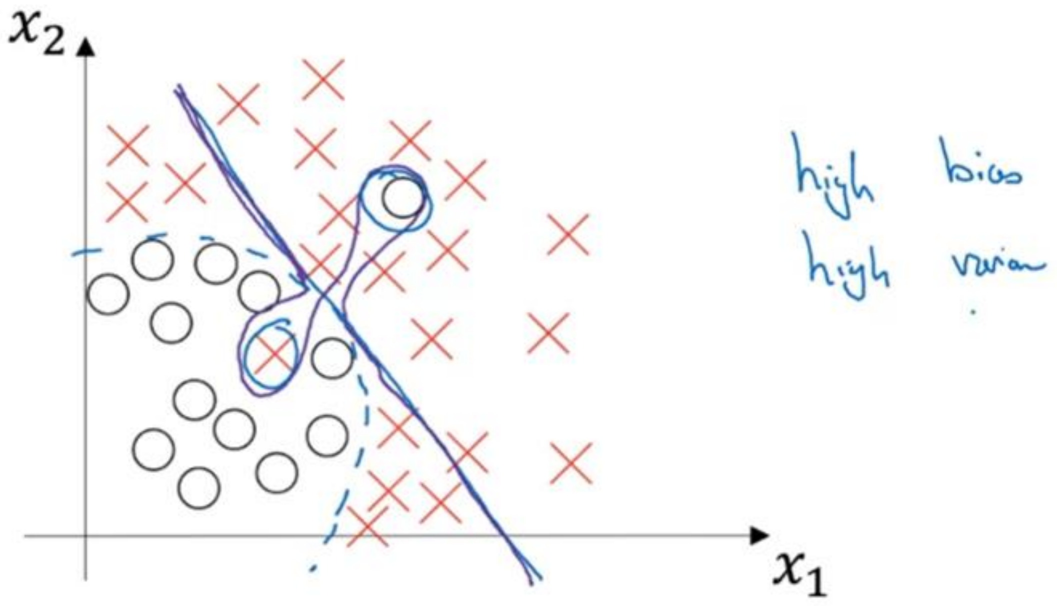
- Suppose we have a cat binary classifier:
  - If the model has a low error with the train set but a bigger one in the Dev set : High variance ⇔ Overfitting to the training data

- If the model has a high error with the train set and has almost the same error in Dev set: high bias ⇔ Underfitting
- If the model has a high error with the train set and a bigger one in the Dev set: High Variance & High Bias ⇔ Bad model
- If the model has a low error on the training set and a lower one also in the Dev set: Low Bias & Low Variance ⇔ Good model

Important Remark: these judgments was made by supposing that the human error to distinguish between the cat photos from the other pics is 0%

- If the Human error in a manual task is around 15% : then the second case (15% error in training set and 16% on the test set) is considered the good model
- The human error is called: **the Bayes Error**

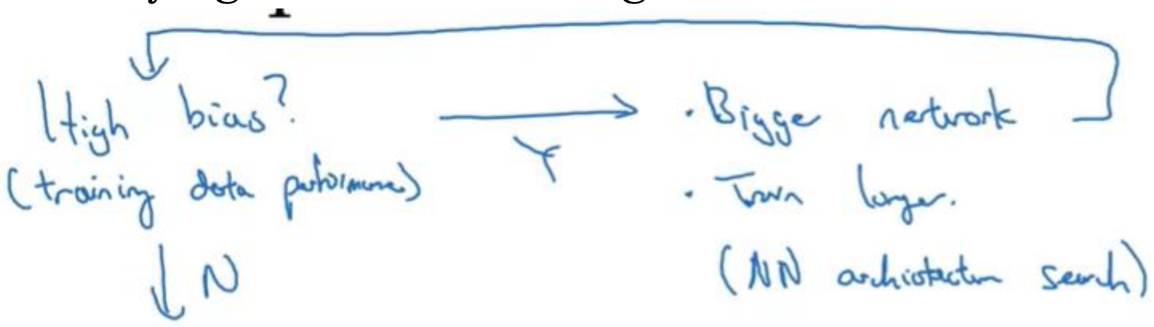
How does a model with High bias and high variance looks?



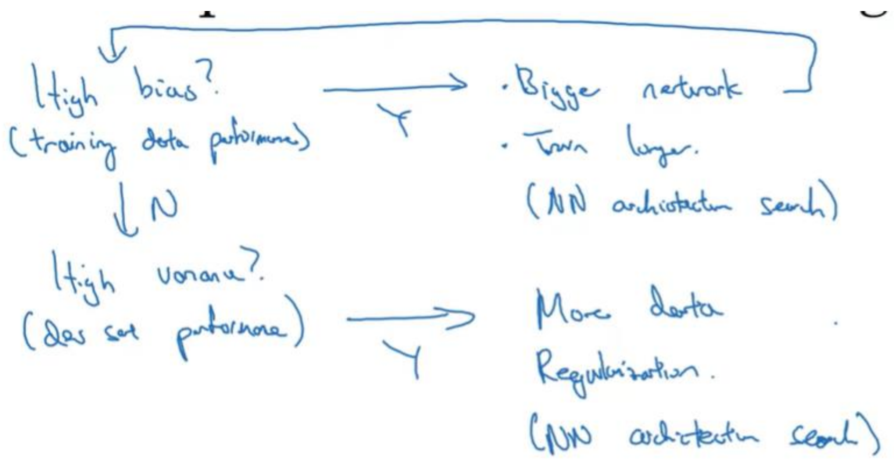
- It doesn't fit well to the data
- It isn't just a linear classifier, he has some variance that tries to overfit the data

V. Basic Recipe for Machine Learning:

A. Trying to remove the high bias of the model :



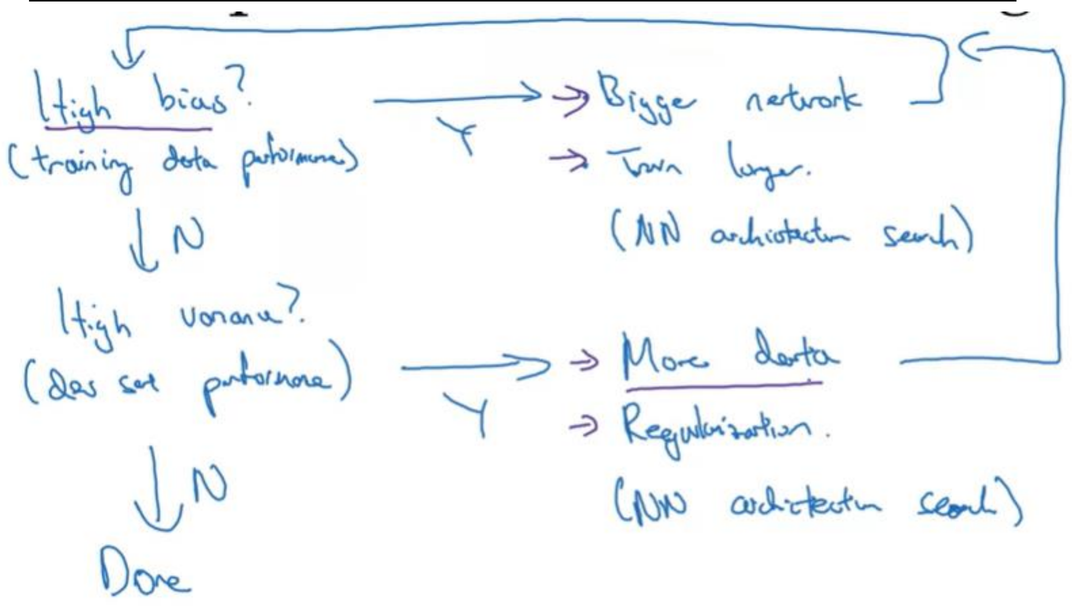
- We begin by testing our model performance on the training set by checking the bias ,If our model has high bias then we have some potential solutions to that :
  - Trying to make our network bigger by adding additional layers
  - Train the model for more epochs
  - Using specific type of NN architectures that will help to reduce the bias



B. Trying to remove the high variance from the model :

- Once we solve the bias problem, it’s the time to see if the model has a problem on fitting the dev Set by checking its variance, if so : we have potential solutions :
  - Giving our model more data
  - Regularization
  - Using specific type of NN architectures that will help to reduce the variance

C. Recheck the bias and the variance again until we have a model with low bias and low variance

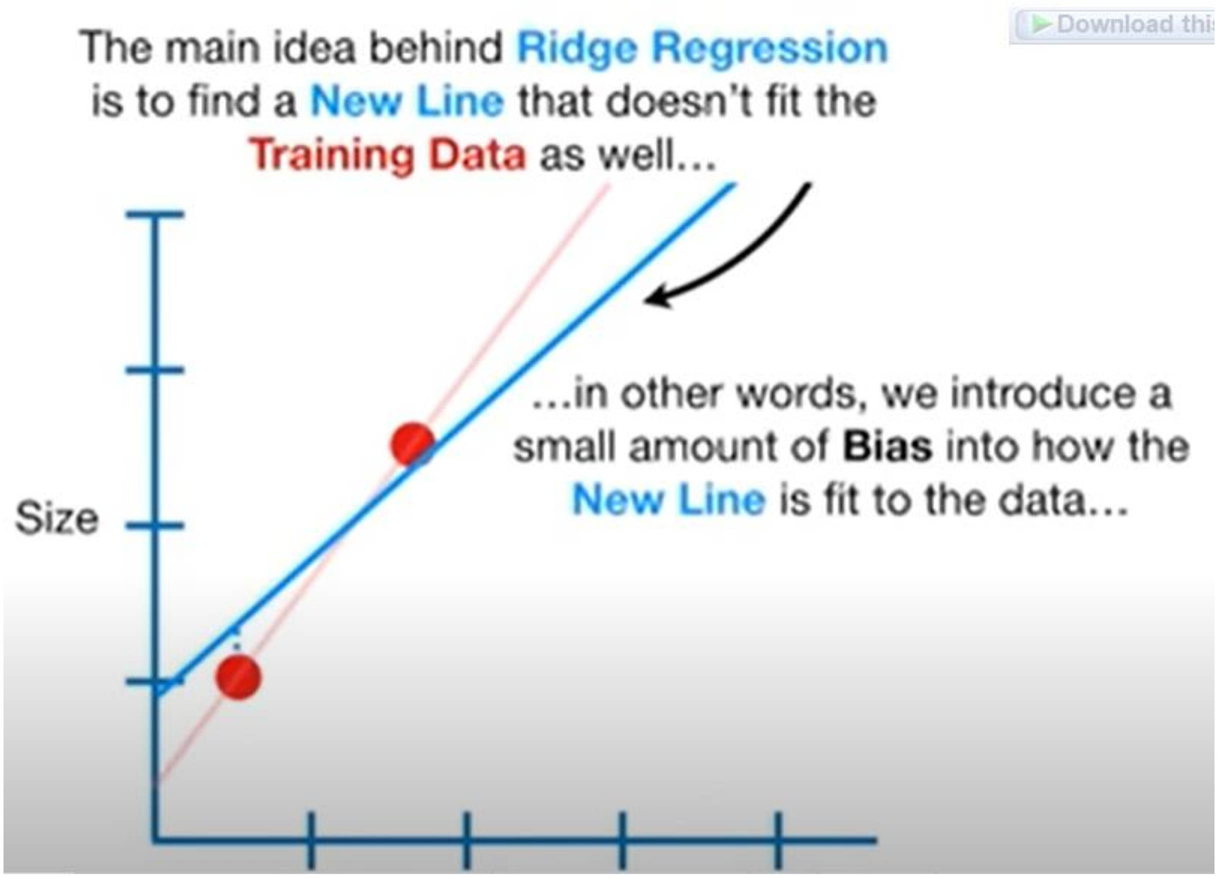


Important remark : In the earlier era of ML and DL , there was a discussion about the tradeoff of bias/variance , Because there was a solutions to decrease the bias and increase the variance, or to decrease the variance and increase the bias . But in the modern era of Machine Learning, it started to have tools to just decrease the bias or to just decrease the variance as long as we regularize properly.

- That's why Machine learning is considered so useful for supervises learning due to not having this tradeoff
- And now training with bigger networks never hurt (if we ignore the computation time issues) as long as we are regularizing

VI. Regularization (L2) and how it reduces the overfitting :

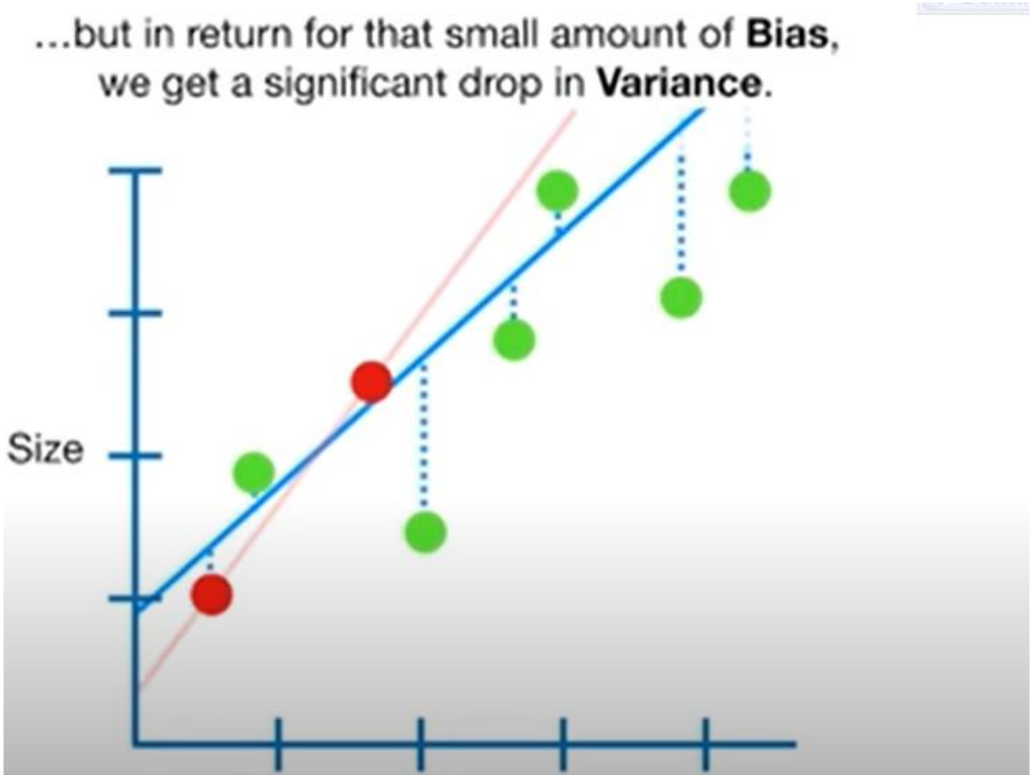
Learned it better by watching https://www.youtube.com/watch?v=Q81RR3yKn30


The main idea behind **Ridge Regression** is to find a **New Line** that doesn't fit the **Training Data** as well...

...in other words, we introduce a small amount of **Bias** into how the **New Line** is fit to the data...

- Suppose we have a few data (red points) , then trying to do a regression on it will make our model overfit easily (the red line) (Because the model can adapt easily to all our points so the cost function will be easily close to 0)
- The intuition of L2 regularization is to give our generated model a little **Bias** so he will not fit perfectly our data (the blue line) (fit it just good enough but not much)

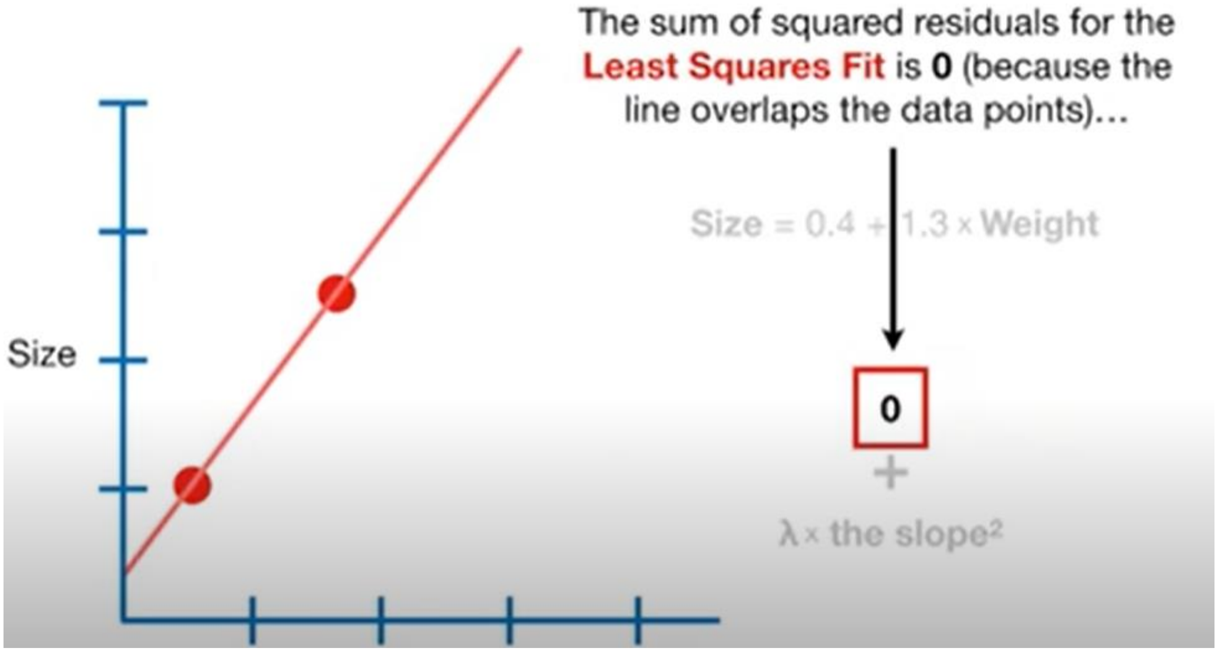
to avoid the overfitting to the training set



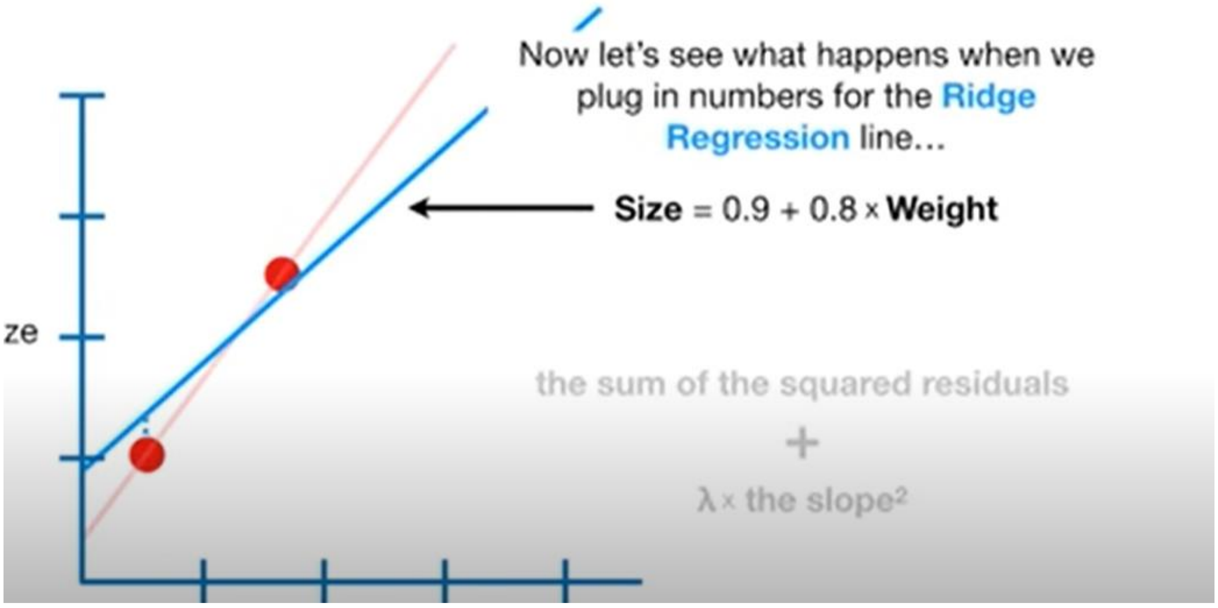
- And by doing that we are reducing the error in the dev set (or in the test set later)

<u>Conclusion:</u> L2 regularization helps on getting a better long term predictions

D.<u>How L2 regularization works:</u>


The sum of squared residuals for the Least Squares Fit is 0 (because the line overlaps the data points)...
Size = 0.4 + 1.3 x Weight
0
+
λ x the slope²
Size

- The bias added by the L2 is (lambda x the slope^2) to the error (which is already 0 due to the total overfitting to the data)
  - The slope is simply the coefficients of the model (W = (a0, a1, a2 , a3 , ...) without the a0)
  - In the 2D plan , we just have W=(a0 , a1) = (0.4 , 1.3) so the slope is just (1.3)^2 = 1.69
  - By having lambda=1 , In this example , the error with L2 regularization will just be sum of error + (lambda x the slope^2) = (0 + 0) + (1.69) = 1.69


Now let's see what happens when we plug in numbers for the Ridge Regression line...
Size = 0.9 + 0.8 x Weight
the sum of the squared residuals
+
λ x the slope²
ze

- For this blue which doesn't overfit , itq equation is Size = 0.9+ 0.8*Weight

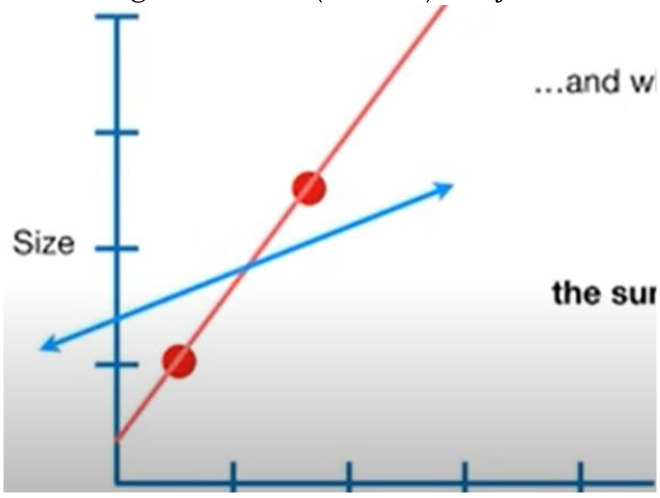
- The slope = 0.8^2 = 0.64
- And the sum of the errors = 0.09 + 0.01 = 0.1
- So the Total error = 0.1 + 1*0.64 = 0.74 which is better than the red line , So by using the L2 , we have as a result a new equation which doesn't' overfit to the training data

E. How L2 regularization prevents the overfit :

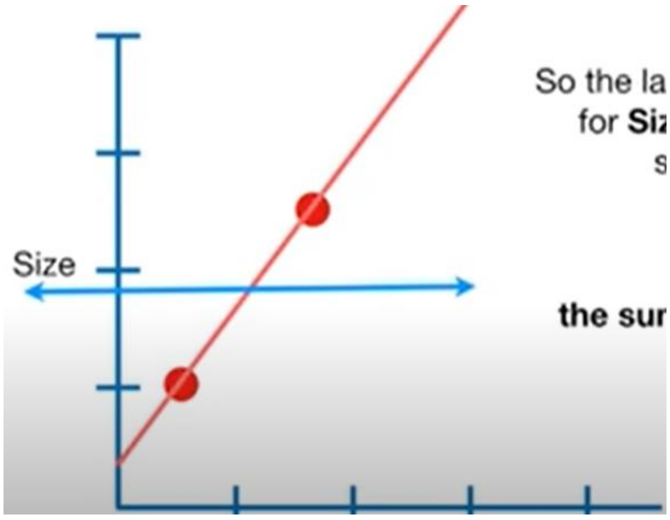
Since the additional bias related directly to the slope , so having a big coefficients of the model will just make this error gets higher , and that's why L2 resulted a line with small slopes (small coefficients) , so the predictions will get more sensitive to the Model attributes (which is weight in this case)

F. Lambda value (hyper parameter)

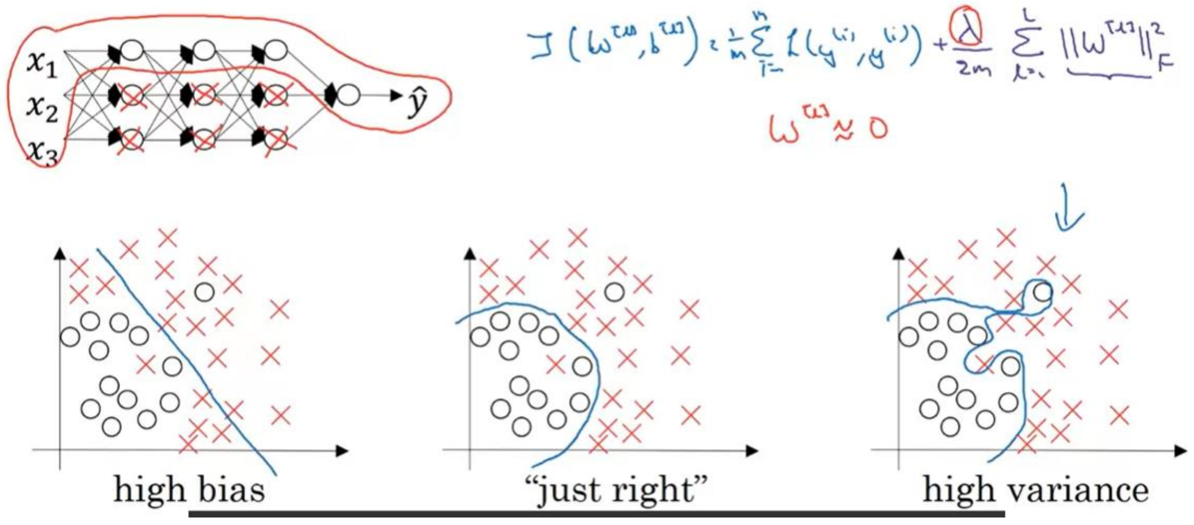
- Choosing a lambda=0 , will just cancel the L2 regularization and we will gte back to the original model
- Increasing the lambda (1 , 2 , 3) will just make our slopes gets lower and lower



- Having lambda going to infinity will just zeroing all the coefficients and we going
- to have W=(a0 , 0 , ...,0) so it just gonna be a straight horizontal line



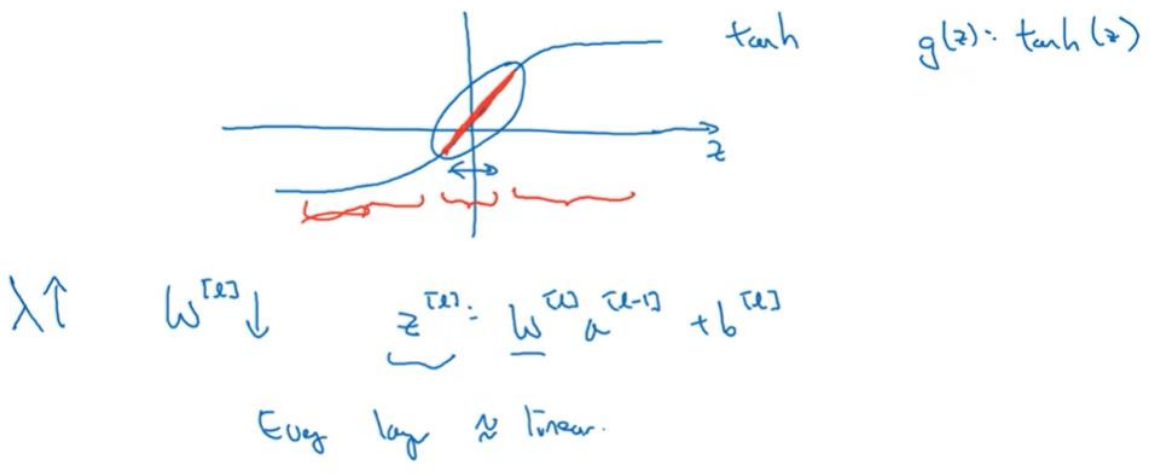
G. L2 regularization with Deep learning



x_1
x_2
x_3
$\hat{y}$
$J(w^{[l]},b^{[l]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)},y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$
$w^{[l]} \approx 0$
high bias
"just right"
high variance

- Having a bigger lambda will nullify the weights of units , so Some units in our Model are going to be totally ignored (the ones who reached a values close to zeros) and by that the model will be simpler and smaller will jump from "high variance" to "just right"
- **Warning :** Having a big value lambda will make us have a risk of having a model that will jump to "high bias" (underfitting) and this is due to nullifying the most of our NN units , So it's important to have a right value of lambda (not so low , not so big)

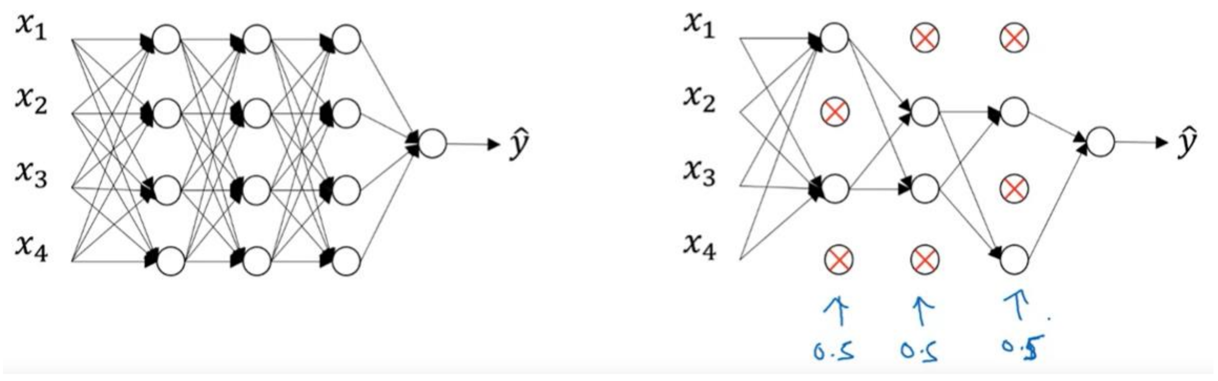
➢ How L2 cancel the effects of some units



- In the activation functions (and let's take tanh as an example here) , Having a tiny value of the slopes (W) will make tanh(W) give us values close to 0 (as tanh graph is showing in the middle) , and by that some units will have coefficients close to 0 => Nullifying the effect of this unit

VII. Dropout regularization: another technique to avoid the overfitting

H. What is dropout:



- The dropout is simply having the same neural network and where for each layer, we are going to ignore the effects of some units and we will end up with a smaller network.
- Nullifying the effect of some units will be by setting for each hidden layer a hyper parameter which defines the possibility of keeping its units
- In this example, we are having 3 random layers with 4 units each , by setting keep_prob=0.5 , we end up only by 2 functional units in each layer

I. Implementing dropout with "inverted technique"

Illustrate with layer $\underline{l=3}$. keep-prob = $\underline{0.8}$ $\underline{0.2}$

→ $\underline{d3}$ = $\underline{\text{np.random.rand(a3.shape[0], a3.shape[1])}}$ < $\underline{\text{keep-prob}}$

---

1. We are going to illustrate that in the third layer (l=3) and with keep_prob=0.8 and that means we are going to keep 80% of the units of this layer
2. The d3 is going to be array of [4 , 4] of True and False (80% True=1 , 20% False=0)

$\underline{a3}$ = np.multiply(a3, d3) # a3 *= d3.

3. Then, we are going to multiply the activation functions with the d3 Boolean array

   a. The a3 is going to be array of activation values and zeros , having zero in (x,y) means we are going to ignore the Yth unit and the X units of the previous layer lied to it
   b. If We are having a 50 units , then we are going to have 10 units shut off (0 value to their activation function)

$$\underline{a3} \mathrel{/}= \cancel{0.8}\ \text{keep-prob}$$

4. We are then going to divide the activations array with the keep_prob (which is 0.8 in this case)
   a. **<u>Why doing that :</u>** because since we eliminated 20% in the a3 matrix , and then the expected out put from this layer Z[4] = W[4] a3 + b[4] is going to have a value reduced by 20%
      i. So in order to refund the lost value from Z[4] we need to multiply the non-zero activation values by the prob_value (0.8) which means making it their values higher by 120% , and by that we are going almost the same expected output without having a values scaling down to 80%
   b. **<u>This line is the inverted dropout technique</u>** which is the most common technique to implement the dropout

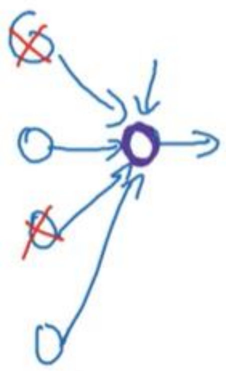
J. <u>Dropout regularization During training phase</u>

<u>Important remark:</u> During the training phase, and for each iteration (epoch) we are going to randomly zero out different units , so for each iteration we are going to have different smaller networks

K. <u>Dropout regularization During the prediction phase:</u>

<u>Important remark:</u> During the prediction phase, there will not be any dropout , because we are not wanting ton have a different random output each time we try to do the prediction on the same data + The dropout effect is important only on the training phase

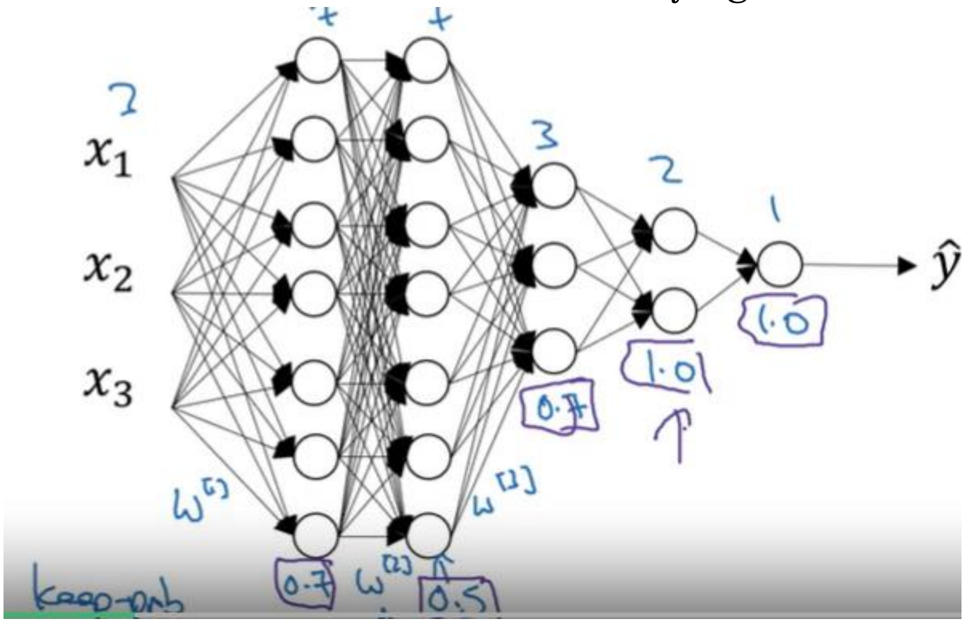
L. How can Dropout make us avoid the overfitting :



- The dropout assure that our model will not rely to any unit more than the other ones , because in each iteration we are going to removing randomly some units , so the weight of the units will be spread out between all of them and shrinking it without giving a huge value to anyone of them

VIII. Good practices with Dropout regularization

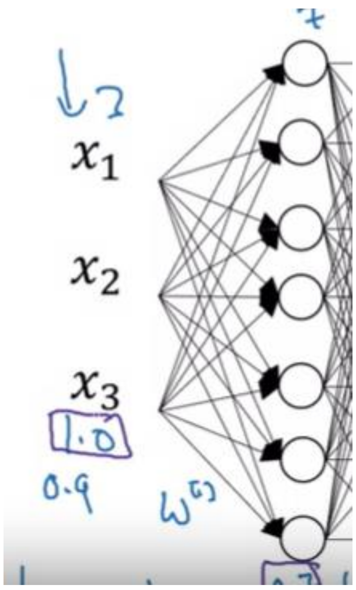
➢ Giving each layer its keep_prob depending on the number of units he is relaying to



- The number of the units is related to a layer is the number of the units of the current layer * the units of the previous layer

- The much the relied units to a layer , the higher the possibility to have a useless overfitting we have to get rid of , <u>so we are going to give a lower rate of keep_prob to the layer who depends to the maximum number of layers , and that's Why Andrew gave only 0.5 keep_prob to the third layer and 1 to the latest layers</u>

  - <u>It's possible to do dropout to the input layer , but it's not really recommended</u>



- We usually set it to 1 (or value closed to 1 , like 0.9)

  - <u>The most successful Dropout technique is in Computer vision field</u>

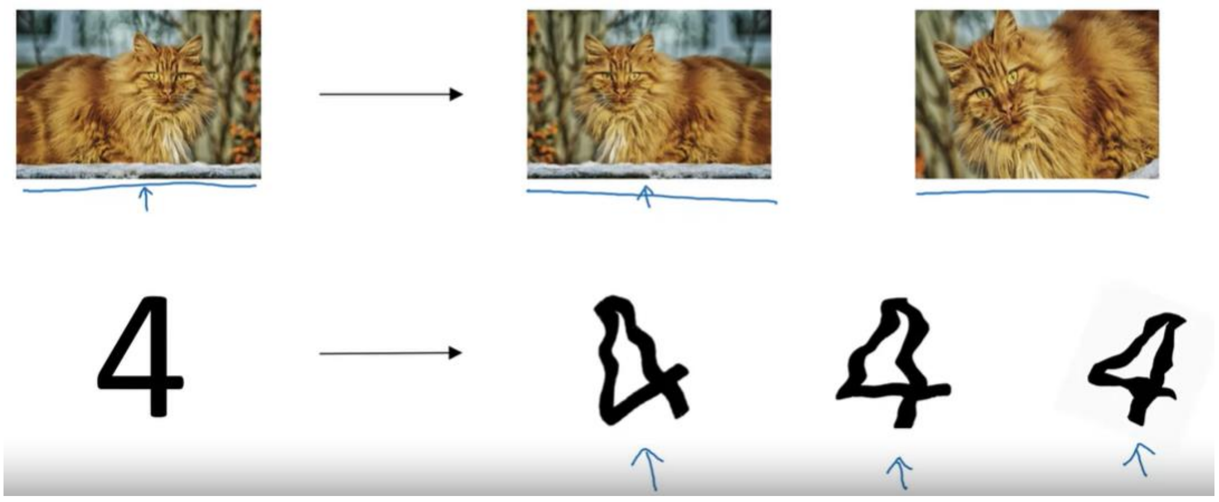
- This because in the Computer vision tasks, you don't have much data to train on, so the overfitting on it would be easy

  - <u>Use Dropout only when there is overfitting</u>
  - <u>Once we use Dropout in at least a layer: the cost function is no longer defined</u>

- This is because for each iteration we are going to randomly cut off some iterations, so we cannot calculate the global cost function of our model
- The best practice to observe the downgrade of the cost function In the graph to assure it really downgrading without the dropout (drop_prob=0) , and then start using the dropout.

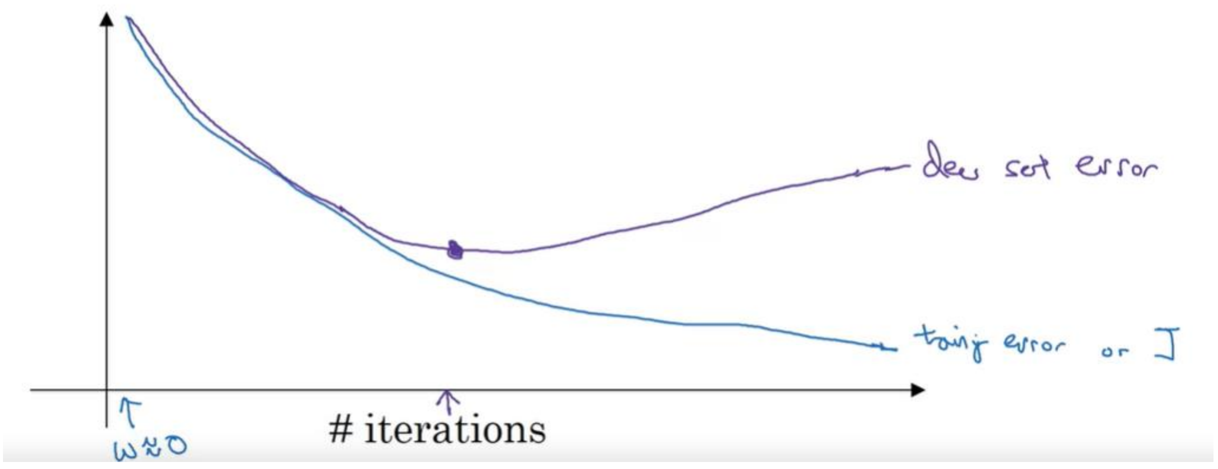
IX. Other techniques to avoid the overfitting:

M. Data augmentation:



- To resolve the lack of the data problem , we can By doing some flipping , translation and eventually waving up our dataset , we can get for free an additional data that will be treated as a fake training example , and by that we will avoid the overfitting that can be caused from the lack of the training set

N. Early stopping:



- Once we observe that test error starting to increase (which means the overfitting to the training set) , we stop the training of our model at this down peak

- The early stopping means stop the training phase of our model earlier

➢ Early stopping has a downside

Usually, during building a ML model we want firstly to optimize the cost error function and then , we try to remove the overfitting behavior from the trained model and Andrew recommends separating the tools that we use to optimize the cost function (Rms prop , momentum ..etc) from the tools we use to avoid the overfitting (regularization , adding data …) so in each tool we have only a single main focus , and this methodology is called **orthogonalization**

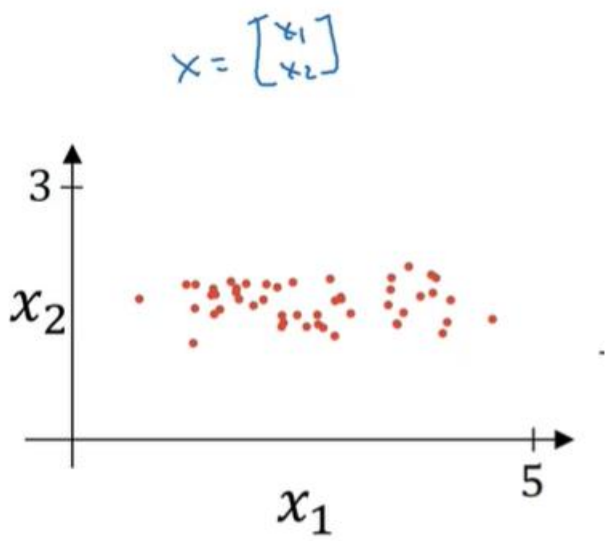
- The early stopping metho doesn't work with the orthogonalization methodology , because it's combines the two goals in a single tool since the goal of the training phase is to optimize the cost function and the stopping aims to stop the overfitting but also stopping the optimization of the loss function !

➢ L2 regularization is a better alternative to the early stopping

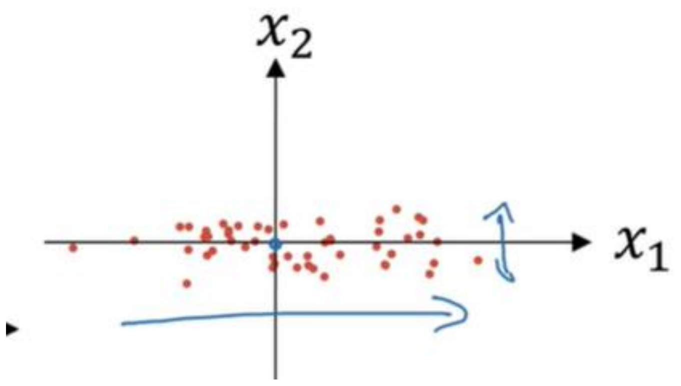
X. Normalizing the inputs in the training set and its importance:

O. How to do the normalization:

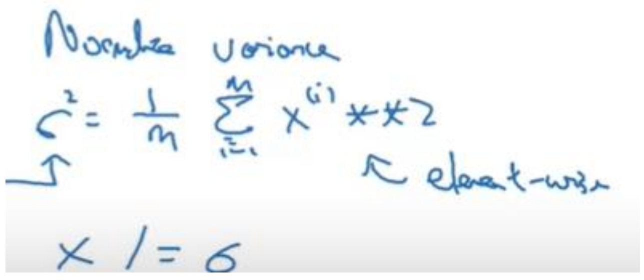
1. Starting from data that looks random like that in 2D (two attributes X1 and X2) :



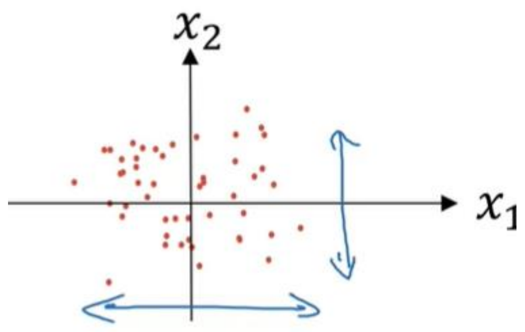
2. Firstly , We subtract from each point the mean vector (the mean of x1 , the mean of x2) : X = X-u , and now the mean of the values is 0



3. And then we normalize the variance by dividing by the variance: X = (X- u) / sigma , so now the point distributions in the X axis is the same on Y axis



Normalize variance
$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2$
element-wise
$x /= \sigma$



- In the training set , we use the same u and sigma of the training set ,because we don't want to do the normalization differently

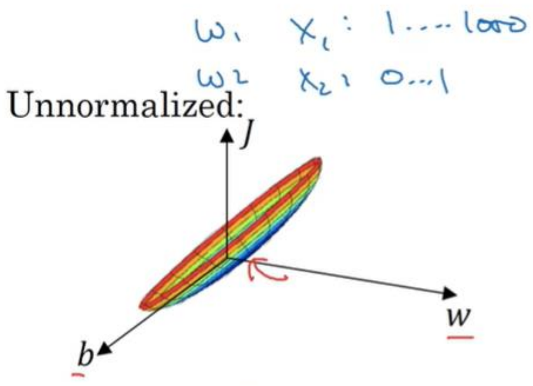
P. Why normalizing the inputs is efficient?

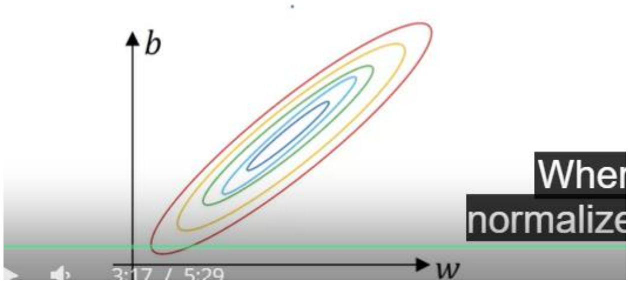
- Reminder: this is the cost function:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

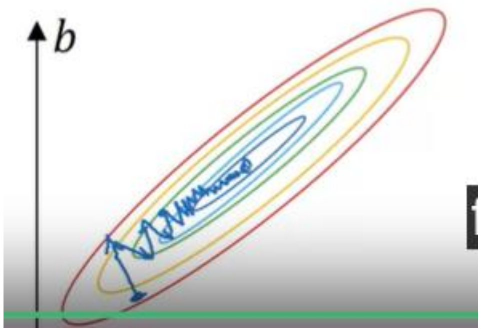
➢ Unnormalized Data case :

- If we have a unnormalized data , let's say we have two features W1 which varies between 1 and 1000 , and W2 varying between 0 and 1 .



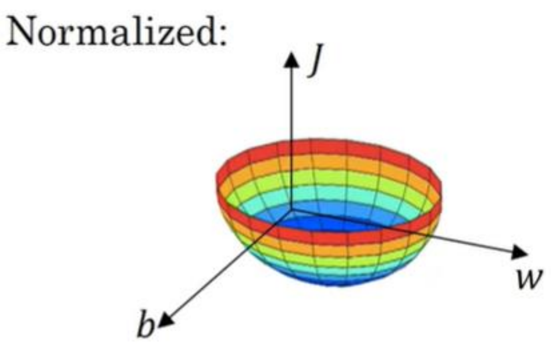


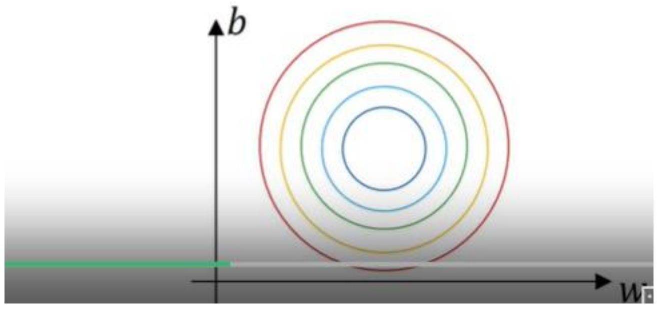
- Then the distribution of b,w and the cost function J is like that , while the little red arrow in the image represents the optimal value for the cost J
- The curve look very elongated



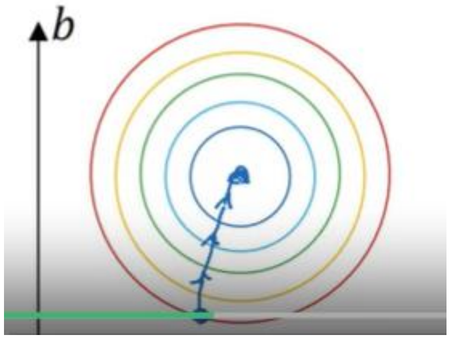
- In the case of unnormalized data , we have to go for a small learning late which will take many little steps osculating back in the gradient descents until getting to the optimal value of J (which is the center)

➢ The normalized data





- The normalized data look perfectly rounded, regular and symmetric



- In the normalized data, no matter where we will start, the gradient descent can go directly straight to the optimal value

Conclusion: The cost function is easier to optimize when the inputs are in the same scale

➢ Tips about the input normalization:

- It isn't really important to do the normalization if the inputs are in similar scale and range like these ones:

$x_1 : 0 \dots 1$

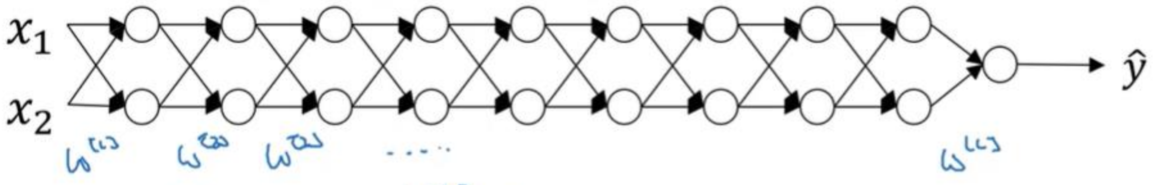
$x_2 : -1 \dots 1$

$x_3 : 1 \dots 2$

- So, it’s important to do the input normalization when the inputs come from very different ranges; otherwise n it’s not really efficient
- In both cases, the ization will not harm your data, so doing it anyway will be efficient more or less

XI. Vanishing / Exploding Gradient

Let’s suppose we have a **Very** deep network, the activation function g is a linear function (g(z) = z) and b =0



$g(z) = z$. $b^{[l]} = 0$.

$$y = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} W^{[1]} x$$

- Let’s suppose we have L layers , so we have L arrays W[l] of slopes and the out put Y is the multiplication of them by the input layer (W[L] * W[L-1] * …. * W[1] X)
- We have also Z[1] = W[1]X and a[1] = g(Z[1]) = Z[1] and a[2]=g(Z[2]) = g(W[2] x a[1]) = W[2] * a[1] = W[2] * W[1] * X and so on
- So y^= a[l] = g(Z[l]) = g (W[l] * a [l-1]) = W[l]*a[l-1] = W[l] * W[l-2]* W[l-3] * …. *W[1] X

$\underline{g(z) = z}$. $b^{[l]} = 0$.

$$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots \underbrace{\boxed{W^{[3]} \boxed{W^{[2]} W^{[1]} x}}}_{a^{[3]}}$$

$$z^{[1]} = W^{[1]} x$$

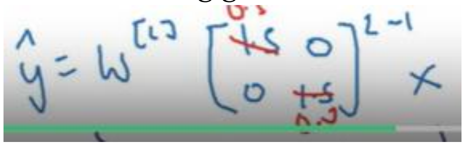
$$a^{[1]} = g(z^{[1]}) = z^{[1]}$$

$$a^{[2]} = g(z^{[2]}) = g(W^{[2]} a^{[1]})$$

- Let’s suppose that each W[l] is a little bigger than the identity matrix:

$$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

- So the y^= 1.5$^{(L-1)}$ X and all a[l] will have a growing values , which means the deeper was your NN (L become bigger) , your Y grows exponentially which means **<u>an exploding gradient</u>** and it will have quickly a huge value
- The same thing goes for a matrix little less than the identity :



$$\hat{y} = W^{[L]} \begin{bmatrix} \overset{0.5}{\cancel{1.5}} & 0 \\ 0 & \underset{0.5}{\cancel{1.5}} \end{bmatrix}^{L-1} X$$

It will give us y^= 0.5$^{(L-1)}$ X , all a[l] will have a lowering values close to zero which means the deeper was your NN (L become bigger) , your Y shrinks exponentially which means **<u>a vanishing gradient</u>** and it will have quickly a value close to 0

- By analyzing these two cases we can conclude that the output values can easily explode or vanish in the very deep networks if we don't control that

➢ <u>The solution ?</u>

- A partial solution of that is the careful choice of the initial weights of our DNNs

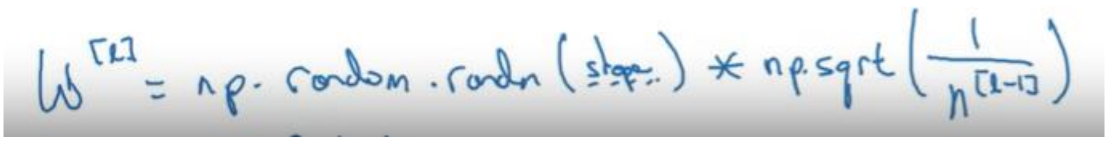
Q. <u>Weight initialization in DNNs</u>

➢ <u>Intuition</u>

- Since Z= W1X1+W2X2+….+WnXn , so the larger n is (the number of inputs : variables) , the less the W values we want (because Z is the sum of these Ws multiplied by Xs)

➢ <u>Reasonable solution :</u>

Set the variance of the initialized weights to be V(W) = 1/n (n = number of input features In the lth layer)



$$W^{[L]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[L-1]}}\right)$$

- Shape is the number of units (we are going to generate a matrix of the same shape as the current layer) in the lth layer and n[l-1] is the number of units in the previous layer (because it's the number of the input features that are going to be fed to the lth layer

➢ <u>For Relu there is a specific recommended variance</u>

- For relu activation , it's recommended to set the variance(W) = 2/n (instead of 1/n)

- **For tanh there is a specific recommended variance :**

- For relu activation , it's recommended to set the variance(W) = sqrt(1/n) (instead of 1/n) (Xavier initialization) or Variance(w) = sqrt(2/(n[l-1] + n[l])

- **Using randn helps solving the exploding/vanishing gradient:**

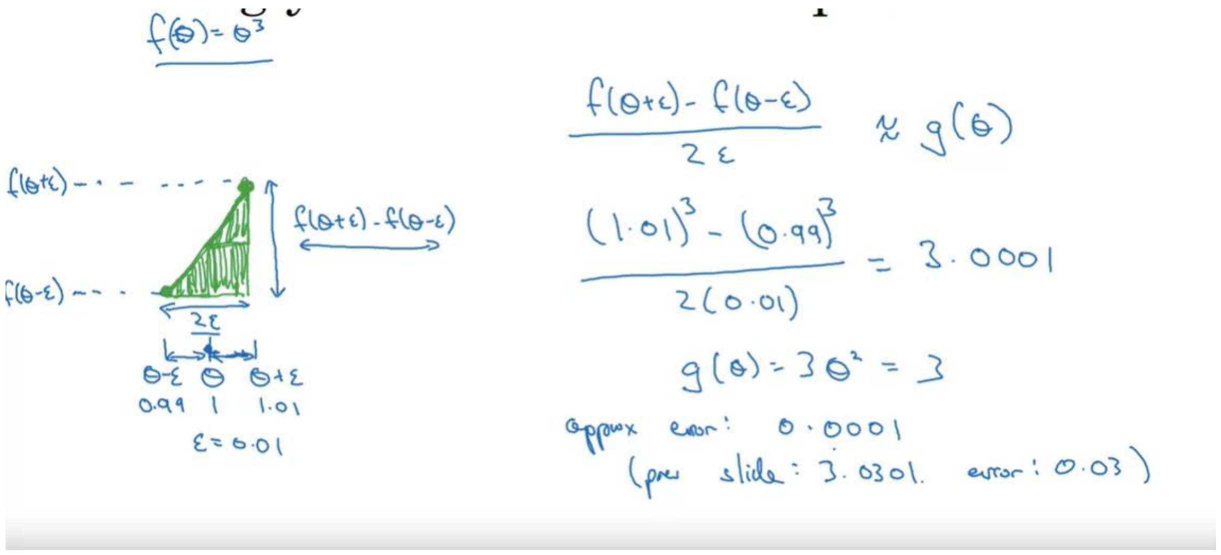
- Since randn gives us a matrix of values with "normal distribution" which means values close to 0 + with a standard variance : we are going to get values close to 1 so there wouldn't be neither a vanishing nor an exploding gradient

I have to understand more why grad check is used

XII. Gradient checking

R. Gradient checking in general

- Gradient checking is used to verify and debug the correctness of the implementation of the back propagation of our NN (sometimes you write all these equations and you're just not 100% sure if you've got all the details right and internal back propagation)
- And this is done by comparing the difference between the function derivative f'(x)=g(x) and the real definition of the derivative (using limits of the small bias called epsilon) :



$f(\theta)=\theta^3$
$f(\theta+\varepsilon)$
$f(\theta-\varepsilon)$
$f(\theta+\varepsilon)-f(\theta-\varepsilon)$
2ε
$\theta-\varepsilon$ θ $\theta+\varepsilon$
0.99 1 1.01
$\varepsilon=0.01$
$\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$
$\frac{(1.01)^3-(0.99)^3}{2(0.01)} = 3.0001$
$g(\theta)=3\theta^2=3$
approx error: 0.0001
(prev slide: 3.0301. error: 0.03)

- In this example , we built our model to have a gradient f(x) = x^3 , to verify the correctness of the real gradient at the point θ = 1 , we compare f'(1)=g(1)=$3\theta^2$ = 3 , and the real derivative using the definition , and they must have a very close values

S. Gradient checking in NN :

Take $\boxed{W^{[1]}}, \boxed{b^{[1]}}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector $\underline{\theta}$.

Concatenate

$J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = J(\theta)$

Take $\boxed{dW^{[1]}}, \boxed{db^{[1]}}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $\underline{d\theta}$.

concatenate

Is $d\theta$ the gradnt of $J(\theta)$.

1. We regroup all the parameters of each layer (W[1] , b[1] , ... W[L] , b[L]) of our NN into a big vector , this is our θ !
2. We take the partial derivative of each parameter and reshape them into a big vector , this is our dΘ
3. Our goal is to have dΘ equals to the slopes of the cost function J(Θ)

➢ How the grad checking is done in NN :

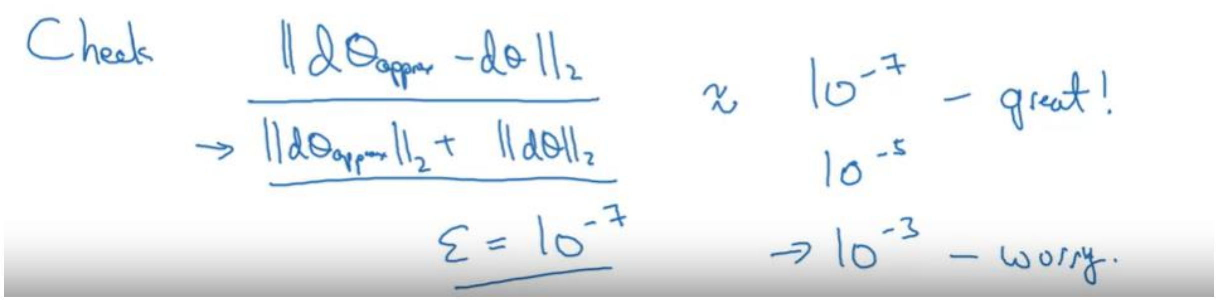
for each i :

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i} + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i} - \varepsilon, \dots)}{2\varepsilon}$$

$$\approx \underline{d\theta}[i] = \underline{\frac{\partial J}{\partial \theta_i}}$$

1. We consider each component in Θ as Θi (Ws and bs)
2. For each one , we calculate its partial derivative with the definition of the cost function J , and with the formal definition and we compare the results

- <u>When we cay say , that the implementation is correct and there is nothing to worry about :</u>



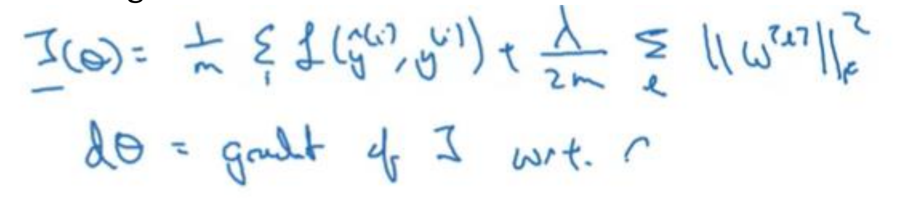
Check $$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx 10^{-7} \text{ - great!}, \quad 10^{-5}, \quad 10^{-3} \text{ - worry.}$$

$$\varepsilon = 10^{-7}$$

- This fraction should give us a little tiny value (10^-7 => great , 10^-5 => good , 10^-3 => we have to get worried)

- <u>Gradient checking implementation tips :</u>

- **<u>Don't use the grad check in training only to do debugging</u>**, because it's very slow in term of computation, do it just few times to check if the gradient is working correctly then turn it off
- **<u>If the grad checking test failed, the first solution is to look manually to the compounents Ws and Bs :</u>** If you notice that for example , the computations in Ws are acceptable but they are not in Bs , then you can realize that the computation faults concerns only the Bs
- **<u>If we are doing regularization, then we have to take it in consideration while calculating the derivatives:</u>**



$$J(\theta) = \frac{1}{m}\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_l \|W^{[l]}\|_F^2$$

$d\theta$ = gradt of J wrt.

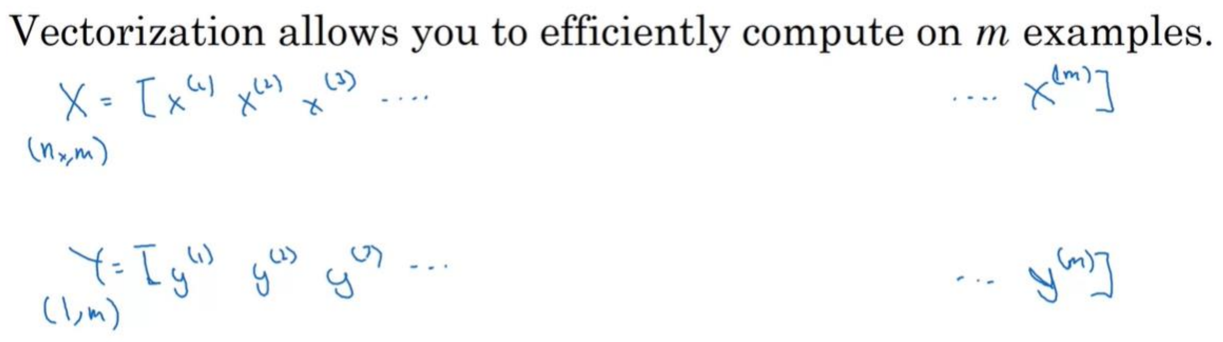
- **<u>You can't do the grad check while using the dropout :</u>** Since the cost J function is not defined due to the random elimination of the units during the training , If we wanna do the grad check , we must set manullay the keep_prob=1 in order to do the grad check
- **<u>For a precise checking, Do the grad check during the weight initializations, then redo it after some training iterations :</u>** This is because sometimes , the grad check works fine in the weight initialization when the weights of Ws and Bs are close to 0 , but once their values get bigger, It starts to appear some calculation faults that doesn't appear in the small values of the weights

XIII. The role of the optimization algorithms:

Set of techniques to Make your NN train faster

XIV. Mini batch Gradient

T. First, What is vectorization ? :



Vectorization allows you to efficiently compute on m examples.

$\underset{(n_x,m)}{X} = [x^{(1)} \; x^{(2)} \; x^{(3)} \; \dots \dots \; x^{(m)}]$

$\underset{(1,m)}{Y} = [y^{(1)} \; y^{(2)} \; y^{(3)} \; \dots \dots \; y^{(m)}]$

Suppose, you have M examples (each example X is a vector of nx attributes) in your data to train on , Instead for having an explicit For-loop which will pass by the M examples : one to one (and for each iteration he gives you corresponding output of a one single input) .... We will just stack them in a matrix with X[nx , m] values where each column presents an example from the M examples : this is the vectorization input ! and the input is a matrix Y[1 , m]

And with the that we gain a lot in term of computation time

U.Vectorization downside :

What if M was a huge number : (5 millions example or more) , your gradient descent algorithm have to pass by the whole 5 millions example in order to make one step to the optimal cost function!

It turns out that letting your gradient descent algorithm make a learning progress just by passing by a part of your data , and here where it comes Mini batch gradient

V. Mini batch intuituition :

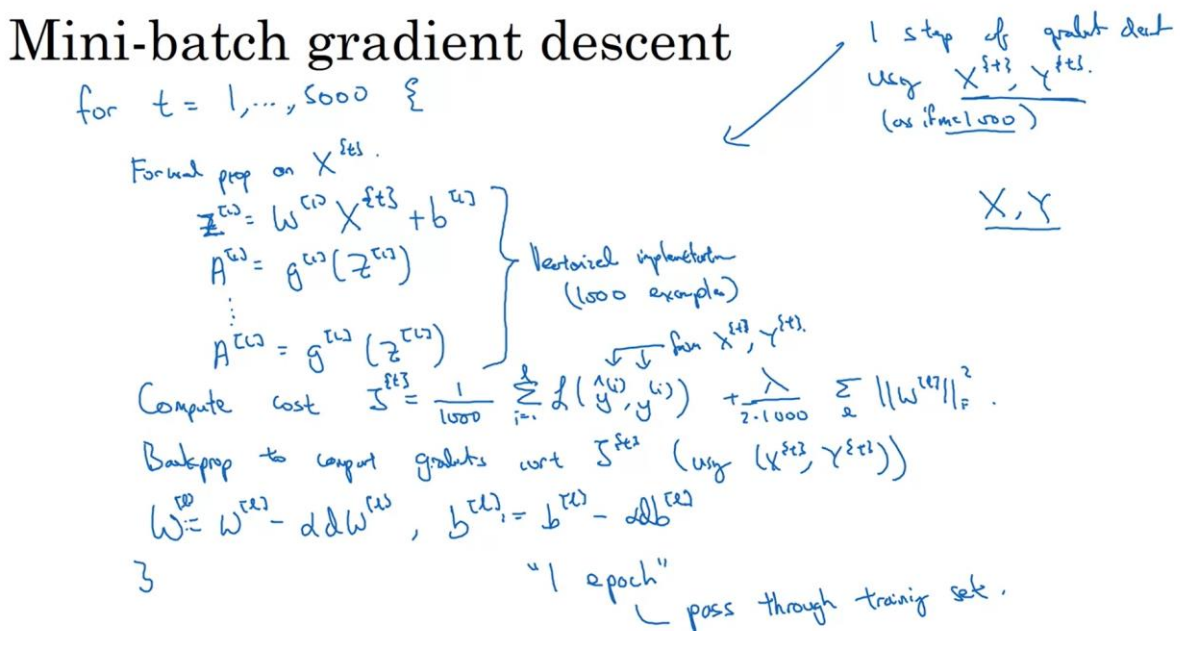
In order to make an intermediate solution between the stochastic gradient descent (passing the data one by one) and the Vectorization (passing the whole data as a single input) : let's split our data into single baby training sets (1000 examples each for example)

$$\underset{(n_x,m)}{X} = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)} \dots x^{(1000)}}_{X^{\{1\}}\ (n_x,1000)} \mid \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}\ (n_x,1000)} \mid \dots \mid \underbrace{\dots x^{(m)}}_{X^{\{5,000\}}\ (n_x,1000)}]$$

$$\underset{(1,m)}{Y} = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \dots y^{(1000)}}_{Y^{\{1\}}\ (1,1000)} \mid \underbrace{y^{(1001)} \dots y^{(2000)}}_{Y^{\{2\}}\ (1,1000)} \mid \dots \mid \underbrace{\dots y^{(m)}}_{Y^{\{5,000\}}\ (1,1000)}]$$

- For 5 million examples and with batch_size=1000 , we will have 5000 batches , each one has as dimension (nx , 1000) as input X{i} and corresponding output Y{i} with [1,1000] as dimension

W. Mini Batch gradient algorithm :



Mini-batch gradient descent

```
for $t = 1, \dots, 5000$ {
    Forward prop on $X^{\{t\}}$.
        $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
        $A^{[1]} = g^{[1]}(Z^{[1]})$
        ⋮
        $A^{[L]} = g^{[L]}(Z^{[L]})$
    Vectorized implementation (1000 examples)
    Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$.
    Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)
    $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$, $b^{[l]} := b^{[l]} - \alpha db^{[l]}$
}
```

from $X^{\{t\}}, Y^{\{t\}}$.

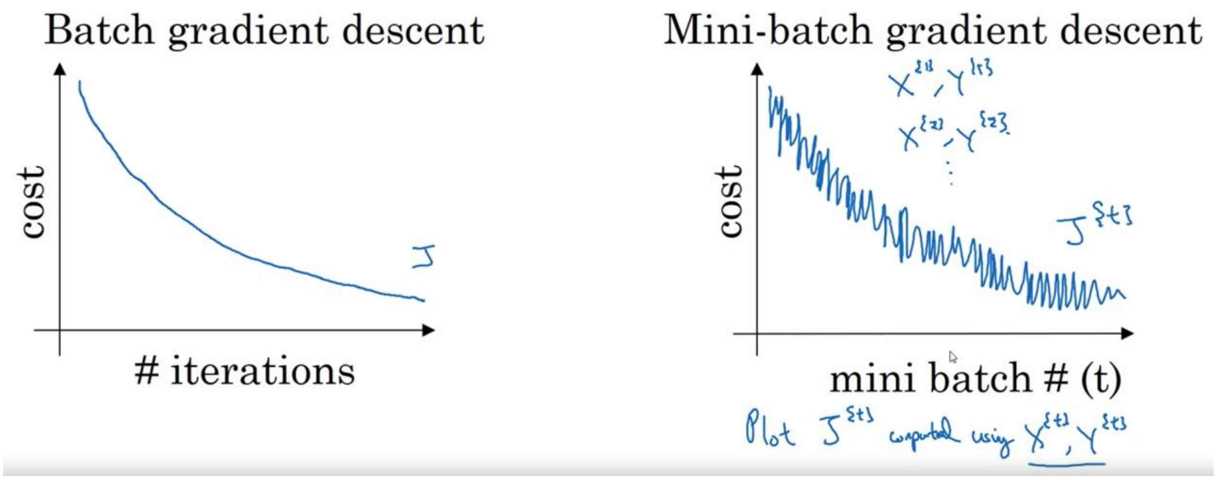
1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$. (as if $m = 1000$)

X, Y

"1 epoch" pass through training set.

- After splitting our 5M data into 5000 batches, for each batch we will do :
  - Forward prop on this mini batch
    - Passing the 1000 inputs (as a single matrix) into the L layers, and we calculate the activation function for each layer until we reach the final layer L (a mini-vectorization example)
  - We compute the cost function J{t} related to this batch (it may have eventually the regularization factor)
  - We do the backprop in order to update the weights of each unit
  - And by doing that , we passed 1 epoch and made a 1 step in Gradient descent using X{t} , Y{t}

X. Mini Batch gradient in details :

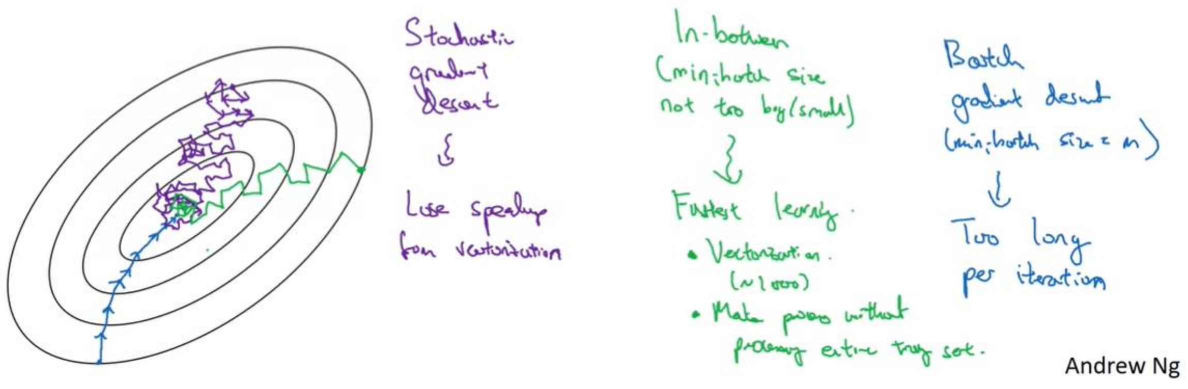


- With classic Batch gradient descent (vectorization) , we notice that the cost function decreases with each step
- Mini Batch in contrast, It has a noisy oscillated step but its trend is on decrease obviously
    - That's because the cost function will be calculated for each batch , It might happen that the X{1} , Y{1} is an easy batch to predict while X{2},Y{2} is a hard batch … so the cost value depends on the current batch
    - But in overall the cost function is decreasing although it's noisy behavior which means there is really a learning

XV. Choosing the mini batch size :

- Batch_size =1 => Stochastic gradient descent
- Batch_size=M => Vectorization (X{1} , Y{1}) = (X , Y)
- Batch_size between 1 and M : Mini Batch gradient descent

Y. Learning progress in each case :

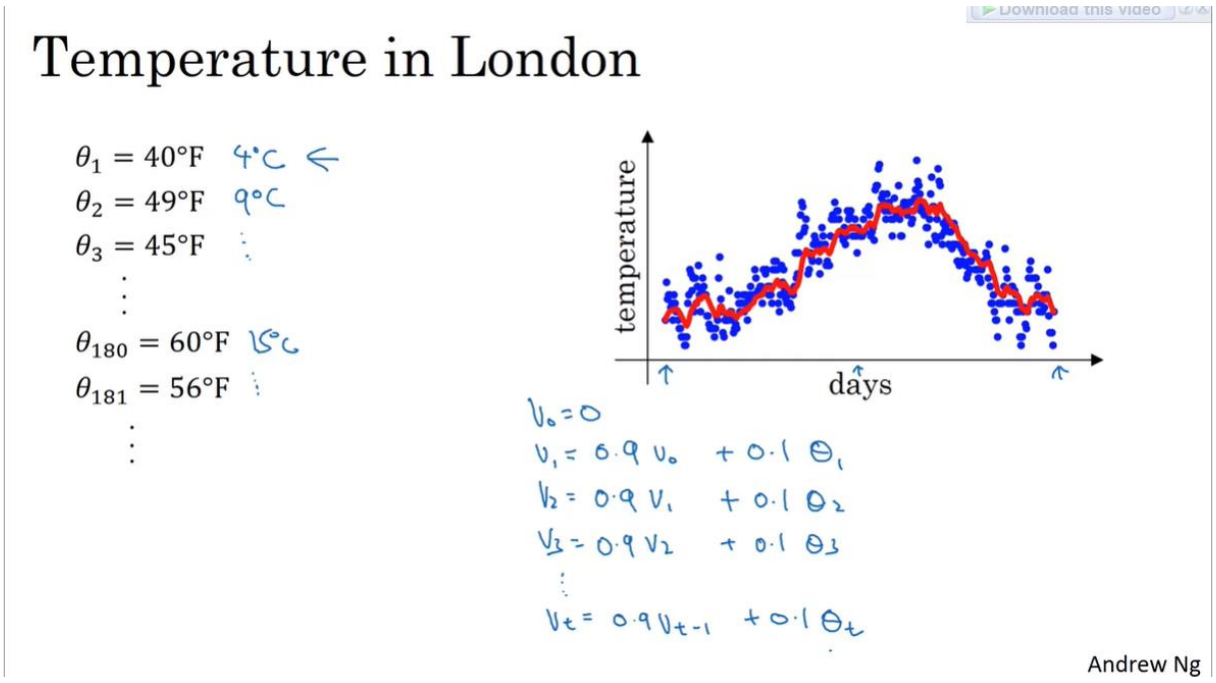

Stochastic gradient descent
Lose speedup from vectorization
In-between (mini-batch size not too big/small)
Fastest learning.
• Vectorization. (~1000)
• Make progress without processing entire training set.
Batch gradient descent (mini-batch size = m)
Too long per iteration
Andrew Ng

- Stochastic gradient descent (m = 1) :
  - In every iteration we make a very little progress to the optimal value , and this little progress , may cause the next iteration to be on another direction (but always oriented to the center which represents the optimal value) , so it's oscillating but converge to the center with many noise , but never ends in the center , it just keeps moving around the center
    - Cons :
      - Lose the speed of handling all the data in a single iteration
- Vectorization (m=M) :
  - Large straightforward steps to the center which become more little
    - Cons :
      - Take too much time per iteration of learning (it treats all data for each iteration)
- Mini Batch Gradient descent (in between) :
    - Pros :
      - Fast learning + mini vectorization (making progress without the need to wait to process the whole data)

Z. Tips to choose the most suitable batch size :

- If we have a small training set (M < 2000) :
  - Use Batch gradient Descent (vectorization)
- For a large training set :
  - Use typical batch sizes (power of 2 , between 32 and 512 and rarely 1024) + the size can be fitted in the CPU/GPU memory :
    - Why power 2 ? :
      - Because the computer runs faster with the power of 2

XVI. Exponentially Weighted Average (moving average) :



Temperature in London

$\theta_1 = 40°F$ 4°C ←
$\theta_2 = 49°F$ 9°C
$\theta_3 = 45°F$
⋮
$\theta_{180} = 60°F$ 15°C
$\theta_{181} = 56°F$
⋮

$V_0 = 0$
$V_1 = 0.9 V_0 + 0.1 \theta_1$
$V_2 = 0.9 V_1 + 0.1 \theta_2$
$V_3 = 0.9 V_2 + 0.1 \theta_3$
⋮
$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

Andrew Ng

- To get the moving average in the instant t : V(t) : we took 0.9*V(t-1) + 0.1 theta(t) (theta is the observed value at the time t) , and by applying that we got the red line which represents the average of the observed time Series

AA. General Formula :

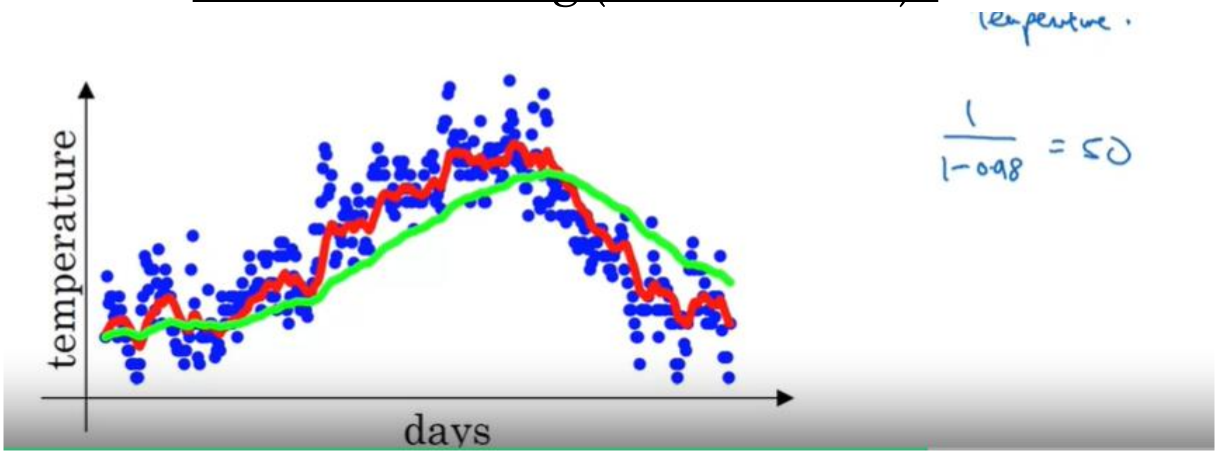
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$
$$\beta = 0.9$$

- The general formula to get the V(t) is to multiply the V(t-1) by beta and theta(t) by Beta-1
  - In the above example : Beta was equal to 0.9

➢ What is exactly V(t) :

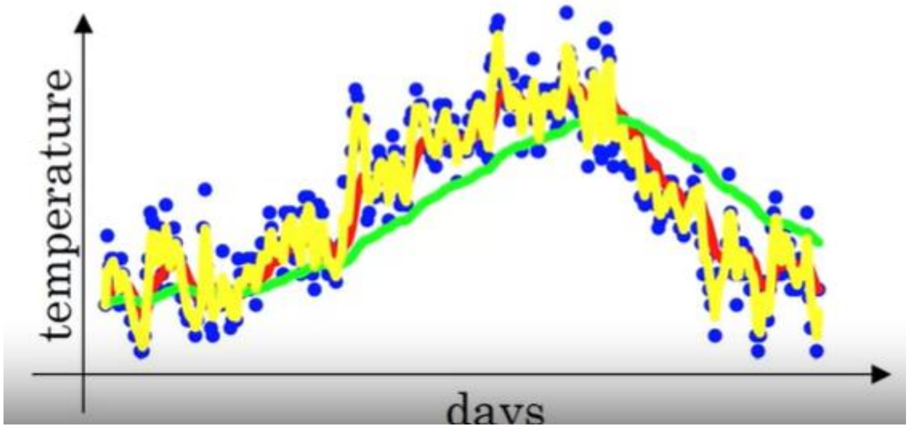
V(t) is approximately the average of the previous (1/(1-Beta)) , so with Beta=0.9 we was averaging by the past 10 days .

BB. If Beta was so big (so close to 1) :



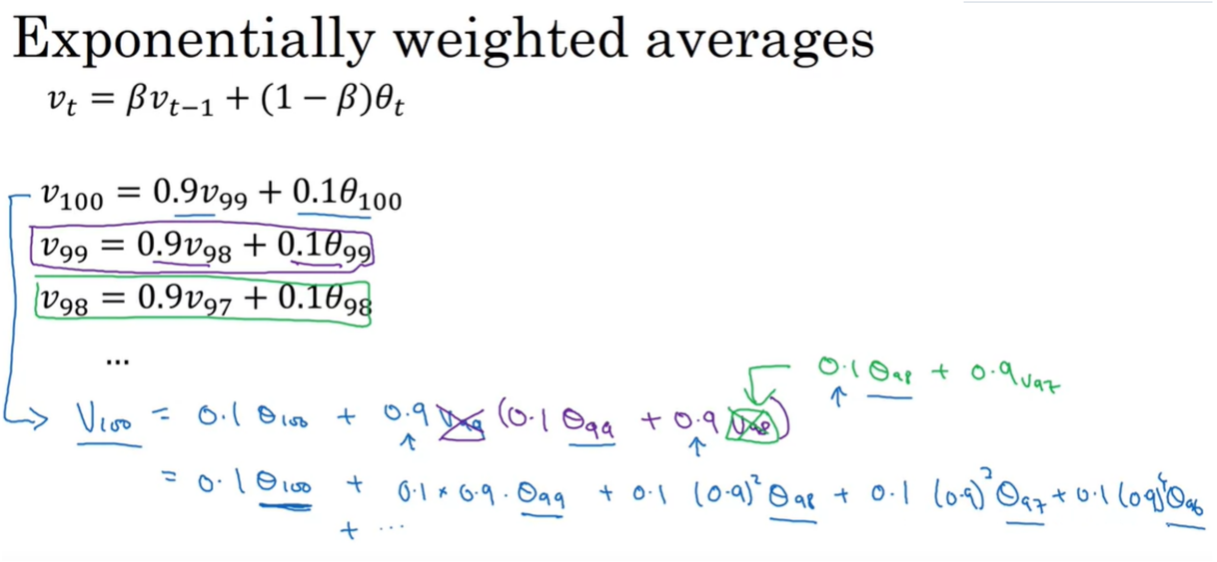
- By taking Beta=0.98 , V(t) will average the past 50 days , and by that we will get the green line which is :
  - Smoother and less wavy ; because we are averaging past 50 values which will make having a new value making a slow effect in the previous average
  - Shifted to the right : this is because by trying to od the average of many previous day, this will make V(t) adapts slowly to the current value Theta(t) because Beta is now giving a bigger weight to the past values

CC. If Beta was so little (far from 1) :



- By taking Beta= 0.5 , V(t) will average the past days only , and by that we will get the yellow line which is :
  - So noisy , because the window to average contains only two values , and more sensitive to the outliers
  - Adapts very quickly to the current temperature Theta(t)

DD. Understanding Exponentially weighted average in details :



Exponentially weighted averages

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

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$

$$\boxed{v_{99} = 0.9v_{98} + 0.1\theta_{99}}$$

$$\boxed{v_{98} = 0.9v_{97} + 0.1\theta_{98}}$$

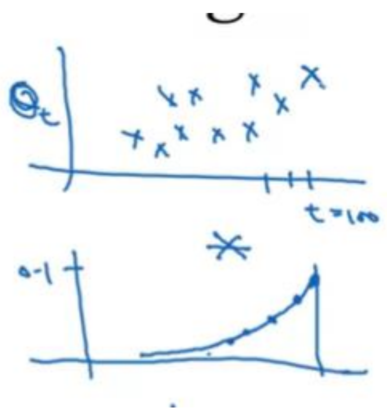
…

$$0.1\,\theta_{98} + 0.9\,v_{97}$$

$$v_{100} = 0.1\,\theta_{100} + 0.9\,\cancel{v_{99}}\,(0.1\,\theta_{99} + 0.9\,\cancel{v_{98}})$$

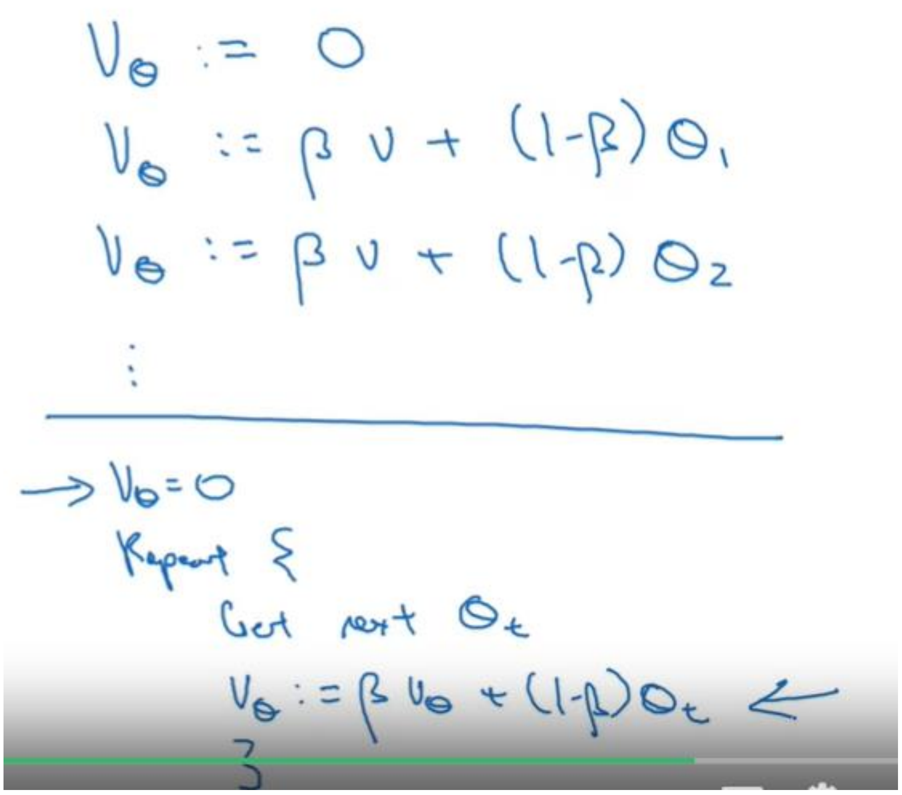
$$= 0.1\,\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1\,(0.9)^2\,\theta_{98} + 0.1\,(0.9)^3\,\theta_{97} + 0.1\,(0.9)^4\,\theta_{96} + \cdots$$

- By taking Beta=0.9 and trying to calculate V(t=100) :
  - We will get V(100) = 0.1theta(100)+(0.1*0.9)Theta(99)+(0.1*0.9*0.9)Theta(98) + …..+
    - We are noticing that the more we go to the past values , the more their weight got decreased exponentially , and that's why this computation is called exponentially weighted !



- In nutshell , we are trying to do wise multiplication between the last 100 temperature values with the weight decay exponentially like it's shown in the image above
- We have said that having beta =0.9 will make the V(t) took the average of approximately the past 10 days because (0.9)^10 is already less than 0.35 and for other days will decay so quickly , so the other days will get a very little neglected weight (and the same goes to beta=(0.98)^50 is less than 0.35)

EE. <u>Final Results (who exponentially weights really would look like while computation) :</u>



$$V_\theta := 0$$
$$V_\theta := \beta v + (1-\beta)\theta_1$$
$$V_\theta := \beta v + (1-\beta)\theta_2$$
$$\vdots$$

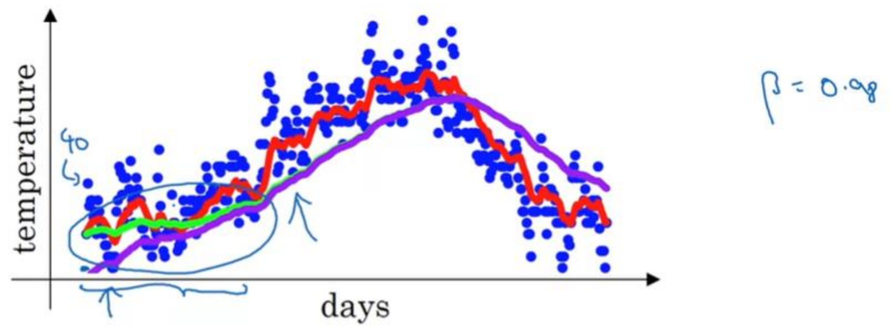
```
$\rightarrow V_\theta = 0$
Repeat {
    Get next $\theta_t$
    $V_\theta := \beta V_\theta + (1-\beta)\theta_t \leftarrow$
}
```

FF. <u>advantages of this exponentially weighted average formula:</u>

- <u>**it takes very little memory**</u>. You just need to keep just one row number in computer memory, and you keep on overwriting it with this formula based on the latest values that you got.
- <u>**the efficiency**</u>, it just takes up one line of code basically and just storage and memory for a single row number to compute this exponentially weighted average.

GG. Bias correction in the exponentially weighted average :

Bias correction



- In reality , and without the bias correction : the red line (beta = 0.98) would look like the purple line which is clearly not accurate as the biased one (the red one)

➢ The intuition behind the need to a bias :

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

$$V_0 = 0$$

$$V_1 = \cancel{0.98 V_0} + 0.02\,\theta_1$$

- According to the previous formula , we begin with V0=0 , and then V1 would equal to 0.98V0 (= 0) + 0.02 theta1 and that's why the first weights in the purple line (in the purple circle) looks very close to zero and far from being in the average

➢ How the bias would fix the average of the first days :

$$\frac{V_t}{1-\beta^t}$$

$$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$$

$$\frac{V_2}{0.0396} = \frac{0.0196\,\theta_1 + 0.02\,\theta_2}{0.0396}$$

- We replace the V(t) by V(t) /(1- beta^t) :

- **When t is little is close to 0:** (1- beta^t) will be close to 0 and that would make the V(t) scale to bigger values
- **When t is little is too big:** _beta^t) will be close to 0 (1-beta^t close to 1) and that would make the V(t) stuck on its current value (or scaling in a very little way)

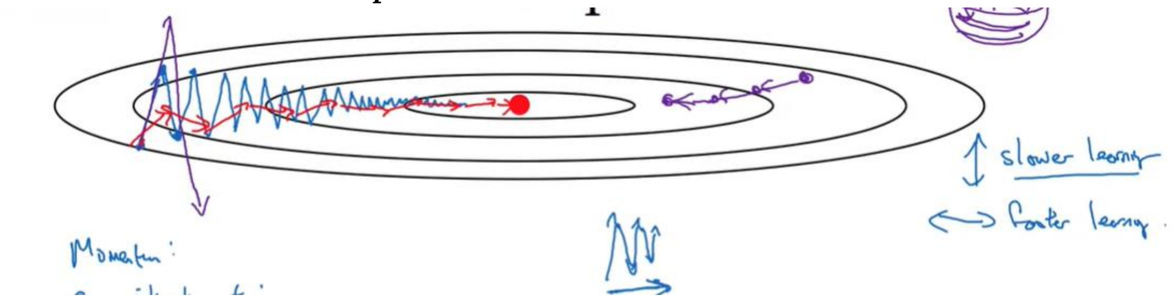
And by that, we assured to scale up the V(t) values only in the very first days only and that would make the purple line becomes the red line .

XVII. Gradient descent with momentum: for quicker learning

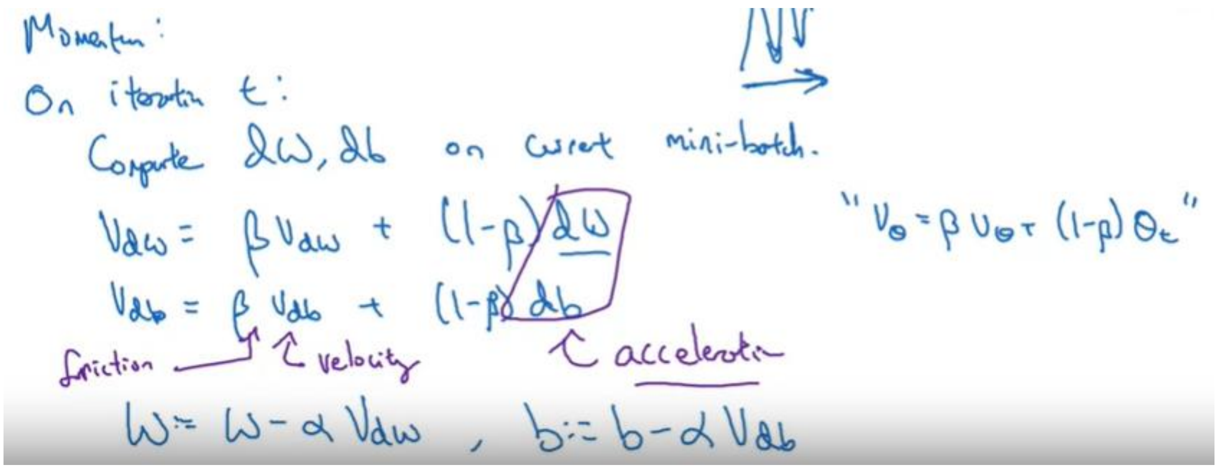
➢ The idea :

Use the exponentially weighted average to update the gradient weights

➢ How's that possible :



- **The purple oscillation**: the gradient descent with a huge learning rate (not recommended at all)
- **The blue oscillation:** the gradient descent with an accurate learning rate
- **The red oscillation:** the gradient descent with an accurate learning rate + the momentum with an accurate beta, the learning is quicker horizontally and slower vertically which makes our gradients weights get updated in an optimal way and quicker way to the optimal values, **we can see that the red oscillation is the moving average of the blue one, and that's how the exponentially weighted average idea is involved in the momentum!**



Momentum:

On iteration t:

Compute dW, db on current mini-batch.

$V_{dW} = \beta V_{dW} + \boxed{(1-\beta)\underline{dW}}$

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

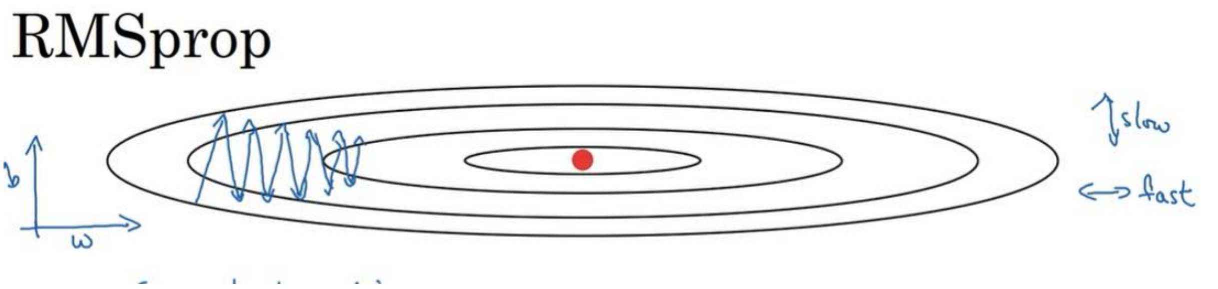
friction $\rightarrow$ $\uparrow$ velocity $\uparrow$ acceleration

$W := W - \alpha V_{dW}$, $b := b - \alpha V_{db}$

"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

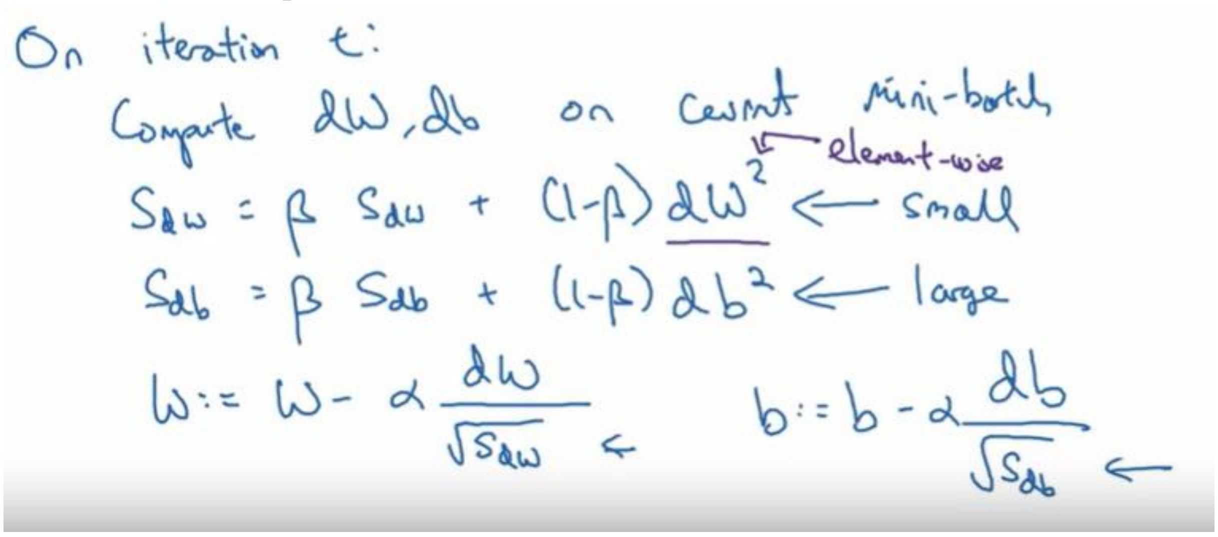
- And the weights are updated like it's shown above, the weights are no longer represented by dW for the weights and db for the biases but with the Vdw and Vdb which are the same of V(t) in example shown for exponentially weighted average.
- The more beta is bigger (close to 1) , the more the update of the weights becomes smoother , **the perfect value of momentum is often 0.9**

XVIII. RMSPROP (Root Mean Square Prop): another way to speed up the learning rate efficiently



- For the intuition sake , let's say that the X-axis represents the W weights and Y-axis represents the b weights , our main goal then is to scale up the W weights in each gradient descent iteration and scale down the b's in order to have a faster learning horizontally and slower one vertically .

- <u>Computation to do in each iteration :</u>



On iteration t:

Compute dW, db on current mini-batch

$S_{dW} = \beta S_{dW} + (1-\beta)\underline{dW^2} \leftarrow$ small

element-wise

$S_{db} = \beta S_{db} + (1-\beta) db^2 \leftarrow$ large

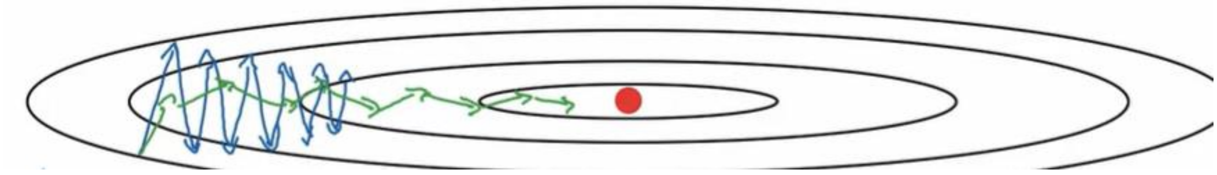
$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}} \leftarrow$ $\qquad b := b - \alpha \frac{db}{\sqrt{S_{db}}} \leftarrow$

- Instead of updating the weights W by W-alpha*dw (alpha = learning rate) , we are gonna add an additional factor which is the root of Sdw (the same goes for b weights with Sdb)
- The Sdb and Sdw are going updated using a hyper-parameter Beta (which has a similar impact to Vt values in exponentially weighted average)

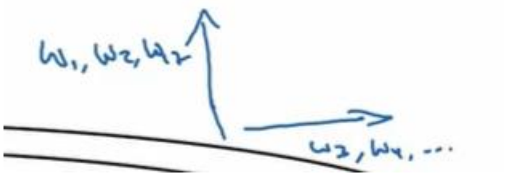
- <u>The intuition behind dividing by Sdb and Sdw</u>

- b weights are big (and need to scale down) => db^2 has a big value too => Sdb has a big value => b weights (which got Sdb as denominator now) are gonna be lower on the next iteration
- The same intuition goes to W weights which are small and need to scale up in order to make a bigger advance horizontally

- <u>Final results (the green line) :</u>



- <u>What X-axis and Y-axis represents in reality? :</u>

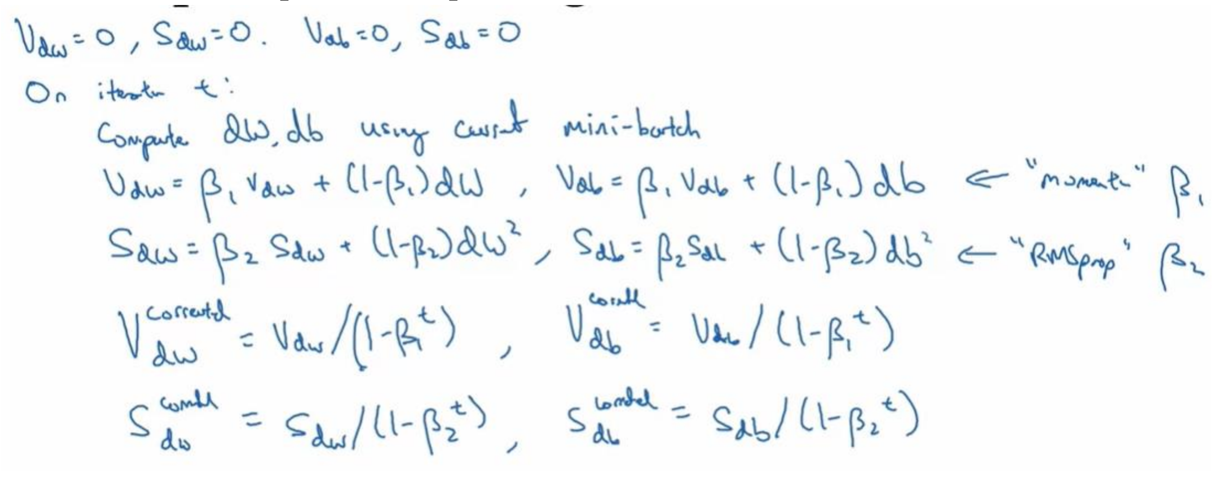


- Since we are handling multi-dimensional data and trying to represent the cost function in 2D only then both axis X and Y represents a set of weights , the X axis represents the set of weights (mixture of some Ws and some Bs) we need to scale up to have a lower value to the cost function while the Y's represents the other weights we need to scale down in order to avoid these useless oscillations (Sdb and Sdw are actually vectors and each element (weight wi/bi) will be affected according to its current weight , if it's too big that the Sdwi/Sdbi will scale it down and the opposite is true also .

XIX. Adam optimizer (Adaption moment optimizer) : a combination of momentum and RMSprop:

- The power of momentum, RMSprop and Adam optimizer that unlike the others who performs very well on some architectures and not on the others : these three works well in almost every neural architecture.

➢ <u>Computation process :</u>



$V_{dw}=0,\ S_{dw}=0.\quad V_{db}=0,\ S_{db}=0$

On iteration t:

Compute dW, db using current mini-batch

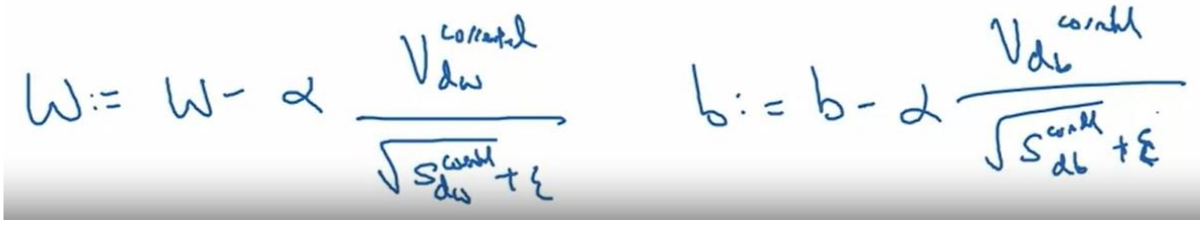
$V_{dw}=\beta_1 V_{dw}+(1-\beta_1)dW,\quad V_{db}=\beta_1 V_{db}+(1-\beta_1)db \leftarrow$ "moment" β_1

$S_{dw}=\beta_2 S_{dw}+(1-\beta_2)dW^2,\quad S_{db}=\beta_2 S_{db}+(1-\beta_2)db^2 \leftarrow$ "RMSprop" β_2

$V_{dw}^{corrected}=V_{dw}/(1-\beta_1^t),\quad V_{db}^{corrected}=V_{db}/(1-\beta_1^t)$

$S_{dw}^{corrected}=S_{dw}/(1-\beta_2^t),\quad S_{db}^{corrected}=S_{db}/(1-\beta_2^t)$

1. As we have already seen , we initialize Vdw , Sdw , Vdb and Vdw to 0
2. In each iteration :
   a. We will cmpute Vdw and Vdb using the hyper parameter beta1 (the same one used with momentum) and Sdb with Sdw using the hyper paremetrer beta2 (the same one used with RMS prop)
   b. We apply the bias correction to these 4 paremeters by dividing by (1-beta1^t) for Vdb and Vdw , and the same goes to Sdw nd Sdb with (1-beta2^t) to obtain the corrected versions



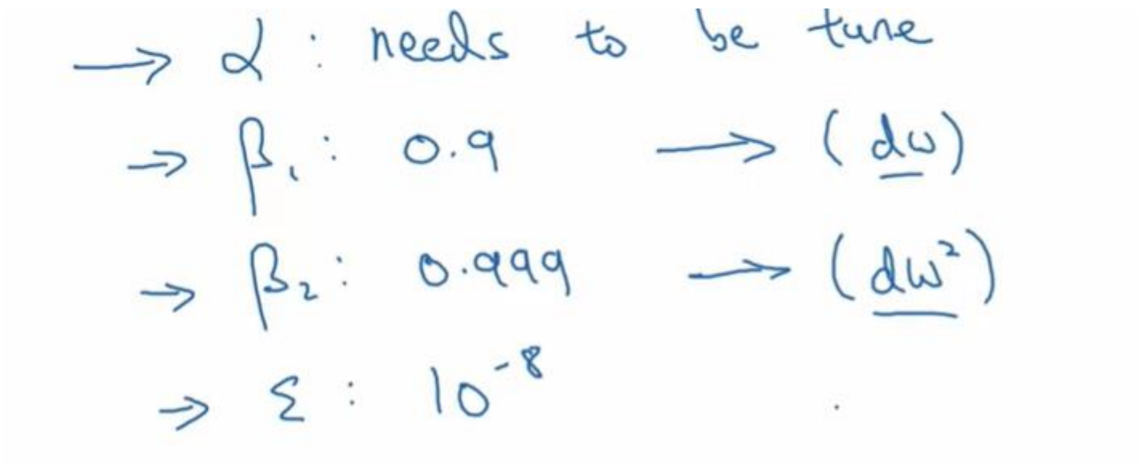
$$W := W-\alpha\frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}}+\varepsilon}\qquad b := b-\alpha\frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}}+\varepsilon}$$

c. We update the W and b weights using alpha (the learning rate) , Vdw corrected from momentum and Sdw from the RMS prop , and the same goes to b
   i. We added epsilon (which will have a very little value btw , like 10^-8) , just to assure that the denominator will not be very close to zero to guarantee that there will not be any weights exploding

➢ Hyperparameters choosing :

We saw that while working with Adam optimizer we will have 4 hyper parametres to deal with :

1. Alpha : the learning rate
2. Beta1 : the momentum's hyperparameter
3. Beta2 : the RMSprop's hyperparameter
4. Epsilon : to avoid the weights explosion

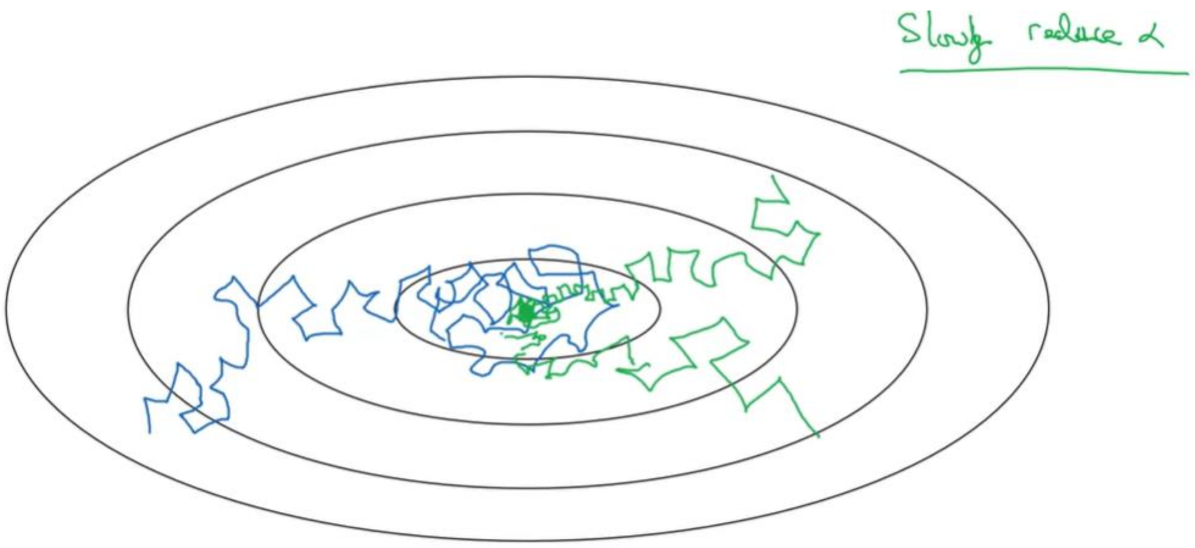

$\rightarrow \alpha$: needs to be tune
$\rightarrow \beta_1$: 0.9 $\longrightarrow$ ($\underline{dw}$)
$\rightarrow \beta_2$: 0.999 $\longrightarrow$ ($\underline{dw^2}$)
$\rightarrow \varepsilon$: 10^{-8}

- Generally , we will fix beta1 , beta2 and epsilon to these values (no need to change them In the most of cases) and we will try to finetune our architecture by trying multiple values for the learning rate , it's the only parameter that really needs to play with .

XX. Learning rate decay; because we need huge steps only in the first iterations

HH. Why we might need the learning rate decay :

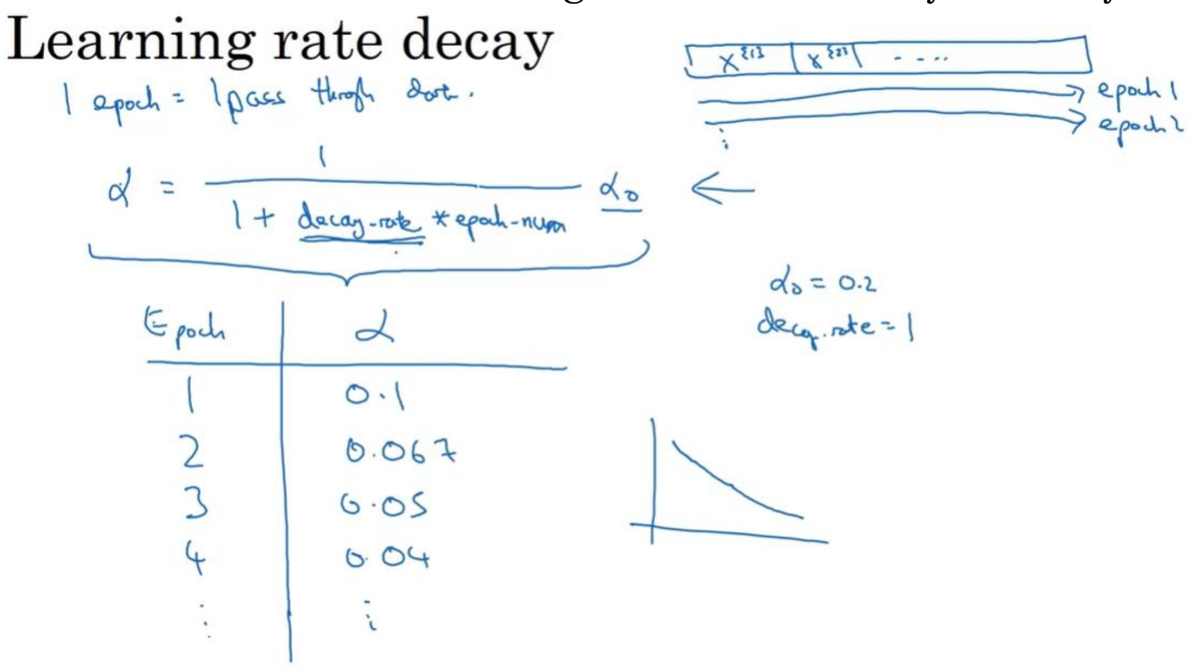
-



- the blue line shows what might happen to our gradient descent, we see that once its weights approach the optimal values, it becomes moving around it and being further from it

- these green lines show how it could be if we dynamically reduce the learning rate in the further iterations, since we've already approached the optimal value, we will not need ready to make big steps in order to have more accuracy, we will need instead to reduce the learning rate so we will not move far away from the optimal cost value and instead, we will just approach it slowly (but surely) .

➢ How can the learning rate be reduced dynimacally :



Learning rate decay

1 epoch = 1 pass through data.

$$\underbrace{\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0}$$

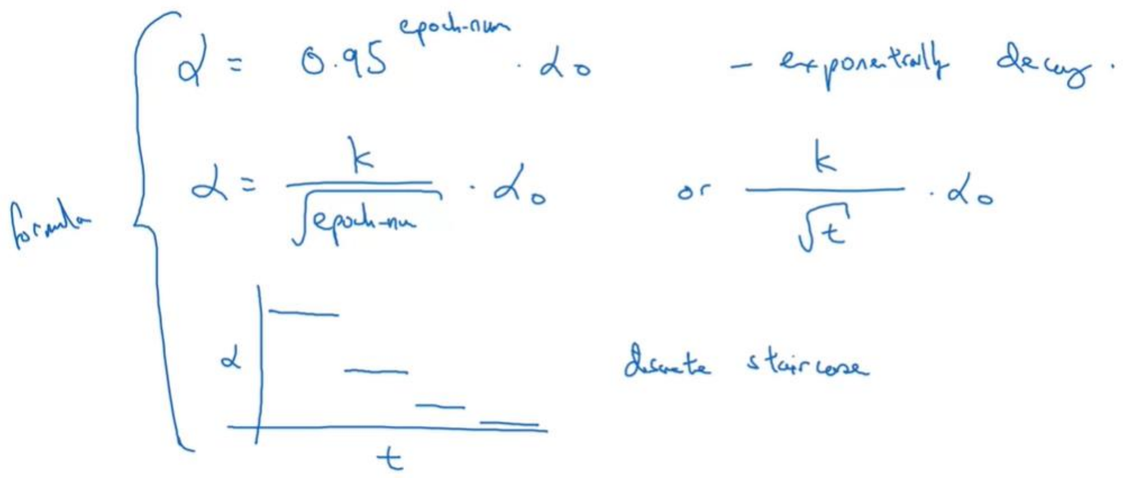
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	⋮

$\alpha_0 = 0.2$

decay.rate = 1

- The lmearning rate (alpha) will not be just a constant anymore, but it's now a formula with three parameters :
  - **Alpha0 :** represents the initial learning rate (at the epoch = 1)
  - **Epoch_num :** the number of epochs already happened , the formula will use the increase of this parameter to reduce dynamically the learning rate after each epoch passed
  - **Learning_rate :** a factor (usually between 1 and 0) connected to epoch_num whioch will tells the decay acceleration , using a big learning_rate will make the learning rate decrease faster

➢ Another ways to implement the learning rate decay :



formula

$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$ - exponentially decay.

$\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \cdot \alpha_0$

- The first method is called exponential decay
- The second method uses k as a hyper parameter constant and t which represents the batch number .

- The third method is called descent staircase : for each fixed number of epochs we will just devide the current learning rate by 2 (which will make the learning rate tend to zero)

➢ <u>Manual Decay learning rate :</u>

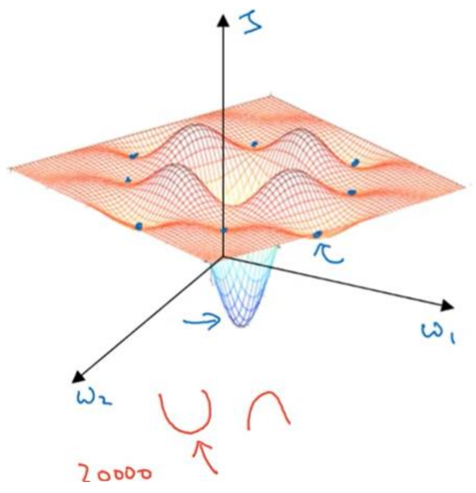
This works with model whioch takes many hours and days to be trained , we will just watch how the training is progressing hour-by-hour or day-by-day and we will go just lmike "oh ! the things starts to getting worse from this moment , so I should decay the learning rate at this instant"

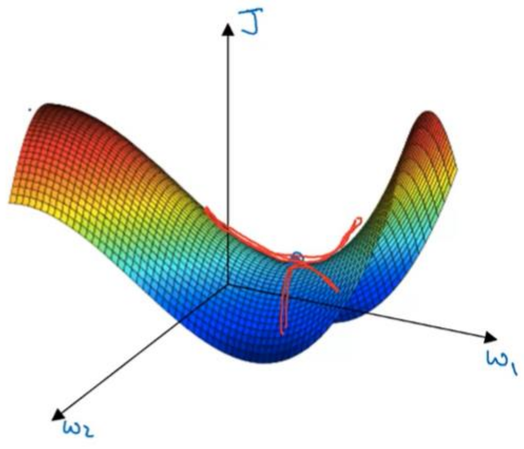
➢ <u>Learning rate decay should be a choice with a low priority:</u>

- We shouldn't go directly to implement the learning rate decay.

XXI. Local optima problem :

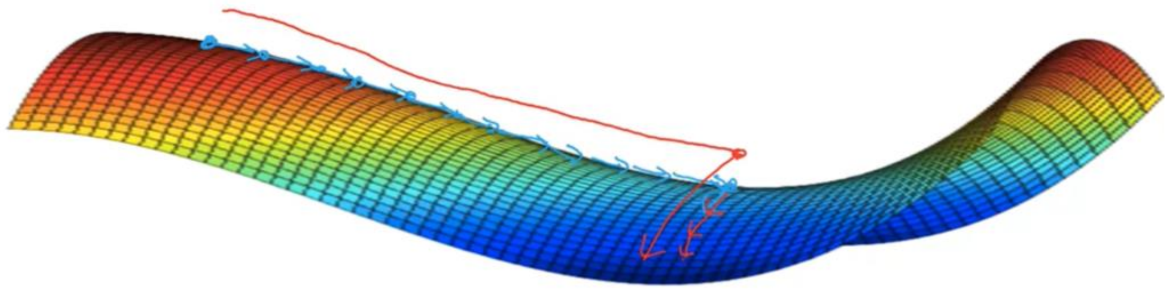
➢ <u>Local optima shape :</u>





- In reality , the local optima doesn't look often like the left chart that we usually see in 2D axis , unless all the parameters (thousands) have the same point to be in their minimal value , but most of the local optima in high-dimension plan looks like the right shape where some of them are in a local minimal value while the others are not , it's called "saddle" points and In deep learning with a very high-dimension space , a many epochs to train on and suitably learning rate : the local optima will not prevent our weights to pss throw it and reach the global optima

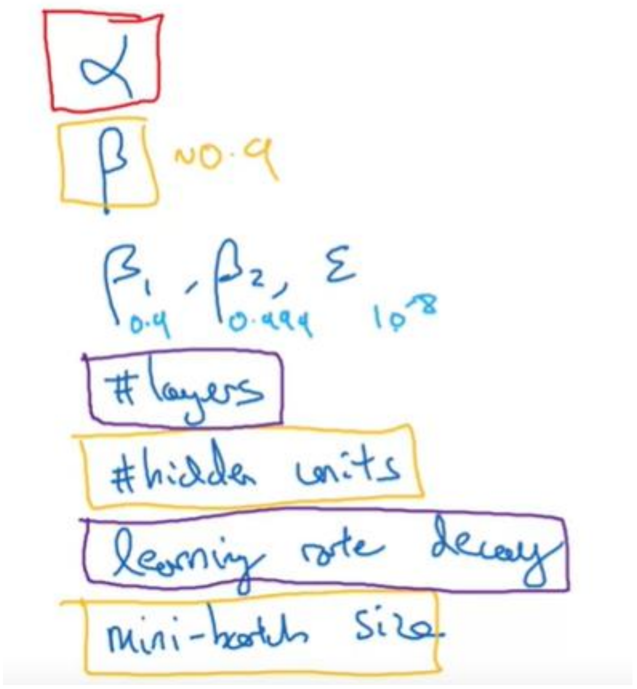
➢ Plateaus problem :



- The plateaus are regions where the weight derivatives are so close to zero for a long time , and that's will cause our gradient descent to learn very slowly (almost flat space => weight gradients tends to zero) , and that will cause our algorithm consume many epochs to surpass that plateaus (blue arrows)which describes a false global optima and having a very slow learning there before to getting some points where he will leave it (red arrows)
- **In a situation like these : momentum , RMSprop and Adam optimizer helps the gradient descent to accelerate its movements horizontally like we've already seen**

XXII. Hyperparameters tuning process:

II. We have multiple hyperparameters to deal with :



$\boxed{\alpha}$

$\boxed{\beta}$ ≈ 0.9

$\underset{0.9}{\beta_1}, \underset{0.999}{\beta_2}, \underset{10^{-8}}{\varepsilon}$

$\boxed{\#\text{layers}}$

$\boxed{\#\text{hidden units}}$

$\boxed{\text{learning rate decay}}$

$\boxed{\text{mini-batch size}}$

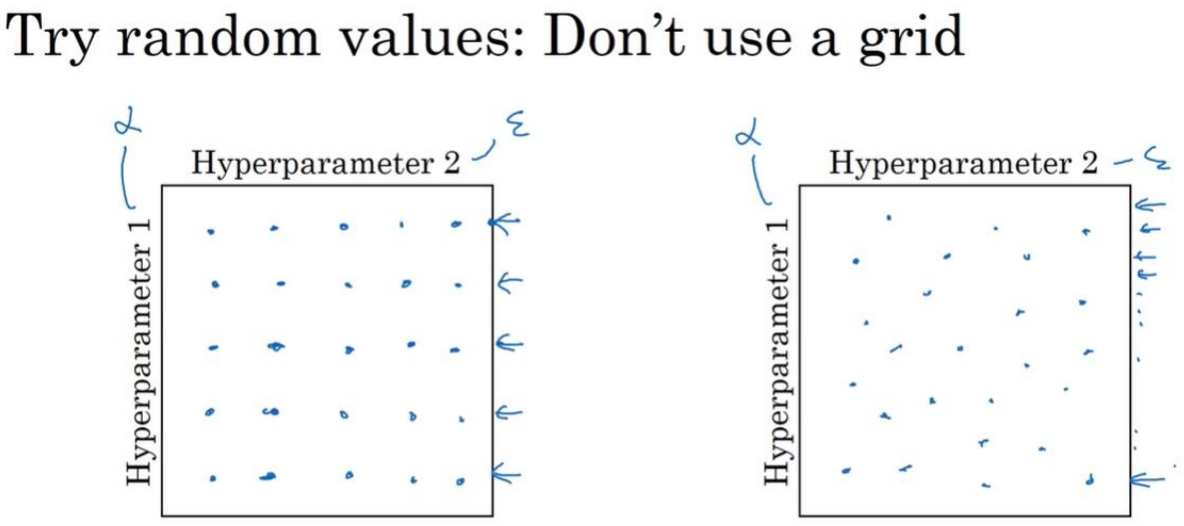
- So far, we have:
  - the learning rate (alpha),
  - the momentum hyperparameter (beta)
  - the Adam optimizer hyperparameters (beta1 for the momentum part and beta2 for RMSprop)
  - The number of layers
  - The number of hidden units inside each layer
  - The learning rate decay which defines the decay acceleration
  - The mini_batch size for the data splitting

JJ.<u>The most common hyperparameters to tune :</u>

- According to Andrew Ng:
  1. Red circles : Alpha, the learning rate is the most important hyperparameter to deal with
  2. Yellow circles : Momentum beta, the number of hidden layers and the learning rate decay are more or less important to tune
  3. Purple circles : number of layers and learning rate decay are the less important one s, even tho the number of layers can be a major factor of the model accuracy
  4. And The Adam optimize : it's almost always the same hyperparameters (0.9 for momentum , 0.999 for RMSprop and 10^-8 for epsilon)

KK. <u>Choosing hyperparameters values process :</u>

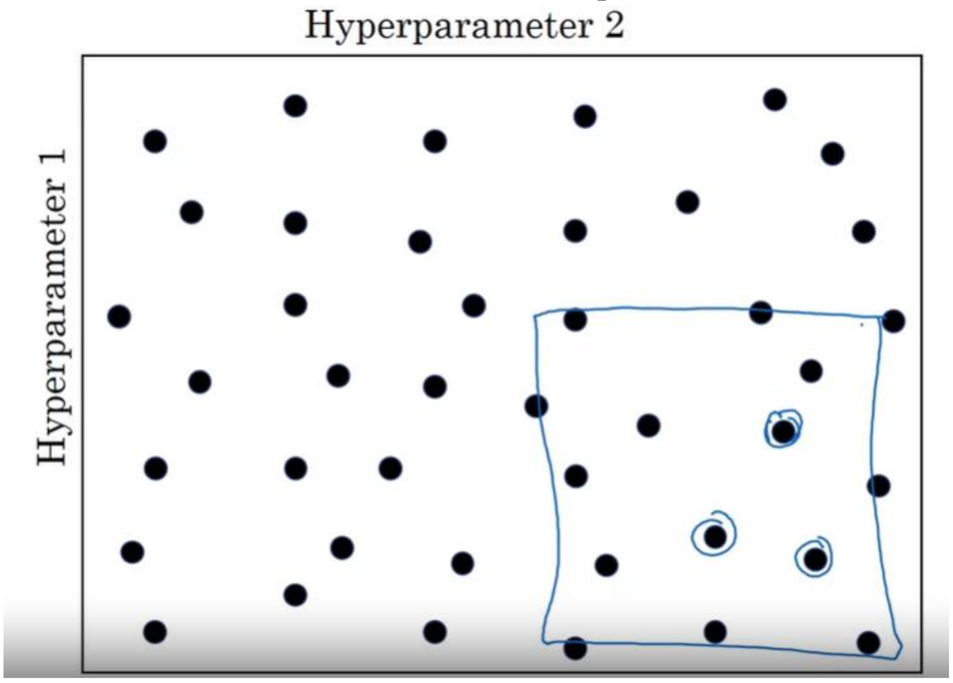
➢ <u>The process is simply choose a random values , BUT NOT IN A GRID SHAPE :</u>



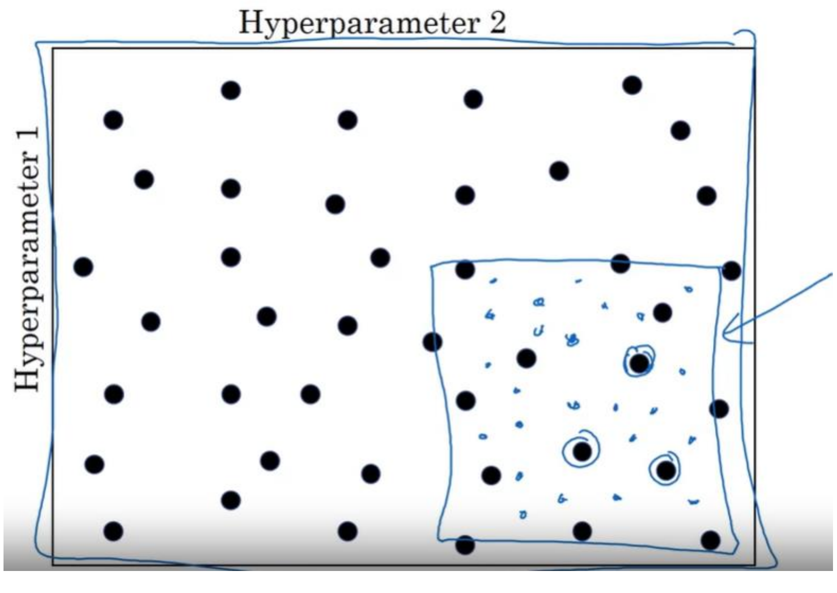
- Let's say for example we are trying to deal with only two hyperparameters (2D plan) : alpha and epsilon , If we try to do the random values (the right square) instead of the grid way (5x5) , we will be dealing with multiple values of alpha , not only 5, since the points coordinates in X axis haven't only five possibilities (or limited numbers generally)

- And that would be more crucial while dealing with multiple hyperparameters at the same time (multi-Dimensional plan)

➢ Coarse to fine technique :



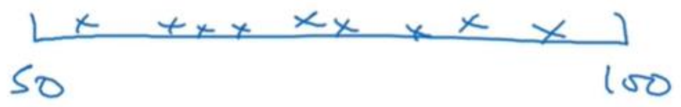
- Let's say after trying random values of the hyper parameters, we've found a set of points which gives an interesting results (the circled ones) , so we will try to repeat the random values process inside these mini-space which is more narrow



XXIII. Choosing the right scale to pick random hypermeters values matters !

LL. <u>For some hyperparameters, it's okay to do the random values picking at the linear scale:</u>

$\rightarrow n^{[l]} = 50, \ldots, 100$



$\rightarrow$ #layers $L: 2 - 4$

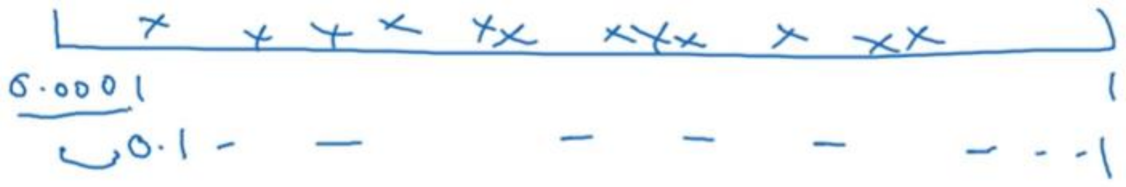
- While choosing the number of layers , or the number of units per layer , it would be okay to do the random picking in a linear scale

  - <u>What is linear scale :</u>

- In the linear scale , the probability of getting a value inside an interval is uniform
  - If the number of layers should be in [2,4] , the p(2) =p(3)=p(4)=33%

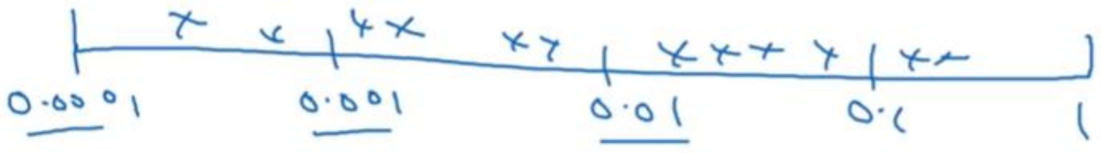
MM. <u>For the learning rate , the linear scale wouldn't be efficient :</u>

$\alpha = 0.0001, \ldots, 1$



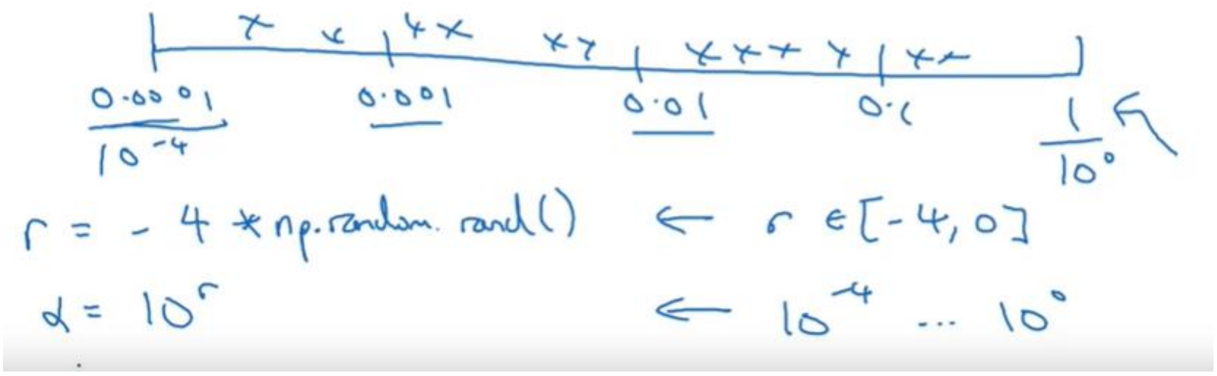
- If we suppose that learning rate must be between [0.0001 , 1], then we will get only 10% possibility to have a value between [0.0001 , 0.1] and the 90% other is for [0.1 , …., 1] which doesn't seem right for a hyperparameter like the learning rate

➢ The correct scale to choose on :



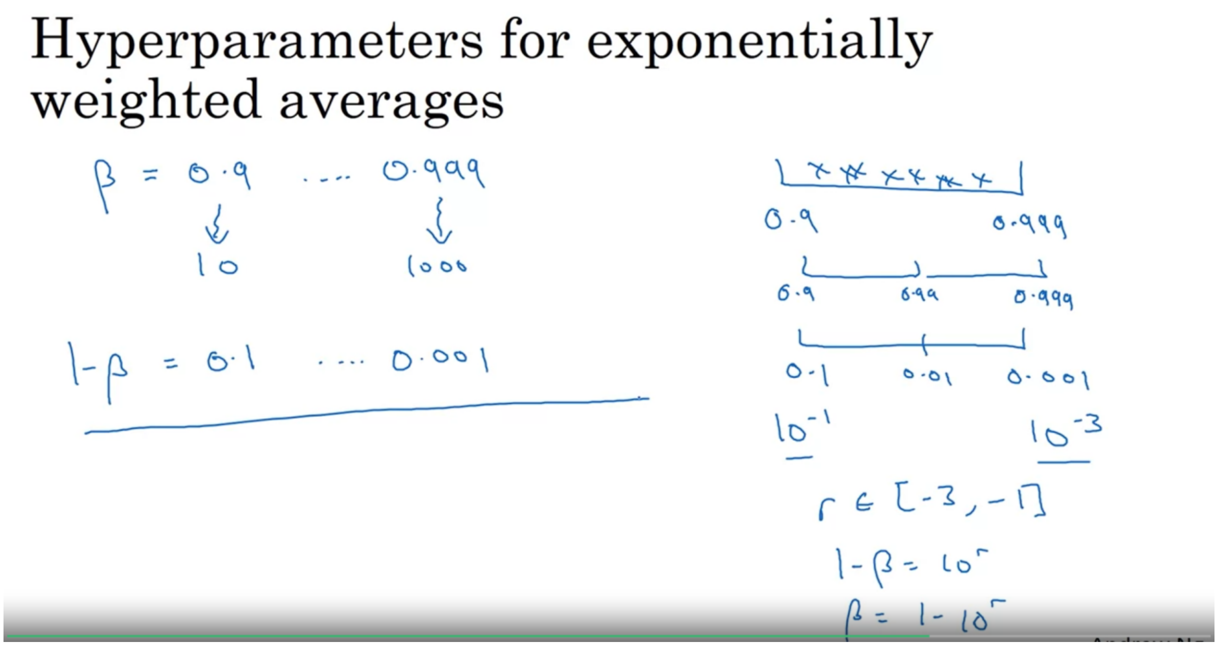
We would want to have a specific scale, where the probability of p([0.0001 , 0.001]=p([0.01 , 0.1]) = p([0.1 , 1]) , this called a **logarithmic scale**

➢ How to get the logarithmic scale in a specific interval



- In this case we wanna get learning rate in [0.0001, 0.1] which means [10^-4 , 10^0] .
  - Then alpha (learning rate) = 10^r where r in [-4 ,0] with a linear scale . and with that the possibility to have a value between 10^-4 and 10^-3 is equal to the one between 10^-3 and 10^-2 ..etc

NN. The logarithmic scale for beta momentum :

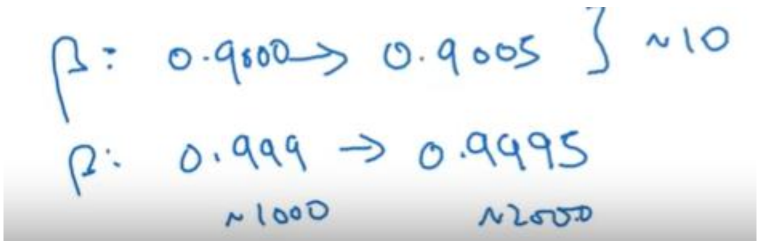

Hyperparameters for exponentially weighted averages
$\beta = 0.9 \ldots 0.999$
10
1000
$1-\beta = 0.1 \ldots 0.001$
0.9
0.999
0.9
0.99
0.999
0.1
0.01
0.001
10^{-1}
10^{-3}
$r \in [-3,-1]$
$1-\beta = 10^r$
$\beta = 1-10^r$

- The beta momentum should have value between [0.9 , 0.999] (although we will try to set it in default for 0.9 , unless we wanna tune it after)
- This hyper parameter should have a logarithmic scale for its random weights also (because the values close to 0.999 are very extreme values , like the 1 for learning rate)

➢ The process for 0.9 ....0.999 :

- The tricky way is to do the logarithmic scale for 1-beta which is between 0.1 ...... 0.001 (which is [10^-1 , 10^-3]) and for that we will have beta = 10^r where r is a random value between 1 and 3 .

XXIV. Why having a linear scale for some hyperparameters is a bad idea, why would we need to log scale ? :


$\beta: 0.9000 \rightarrow 0.9005$ } ~10
$\beta: 0.999 \rightarrow 0.9995$
~1000
~2000

- when beta is close to 1, the sensitivity of the results you get changes (becomes super sensitive) , even with very small changes to beta. So if beta goes from 0.9

to 0.9005 (we will add more 10 previous values for the weighted average) , it's no big deal, this is hardly any change in your results. But if beta goes from 0.999 to 0.9995, this will have a huge impact on exactly what your algorithm is doing (we will add more 1000 previous values to the weighted average) .

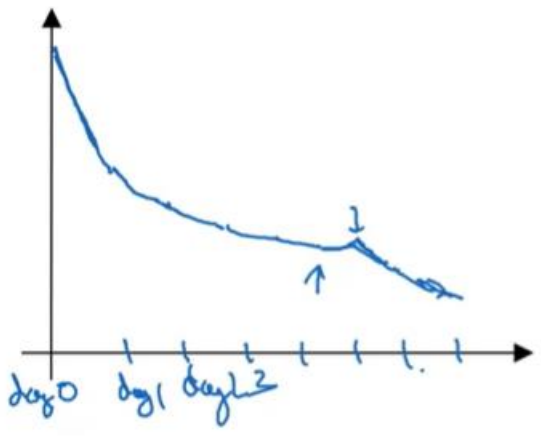
➢ <u>Linear scale isn't a very bad idea anyway :</u>

- Even for sensitive values like the learning rate or the momentum beta , dealing with linear scale isn't a devastative idea especially when doing the random weights with "Coarse" technique

XXV. Hyperparametres in practice : between focusing on single model or training multiple models at the same time

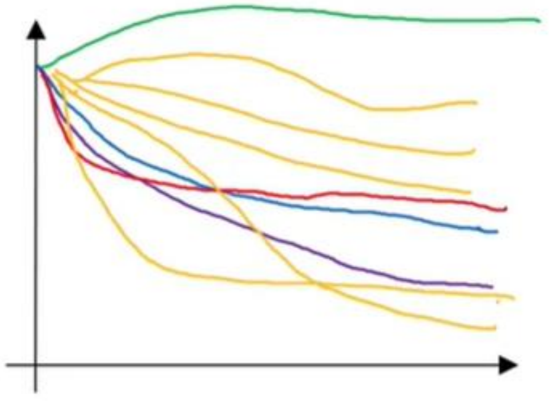
➢ <u>Babysitting single model (Panda way):</u>

- We go for this approach if we have a huge data to deal with but we are suffering from GPU/CPU lack of resources



- For every day (or a period we should fix) we observe the development of the cost function and we try eventually to change the learning rate (or even evolving the momentum) starting from a certain day in order to achieve better results after

➢ <u>Training many models in parallel (caviar way) :</u>

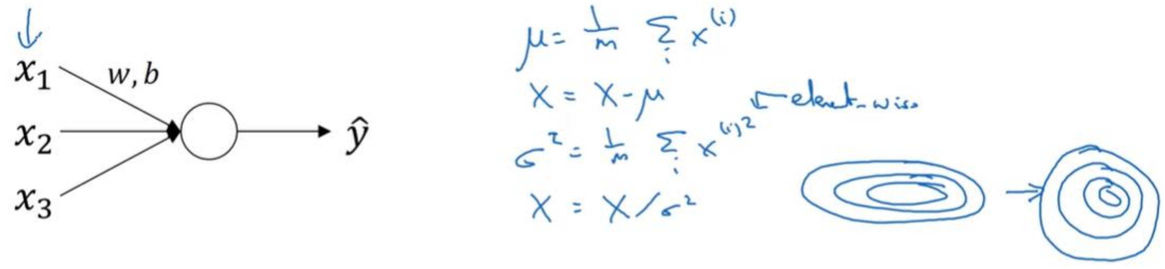


- We will launch many models (same architecture with different hyper parameters) at the same time and by the end of the execution of all the models we will extract the best one to choose .

XXVI. Batch normalization for quicker training:

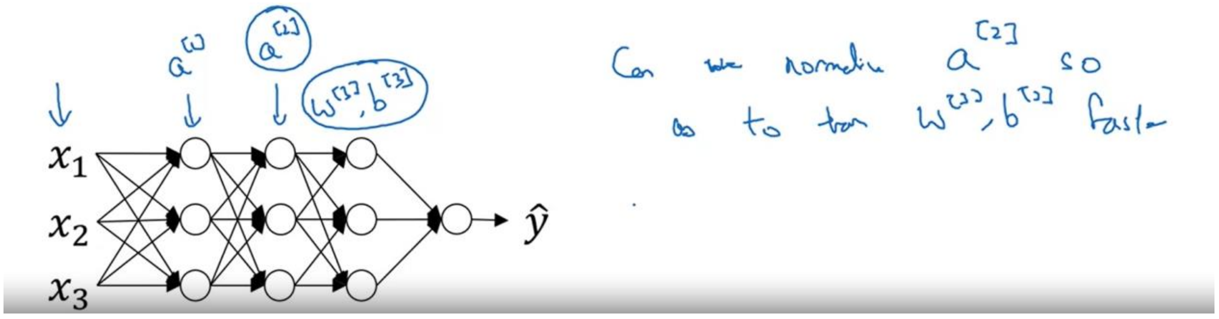
OO. <u>Already seen that the input normalization makes the NN training faster:</u>

Normalizing inputs to speed up learning


x_1
x_2
x_3
w, b
$\hat{y}$
$\mu = \frac{1}{m} \sum_i x^{(i)}$
$X = X - \mu$
element-wise
$\sigma^2 = \frac{1}{m} \sum_i x^{(i)2}$
$X = X / \sigma^2$

- By subtracting the inputs means and dividing by the standard deviation (sigma not sigma square) ; we've already said that will make the cost function contour becomes in a round way instead of the oval one which will make our gradient descent reach the optimal value quicker

PP. Batch normalization idea :



Can we normalize $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster

Since normalizing the inputs (first layer inputs to have W[1] , b[1]) values will make our model training faster, what about doing the same for the deeper layers , For example : normalizing the second layer inputs (a[2]) for the W[3] ,b[3] training in order to make it faster ? and doing the same normalization process in every intermediate layer.

- **Technically , we will normalize z[2] and not a[2] :**

- Andrew ng said that usually we normalize the z[l][1] instead of the a[l]

QQ. Batch Normalization implementation :

Given some intermediate values in NN $\underbrace{z^{(1)}, \ldots, z^{(m)}}_{z^{[l](i)}}$

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

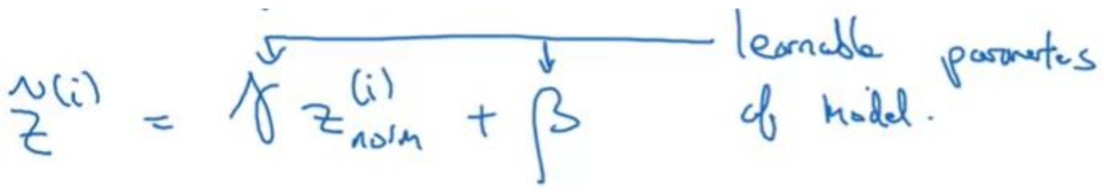
$$\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

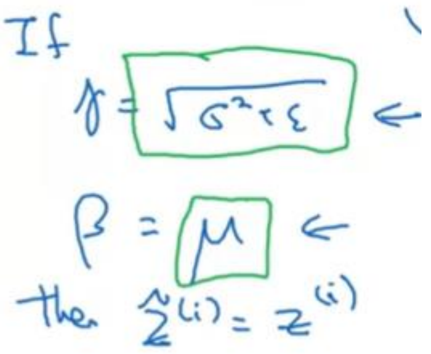
- SO as we said , we will do the normalization in Z instead of A of a certain layer [l]
  - We susbtract from each z(i) : the mean and we divided it by np.sqrt(sigma^2 + epsilon) , the epsilon is added in order to avoid the explosion (if sigma is close to 0)
- And by that we will get z(i) with mean=0 and variance=1

[1] z[l] is the activation input: a[l]= g(z[l]) = a[l](W1[l]X1[l] + W2[l]X[l] +........ b[l]) where g is the applied activation function

RR. The Z doesn't necessary have the same distribution :

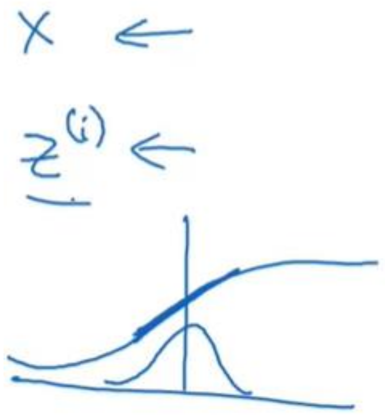

$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$
learnable parameters of model.

- Not very Z~(i) can have mean=1 and variance=0 , so instead of passing the normalized Z(i) to the activation function directly , we will pass Z~ (i) which is equaled to GAMMA*normalized Z(i) + BETA , where GAMMA and BETA are learnable parameters for each normalized z(i)


If
$\gamma = \sqrt{\sigma^2+\varepsilon}$
$\beta = \mu$
then $\tilde{z}^{(i)} = z^{(i)}$

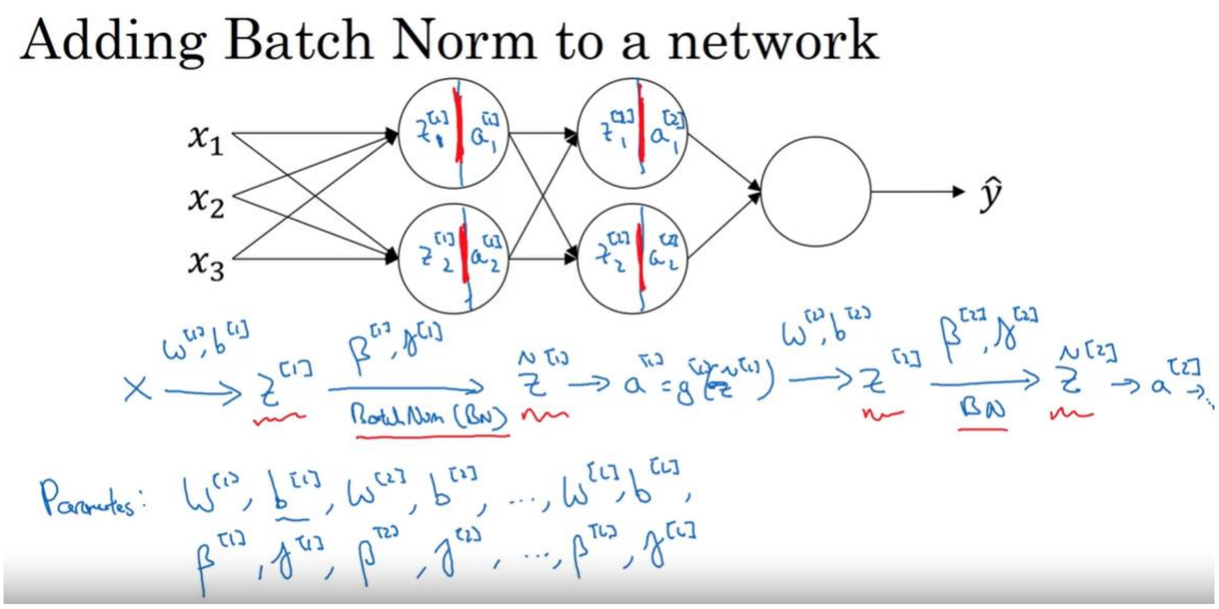
- Note that by having these specific values of GAMMA and BETA , we are inverting the normalization step and we are making Z~ (i)= z(i) , so by using these parameters we are more flexible to Z(i) distribution and we can even omit the normalization for certain z(i)

➢ The importance of GAMMA and BETA :



- If we are forcing the z(i) to have all the same standard distribution , we are just forcing there values to be mostly close to 0 and applying the sigmoid function to these z(i) will make most of them a(z) = 1 or closer to 1 , which isn't really the real case , every z(i) should have its own value independently from the others distribution , and that's why we add GAMMA and BETA in order to make the mean and the variance more dynamic .

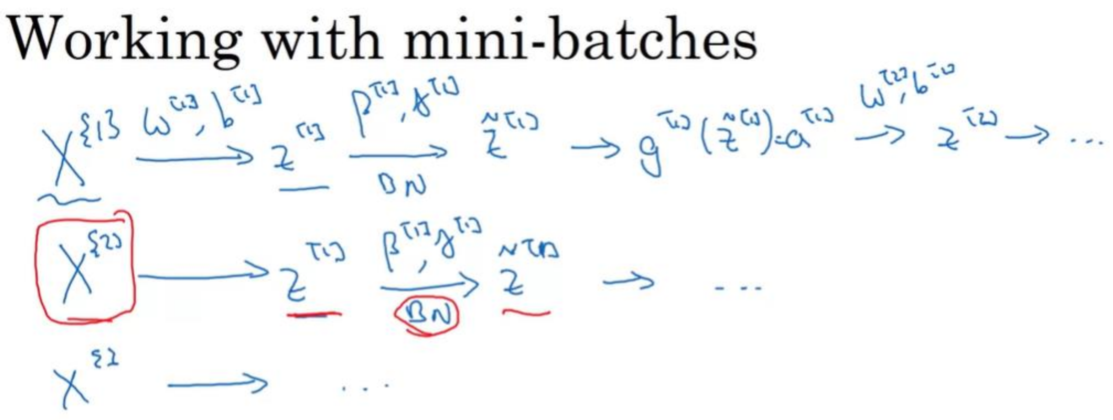
XXVII. Implementing Batch Normalization in the NN:


Adding Batch Norm to a network
x_1
x_2
x_3
$\hat{y}$
$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \to a^{[2]} \to \dots$
Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$

- In each neuron and with batch norm, and instead of calculating ai[l] directly , we will have an additional calculation for Z~(i) before having the layer outputs :
  - We start from the inputs X, we use them with W[1] and b[l] to calculate z[l] and then we apply the batch norm with GAMMA[l] and BETA[1] to have the z~[1] that we will use with activation function g to get the a[2] and so on ….
  - So additionally to the usual trainable parameters W[1] ,b[1] , ….. W[L], b[L] : we will have also GAMMA[1] ,BETA[1] , ….. GAMMA[L], BETA[L] (each layer will have its own GAMMA and BETA parameters to calculate the Z~[l] , **and actually we will not need to b[l] anymore , see the section : With batch normalization , there is no need to parameters b[l] :**)

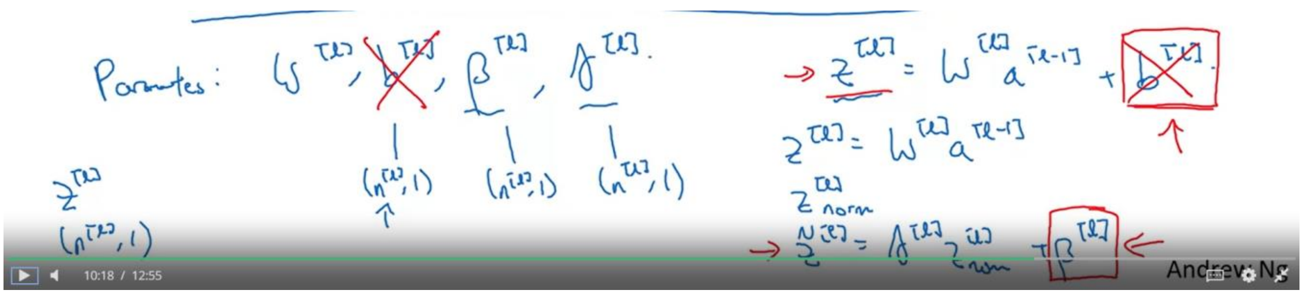
Important remark: with DL frameworks such as TensorFlow , we don't need to calculate Batch Norm manually but we will just added as an intermediate layer

SS. Batch Normalization with Mini-batches :


Working with mini-batches
$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow g^{[1]}(\tilde{z}^{[1]}) = a^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \rightarrow \dots$
$X^{\{2\}} \rightarrow z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow \dots$
$X^{\{3\}} \rightarrow \dots$

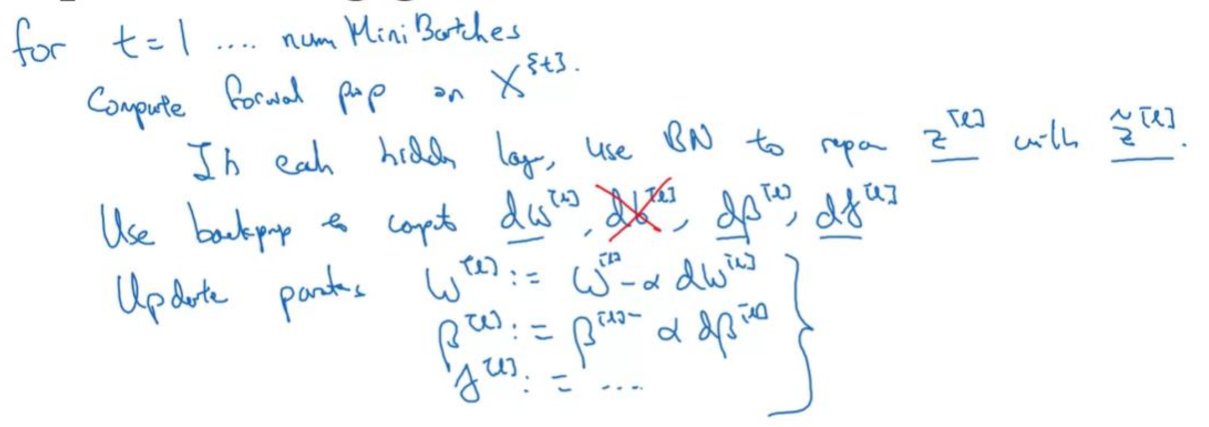
- We will do a batch normalization for each batch X{l} separately , which means every batch has its own mean and its own standard deviation defined by The inputs X of the same batch l

TT. With batch normalization, there is no need to parameters b[l]:


Parameters: $W^{[l]}, \cancel{b^{[l]}}, \beta^{[l]}, \gamma^{[l]}$.
$(n^{[l]}, 1)$ $(n^{[l]}, 1)$ $(n^{[l]}, 1)$
$z^{[l]}$ $(n^{[l]}, 1)$
$z^{[l]} = W^{[l]} a^{[l-1]} + \boxed{\cancel{b^{[l]}}}$
$z^{[l]} = W^{[l]} a^{[l-1]}$
$z^{[l]}_{norm}$
$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \boxed{\beta^{[l]}}$
10:18 / 12:55
Andrew Ng

- Since normalized Z(l) zeroes out the mean , there would be no need to the bias b[l] and it will be replaced by BETA[l] of the Z~[l] , so the Z[l] equation will just be W[l]a[l-1] (without +'b[l])

UU. Batch normalization algorithm in resume :



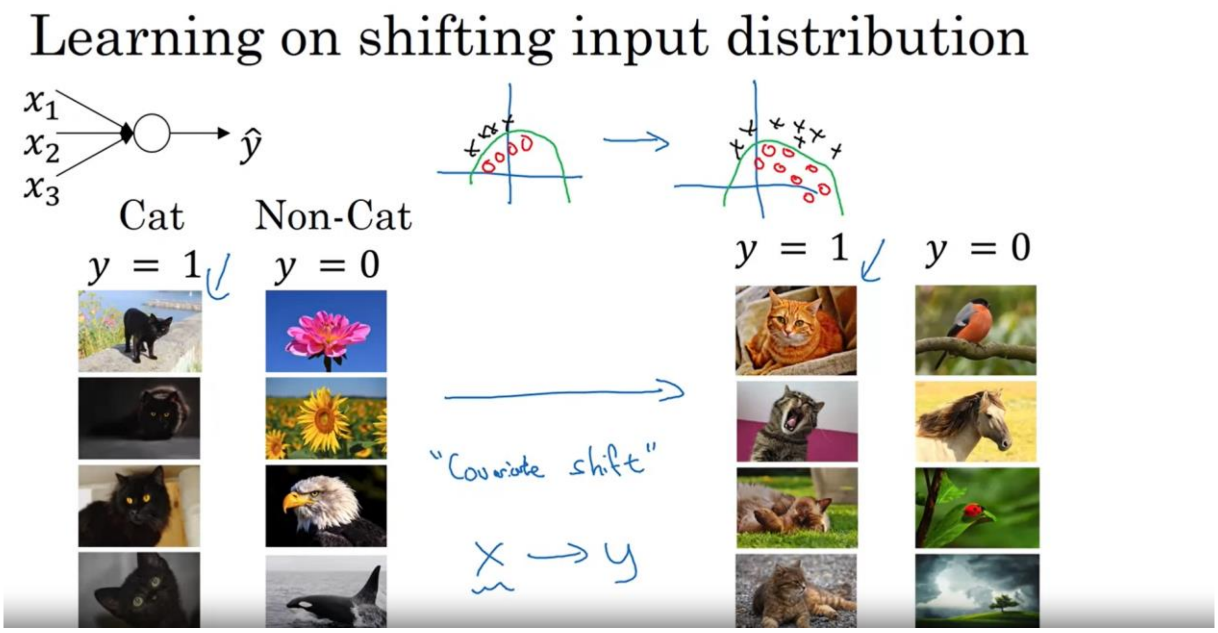
```
for t = 1 .... numMiniBatches
    Compute forward prop on $X^{\{t\}}$.
        In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$.
    Use backprop to compute $dW^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$
    Update params $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$
                  $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$
                  $\gamma^{[l]} := \dots$
```

- For each batch t :
  - We do the forward propagation for the batch X{t}
    - In each hidden layer we compute Z~[l] using Z[l] and we pass it to the activation function a[l]
    - We do the back propagation of dW[l] , dBETA[l] and dGAMMA[l] by updating the parameters W[l] , BETA[l] , GAMMA[l] using the gradients and the learning rate as always

<u>Important remark:</u> the Batch Normalization can be applied with momentum, RMS prop and Adam optimizer too !

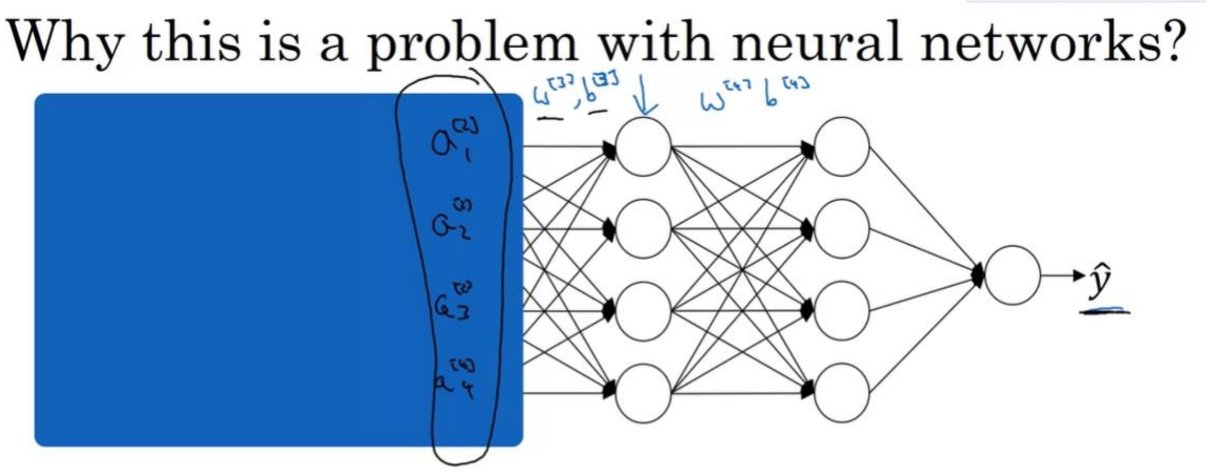
VV. <u>How Batch normalization exactly works:</u>

➢ <u>Beside making the training faster , it does the shifting of the input distrtibution !</u>

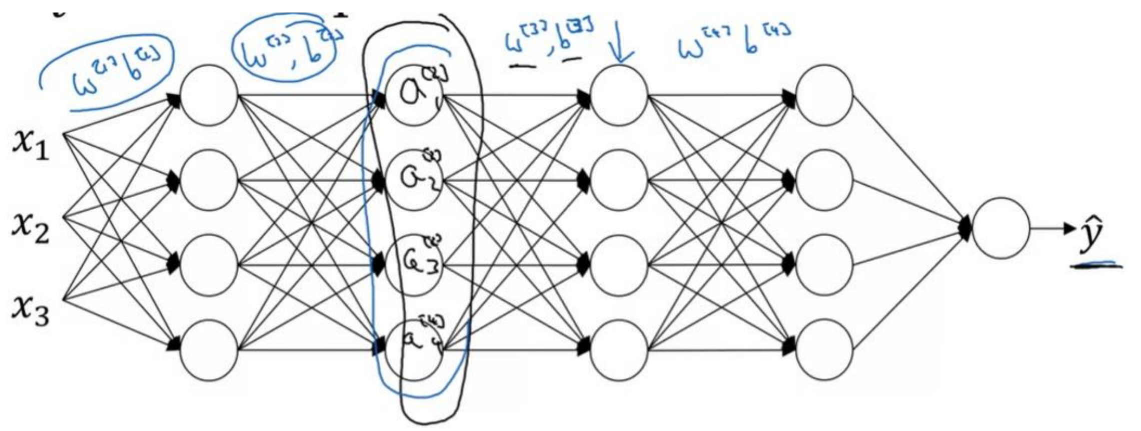


- Let’s say we’ve trained our model who will predict the cats photos , but our training set contains only black cats : our model might have a problem of generalization , he might not detect the cats with other colors correctly (our binary classification model would look like the 2D plan in the left .
- But thanks to batch norm, our model will have the ability to generalize the idea of being or not a cat , and that’s all thanks to “Covariate Shift”
    - The idea of Covariate shift is that if you've learned some X to Y mapping, and the distribution of X changes, then you might need to retrain your learning algorithm. And this is true even if the function, the ground true function, mapping from X to Y, remains unchanged, which it is in this example, because the ground true function is, is this picture a cat or not

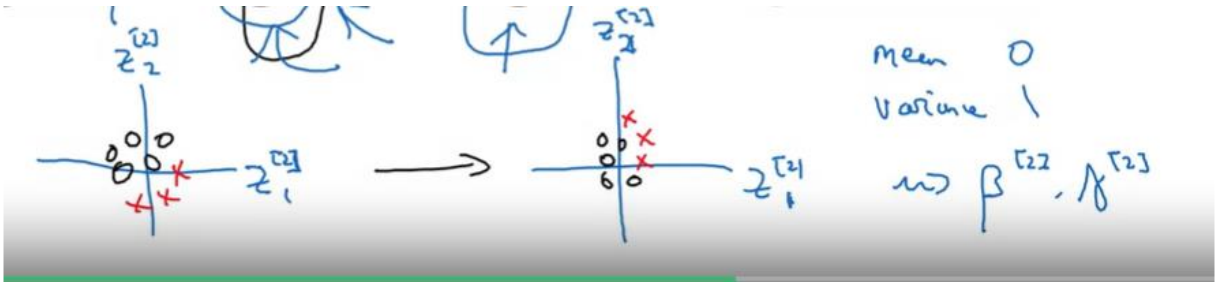
WW. <u>How can Batch Norm resists to the Covariate shift and do the generalization :</u>



- We will talk from the prespective of the third layer for example :
    - It's goal is to take inputs (which are the a[2] vector , they are like the input vector X from the prespective of the first layout) and try to train the W[3] and b[3] weights in order to map these inputs to the y^ value which is the output and the expected class

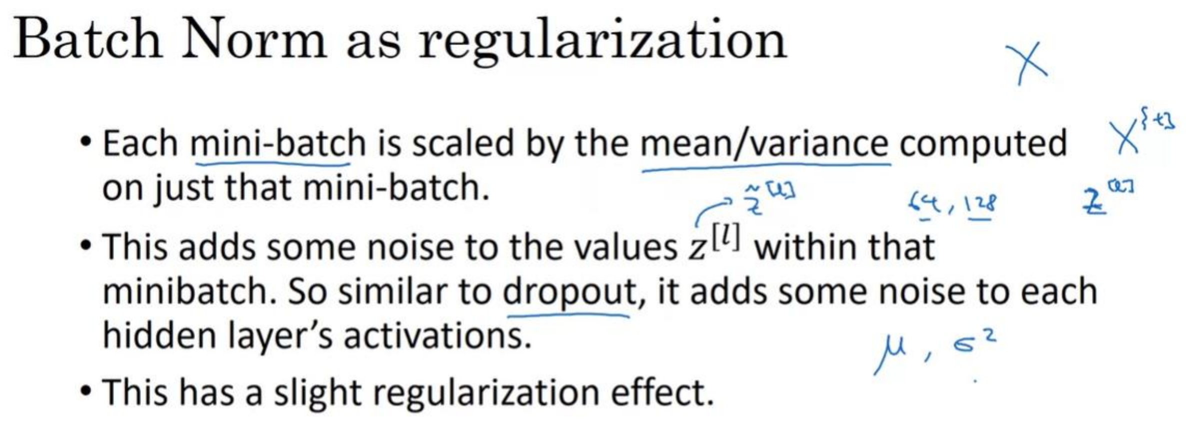


- The problem is the input vector of this third layer (the a[2]) are changing by the changes of W[1],b[1] and W[2] , b[2] ....so it would be a problem of generalization of the a[2] values , **and here where it comes the utility of Batch normalization : it reduces the amount of sihifting of the distribution of the a[2] values (and wherever the Batch Norm is applied)** , and each layer become less dependent to the previous layers inputs/outputs since the values will be normalized before getting trated by the targeted layer



- and this is by normalization the values of Z[2] vector and forcing the to have mean=0 and variance=1 (or related to BETA[2] and GAMMA[2]) and went from the graph on the left to the graph on the right !

  - <u>Batch norm is a slight regularizer too !</u>



Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

- Since while doing the mini-batch gradient descent : the mean and variance distribution of a layer input is totally dependent to the particular batch content : this as we already know made the mean and the variance in each batch has a different value (noisy) and that noise has a similar effect to the dropout which assures the independency between the layouts , but this regularization effect is really implicit and has a tiny impact compared the the other regularization techniques .

<u>**Important remark:**</u> By increasing the batch size : we are reducing the noise and the difference between the mean and variance distribution of each batch which will eliminate this regularization effect

XX. Batch normalization at the test time :

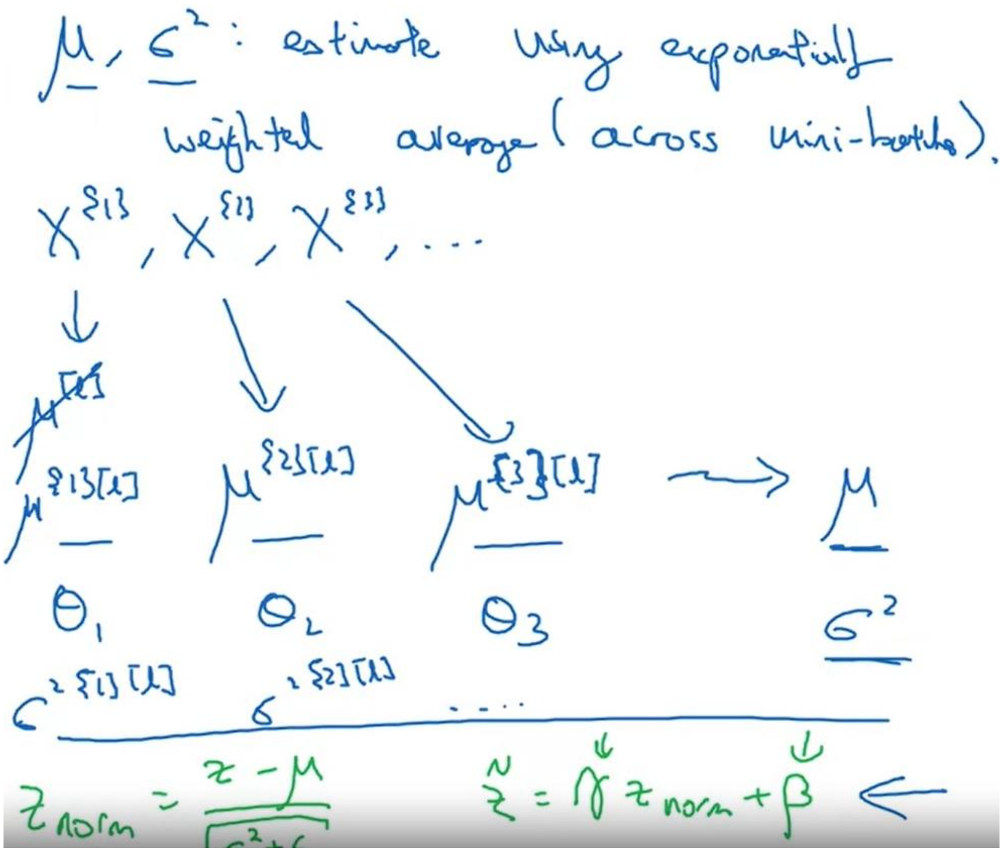
➢ <u>At the prediction phase , the process goes in a different way</u>

$$\boxed{\begin{aligned} \rightarrow\quad \underline{\mu} &= \frac{1}{\boxed{m}}\sum_i z^{(i)} \\ \rightarrow\quad \underline{\sigma}^2 &= \frac{1}{m}\sum_i \left(z^{(i)}-\mu\right)^2 \end{aligned}}$$

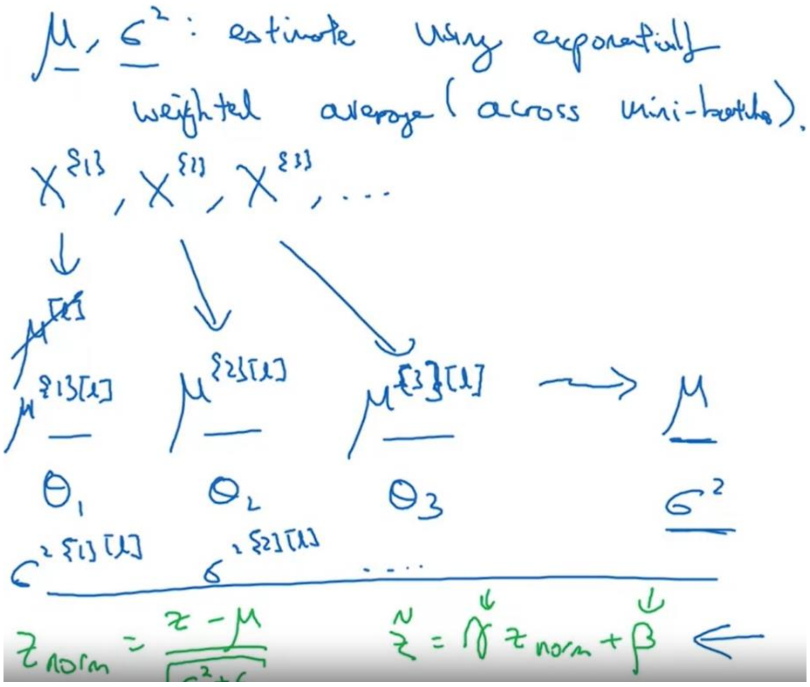
$$\rightarrow\ \boxed{z^{(i)}_{\text{norm}} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\underline{\varepsilon}}}}\ \leftarrow$$

$$\rightarrow\ \boxed{\tilde{z}^{(i)} = \gamma \underline{z^{(i)}_{\text{norm}}} + \beta}$$

- While training , we were calculating the mean and the variance of a layer inputs Z using the current batch {l} , but in the prediction phase , we will do the computations for a single value X , so we will not have a mean or variance


μ, σ²: estimate using exponentially weighted average (across mini-batches).
X{1}, X{2}, X{3}, ...
μ{1}[l]
μ{2}[l]
μ{3}[l]
μ
θ1
θ2
θ3
σ²
σ²{1}[l]
σ²{2}[l]
....
z_norm = (z − μ) / √(σ² + ε)
z̃ = γ z_norm + β

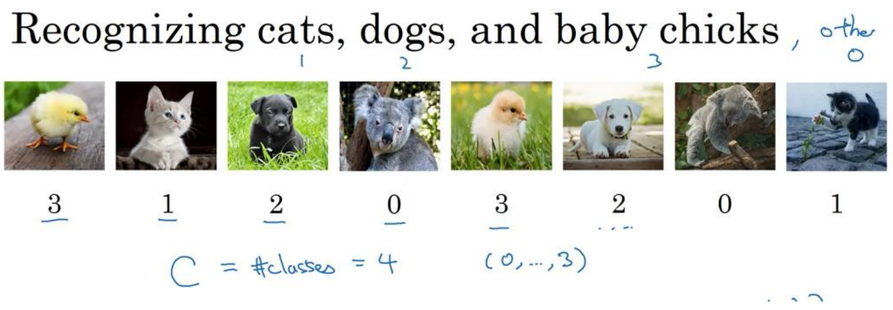
- <u>Moving average technique (Exponentially weighted) is the solution:</u>


μ, σ²: estimate using exponentially weighted average (across mini-batches).
X{1}, X{2}, X{3}, ...
μ{1}[l]
μ{2}[l]
μ{3}[l]
μ
θ1
θ2
θ3
σ²
σ²{1}[l]
σ²{2}[l]
....
z_norm = (z − μ) / √(σ² + ε)
z̃ = γ z_norm + β

- For each layer l , the mean and the variance of each batch would be the Theta{i} (i is the batch number) and the mean of the coming input to predict is like calculating V(t+1) where t is the total batch numbers and the same goes to the variance (sigma) , and then everything will be calculated by the inputs (Z) and the trainable parameters Gamma[l] and BETA[l]

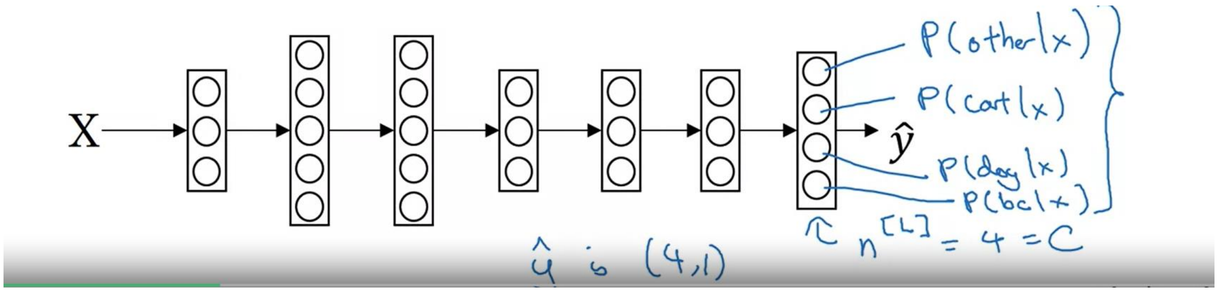
XXVIII. Multi classification with Softmax : the mathematical theory behind

YY. Reminder of Multi-Cassification principle :



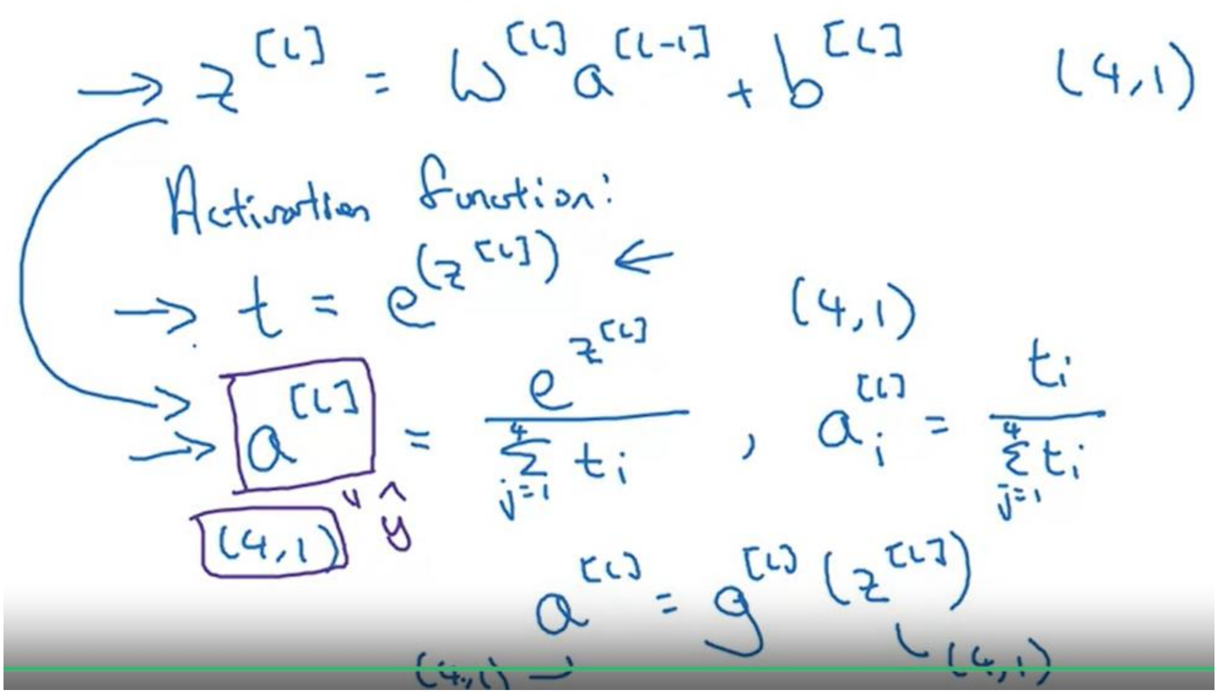
- Instead of having a binary classification (sogmoid value between 0 and 1) , the multi classification task assign to each classe an index (1 for the first class , 2 for the second class ...etc)
- Usually, we have an additional class denoted by"0" which describes "None of the possible classes" which describes the data that doesn't belong to any class
- We denote the number of classes by C , and the possible output values goes from 0 to C-1

➢ The NN multi-classification architecture :



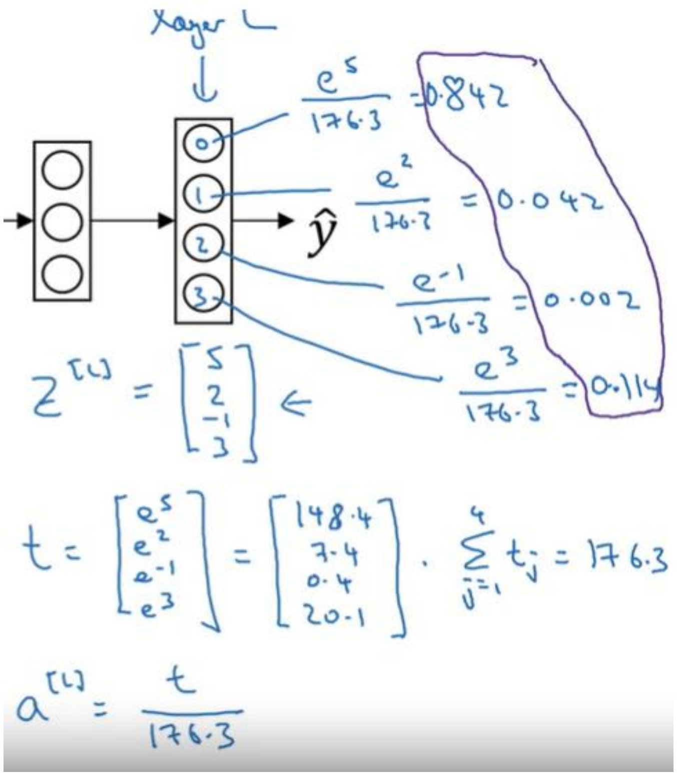
- The output layer is no longer a single value , the y^ is now a vector of values , its length is equal to the classes number C
- Each value represents the possibility of X to belong to a class (P(class_name[i] | X)) : their sum is 1
- The NN model will give to the class that it predicts : a higher value of probability , and this is all thanks to the softmax activation function .

ZZ. The sofmax activation function computation :


$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$
(4,1)
Activation function:
$t = e^{(z^{[L]})}$
(4,1)
$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i}$
$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$
(4,1)
$\hat{y}$
$a^{[L]} = g^{[L]}(z^{[L]})$
(4,1)
(4,1)

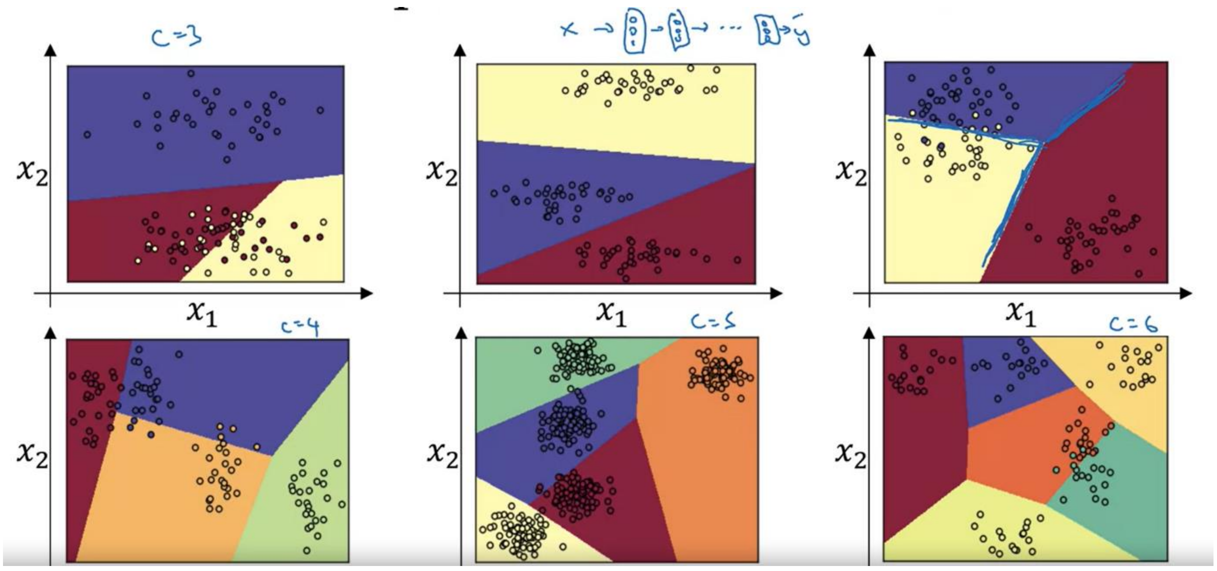
- The softmax activation function and unlike the others , it will take a vector of values instead of a single value using the vector Z[l] (which has the shape (class numbers , 1))
  - The process starts by calculating an intermediate variable t (with shape (class numbers , 1) always)
  - Then, the activation value a[l] will be equal to that vector but normalized to 1 (so the sum of the values will be equal to 1) by dividing each vector component ai[l] by the sum of the components.

➢ Concrete example of the Softmax computations:


Layer L
$\hat{y}$
$\frac{e^5}{176.3} = 0.842$
$\frac{e^2}{176.3} = 0.042$
$\frac{e^{-1}}{176.3} = 0.002$
$\frac{e^3}{176.3} = 0.114$
$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$
$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$. $\sum_{j=1}^{4} t_j = 176.3$
$a^{[L]} = \frac{t}{176.3}$

- Nothing to say , everything is explained in the picture .

➢ How The multi classification separation looks like :

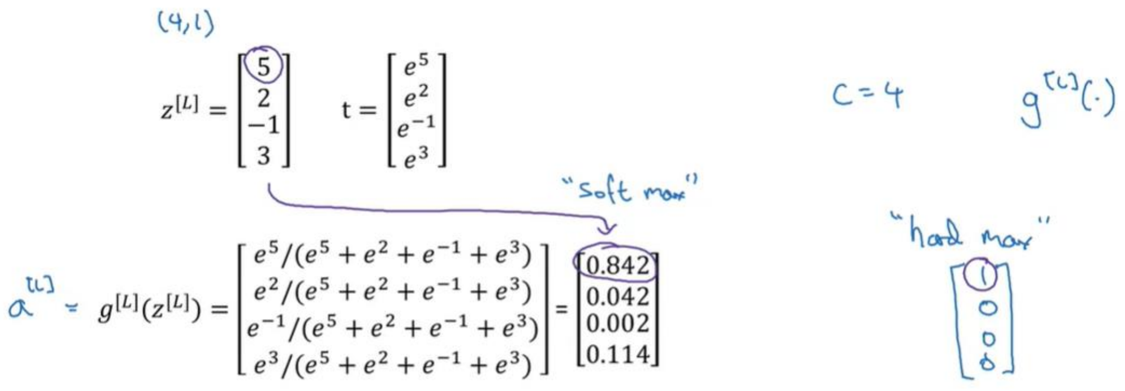


➢ Softmax regression is a generalization of logistic regression:

Multi classification is a generalization of logistic regression with sort of linear decision boundaries, If C=2 then the softmax regression is equivalent to logistic regression .

AAA. Training Softmax Classifier:

➢ Softmax vs Hardmax :

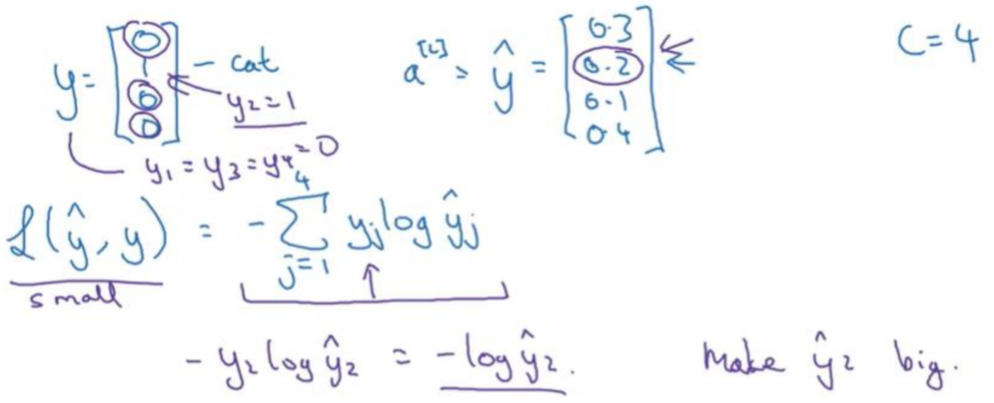


$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- The name softmax comes from contrasting it to what's called a hard max , which will look at the elements of Z and just put a 1 in the position of the biggest element of Z and then 0s everywhere else. And so, this is a very hard max where the biggest element gets a output of 1 and everything else gets an output of 0. Whereas in contrast, a softmax is a gentler mapping from Z to these probabilities.

➢ Defining Loss function L for the softmax classifier



$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad y_2 = 1 \quad y_1 = y_3 = y_4 = 0 \qquad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

$$\underbrace{\mathcal{L}(\hat{y}, y)}_{\text{small}} = \underbrace{-\sum_{j=1}^{4} y_j \log \hat{y}_j}$$

$$-y_2 \log \hat{y}_2 = \underline{-\log \hat{y}_2}.$$

- The y is the expected output which is a "hardmax" vector : in the correct class we will have '1' , and the y^ is the softmax vector obtained from our trained model
  - For each input , we define its loss function with L(y , y^) = - Sum(yj *log(yj^))
    - Since the y vector contains a single not-zero value so the real formula of the lost function is :

-y[correct_class]*log(y^[correct_class]) = -log(y^[correct_class])

- So the loss function of an input depends only in the y^[correct_class] (the probability value that our model gives to the correct class)
  - If y^[correct_class] = 1 then , Loss=0
  - If y^[correct_class] close to 0 then the loss goes to the infinity

➢ <u>Defining the global cost function J for the softmax classifier</u>

$$J(w^{[1]}, b^{[1]}, \dots) = \underline{\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$$

- It's simply the average of the losses of each input

➢ <u>How Y and Y^ would look like for the whole training data</u>

$$Y = [y^{(1)} \; y^{(2)} \dots, y^{(m)}] \qquad \hat{Y} = [\hat{y}^{(1)}, \dots, \hat{y}^{(m)}]$$

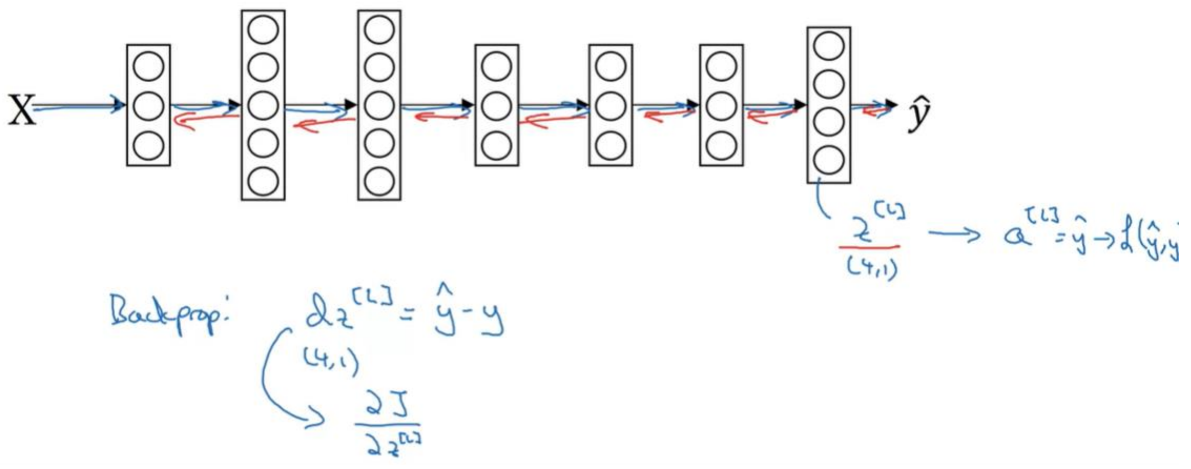
$$= \begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix} \qquad = \begin{bmatrix} 0.3 & \\ 0.2 & \\ 0.1 & \dots \\ 0.4 & \end{bmatrix}$$

$$(4, m) \qquad\qquad (4, m)$$

7:32 / 10:07

- <u>**Y :** It's a hardmax matrix , each row is a training data row where we give 1 to the correct class</u>
- <u>**Y^ :** It's a softmax matrix computed with softmax activation function , each row is a prediction of the output class row where the sum is ; 1 and the max value is given to the predicted class</u>

BBB. Back propagation in the Softmax classifier :


X
$\hat{y}$
$z^{[L]}$
(4,1)
$a^{[L]} = \hat{y} \rightarrow \mathcal{L}(\hat{y}, y$
Backprop:
$dz^{[L]} = \hat{y} - y$
(4,1)
$\frac{\partial J}{\partial z^{[L]}}$

- The gradient (partial derivative of Z) of each Z is simply : y-y^ (we can find that by ourselves by calculating the partial derivative of the SoftMax function)