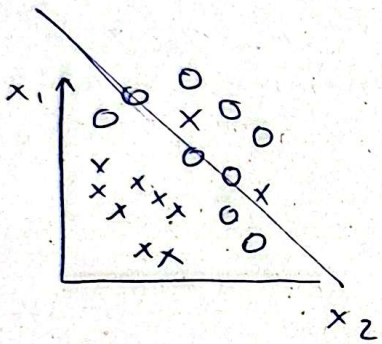
Course 2

Improving Deep Neural Networks:- (Week 1)

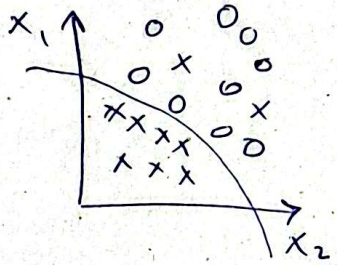
Hyper parameter:- it's the parameter that affects the resulting other parameters → α.

Bias and Variance



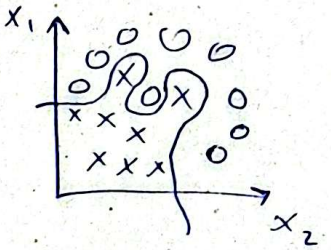
high bias

in this case there is high bias because the model is underfitting the data because the solution is biased thanks to our assumption that a linear desicion boundry would fit.



Just right

the desicion boundry is not underfitting nor over fitting the data so it actually has good bias and good variance.



high variance

here the decesion boundry is overfitting the data resulting in a high variance

Train set →	1 %	15 %	0.05 %
Dev set →	11 %	16 %	1 %
	↓	↓	
in high dimensional plots you can't visualize the D.B	~~high bias~~ high variance over fitting	high bias underfitting	just right

Dev set 30 % → high variance.

Train set 15 % → high bias.

Basic Recipe for ML

if you have

High bias? (training data ~~overfitting~~ underfitting) —try→ Bigger network (more hidden units) (more hidden layers) Train the network longer *(Change the Arch)

These options can fix the high bias problem.

High variance? (Dev set performance?) ——→ More Data Regularization *(Change the Arch)

if the train error is low but the Dev error is high then you're overfitting the training data

High variance

---

* in Deep Learning there is not much talk about the Bias Variance trade off because you don't have to affect one by changing the other Like ML

Regularization

$[w \in \mathbb{R}^{n_x}, b \in \mathbb{R}]$

Logistic regression :-

$$\min_{w,b} J(w,b) = \mathcal{L}(\hat{y}, y)$$

(regularization parameter)

$$= \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

↳ L2 regularization

↳ norm of w^2

L1 regularization → $\frac{\lambda}{2m} \sum_{j=1}^{n_x} |w| = \frac{\lambda}{2m} \|w\|_1$

For a neural network

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|^2$$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

↳ This is called the "Frobenius norm"

$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

This is also called weight decay ↳ regularization term

$$w^{[l]} := w^{[l]} - \alpha \, dw^{[l]}$$

↳ regularized.

$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha (dw^{[l]})$$

* Drop out Regularization :-

→ in drop out regularization we drop some of the hidden units at random, drop out has multiple implementation methods.

* Inverted dropout :-

for Layer = 3. → l = 3 , keep_prob = 0.8

d3 → dropout vector

d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep-prop

→ each activation like (a3) has a value ranging from 0 to 1 so if a3 < keep_prob and keep_prob is 0.8 that means that if a is ~~bigger~~ less than 0.8 set the d3 which is the dropout vector to 1 else set to 0

a3 = [0.5 0.9 0.4 0.3] , keep_prob = 0.7

d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob

print(d3)

↳ [1 0 1 1]

→ we used random.rand because d3 don't actually have a3 but in implementation

* to use the dropout vector :-

d3 = a3 < keep_prob

a3 = np.multiply(a3, d3)

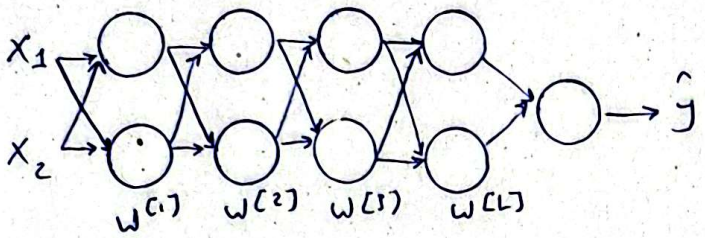
print [a3]

↳ [0.5 0 0.4 0.3]

a3 = a3 / keep_prob

↳ to maintain the value of a which we reduced by eliminating one of the hidden units values.

Vanishing and exploding gradients.



$g(z) = z \longrightarrow$ Linear activation function.

$z^{[1]} = W \cdot X = y$

$\hat{y} = W^{[1]} W^{[2]} \ldots W^{[L]} \cdot X$

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

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

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

$z^{[2]} = W^{[2]} \cdot z^{[1]}$

$z^{[L]} = \hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \ldots W^{[3]} W^{[2]} W^{[1]} X$

$z^{[1]}$ $z^{[2]}$ $z^{[3]}$

Let's suppose that

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

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

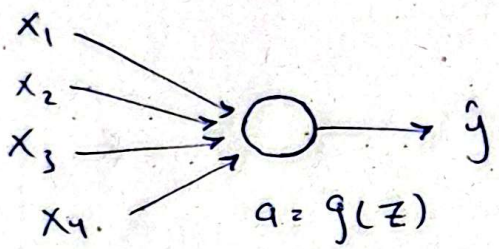
↓ this weight matrix is usually a bit larger than the rest.

---

since this equation will result in multiplying X with 1.5 (L-1) times then the value of $\hat{y}$ will likely explode and ; if we do the same setting the value of $W^{[l]}$ to $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ then the value of $\hat{y}$ will decrease until its vanishes.

* a partial solution to the problem of vanishing and exploding gradients is:

- Careful choice of the random initialization parameters:



* $z = w_1 x_1 + w_2 x_2 + \dots w_n x_n + b \to 0$

→ to solve the problem of vanishing and exploding gradients for a nueralnetwork with l number of layers

* if l is large you want to minimize $w^{[l]}$

→ because the vanishing gradient can change as the number of layers get bigger.

$$z = \sum_{i=1}^{l} w_i x_i$$

→ a way to deal with this problem is setting the variance of (w_i) to be $\frac{2}{n}$, this helps with the gradients stablizing and not growing or deminishing.

how to set $Var(w_i) = \frac{2}{n}$?

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

setting the $var(w_i)$ to $\frac{2}{n}$ is required when dealing with relu because Relu only activates half the inputs.

* this only helps to stabilize z and reduce the Vanishing and exploding gradient's problem.

* why did we devide by $n^{[l-1]}$ when setting the variance to be $\frac{2}{n}$?

↳ because in large nueral nets the Layer = L has n-1 inputs so the $n^{[l-1]}$ is the correct input to layer L.

---

* Why this solution helps ~~with~~ with the vanishing and exploding gradients problem?

↳ we can control the variance of z through the variance of w_i and x_i

$z = \sum_{i=1}^{n} w_i x_i$

$Var(z) = n \cdot Var(w_i) \cdot Var(x_i)$ → assuming x has mean 0 and σ of 1.

$Var(z) = n \cdot \frac{1}{n} \cdot 1$

↳ we set the $Var(w_i) = \frac{1}{n}$ to ensure that the variance of z does not grow as the value of n increases and this stabilizes z.

Why scaling w_i by $np.sqrt(\frac{2}{n})$ works?

* np.random.randn generates N(0,1) with Gaussian distribution

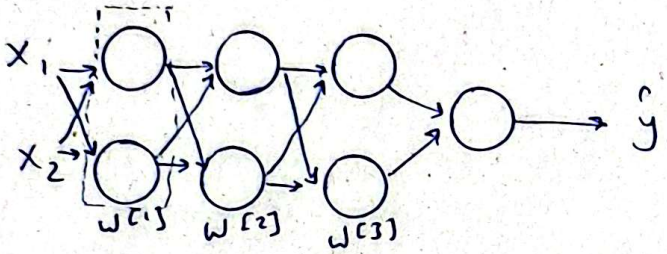
$Var(w_i) = (\sqrt{\frac{2}{n}})^2 \cdot Var(0,1) = \frac{2}{n}$

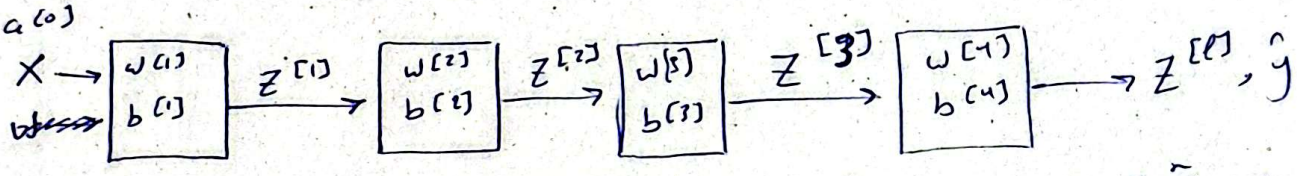
↳ ~~this multiplys the samples~~

this is why we multiplyed the samples with the square root of $\frac{2}{n}$ because the variance of $\sqrt{\frac{2}{n}} = (\sqrt{\frac{2}{n}})^2 = \frac{2}{n}$

* why stabilizing Z is the solution that we have to help reduce this problem?

↳ because the value that is being fed to the neural network's layers is Z





→ since Z is the value that is being fed because $Z = \sum_{i=1} w_i x_i$ and the main component responsable for the problem of vanishing and exploding gradients is w (supposing x has mean 0 and σ of 1).

• then ~~to scale~~ scaling w scales the value of Z helping maintaining stable distribution and ensuring the signal does not deminish or explode through random initialization of w with a certain distribution that has variance $= \frac{1}{n}$

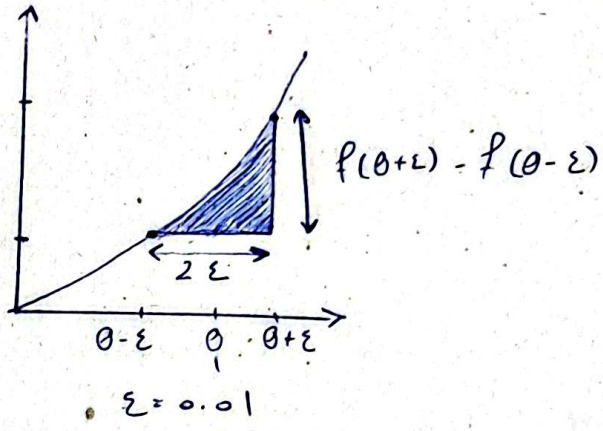
$$\text{Var}(Z) = n \cdot \text{Var}(w_i) \cdot \text{Var}(x_i)$$

Numerical approximation of gradients :

$$\text{slop} = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

$$= \frac{(1.01)^3 - (0.99)^3}{2 \times 0.01}$$

$$= 3.0001$$



$\varepsilon = 0.01$

$f(\theta) = \theta^3 \longrightarrow g(\theta) = f'(\theta) = 3\theta^2 = 3 \times 1^2 = 3$

* approximation error = $3.0001 - 3 = 0,0001$

↳ From this we can be confident that $g(\theta) = 3\theta^2$ is a correct implementation of the derivative of $f(\theta) = \theta^3$.

Gradiant checking

- First step of implementing gradiant checking is taking all our parameters like $w^{[1]}, b^{[1]}, w^{[2]} b^{[2]} \dots w^{[L]}, b^{[L]}$ an reshaping (concatinating) them into 1 vector θ.

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

- next we take the parameters $dw^{[1]}, db^{[1]} \dots dw^{[L]}, db^{[L]}$ from backprob and concatinate them into 1 vector $d\theta$

- The question now is → Is $d\theta$ the gradiant $J(\theta)$

* $J(\theta) = J(\theta_1, \theta_2, \theta_3, \theta_4 \dots)$

for each i :

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

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

now we need to check ; if $d\theta_{approx} \approx d\theta$

* check $$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$$

- the main thing that we did is calculate the numerical gradient for each θ_i

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

and doing that for each θ_i

now we have $d\theta_{approx} = [\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \frac{\partial J}{\partial \theta_3} \dots \frac{\partial J}{\partial \theta_n}]$

↳ after each time we calculated $d\theta_{approx}[i]$ we added the result to the vector $d\theta_{approx}$

- now we can go check the difference between our numerical gradient and the gradient we got from backprob, we can measure them by calculating the euclidian distance between them.

check $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx 10^{-7}$ for $\varepsilon = 10^{-7}$

- if we have used $\varepsilon = 10^{-7}$ then the diffrece is supposed to be around 10^{-7}, if else then we need to check our individual components.

Course 2 • week 2 •

Mini batch GD :-

Mini batch gradient descent is just about processing mini or smaller batches or chunks of the training data at a time because processing the whole data set would consume alot of time and resources.

* a mini-batch has >1 and $\leq m$ size

$$1 < \text{mini\_batch\_size} < m$$

if mini_batch = m then it's just batch GD and if m = 1 then it's called stochastic GD.

* Mini batches typically have a size that is the exponential of 2 Like 2^2 or 2^4 or 2^6 so the sizes could be 4, 16, 32, 64, 128, 256, 512 etc with the most common mini batch size = 512

$X_{(nx,m)} = [\underbrace{x^{(1)}\ x^{(2)} \ldots x^{(1000)}}_{X^{\{1\}}} | \underbrace{x^{(1001)} \ldots x^{(2000)}}_{X^{\{2\}}} \ldots$

$Y_{(1,m)} = [\underbrace{y^{(1)}\ y^{(2)} \ldots y^{(1000)}}_{y^{\{1\}}} | \underbrace{y^{(1001)} \ldots y^{(2000)}}_{y^{\{2\}}} | \underbrace{y^{(2001)}}_{y^{\{3\}}} \ldots$

Mini_batch GD implementation

→ for $t = 1, \ldots,$ ~~last mini batch~~ num_mini_batch.

→ Forward prop on $X^{\{t\}}$

→ $Z^{[1]} = W^{[1]} X + b^{[1]}$

→ $A^{[1]} = g^{[1]}(Z^{[1]})$

⋮

→ $A^{[L]} = g^{[L]}(Z^{[L]})$

} vectorized implementation

→ Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2000} \sum \|w^{[l]}\|_F^2$

size of mini batch

↳ This is the cost on only 1 mini_batch

→ ~~Cost~~ Back prop to compute gradients with respect to $J^{\{t\}}$

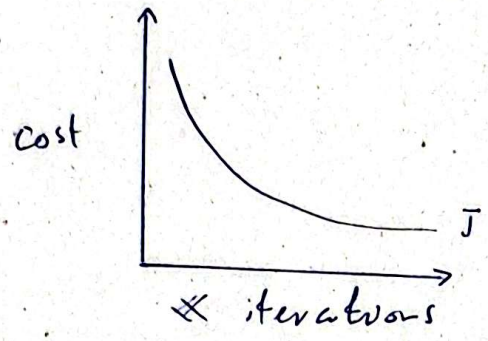
→ $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$

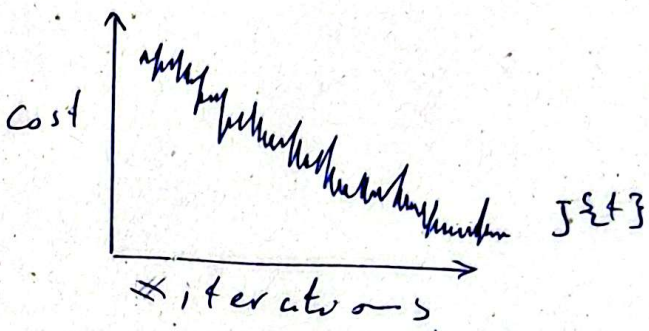
→ $b^{[l]} = b^{[l]} - \alpha db^{[l]}$

This is all called one epoch or a single pass through the training data or in this case a single pass through all the mini_batches.

* Mini_batch GD allows you to make modifications to the weights and biases based on each mini_batch without needing to go through the whole training data to just make one update.

Batch GD vs mini batch GD

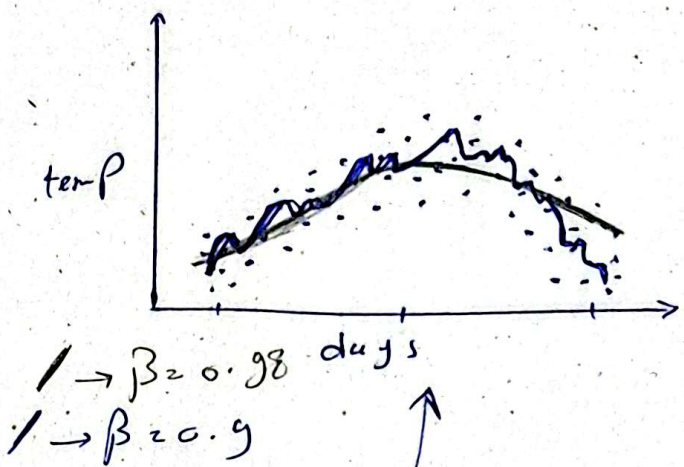




→ the difference between the two graphs is that when we're using batch gradient descent we have to go through all the traing data just to make one weigh update and that surly smothes out the curve because the weight update is based on all the traing data but with min_batch GD the update is based on seprate smaller size mini batches which make the training and weighup dates prone to more bias but surly after runing through all the mini_batches it gets to the same cost as batch GD; if not smaller.

exponentially weighted averages :-

$\theta_1 = 40°f$
$\theta_2 = 49°f$
$\theta_3 = 45°f$
⋮
$\theta_{180} = 60°f$
$\theta_{181} = 56°f$
⋮



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

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

↳ This is the general form for the exponentially weighted averages algorithm that sometimes performs better than Gradient descent.

V_t averages over approximately $\frac{1}{1-\beta}$ training examples

So if $\beta = 0.9$ then it's averaging over $\frac{1}{1-0.9} = 10$ days

→ if you increase the value of β then you're averaging over more past training examples.

$\beta = 0.98 \longrightarrow \frac{1}{1-0.98} = 50$ days

Understanding exponentially weighted averages:-

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

$V_{100} = 0.9\, V_{99} + 0.1\, \theta_{100}$

$V_{99} = 0.9\, V_{98} + 0.1\, \theta_{99}$

$V_{98} = 0.9\, V_{97} + 0.1\, \theta_{98}$

$V_{100} = 0.1\, \theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9(\cdots\cdots$

$V_{100} = 0.1\, \theta_{100} + 0.1 * 0.9\, \theta_{99} + 0.1 * 0.9 * 0.9\, \theta_{98} \cdots$

$= 0.1\, \theta_{100} + 0.1\,(0.9)\,\theta_{99} + 0.1\,(0.9)^2\,\theta_{98} + 0.1\,(0.9)^3\,\theta_{97}$

$+ \cdots\cdots\cdots\cdots$

So basically exponentially weighted moving averages is just computed the averages of two points with giving respect to the previous points, the number of previous points that you want to compute the e.w.m.a of is going to be set based on the value of β like we said before but because the first value does not have any training examples before it we need to correct it's bias.

Bias Correction

When we implement the e.w.a we compute v_t as follows

$v_0 = 0$

$v_1 = \beta v_0 + (1-\beta)\theta_1$

but $v_0 = 0$

so $v_1 = (1-\beta)\theta_1$

$v_2 = \underbrace{\beta v_1}_{\text{very low estimate}} + (1-\beta)\theta_2$

So when we are computing the exponentially weighted moving averages and we start with $v_0 = 0$ then we're estimating a very poor value of v_1, v_2, v_3 till it finally converges at v_t ($\frac{1}{1-\beta}$) or the v(number of days' we're averaging over) So to correct this bias instead of computing v_t we'll compute $\frac{v_t}{(1-\beta^t)}$ So:-

$t = 2 \rightarrow 1-\beta^t = 1-(0.9)^2 = 0.19$

$v_t = \frac{v_t}{0.19} = \frac{0.09\theta_1 + 0.1\theta_2}{0.19} = 0.473\theta_1 + 0.526\theta_2$

So this modification for the first few examples and later when $t = 50$ for example the $(1-\beta^t) = (1-\beta^t) = (1-0.9^{50}) = 0.994$ which is close to 1

So $v_t \simeq v_t$ so it corrects over the first few examples then finally goes to $v_t \simeq v_t$

$t = 1000 \rightarrow (1-\beta^{1000}) = 1 \quad \boxed{v_t = \frac{v_t}{1} = v_t}$

Gradiant descent with momentum:

→ a better idea than using GD alone is combining it with exponentially weighted moving averages which makes GD work faster almost everytime.

This contour plot shows that when we're using GD it takes alot of steps to get to the lowest cost and you wouldn't even be able to use a larger α → learning rate to prevent it from overshooting but if we plug in the exponentially weighted moving averges in the process of updating the parameters to take into consideration the parameters of the previous min_batches parameters this would smooth out the oscillations and yeild better results altogether.

---

on iteration t:

Compute dw, db

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

$$v_{dw} = \beta v_{dw} + (1-\beta)\, dw$$
$$v_{bd} = \beta v_{db} + (1-\beta)\, db$$
$$w = w - \alpha v_{dw}$$
$$b = b - \alpha v_{db}$$

Hyper parameters

α, β

RMS prop → (Root mean squared error prob)

on iteration t:

Compute dw, db on t:-

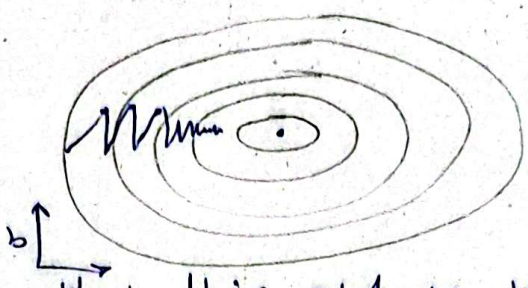
$$sdw = \beta_2 sdw + (1-\beta_2) dw^2$$

$$sdb = \beta_2 sdb + (1-\beta_2) db^2$$

$$w = w - \alpha \frac{dw}{\sqrt{sdw} + \varepsilon}$$

$$b = b - \alpha \frac{db}{\sqrt{sdb} + \varepsilon}$$

$$\varepsilon = 10^{-8}$$



in this plot we wanted to slow the Learning in direction b and maintain the same speed or increase it in direction w to get to the lowest Cost faster.

→ So our original goal was to minimize the Learning in the direction b so we wanted to decrease the value of b which is why we devided it by $\sqrt{sdb}$ because:- $\sqrt{sdb}$ here is a large number so:-

$b = b - \alpha \frac{db}{\sqrt{sdb}}$. is deviding db by a large number

So $b - \alpha \left(\frac{db}{\text{Largenum}} = \text{small num}\right)$

→ $b - \alpha$ (small num)

So overall we're taking smaller steps in the b direction hence the updates to b are small and the updates to w are big which smooths out the Learning Like in momentum.

→ Finally we add $\varepsilon = 10^{-8}$ to $b = \alpha \frac{db}{\sqrt{sdb} + \varepsilon}$ Just in Case sdb or sdw are very small so don't end up exploding the gradients.

ADAM → adaptive moment estimation

Combining the exponentially weighted moving averages with RMS prop make an algorithm which is even better the both of them seprately.

$vdw = 0$, $sdw = 0$, $vbd = 0$, $sdb = 0$

on iteration ti.

Compute dw, db on current t:-

$$vdw = \beta_1 vdw + (1-\beta_1) dw$$
$$vbd = \beta_1 vdb + (1-\beta_1) db$$
} momentum

$$sdw = \beta_2 sdw + (1-\beta_2) dw^2$$
$$sdb = \beta_2 sdb + (1-\beta_2) db^2$$
} RMS prob

$$v_{dw}^{corrected} = \frac{vdw}{(1-\beta_1)^t}, \quad vdb^{corrected} = \frac{vdb}{(1-\beta_1)^t}$$

$$S.dw^{corrected} = \frac{sdw}{(1-\beta_2)t}, \quad sdb^{corrected} = \frac{sdb}{(1-\beta_2)^t}$$

Finally :-

$$W = W - \alpha \frac{vdw^{corrected}}{\sqrt{sdw^{corrected}} + \varepsilon}$$

$\varepsilon \rightarrow 10^{-8}$

and same for b

→ This algorith is the combination of GD, exponentely weighted moving averages, and lastly RMS prob.

Hyperparameters α, β_1, β_2, ε.

β_1 : 0.9 (dw)

β_2 : 0.999 (dw^2)

ε : 10^{-8}

Course 2 : Weeks 3 Hyper parameter tuning

* So far the hyper parameters that we've come across include:-
→ α
→ $\beta_1, \beta_2, \varepsilon$
→ number of layers
→ number of hidden units for the layers
→ Learning rate decay
→ mini-batch size.

---

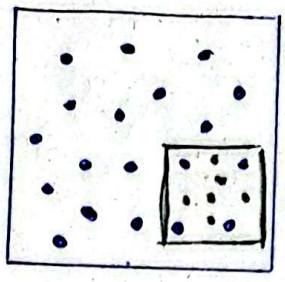
* in the tuning process some hyperparameters are more important to tune than others
→ α is the most important hyper parameter to tune
→ next in importance:-
* if we're using momentum the β is a good start then maybe the mini-batch size and the number of ~~layers~~ hidden units.
* if we're using adam then the default values for β_1, β_2 and ε are pretty much ok and you don't have to tune them.
→ Lastly the least important hyper parameters are the Learning rate decay, the number of layers and the $\beta_1, \beta_2, \varepsilon$ if you're using ADAM.

---

* in the hyper parameter tuning process it's better to search or tryout values at random instead of using Grid search because based on the importance of the change in value of the hyper parameter, Grid search could be wasting alot of time and computational power.

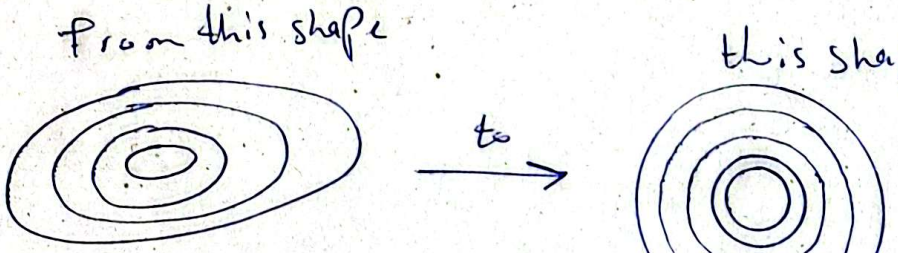
Coarse to fine sampling scheme :-

* In the coarse to fine scheme we look at some randomly sampled values for hyperparameters and if we find some points that are near to each other and result in lowering the cost function J or reducing the time i.e better hyper parameters then we might want to zone in and look for better values in that selected region.

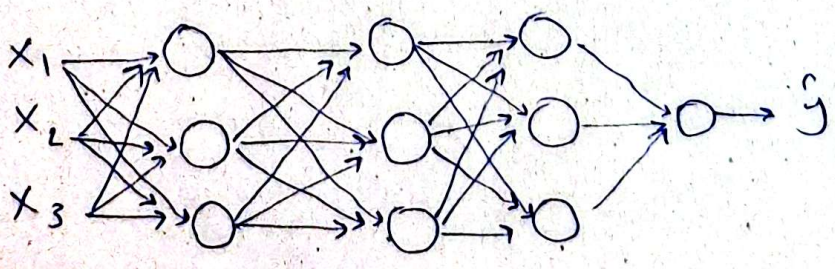


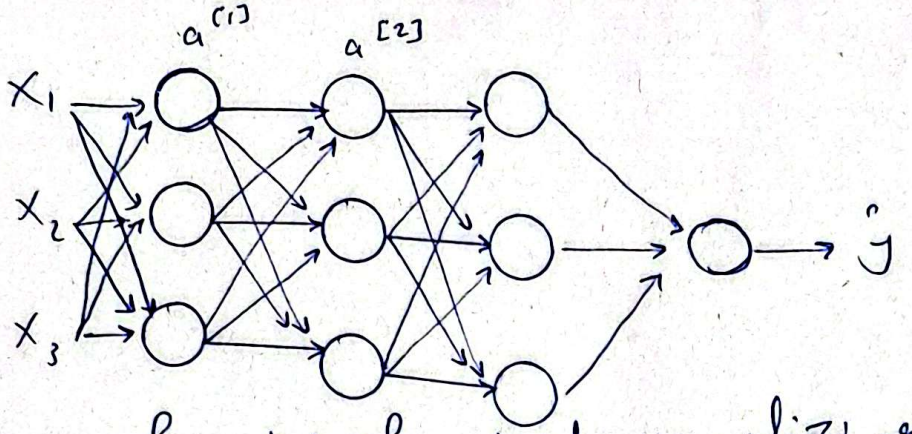
Normalizing activations :-

→ it has been said befor that normalizing the input X speeds up the training process because it helps GD to converge faster as it smooths and rounds the counters of the cost function plot hence making it more convex or so



→ wouldn't also help to normalize the output of each and every layer as they are the input to the next layer so it also speeds up the training process in each layer





→ we found before that normalizing X would help Gradiant descent find the best parameters for $W^{[1]}$ and $b^{[1]}$, so wouldn't normalizing $Z^{[2]}$ also help speeding up the search for parameters $W^{[2]}$ and $b^{[2]}$ that yeild the lowest cost.

→ This is called batch normalization

---

implementing batch norm.

* Given some intermediate values in the NN $Z^{(1)}, \ldots, Z^{(m)}$

$$\mu = \frac{1}{m} \sum_{i=1}^{m} Z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (Z^{(i)} - \mu)^2$$

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

→ now all the Z values have $\mu = 0$ and $\sigma = 1$

→ now we add some parameters to control the distribution of our inputs

$$\tilde{Z}^{(i)} = \gamma Z^{(i)}_{norm} + \beta$$

* Gamma γ and beta β are Learnable parameters.

* how does having the parameters γ and β control the distribution of our inputs?

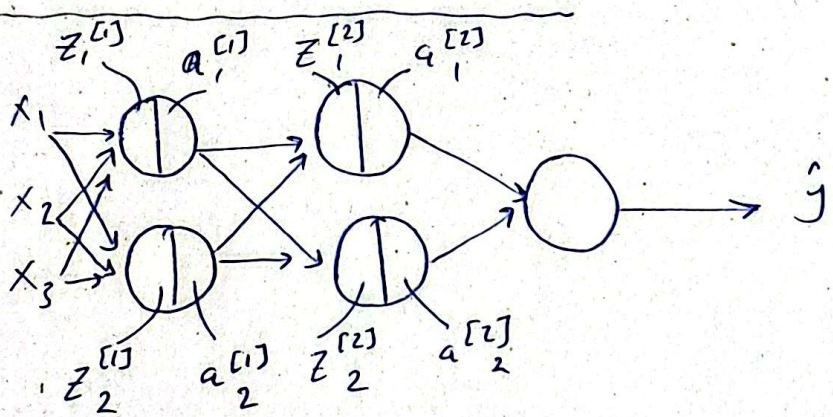
↳ for instance if we set $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$

then

$$\tilde{Z}_{norm} = \sqrt{\sigma^2 + \varepsilon} \frac{Z_{norm} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \mu = Z_{norm}$$

which is the original distribution so the parameters γ and β control the mean and the standard deviation ie. the spread of the data points and their center.

Fitting batch norm in a NN



$$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[\text{Batch norm}]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g^{(1)}(\tilde{Z}^{[1]})} a^{[1]}$$

$$a^{[2]} \xleftarrow{g^{[2]}(\tilde{Z}^{[2]})} \tilde{Z}^{[2]} \xleftarrow[\text{Batch norm}]{\beta^{[2]}, \gamma^{[2]}} Z^{[2]} \xleftarrow{W^{[2]}, b^{[2]}}$$

Parameters $\rightarrow$ $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots W^{[L]}, b^{[L]}$

$\beta^{[1]}, \gamma^{[1]}, \ldots \ldots \beta^{[L]}, \gamma^{[L]}$

working with mini batches

* since we're using mini batches now, we have to redesign our batch normalization for it to work on our mini batches.

$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[B.N]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g^{[1]}(Z^{[1]})} a^{[1]} \longrightarrow$ etc...

First mini batch

$X^{\{2\}} \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[B.N]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g^{[1]}(\tilde{Z}^{[1]})} a^{[1]} \longrightarrow$ etc...

$\vdots$

$X^{\{m\}} \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[B.N]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g^{[1]}(\tilde{Z}^{[1]})} a^{[1]} \longrightarrow$ etc...

* and that is called one epoch.

→ since when we're rescaling $Z^{[l]}$ by $\beta^{[l]}$ and $\gamma^{[l]}$ and we're substracting the $\beta^{[l]}$ and setting $Z^{[l]}$ to $\mu = 0$ and $\sigma = 1$ or so, we're omitting the b value since we're substracting a raw number.

→ since $Z^{[l]}$ has a $\mu = 0$ then $b^{[l]}$ does not matter anymore so we're gonna drop it from the parameters.

parameters → $w^{[l]}, \beta^{[l]}, \gamma^{[l]}$

$Z^{[l]} = W^{[l]} * a^{[l-1]}$

$$\tilde{Z}^{[l]}_{norm} = \gamma^{[l]} Z^{[l]} + \beta^{[l]}$$

for t=1 ... num mini batches
→ compute forward prop
→ compute B.N to replace $Z^{[l]}$ to $\tilde{Z}^{[l]}$
→ compute back prob
* $w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$ → ①
* $\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$ → ②
* $\gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]}$ → ③

use these values $dw^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$ to compute Adam or GD.

Batch norm at test time

→ since mini batch normalization works with multiple data points to figure out μ and σ so it can normalize the distribution of this specific mini batch and this not an option at test since we're dealing with individual data points so we have to come up with one estimate for the mean μ and the S.D σ to normalize the test point so that the classifier i.e. NN can handle them correctly.

→ First we use the exponentially weighted averages across all mini batches for μ and σ.

$X^{\{1\}[l]}, X^{\{2\}[l]}, X^{\{3\}[l]}, \ldots \rightarrow L = 1$

$\downarrow$

$\mu^{\{1\}[l]} \; \mu^{\{2\}[l]} \; \mu^{\{3\}[l]}$ ——— these are the means for the mini batches.

now we calculate the exponentially weighted average for μ and we're going to end up with some value for the last mean:

$V_0 = 0, \; V_1 = \beta V_0 + (1-\beta) V_0 \ldots$

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

until we reach:

$$V_{\text{num\_mini\_batches}} = V_n = \beta V_{n-1} + (1-\beta) V_n$$

→ n reffering to the number of mini batches

* We do the same for σ^2

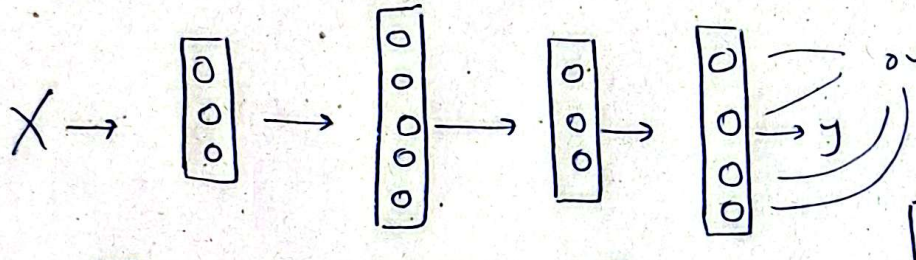
then we use the values that we got from this process *meaning the last μ and σ^2 and computer

$$Z^{(i)[l]}_{norm} = \frac{Z^{(i)[l]} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \rightarrow \tilde{Z}^{(i)[l]} = \gamma Z^{(i)}_{norm} + \beta$$

↳ this $\tilde{Z}$ is calculated before each layer in the Linear forward process per test sample, so it is normalized just like we did to the training data..

Soft max function:

→ softmax function is used when the classes of the model are more than 2 → >2 so we can't use the sigmoid any more or use the binary cross intropy in genral



$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

Activate →

$$t = e^{(Z^{[L]})}$$

$$a^{[L]} = \frac{e^{(Z^{[L]})}}{\sum_{i=1}^{4} t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

output layer units

$$Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.9 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

$$\sum_{i=1}^{4} t_i = 176.3 \rightarrow a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

→ the model classifies that it's gonna be the first class with probability of it being 1, = 84.2%

> softmax algorithm takes the output of the final layer and maps it to some probabilites.

Loss function for soft max

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

→ our ~~trained~~ algorithm outputs that the class is a cat meanwhile it actually isn't, so how would we calculate the cost in this case?

→ in Logestic regression we used to calculate the cost as follows:

$$\text{Cost} = \frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right]$$

this used to work because we only had 2 classes but now we have number of classes $= C$.

→ the Loss function for the softmax algorithm:-

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

* In the previous case we had $y = [0\ 1\ 0\ 0]$

so $\mathcal{L}(\hat{y}, y) = -(0 \log \hat{y}_1) - y_2 \log \hat{y}_2 - 0 \log \hat{y}_3 - 0 \log \hat{y}_4$

$$= -y_2 \log \hat{y} \Big|^{y_2 = 1} = \underline{-\log \hat{y}_2}$$

* so the only way to minimize $\mathcal{L}(\hat{y}, y)$ the cost function is to maximize the value of $\hat{y}_2$ since $\boxed{-\log(1) = 0}$

So this is exactly what happens when we're training the model; the only way to minimize the cost function is to maximize the probability of the true class.

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

$Y_{(4,m)}$

$\vdots$

$Y_{(4,m)}$