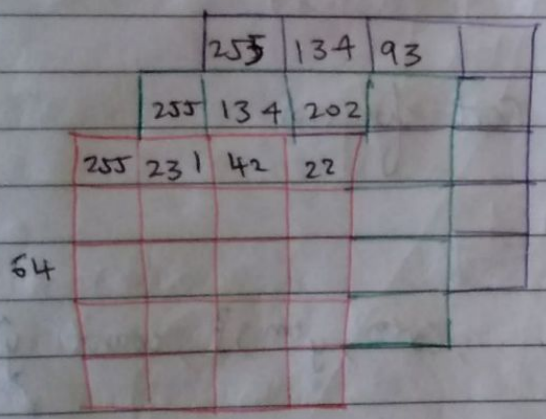
# Deep learning (coursera)

* Supervised learning with neural networks

| Input (x) | output (y) | Application |
|---|---|---|
| Home features | Price | Real estate (Standard NN) |
| Image | object (1, .... 1000) | Photo tagging (CNN) |
| Audio | Text transcript | Speech recognition (Recurrent NN) (RNN) |
| Image, Radar info | Position of other cars | Autonomous driving (custom / hybrid) |




$$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ 255 \\ \vdots \end{bmatrix}$$

64 × 64 × 3 = 12288

$n_x = 12288 = n$

Input feature vector

* Some notations

$(x, y)$ $\quad x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$

($n_x$ dimensional feature vector)

m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})\}$

$m = m_{train}$, $m = m_{test}$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad n_x$$

$\leftarrow m \rightarrow$

$X \in R^{n_x \times m}$ $\quad X.shape = (n_x, m)$

$Y = [y^{(1)}, y^{(2)} \cdots, y^{(m)}]$

$Y \in R^{1 \times m}$

$Y.shape = (1, m)$

## * Logistic Regression cost function

$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$

Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

$\hat{y}^{(i)}$ → over output; $y^{(i)}$ → true label.

Loss (error) function :-

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$$

For single training example

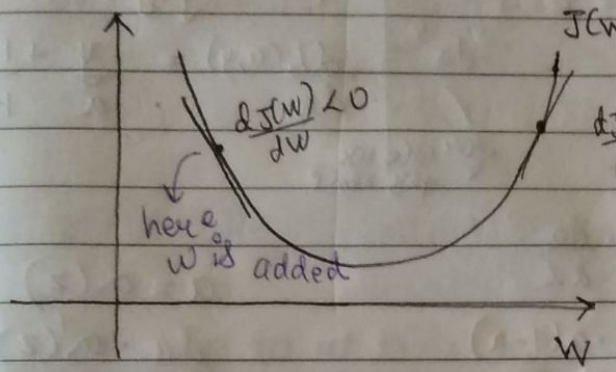
If $y=1$ : $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← want $\log \hat{y}$ large, want $\hat{y}$ large {for small loss}

If $y=0$ : $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← want $(1-\hat{y})$ to be large so, $\hat{y}$ to be small for small loss.

Cost function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \ell(\hat{y}^{(i)}, y^{(i)})$

For entire training set $= \frac{-1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\cdot\log(1-\hat{y}^{(i)})\right]$

* Gradient descent



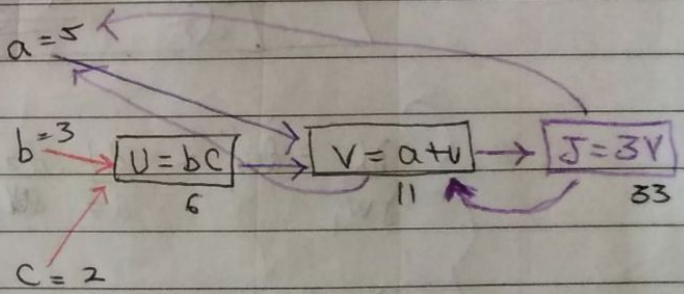

learning rate

$w := w - \alpha \frac{dJ(w)}{dw}$

* Computation graph

$J(a,b,c) = 3(a+bc)$, with $u = bc$ and $v = a+u$

$J = 3v$




$\frac{dJ}{dv} = 3$

$\frac{dJ}{da} = \frac{dJ}{dv}\cdot\frac{dv}{da}$

$= 3 \times 1$

$\frac{dJ}{da} = 3$

$J = 3v$

$v = 11 \rightarrow 11.001$

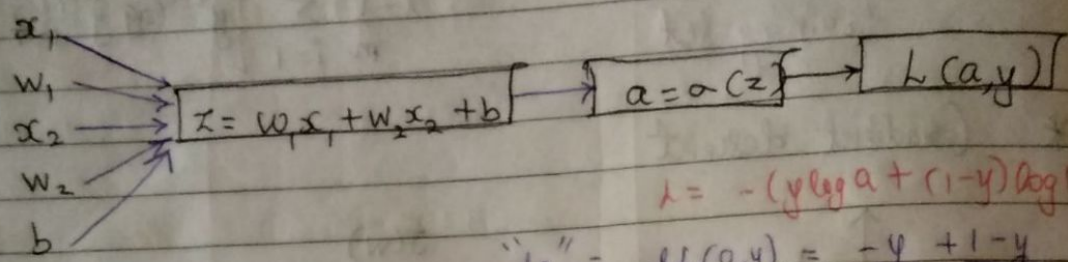
$J = 33 \rightarrow 33.003$

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

* Logistic regression derivatives.




$$L = -(y \log a + (1-y)\log(1-a))$$

$$\text{"}da\text{"} = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

for code use this name

$$dz = \frac{dL}{dz} = \frac{dL}{da}\cdot\frac{da}{dz}$$

$$= \frac{dL}{da}\cdot a(1-a)$$

$$dz = \frac{dL}{dz} = \left[-\frac{y}{a} + \frac{1-y}{1-a}\right] a(1-a)$$

$$= \left[-y + ay + a - ay\right]$$

$$\boxed{dz = \frac{dL}{dz} = a - y}$$

$a = a(z)$

$$\frac{da}{dz} = a'(z)$$

$$= \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right)$$

$$= \frac{e^{-z}}{(1+e^{-z})^2}$$

$$a(z) = \frac{1}{1+e^{-z}}$$

$$= a^2(z)\cdot e^{-z}$$

$$= a^2(z)\cdot(1-a(z))$$

$$a(z)$$

$$= a(z)(1-a(z))$$

$$= a(1-a)$$

$$\frac{\partial L}{\partial w_1} = \text{"}dw_1\text{"} = \frac{dL}{dz}\cdot\frac{dz}{dw_1} = dz \times x_1$$

$$\text{"}dw_2\text{"} = x_2\, dz$$

$$db = dz$$

$$w_1 := w_1 - \alpha\, dw_1$$

$$w_2 := w_2 - \alpha\, dw_2$$

$$b := b - \alpha\, db$$

* Logistic regression on m examples.

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

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b) \qquad dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$$

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})}_{dw_1^{(i)}}$$

* $J = 0 ; dw_1 = 0 ; dw_2 = 0 ; db = 0$

For i=1 to m

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

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$ ↕ n=2 (features)

$db \mathrel{+}= dz^{(i)}$

$J /= m$

$dw_1 /= m$

$db /= m$

$dw_2 /= m$

$dw_1 = \frac{\partial J}{\partial w_1}$, "y for $dw_2$ & $db$

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

* Vectorization [To avoid for loops] (less efficient for bigger data sets)

$z = w^T x + b$

Non-vectorized

```
z = 0
For i in range(n_x):
    z += w^T[i] * x[i]
z += b
```

vectorized (Takes very less time)

```
z = np.dot(w, x) + b
```

* Example :-

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

If you want to calculate exponential of each element of v

$$u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

```
u = np.zeroes((n,1))
For i in range(n):
    u[i] = math.exp(v[i])
```

[non-vectorized]

Now, vectorized

```
import numpy as np
u = np.exp(v)
```

* we use vectorization in our code to get rid of 1 for loop.

$J=0,\ dw_1=0,\ dw_2=0,\ db=0$ $\quad dw = np.zeros((n_x,1))$

for $i=1$ to $m$:

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

$a^{(i)} = \sigma(z^{(i)})$

$J\ +\!= -[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$

$dz^{(i)} = a^{(i)}[1-a^{(i)}]$

for $j=1 \dots n_x$:

$dw_1 += x_1^{(i)} dz^{(i)}$

$dw_2 += x_2^{(i)} dz^{(i)}$

$dw_j += \dots$

$n_x = 2$ $\quad dw\ +\!= x^{(i)} dz^{(i)}$

$db\ +\!= dz^{(i)}$

$J = J/m,\ dw_1 = dw_1/m,\ dw_2/m,\ db = db/m$

$dw /= m$

* Vectorizing logistic regression

$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b \qquad z^{(3)} = w^T x^{(3)} + b$

$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)}) \qquad a^{(3)} = \sigma(z^{(3)})$

$$X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix} \qquad (n_x, m)$$

$$[z^{(1)}\ z^{(2)}\ z^{(3)} \cdots z^{(m)}] = w^T \cdot X + [b\ b \cdots b]_{1\times m}$$

$$= \begin{bmatrix} w^{(T)} \text{ (row vector)} \end{bmatrix} \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix} + [b\ b\ b \cdots]$$

$$Z = [z^{(1)}\ z^{(2)}\ z^{(3)} \cdots z^{(m)}] = [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \cdots \quad w^T x^{(m)} + b]$$

$Z = np.dot(w.T, X) + b$

[Broadcasting in Python]

$$A = [a^{(1)}\ a^{(2)}\ a^{(3)} \dots\ a^{(m)}] = \sigma(z)$$

## * Vectorizing gradient Computation

$$dz = [dz^{(1)}\ dz^{(2)} \dots\ dz^{(m)}]$$

$$A = [a^{(1)} \dots\ a^{(m)}]$$

$$Y = [y^{(1)} \dots\ y^{(m)}]$$

$$dz = A - Y = [a^{(1)} - y^{(1)}\quad a^{(2)} - y^{(2)} \dots\quad a^{(m)} - y^{(m)}]$$

$dw = 0$

$dw\ += x^{(1)} dz^{(1)}$

$dw\ += x^{(2)} dz^{(2)}$

⋮

$dw\ /= m$

(we got rid of this for loop)

$db = 0$

$db\ += dz^{(1)}$

$db\ += dz^{(2)}$

⋮

$db\ += dz^{(m)}$

$db\ /= m$

For this,

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$\boxed{db = \frac{1}{m}\, np.sum(dz)}$$

$$\boxed{dw = \frac{1}{m} X dz^{T}}$$

$$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

$$= \frac{1}{m} [\, x^{(1)} dz^{(1)} + \dots\ x^{(m)} dz^{(m)} \,]$$

# Modified logistic regression

For iteration in range(1000):

$z = w^T x + b$

$z = np.dot(w.T, x) + b$

$A = \sigma(z)$

$dz = A - Y$

$dw = \frac{1}{m} X dz^T$

$db = \frac{1}{m} np.sum(dz)$

$w = w - \alpha\, dw$

$b = b - \alpha\, db$

# Broadcasting example

Calories from Carbs, proteins, Fats in 100g of different foods:

| | Apples | Beef | Eggs | Potatoes |
|---|---|---|---|---|
| Carb | 56.0 | 0.0 | 4.4 | 68.0 |
| Protein | 1.2 | 104.0 | 52.0 | 8.0 |
| Fat | 1.8 | 135.0 | 99.0 | 0.9 |

$= A_{(3,4)}$

Suppose you want to calculate % of calories from carb, protein, Fat.

① $cal = A.sum(axis = 0)$ — means it will sum i.e a row vector [ , ]

If (axis = 1) it will sum → i.e Column vector [ ]

② $percentage = 100 * A / (cal)$

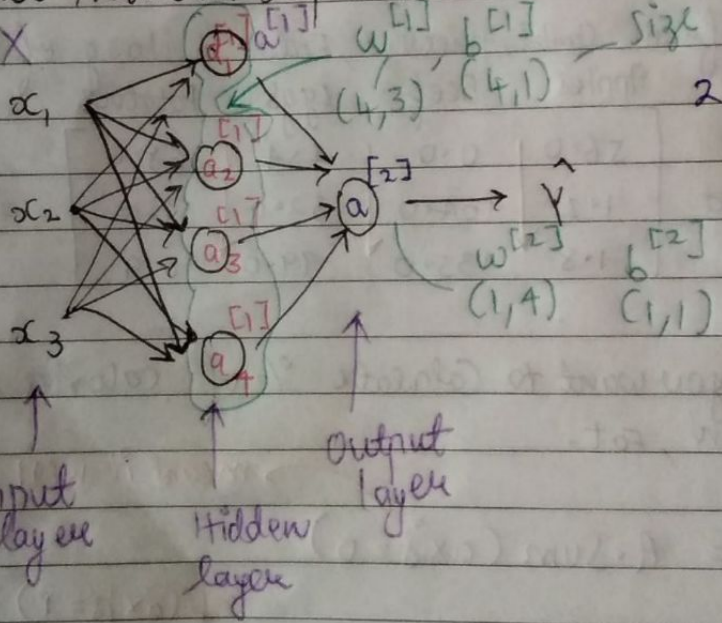
## * Broadcasting example

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} \quad 100 \quad = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix} \rightarrow \begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$$

## * Neural network representation

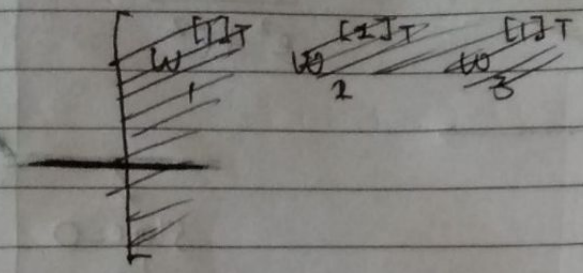
$a^{[0]} = X$




2 layer NN (we don't count input layer)

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$$
$$a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w^{[1]T} x + b_2^{[1]}$$
$$a_2^{[1]} = \sigma(z_2^{[1]})$$

$a_i^{[l]}$ — layer, node in layer



$$\begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix}_{4\times 3} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3\times 1} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}_{4\times 1}$$

(weight betⁿ 0th & 1st layer)

* Vectorizing across multiple examples.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}_{(n_x, m)}$$

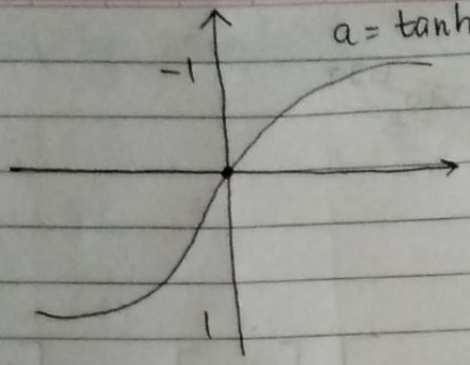
$$Z^{[1]} = \begin{bmatrix} z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \end{bmatrix}$$

layer no. / training example no.

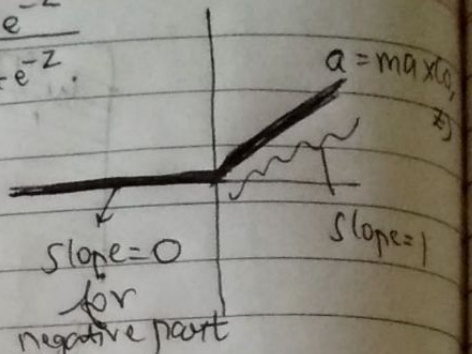
$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

hidden units ↕ ; Training examples ↔

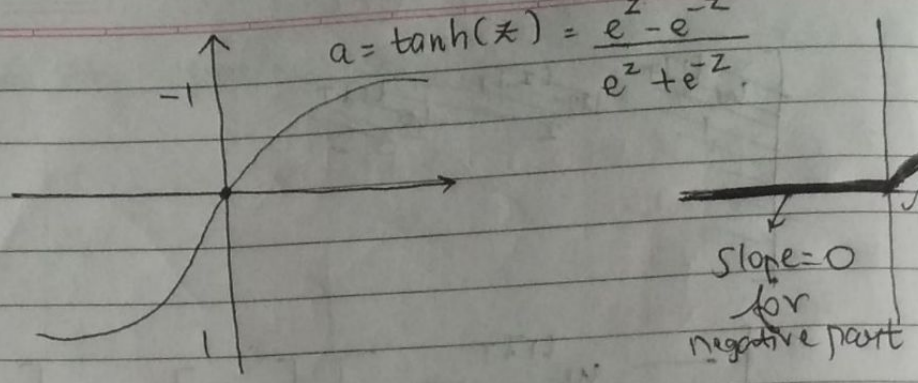
*




$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$




* For output layer you can use Sigmoid fn.
* For hidden layer you can use Relu.




* For output layer you can use Sigmoid fn.
* For hidden layer you can use Relu.

## * Gradient descent for neural networks.

Parameters: $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$ — $(n^{[2]}, 1)$

$(n^{[1]}, n^{[0]})$ $(n^{[1]}, 1)$ $(n^{[2]}, n^{[1]})$

Cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} \ell(\hat{y}, y)$ ( $a^{[2]}$

## Formulas for Computing derivatives

Forward propogation:

$$z^{[1]} = w^{[1]} x + b^{[1]}$$
$$A^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$$

Back propagation:-

$$dz^{[2]} = A^{[2]} - Y \qquad Y = [y^{(1)}\ y^{(2)} \ldots y^{(m)}]$$

$$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=True)$$

prevent python from shaving $(n^{[2]}, )$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$(n^{[1]}, m)$ $\quad$ $(n^{[1]}, m)$ $\quad$ $(n^{[1]}, m)$

element wise product.

$$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True)$$

would prevent python from shaving $(n^{[1]}, )$

For proof of:-

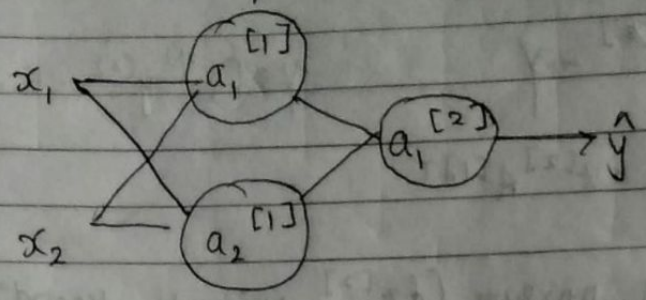
$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$$z^{[1]} \to a^{[1]} \to z^{[2]} \to a^{[2]} \to L$$

Now,

$$dz^{[1]} = \frac{dL}{dz^{[1]}} = \frac{dL}{dz^{[2]}} \cdot \frac{dz^{[2]}}{da^{[1]}} \cdot \frac{da^{[1]}}{dz^{[1]}}$$

## * Random initialization




$w^{[1]} = np.random.randn((2,2)) * 0.01$

$b^{[1]} = np.zero((2,1))$

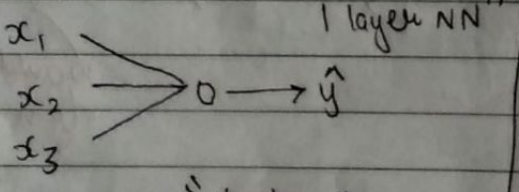
similarly,

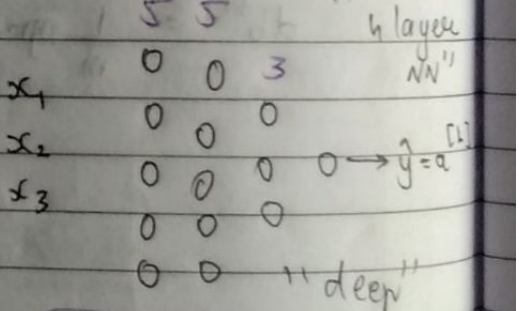
$w^{[2]} = \ldots$

$b^{[2]} = \ldots$

To make random no's smaller (as we are using Sigmoid & other activation fn)

## * what is a deep neural network?




$L = 4$

$n^{(l)}$ = no of units or nodes in layer $l$

$n^{[1]} = 5, \; n^{[2]} = 5, \; n^{[3]} = 3, \; n^{[4]} = n^{[L]} = 1$

$n^{[0]} = n_x = 3$

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

$w^{[l]}$ = weights for $z^{[l]}$

* Forward propogation in a deep network.

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

* $W^{[l]} = (n^{[l]}, n^{[l-1]}) = dw^{[l]}$

dimention of $W^{[l]}$

$$b^{[l]} = (n^{[l]}, 1) = db^{[l]}$$

* Vectorized implementation

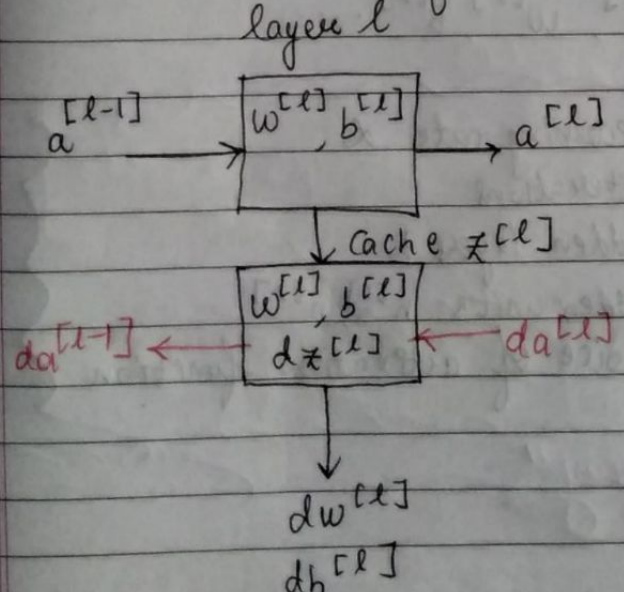
$$Z^{[l]}, A^{[l]} = (n^{[l]}, 1)$$

$$Z^{[l]}, A^{[l]} = (n^{[l]}, m) = dZ^{[l]} = dA^{[l]}$$

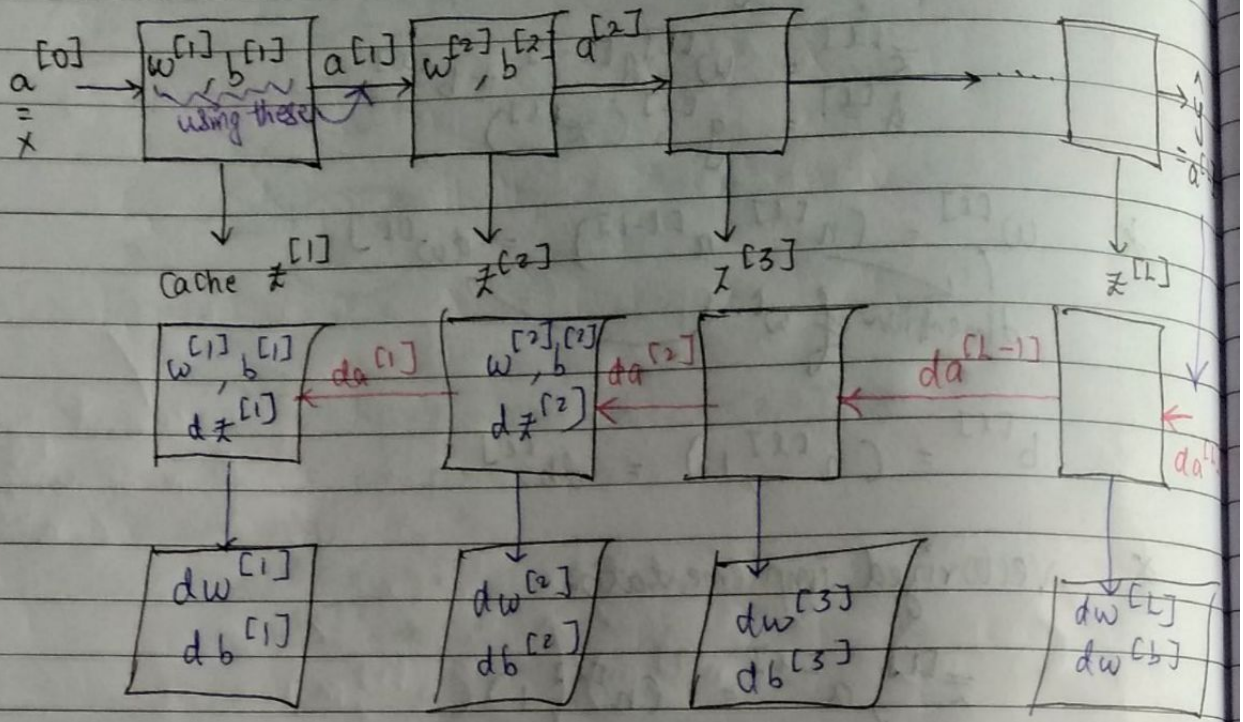
when $l = 0$,

$$A^{[0]} = X = (n^{[0]}, m)$$

* Forward & backward functions.



# One iteration of gradient descent

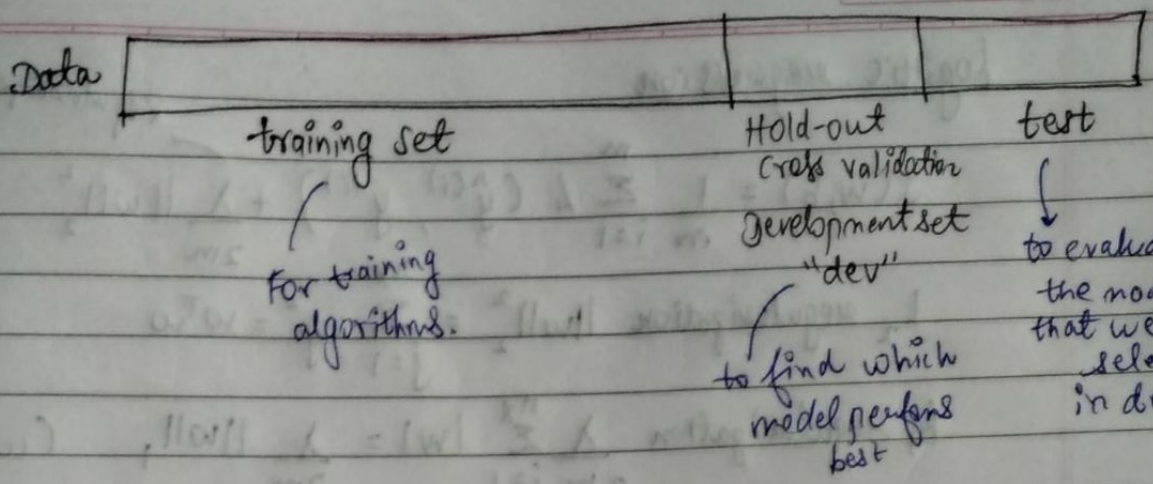



$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

* What are hyperparameters :-

Parameters : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots\ldots$

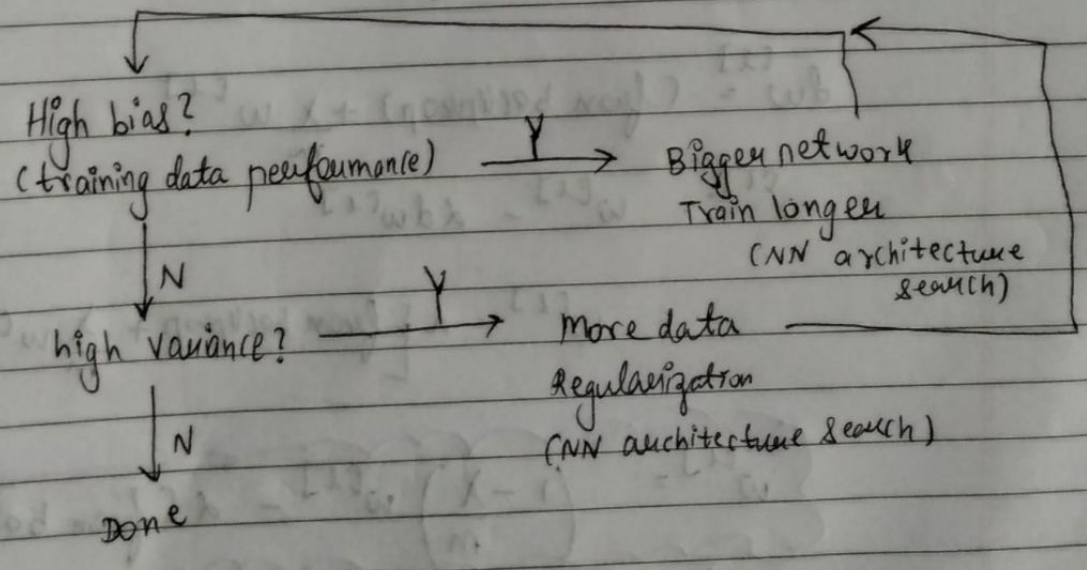
hyperparameters : learning rate $\alpha$
Iterations
hidden layers L
hidden units $n^{[1]}, n^{[2]} \ldots$
choice of activation function




→ Make sure dev & test come from same distribution.

## Bias and Variance

| | | | | |
|---|---|---|---|---|
| Train set error: | 1% | 15% | 15% | 0.5% |
| dev set error: | 11% | 16% | 30% | 1% |
| | high variance | high bias | both high | low bias low variance. |



Logistic regression

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

(Regularization parameter)

$L_2$ regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

$L_1$ regularization $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w| = \frac{\lambda}{2m}\|w\|_1$ (w will be sparse it will have lot of zeros in it)

For Neural network :-

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

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

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

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

$$= w^{[l]} - \alpha\left[\text{from backprop} + \frac{\lambda}{m}w^{[l]}\right]$$

$$w^{[l]} = \left(1 - \frac{\lambda}{m}\right)w^{[l]} - \alpha[\text{from backprop}]$$

"Weight decay"

2) Implementing dropout ("Inverted dropout")

Illustrate with layer $l = 3$

$d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\_prob$

dropout vector for layer 3

Say keep_prob = 0.8

$a3 = np.multiply(a3, d3)$

$a3 /= 0.8$ (keep_prob)

Lets say you have 50 units → 10 units are shut off

$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$

(reduced by 20%)

To maintain original value of $z^{[4]}$ we divide a3.

* Try to keep_prob high where you think there is not much need of removing neurons. ex: For (4, 3) matrix (weight)

But if you have (7, 7) matrix where chances of overfitting are high you can make keep_prob lower.

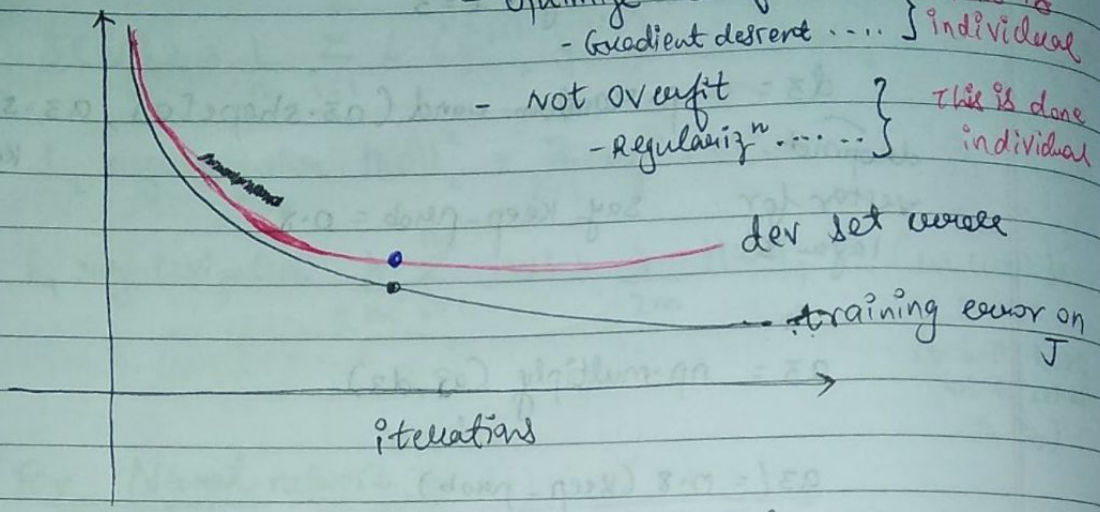
3) * Data-Augmentation

4 → 4 4 4

you can use some image & perform some transform" on it.

4) Early Stopping

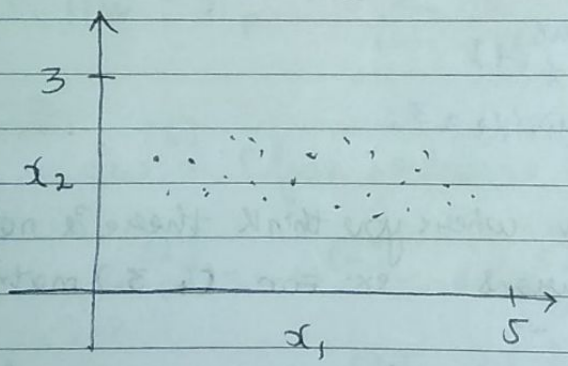
- optimize cost fn J — Gradient descent .... } This is individual
- not overfit — Regularizn .... } This is done individual




you are stopping gradient descent in between.
By Early stopping, it mixed both optimizn & overfitting reduce methods.

* To Speed up training. (NORMALIZATION)

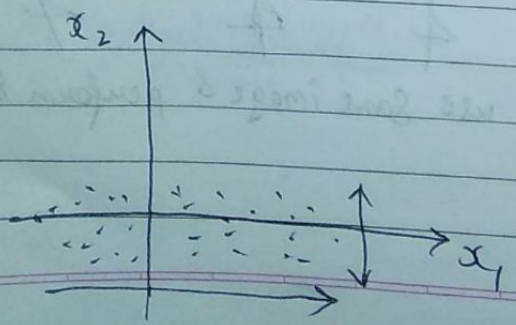
1)




Subtract mean:-

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

$$X = X - \mu$$

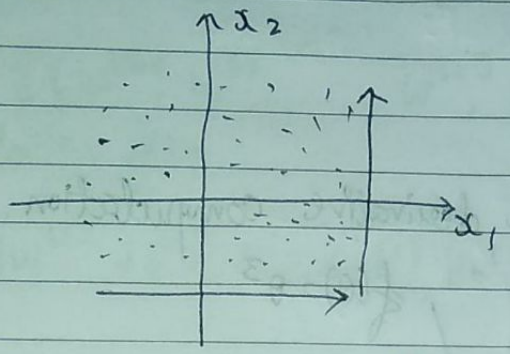


Normalize Variance

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2$$

↳ element wise Squaring.

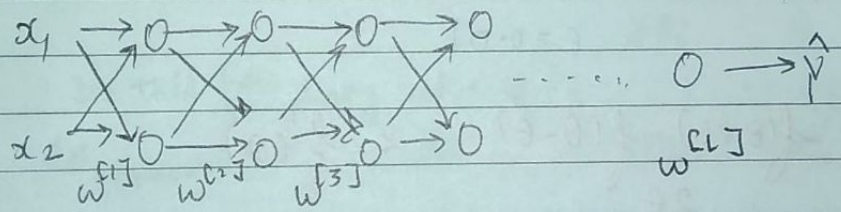
$$X /= \sigma^2$$




Now we have same variance of $x_1$ & $x_2$

If you use $\mu, \sigma^2$ in training data, use same values in test set.

2). Vanishing / exploding gradients.




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

$$\hat{y} = w^{[l]} w^{[l-1]} \cdots w^{[2]} w^{[1]} x$$

Assume each of this is $z^{[1]} = a^{[1]}$

$$w^{[1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \qquad \hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1}$$

For deep layer:

if $w^{[l]} > I$ (I is identity matrix)
(Activations can explode exponentially)

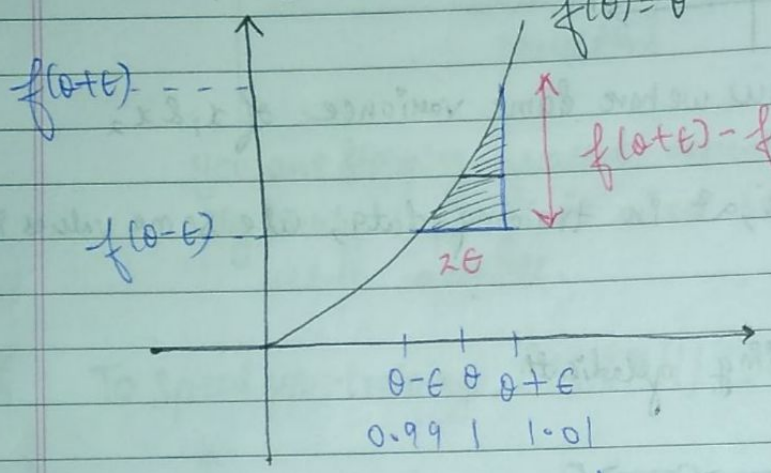
$w^{[l]} < I$
(Activation can decrease exponentially)

* weight initialization (To avoid exploding & vanishing)

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

$Var(w_i) = \frac{2}{n}$

* Checking your derivative computation




$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx f'(\theta)$$

$$\frac{(1.01)^3 - (0.99)^3}{2\times 0.01} \approx 3.0001 \approx 3\theta^2 \approx 3$$

when you take $\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} \approx 3.0301$

error is 0.03 which is more

so prefer

## * Gradient checking.

Take $W^{[1]}, b^{[1]} \ldots W^{[L]}, b^{[L]}$ & reshape into big vector $\theta$.

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

Similar for,

$dW^{[1]}, db^{[1]}, \ldots dW^{[L]}, db^{[L]}$ into big vector $d(\theta)$

$$J(\theta) = J(\theta_1, \theta_2, \theta_3 \ldots)$$

for each $i$:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \ldots \theta_i + \epsilon, \ldots) - J(\theta_1, \theta_2 \ldots \theta_i - \epsilon, \ldots)}{2\epsilon}$$

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

Do this for every value of $i$,

$d\theta_{approx} \overset{?}{\approx} d\theta$ (check whether they are equal?)

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx 10^{-7}$ - great!

Take $\epsilon = 10^{-7}$ & you get

* Don't use in training - only to debug
If algorithm fails, look at components to try to identify bug.
Remember regularization
Doesn't work with dropout

# * Optimization algorithms *

$X = [\, x^{(1)} \; x^{(2)} \; x^{(3)} \; \dots \; x^{(1000)} \;|\; \dots \; x^{(m)} \,]$

$(n_x, m)$ — $X^{\{1\}}$ $(n_x, 1000)$ — $X^{\{5000\}}$ $(n_x, 1000)$

$Y = [\, y^{(1)} \; y^{(2)} \; y^{(3)} \; \dots \; y^{(1000)} \;|\; \dots \; y^{(m)} \,]$

$(1, m)$ — $Y^{\{1\}}$ $(1, 1000)$ — $Y^{\{5,000\}}$ $(1, 1000)$

This is if you have $m = 5{,}000{,}000$

5,000 mini-batches of 1000 each

mini-batch $t$: $X^{\{t\}}, Y^{\{t\}}$

* How mini-batch works.

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

Forward prop on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

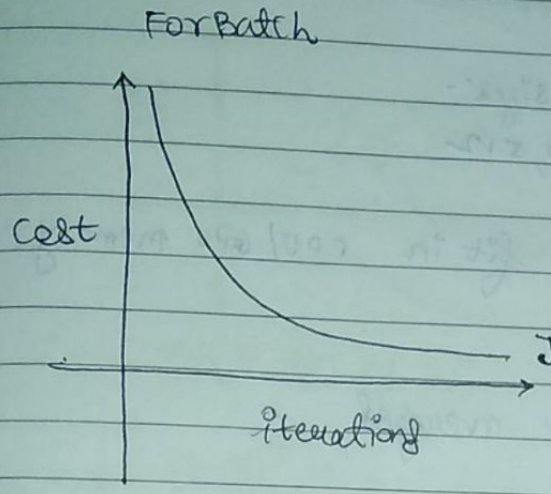
Compute cost $J = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum \|w^{[l]}\|_F^2$ — For $X^{\{t\}}, Y^{\{t\}}$

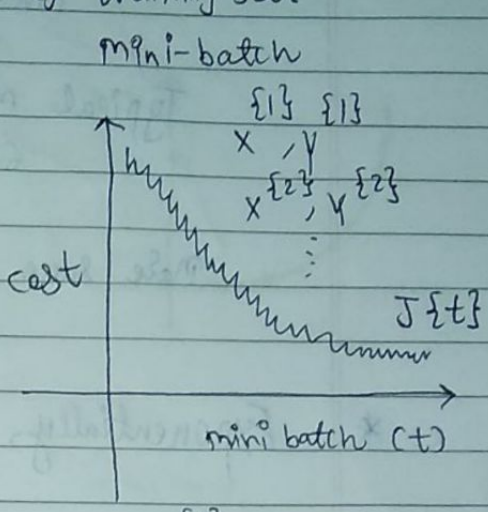
$m = 1000$

Back prop to get gradients w.r.t $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha \, db^{[l]}$$

This is called "1 epoch"
" single pass through training set.




Plot $J^{\{t\}}$ computed using $x^{\{t\}}, y^{\{t\}}$

* Choosing your mini-batch size

If mini-batch size = m : Batch gradient descent $(x^{\{1\}}, y^{\{1\}}) = (X, Y)$

If mini-batch size = 1 : Stochastic gradient descent

$(x^{\{1\}}, y^{\{1\}}) = (x^{(1)}, y^{(1)})$
⋮
so on.

(Every example is its own mini-batch.

In practice: mini-batch size betn 1 & m

| Stochastic gradient descent | In betn (mini-batch size (middle not much big or small) | Batch gradient descent (mini-batch size = m) |
|---|---|---|
| ↓ | ↓ | ↓ |
| lose speading from vectrization. | Fastest learning | Too much time per iteration. |

If Small training set: use batch gradient descent
(m < 2000)

Typical mini-batch sizes:-
64, 128, 256, 512

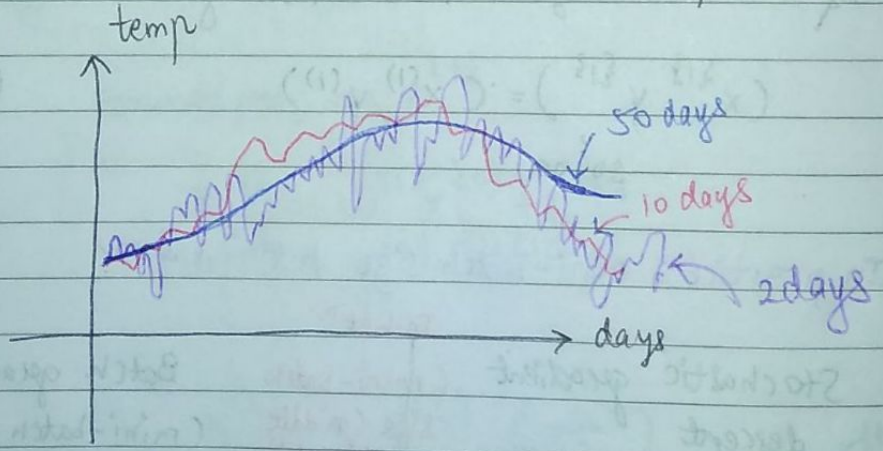
Make sure mini-batch fit in CPU/GPU memory.

## * Exponentially weighted averages

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

$\beta = 0.9$ : $\approx$ 10 days temp.
$\beta = 0.98$ : $\approx$ 50 days
$\beta = 0.5$ : $\approx$ 2 days

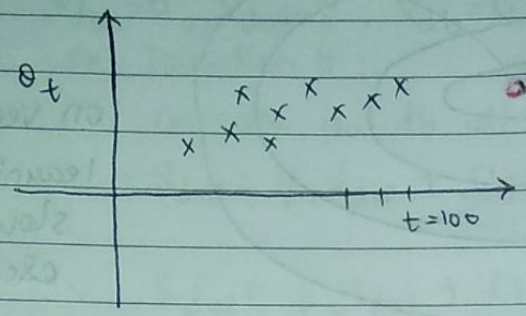
$V_t$ approximately averages over $\approx \frac{1}{1-\beta}$ days temp.

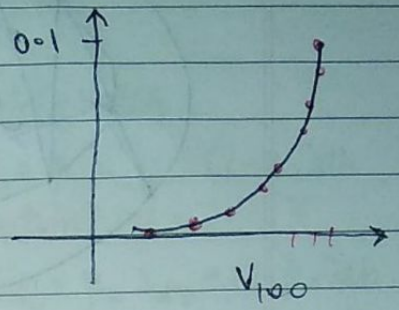



$V_{100} = 0.9 V_{99} + 0.1 \theta_{100}$
$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$
$V_{98} = 0.9 V_{97} + 0.1 \theta_{98}$

$V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(V_{98}))$
$= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.9V_{97} + 0.1\theta_{98}))$
$+ \ldots$

$$V_{100} = 0.1\theta_{100} + 0.1\times0.9\times\theta_{99} + 0.1\times(0.9)^2\times\theta_{98} + 0.1(0.9)^3\times\theta_{97} + \cdots\cdots$$

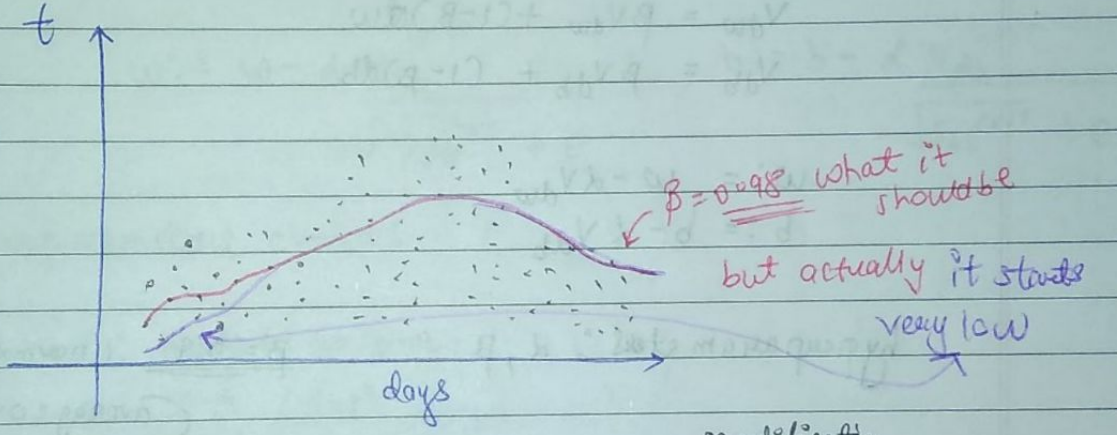



This is for $\beta = 0.9$

$(0.9)^{10} \approx 0.35 \approx \frac{1}{e}$

* Bias correction in weighted averages.




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

$V_0 = 0$

$V_1 = 0.98V_0 + 0.02\theta_1$

$V_2 = 0.02\theta_2 + 0.98V_1$

$= 0.98(0.02\theta_1) + 0.02\theta_2$

$V_2 = 0.01960_1 + 0.02\theta_2$

modification

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

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

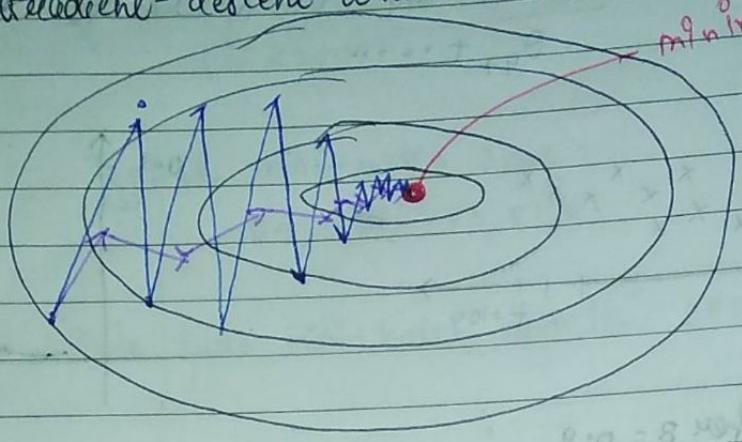
$= 0.0396$

$\therefore V_2 = \frac{V_2}{0.0396}$

$V_2 = \frac{0.01960_1 + 0.02\theta_2}{0.0396}$

So when t is very large $\beta^t$ doesn't make any diff. so it helps for initial temperatures.

* Gradient descent with momentum




on vertical axis, learning should be slowere to reduce oscillations.

on horizontal axis, faster learning.

Momentum:-

on Iteration t:

Compute $dw, db$ on current mini-batch

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

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

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

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

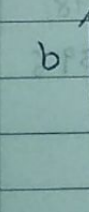
hyperparameters: $\alpha, \beta$ $\beta = 0.9$ (normally use)

(average over last 10 gradients)

* RMSprop

on iteration t:-

Compute $dw, db$ on current mini-batch




element wise

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ ← small

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

Ex:- we want b to be small & w to be large

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \epsilon}$ small so to speed up in w direc$^n$

$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$ large so to slow down in b direc$^n$

For numerical stability, $\epsilon = 10^{-8}$

## * Adam Optimization algorithm.:

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

On iteration $t$:

compute $dw, db$ using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)dw,\quad V_{db} = \beta_1 V_{db} + (1-\beta_1)db$

"momentum" $\beta_1$

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

"RMS prop" $\beta_2$

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

$$S_{dw}^{\text{correct}} = S_{dw} / (1-\beta_2^t),\quad S_{db}^{\text{correc.}} = S_{db} / (1-\beta_2^t)$$

$$w := w - \alpha \frac{V_{dw}^{\text{corr.}}}{\sqrt{S_{dw}^{\text{corr}}} + \epsilon},\quad b := b - \alpha \frac{V_{db}^{\text{corr.}}}{\sqrt{S_{db}^{\text{corr}}} + \epsilon}$$

hyperparameters choice:

$\alpha$: needs to be tune.

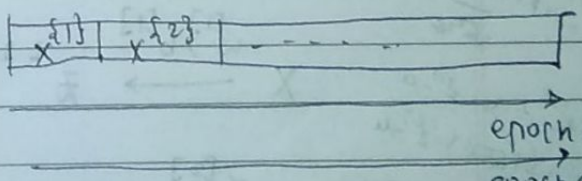
$\beta_1$: 0.9 ($dw$)

$\beta_2$: 0.999 ($dw^2$)

$\epsilon$: $10^{-8}$

## * Learning rate decay.

1 epoch = 1 pass through data




$$\alpha = \frac{1}{1 + \text{decayrate} * \text{epoch\_num}} \times \alpha_0$$

let $\alpha_0 = 0.2$

decayrate = 1

| Epoch | $\alpha$ |
|---|---|
| 1 | 0.1 |
| 2 | 0.67 |
| 3 | 0.5 |
| ⋮ | ⋮ |

## * Implementing batch norm

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

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

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

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

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

($\gamma$, $\beta$: learnable parameters of model.)

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

## * Adding Batch norm to a network.

$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{B.N}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \rightarrow z^{[2]} \ldots\ldots$$

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots$ $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots$

$d\beta^{[1]}$

$\beta^{[1]} := \beta^{[1]} - \alpha\, d\beta^{[1]}$

# Multi-class classification.

## * Softmax function

$$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]} \qquad (4,1)$$

Softmax Activation function:

$$\rightarrow t = e^{(z^{[L]})} \qquad (4,1)$$

$$\rightarrow a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i} = a, \quad a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

$(4,1)$

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

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

softmax →

$$= \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

$$\text{hard max} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(∵ 5 was biggest it gave 1 to it)