

Deep learning

Applications

→ CNN → Image based

LSTM, RNN → mostly on sequential data like Audio & text

There can also be custom Neural Networks.

- Structured data -: In tabular form, having rows & columns.

Unstructured data -: Audio, Image & text

Sigmoid is ^{worse} better than relu because it helps with slow's down gradient descent

Notations -:

A single training example → (x, y)

$$x \in \mathbb{R}^{n_x}$$

↳ x is an n_x dimensional feature

$$y \in (0, 1)$$

↳ y is an element of a set which has two values $(0, 1)$

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

* m_{train} : no of training examples
 m_{test} : no of test examples

"matrix of capital x"

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

↑
 n_x
↓

← m →

Here, no. of rows = n_x or no. of examples in feature vector x .

, no. of columns = m , no. of training examples

$$X \in \mathbb{R}^{n_x \times m}$$

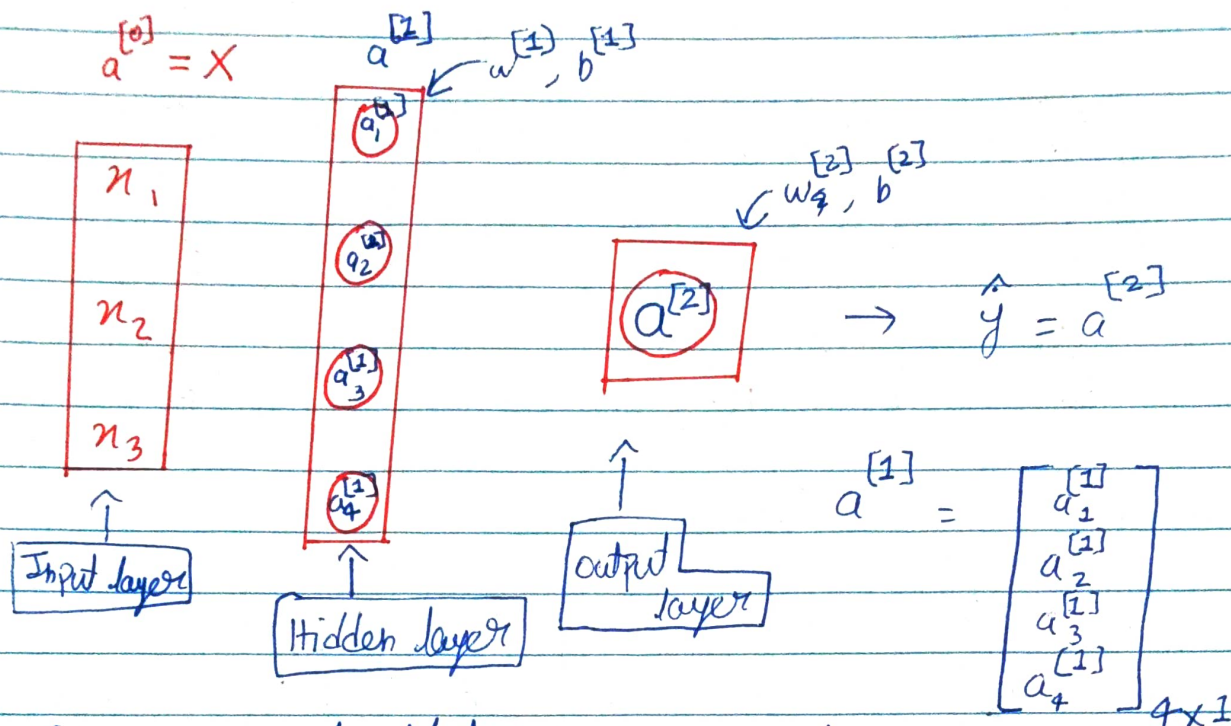
set of all real

→ x is an element of a $2d$ matrixes, $(n_x \times m)$ with dimension
a matrix

⇒ The loss function computes error from single training example,

The cost function is the average of errors from all the training example.

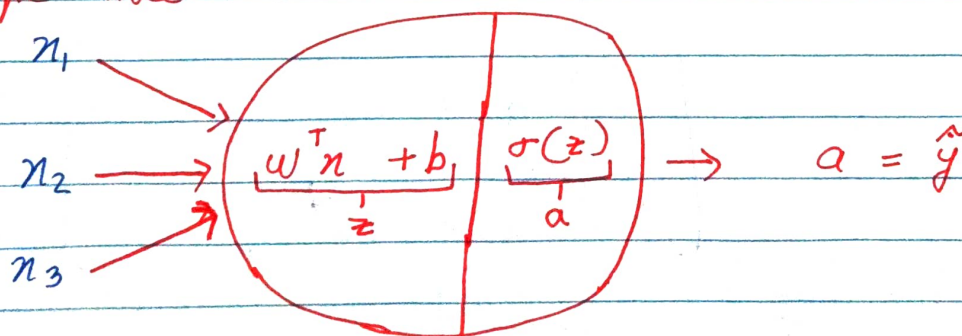
Neural Net's representation



This Neural Network has only 2 ~~hidden~~ layers
we don't count input layer

$a_i^{[l]}$, $l = \text{layer no}$
 $i = \text{node in layer}$

A single node



$$z = w^T x + b$$

$$a = \sigma(z)$$

in the given neural net,

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

matrix representation

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

$$\begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Vectorizing across multiple example

We know that,

$$\begin{aligned}z^{[1]} &= W^{[1]}x + b^{[1]} \\a^{[1]} &= \sigma(z^{[1]}) \\z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\a^{[2]} &= \sigma(z^{[2]})\end{aligned}$$

Here,

$$x \longrightarrow a^{[2]} = \hat{y}$$

$$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$

$$x^{(2)} \longrightarrow a^{2} = \hat{y}^{(2)}$$

$$\vdots \qquad \vdots \qquad \vdots$$
$$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$$

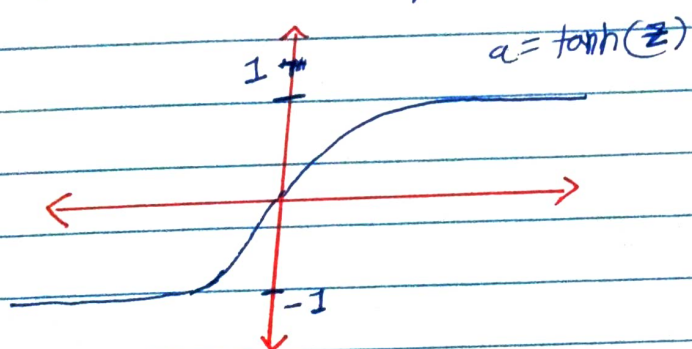
Here,

$a^{[2](i)}$
└─ example i
└─ layer 2

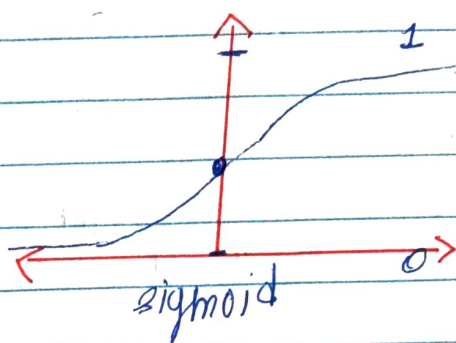
so, this means that output of the neural network, when it's fed the example (i) .

Activation functions

One of the major problems with activation fn like sigmoid & tanh is that if the value of "z" is very large or very small, then the gradient of the function becomes very small. The slope is close to "0", which slows down the gradient descent.

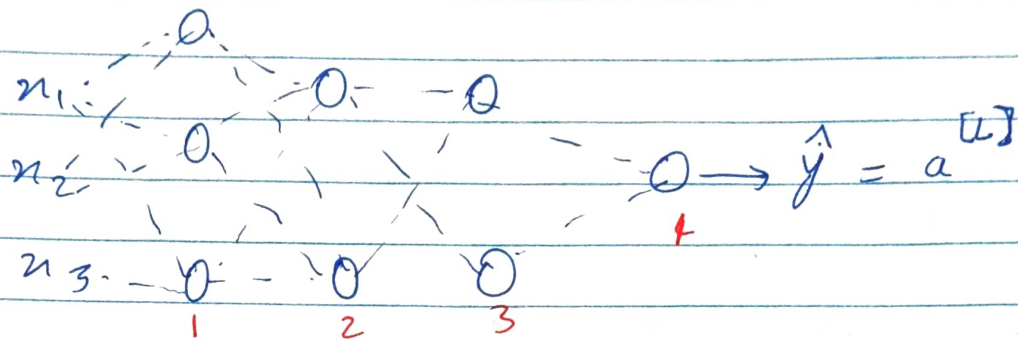


$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



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

Deep Neural Net's Notations



$L =$ no of layers in the network $= 4$

$n^{[l]}$ = no of units/nodes in layer l

ex $\rightarrow$

$$n^{[2]} = 2$$

$$n^{[3]} = 2$$

$$n^{[4]} = 1 = n^{[L]}$$

these are same

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

$a^{[l]}$ = activations in layer l

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

$w^{[l]}$ = weight's for $z^{[l]}$

Forward prop in a deep Network

	0	0		
x_1	0	0	0	
x_2	0	0	0	0
x_3	0	0	0	
	0	0	3	4
	1	2		

$$X: \quad z^{[1]} = w^{[1]} \cdot x + b^{[1]} \quad \text{or} \quad z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$

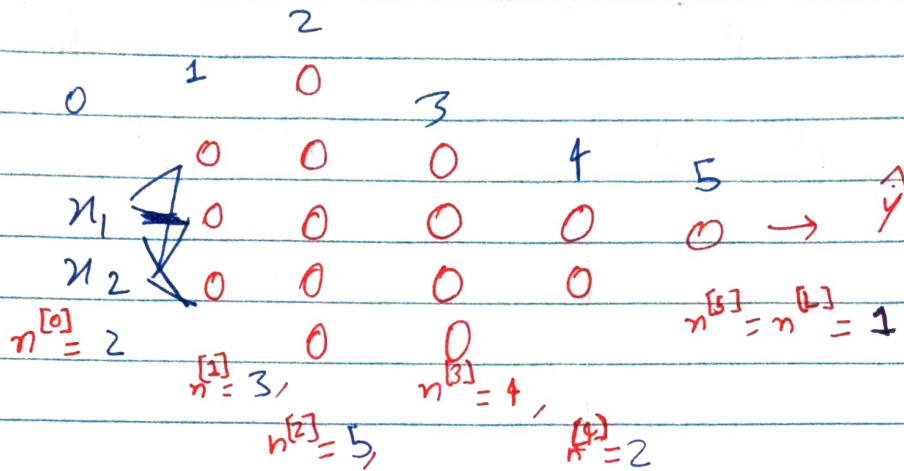
$$z^{[3]} = w^{[3]} a^{[2]} + b^{[3]}$$
$$a^{[3]} = g^{[3]}(z^{[3]})$$

$$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$$
$$\hat{y} = a^{[4]} = g^{[4]}(z^{[4]})$$

⇒ You can use a for loop here
for i in range (1 to 4) like that

Getting your matrix dimension right

$x \rightarrow$



$$z^{[1]} = W^{[1]} \cdot x + b^{[1]}$$

dimensions: $(3, 1)$ $(3, 2)$ $(2, 1)$ $(3, 1)$

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

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

dimension generally,

$$W^{[l]}: (n^{[l]}, n^{[l-1]})$$

vectorized implementation

$$z^{[l]} = W^{[l]} \cdot x + b^{[l]}$$

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

$(n^{[l]}, n^{[l-1]})$

↳ it gets fixed in broadcasting.

Forward prop for layer l

$$z^{[l]} = w^{[l]} \cdot a^{[l-1]} + b^{[l]}$$

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

vectorized:

$$z^{[l]} = w^{[l]} \cdot A^{[l-1]} + b^{[l]}$$

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

Backward prop for layer l

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$$

$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$$

Parameters vs Hyperparameters

Hyperparameters :- learning rate " α "

iterations

hidden layers " L "

hidden units $n^{[1]}, n^{[2]} \dots$

Choice of activation fn

momentum, mini batch size,

regularization parameters

Parameters :-
Weights
biases
embedding vectors

Do all of these stuff from the ML notes
I ain't going right same stuff again.

Train/Dev/Test sets

→ To train the model.
→ To tune hyperparameters.
→ To test the model.

→ Make sure that dev/test set come from same distribution as as Train set

Bias / variance

error -:

ex →	Train set -:	1 %.	15 %.	15 %.
	Dev set -:	11 %.	16 %.	30 %.
		high variance	high bias	high bias / high variance

Basic recipe for machine learning

high bias?

solⁿ →

Bigger network

Train longer

(NN architecture search)

high Variance?

dev set

Performance

↓

→ more data

→ regularization

→ Try different NN architecture

Regularization

ex:

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

here $\|w\|_2$ means the L2 norm of the vector containing all the weights which is $\sqrt{(w_1)^2 + (w_2)^2 + \dots + (w_n)^2}$, or euclidean ~~at~~ norm.

in geometry it tells the distance between a point vector & the origin.

in neural network $\therefore$

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

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

for "Frobenius norm"

Intuition on why regularization prevent overfitting?
So, think of it this way when you use regularization it decreases the weights of the model ("The value of the weight") making the model on those weights ineffective, so in the essence, now you are left with a model with less no of weights & that doesn't overfit

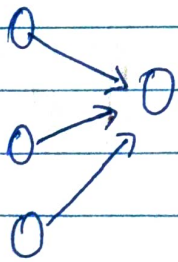
Dropout regularization

→ in dropout regularization we randomly shut off ~~few~~ ^{perceptrons} ~~weights~~ in the neural network, ~~so~~ in each pass of the gradient descent ~~so~~ that's how the forward pass & then the gradient descent is applied only on the active perceptrons & that's how we regularize the model.

And we don't shut off perceptrons at the time of inference

How dropout works?

→ Intuition,



in the dropout method, a perceptron can't rely on any one feature, so it has to spread out weights into other perceptrons.

Normalizing inputs

→ 2 steps

(i) Subtract mean:

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

$$x := x - \mu$$

(ii) Normalize variance:

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

$$x /= \sigma$$

use the same μ & σ to normalize the test set as used in the training set.

→ The model converges fast, the gradient descent steps are faster & converge to the minimum faster.

Vanishing / Exploding gradient's

→ when you are training a deep neural network your derivative can be either very very big or very very small & this makes training difficult,

⇒ let's say you have a very deep neural network with linear activation or no activation fn, then

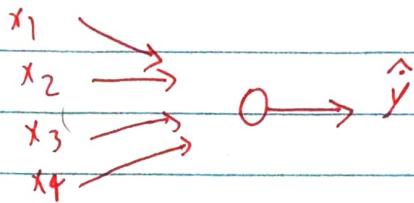
$$\hat{y} = w^{[L]} * w^{[L-1]} * w^{[L-2]} \dots w^{[3]} * w^{[2]} * w^{[1]} x$$

so, when you assume that the weights are close to (1.5) in all the layers, their multiplication for the inference becomes exponentially large example $(1.5)^L$,

& they explode but at the same time if they are around (0.8), they get squashed down & vanish $(0.8)^L$.

Weight initialization for deep networks

ex $\rightarrow$ for a single neuron



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

in order to make z not blow up or not become too small,

large $n \rightarrow$ smaller w_i

To prevent this explosion & implosion, you can set the variance of "W" to be equal to $1/n$.

$$[w_{ij}] = \text{np.random.randn}(\text{shape}) * \text{np.sqrt}\left(\frac{1}{n(l-1)}\right)$$

in ReLU activation fn you can instead set $\text{var}(w)$ to $\frac{2}{n}$,

so it becomes

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

other variants :-

tanh $\sqrt{\frac{1}{n(l-1)}}$

Numerical approximation of gradients

$$\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \approx g(\theta)$$

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001$$

$$g(\theta) = 3\theta^2 = 3$$

approx error: 0.0001

Gradient checking for a neural net

$\therefore$ Take all the parameters $w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}$ & reshape into a big vector θ .

[The cost fn $J(\theta)$ will now be the fn of " θ "] $J(\theta)$ instead of being the fn's of all the parameters weights & biases

Take $dw^{[1]}, db^{[1]}, \dots, dw^{[L]}, db^{[L]}$ & reshape into a big vector $d\theta$.
Using a for loop:
for each i :

$$d\theta_{\text{approx}}[i] = J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots)$$

$$- J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)$$

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

to check how similar they are you can take L_2 norm or euclidian distance between these vectors

$$\|d\theta_{\text{approx}} - d\theta\|_2 \quad - (i)$$

Now you can normalize it by the lengths of these vectors, by dividing by the length of these vectors

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

→ The denominator turns this formula from (i) to a ratio meaning if the weight ~~into~~ any of the vector's come out to be really small or really large.

if it's between,

10^{-7} &

10^{-5}

then it's great

anything above 10^{-3} is concerning.

Gradient checking implementation notes

→ Don't use in training - only to debug

→ If an algorithm fails grad checks, look at Component's to try to identify bug.
for example, when you look at the

vector & find that values of θ or $d\theta$ that are very far off all corresponds to

$db^{[l]}$ for some layer $[l]$, but the components for dw are quite close, ~~right?~~ that's how you

can diagnose it.

→ Remember regularization $\therefore$ if your loss includes regularization then don't forget to include it too.

→ doesn't work with dropout.

→ Run at random initialization; perhaps again after some training

Optimization Algorithms [C2W2]

Mini-batch gradient descent

vectorization allows you to efficiently compute on "m" examples.

But if "m" is very large which is common in the case of Deep learning, it still can be very slow.

That's why we need Batch SGD

We randomly pick up a baby "mini" batch from the main batch & optimize the model using that mini-batch.

Partitioning (Nq) } Best used for, small to medium dataset's (e.g., Iris, MNIST, CIFAR-10).

Like, $y = i_1 + i_2 + i_3 + \dots + i_n$
i are mini batches & y is the batch & the reshuffle & train again.

Random sampling } Massive dataset's (e.g., text corpora, "Korpathy") Pre-training or RL

Naming $\therefore x^{\{t\}}, y^{\{t\}}$

like the first mini batch would be $x^{\{1\}}, y^{\{1\}}$ $\{t\} \rightarrow$ the number of the minibatches we are using for example.

"an epoch meaning a single pass through the training set"

so the cost from a mini batch would look like,

Compute cost $J^{(t)} = \frac{1}{1000} \sum_{i=1}^1 L(\hat{y}^{(i)}, y^{(i)}) + \frac{1}{2 \times 1000}$

$$\sum_{i=1}^n ||w^{(t)}||_F^2$$

mini Batch SGD is faster, than normal Batch

Typical mini Batch sizes.

→ Try different powers of 2

$$\begin{array}{ccccccc} \rightarrow & 64 & , & 128 & , & 256 & , & 512 & - & \dots \\ \rightarrow & 2^6 & & 2^7 & & 2^8 & & 2^9 & & \end{array}$$

Exponentially weighted averages

→ it helps in computing trends, The local average or a moving average or exponentially weighted average.

assuming $V_0 = 0$,

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

for ex → here let's say, V_t is the local average at ~~that~~ current point, V_{t-1} is the local average of the previous point, θ_t is the value of the current point.

to determine how many values we are currently estimating over you can use the formula $(1/(1-\beta))$

β → Smoothing/memory coefficient

it controls how much of the past you remember vs how much you trust the current value

case 1: if $\beta=0$, $\Rightarrow V_t = \theta_t$, extreme cases

→ no memory at all, → You only care about the current value, → very noisy, → No smoothing.

Case 2: $\beta = 0.99$

$$V_t = 0.99 V_{t-1} + 0.01 \theta_t$$

→ strong memory | → very smooth
→ changes very slowly | → resistant to noise

Understanding exponentially weighted averages

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

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

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

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

⋮

$$V_0 = 0$$

Now,

$$V_{100} = 0.9 * V_{99} + 0.1 * \theta_{100} \quad \text{--- (i)}$$

Putting the value of V_{99} in (i) we get

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

$$V_{100} = (0.9)^2 * V_{98} + 0.09 * \theta_{99} + 0.1 * \theta_{100} \quad \text{--- (ii)}$$

Putting the value of V_{98} in (ii) we get,

$$V_{100} = (0.9)^2 * (0.9 * V_{97} + 0.1 * \theta_{98}) + 0.09 * \theta_{99} + 0.1 * \theta_{99}$$

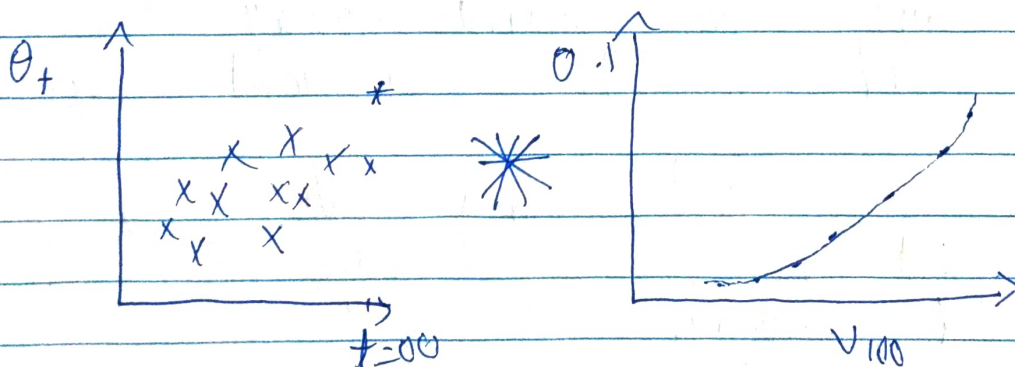
$$V_{100} = (0.9)^3 V_{97} + 0.1 * (0.9)^2 * \theta_{98} + 0.09 * \theta_{99} + 0.1 * \theta_{99}$$

& so on - - - -

so it will become this after multiple iterations

$$V_{100} = 0.1 \theta_{100} + 0.9 * \theta_{99} + 0.1 * (0.9)^2 \theta_{98} + 0.1 * (0.9)^3 * \theta_{97} + 0.1 * (0.9)^4 \theta_{96} + \dots$$

let's say you have a temperature of 100 days & you have an exponentially decaying fn, we are just multiplying these two & plotting it & summing it up
ex $\Rightarrow$



multiply the values above & summing them will give you the plot.

Q $\Rightarrow$ how many days temp is this averaging over?

$\Rightarrow$ it turns out that, $(0.9)^{10} \approx 0.35 \approx \frac{1}{e}$

so,
$$\frac{(1-\epsilon)^{\frac{1}{\beta}}}{0.9} = \frac{1}{e} \quad \text{--- (i)}$$

in this case, epsilon " ϵ " is 0.1,

so, it takes 10 days to the height of the ~~fr~~ to decay to $\frac{1}{3}$ rd of its original value, so when $\beta = 0.9$,

that it's computing exponentially weighted average that focuses on just last 10 days temp, & we say

after 10 days the weight decay is less than about the third of the current decay

So, you can put any value in formula (i) to get how many days it's taking for the weighted average.

ex $\Rightarrow$

$$\epsilon = 0.02$$

$$\frac{(0.98)^{\frac{1}{\beta}}}{0.9} = \frac{1}{e}$$

Here $\frac{1}{\epsilon} \approx 50$

implementing exponentially weighted averages

$$\begin{aligned}v_0 &= 0 \\v_1 &= \beta v_0 + (1-\beta) \theta_1 \\v_2 &= \beta v_1 + (1-\beta) \theta_2 \\v_3 &= \beta v_2 + (1-\beta) \theta_3\end{aligned}$$

it is very efficient
in memory & computational
less expensive,

what you can do is

$$v_0 = 0$$

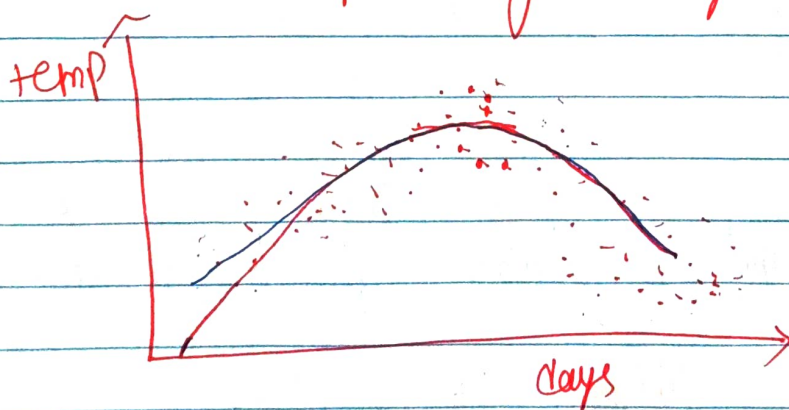
Repeat {

Get next θ_t

$$v_t := \beta v_t + (1-\beta) \theta_t$$

}

Bias correction in exponentially weighted average :-



→ When you implement EMWA, you don't get the curve in blue you get the curve in red

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

$$V_0 = 0$$

$$V_1 = \cancel{\beta V_0} + 0.02 \theta_1 \quad \text{since } V_0 = 0 \text{ \& } \theta_1 = 90$$

$$V_1 = 0.02 \times 90 = 1.8$$

in V_2 ,

$$V_2 = 0.98 V_1 + 0.02 \theta_2$$

$$= 0.98 \times 0.02 \times \theta_1 + 0.02 \times \theta_2$$

$$= 0.0196 \theta_1 + 0.02 \theta_2$$

So, V_2 will be much less ~~too~~ than θ_1 or θ_2 , which is incorrect,

So V_2 isn't

a good estimate of first 2 days of temperature of the year

There is a way to modify this estimate that makes it much better & accurate in the initial phase of the estimate,

instead of taking V_t , take

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

here it's beta raised to the power of "t" where "t" is the

data you are on.

ex,

at

$t = 2$:

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

$$\frac{v_2}{0.0396} = \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396}$$

$\Rightarrow$

~~0.0396~~

Notice that the coefficient of ~~0.0196~~ θ_1 & θ_2 adds up to denominator & it simply becomes the weighted average of θ_1 & θ_2

So as t becomes large $\beta^{(t)}$ will approach zero,

So when " t " is large enough the bias correction makes no difference,

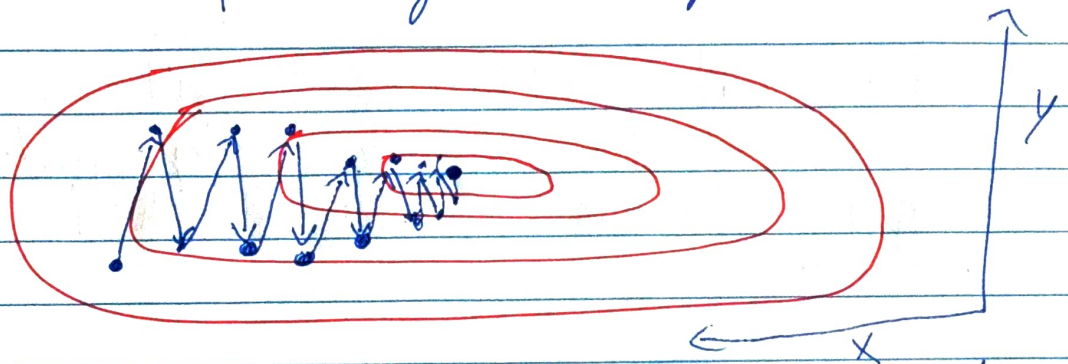
In ML people don't often bother to implement bias correction because, like ~~the~~ θ you can wait for that initial period to pass, but if you are still concerned about it, you can implement bias.

Gradient descent with "momentum"

This works faster than momentum Gradient descent,

The idea is to compute an exponentially weighted average of your gradients, & then use that gradient to update your weight instead

ex $\rightarrow$



Here the gradient descent takes a lot of steps to reach the minima, the gradient descent will slowly oscillate to the minima,

on the vertical axis (y-axis), you don't want those oscillations, you want that learning to be slower,

On the horizontal axis (x-axis), you want your learning to be faster

So, on iteration "t",

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

You compute dw, db on the current mini batch,
then compute v_{dw} & v_{db}

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

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

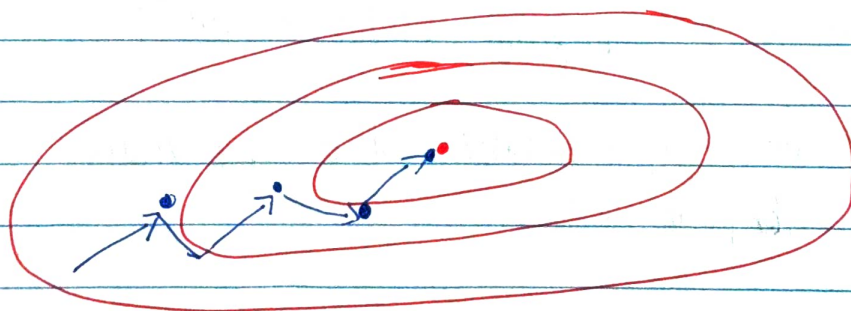
updating weights using v_{dw} & v_{db}

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

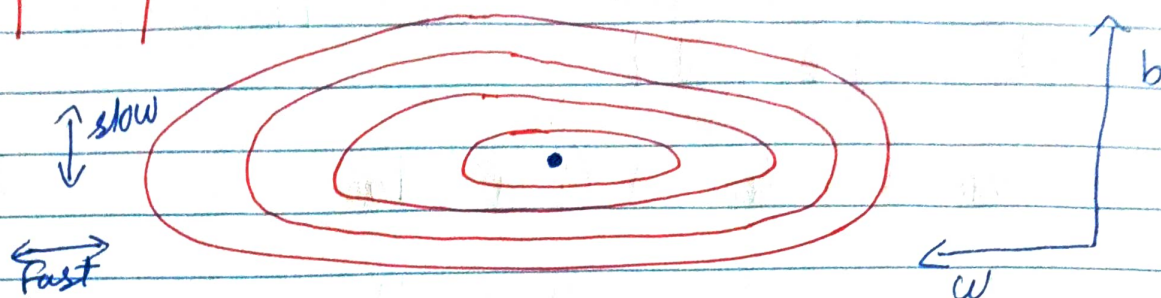
$$b = b - \alpha v_{db}$$

Here,
hyperparameters are:
 α & β | $\beta \rightarrow$ smoothing coefficient
 $\beta = 0.9$ | $\alpha \rightarrow$ learning rate

So, gradient descent with momentum will take much smaller oscillations than the vertical direction, so it's like



RMS prop



for the intuition let's say the vertical direction is parameter b & horizontal direction is parameter w ,

so you want to slow the learning in horizontal direction & increase it in vertical direction or atleast not slow it down.

On iteration t :

compute dw, db on the current mini batch

$$S_{dw} = \beta S_{dw,prev} + (1-\beta) dw^2$$

$S_{dw,prev}$ → means the previous value of S_{dw}
 $S_{db,prev}$ → " " " " " " " " S_{db}

$$S_{db} = \beta S_{db,prev} + (1-\beta) db^2$$

$$w = w - \alpha \frac{dw}{\sqrt{S_{dw} + \epsilon}}, \quad b = b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$$

so, mathematically, the S_{dw} will be smaller than S_{db} , so when we divide the parameter " b " by $\sqrt{S_{db}}$ it becomes smaller &

hence,

The movement in the vertical dimension slows down,

So here's what I learned on my own,

→ there will be n number of params in a big model

→ So how are you going to decide which one are which axis, & which one are causing the movement in "y" direction,

it turns out the parameter with big gradients are the one that cause the movement in "y" direction, so if you just shrink their gradients

by dividing them by their square root of the grad moving average value they become small & hence the movement in y direction increases.

& here " ϵ " epsilon is a very small value like " 10^{-8} ", we add epsilon because what if S_{w} is very close to zero then this operation can go ka boom. ———

Adam optimization Algorithm

It's taking momentum & RMS Prop & putting it together

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

on iteration t :

Compute dw, db using current mini batch

"Momentum
EWMA" $\rightarrow$

$$v_{dw} = \beta_1 v_{dw, prev} + (1 - \beta_1) dw,$$

$$v_{db} = \beta_1 v_{db, prev} + (1 - \beta_1) db$$

s_{dw}

RMS Prop $\rightarrow$ $s_{dw} = \beta_2 s_{dw, prev} + (1 - \beta_2) dw^2,$

$$s_{db} = \beta_2 s_{db, prev} + (1 - \beta_2) db^2$$

In the typical implementation of Adam, you also include bias correction.

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

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

$$W = W - \alpha \frac{V_{dw}^{\text{corrected}}}{\sqrt{S_{dw}^{\text{corrected}} + \epsilon}} \quad \bigg| \quad b = b - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}} + \epsilon}}$$

Hyperparameters Choice:

α : needs to be tuned
 β_1 : 0.9 "default choice"
 β_2 : 0.999 "
 ϵ : 10^{-8}

Here, β_1 is computing the mean of derivatives called the first moment.

& β_2 is used to compute exponentially weighted average of the squares, & that's called the second moment

Adam: Adaptive moment estimation.

Learning Rate decay :-

So your, ~~loss~~ model is trying to converge but it has a big learning rate it might help first when you start to do gradient descent but overtime,

you will be wandering inside a minima periphery of a minima

but to decrease the size of that periphery you need to lower the size of gradient update, hence you decrease learning rate overtime.

formula for learning rate decay is,

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

α_0 = initial learning rate

decayRate $\Rightarrow$ The rate by which you want your learning rate to decay,

Epoch number $\Rightarrow$ The number of times the model has seen the complete data.

The problem of local optima :-

Most points of zero gradient's in a cost function are saddle points, for a large neural net, then they are not local optima,

The problem with large neural net is the problem of plateau, A plateau is a region where the derivative is close to zero for a long time & it can slow the learning, but this is where the adam algo shines & can fix the slow learning process.

Hyperparameter tuning

α β $\beta_1, \beta_2, \epsilon$ # No of layers # hidden units learning rate decay mini-batch size	} The most imp are like, α , no of layers, hidden unit's then mini-batch size, β
---	---

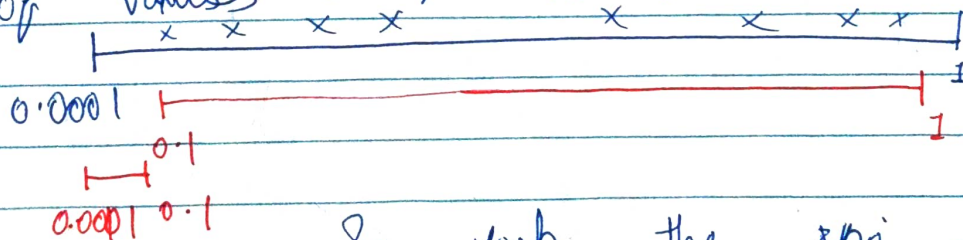
Using an appropriate scale to pick hyperparameter

So,

if you are searching for

α , = 0.0001, ..., 1

90% if you do that linearly, you will sample 90% of values straight between 0.1 & 1

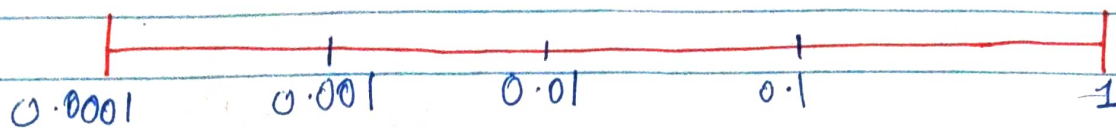


sampled will be even smaller & yeah the no of values between

0.0001 & 0.001, So you can't do that,

instead, you can search for hyperparameter on a log scale

Using the log scale, you will have the distribution like,



So sampling at random on logarithmic scale
Now, you have more resources dedicated searching
between 0.0001 & 0.001, & so

So, in python this is how you will implement it,

$$r = -4 * \text{np.random.rand}() \in r \in [-4, 0]$$

$$\alpha = 10^r \leftarrow \alpha \in [10^{-4}, 10^0]$$

Hyperparameters for exponentially weighted averages

$$\beta = 0.9 \quad \dots \quad 0.999$$

$\downarrow$
10

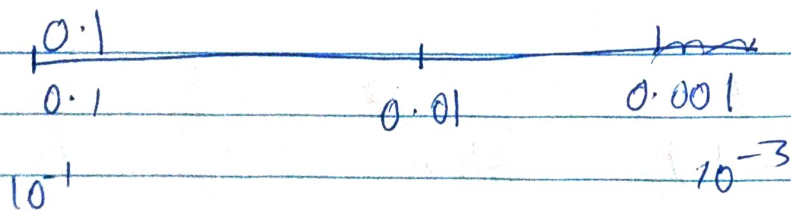
$\downarrow$
1000

$\overbrace{0.9 \quad 0.99 \quad 0.999 \quad 0.9999}^{xx \quad xx \quad x \quad xx}$

use logarithmic scale
instead of this one.

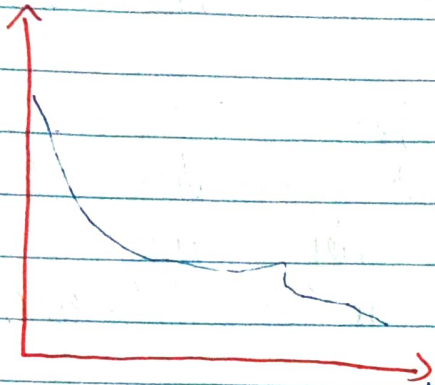
$$1 - \beta = 0.1 \quad \dots \quad 0.001$$

$\underbrace{0.9 \quad 0.99 \quad 0.999}_{\downarrow}$



Hyperparameters tuning in practice: Pandas vs. Caviar

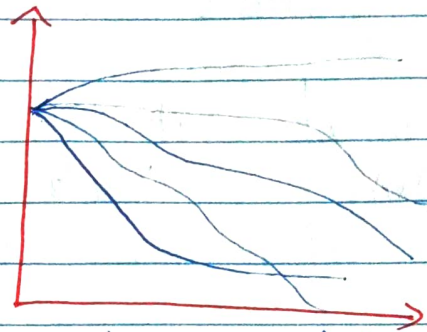
Babysitting one model



→ Picking up one model & then training it several days or hours while continuously changing its learning rate

"Pandas approach"

Training many model in parallel.



→ Training multiple models with different set of parameters all in //

"Caviar approach"

Batch Normalization:

Normalizing activations in a network

So, in a simple NN we normalize the input feature vector's, but when it comes to deep neural network, you can just normalize the pre activations inside of the hidden layers which act's as an input for another layer.

implementing Batch norm

given some pre activation's
 $z^{(1)} \dots z^{(m)}$

$$\mu = \frac{1}{m} \sum_i z^{(i)}, \quad \sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$

$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad \left\{ \begin{array}{l} \text{Here } \epsilon \text{ epsilon is a} \\ \text{very small number for} \\ \text{numerical stability} \end{array} \right.$$

we don't always want the, $\mu=0, \sigma^2=1$
so what we do is we compute

$z_{\text{tilde}}^{(i)}$

 $\leftarrow \{ z_{\text{tilde}}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta \}$ Here γ is the standard deviation & β is the mean

γ, γ & β are trainable parameters

by gradient descent,

Notice bias "b" that you can simply get rid of because in batch norm " β " it does

the same thing.

Batch norm at test time

→ Batch norm processes the data one minibatch at a time, but at test time you will give it 1 example at a time,

so,

you don't have mean & variance over a mini batch,

you can fix this by computing μ & σ^2 , ~~for~~ estimate using exponentially weighted averages across the mini batches

$$\mu_0 = 0$$

$$\mu_1 = \beta \mu_0^{[weighted]} + (1-\beta) \theta$$

here, $\theta \rightarrow$ is the value of a mean of a mini-batch

$$\sigma_0^2 = 0$$

$$\sigma_1^2 = \beta \sigma_0^2 + (1-\beta) Z$$

$Z \rightarrow$ is the value of the variance of a mini batch

& you can use,

$$\mu_n^{[weighted]} \text{ \& } \sigma_n^{2[weighted]}$$

$\mu_n^{[weighted]}$ & $\sigma_n^{2[weighted]}$ insted in the test time

Softmax regression

Softmax lets you recognize ~~two~~ ~~a class~~ in multiple classes.

let's

say you have

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

the $n^{[L]} \Rightarrow 4$ } no of units in the last layer,

$$\text{The } z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]}$$

Activation function:-

$$t = e^z \quad (4, 1)$$

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

Loss function:-

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

$$\text{Cost} \Rightarrow \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

$$\text{Backprop: } dz^{[L]} = \hat{y} - y$$

$$dz^{[L \text{ Last Layer}]}$$