Model and Cost function

Not's :

m : no. of training ex.s

x's : input var.s / features

y's : output var.s / ~~features~~ target var. that we are trying to predict

(x, y) : one training ex. / observ^n

$(x^{(i)}, y^{(i)})$: i^{th} training ex. / observ^n

$(x^{(i)}, y^{(i)})$; $i = 1, 2, \ldots, m$: training set

i : index into the training set

X : space of I/P val.s

Y : space of O/P val.s

n : no. of features

$$\boxed{\text{Hypothesis} : h_\theta(x) = \theta_0 + \theta_1 x}$$

θ_i's : para. of model

h maps from x's to y's

Gradient Descent

Now we have a hypoth. funⁿ and a way of measuring how well it fits into the data. Now we need to estimate the para.s in the hypoth. funⁿ. That's where gradient descent comes in.
(θ_0, θ_1)

Gradient Descent is a first-order iterative optimizⁿ algo. for finding a local min. of a differentiable funⁿ.
It tweaks the funⁿ(s) para.s iteratively to minimize a given funⁿ to its local min.

grad. desc. is a gen. algo. i.e., it applies not only just to cost funⁿ

Outline :

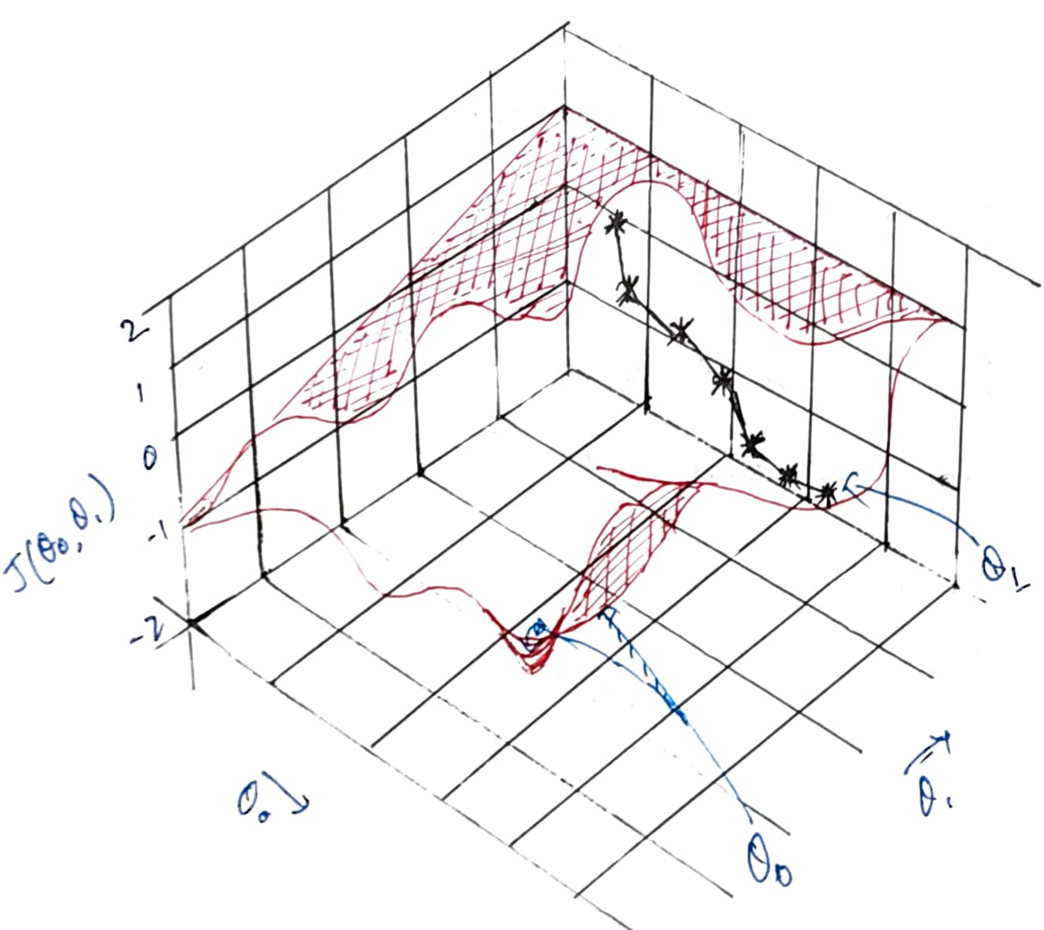
- start w/ some θ_0, θ_1 (any val. say, $\theta_0 = 0, \theta_1 = 0.5$)
- keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a min.

(context : a pt. on a slopy park w. hills like windows xp wallpaper) In grad. desc. what happens is that you look around yourself 360° and ask "if you were to take a step in some direcⁿ and get downhill asap, what direcⁿ do you take that step in ?
You take that step and repeat until you've reached the (lowest) local optimum.
So, 2 things to notice are : direcⁿ of your step, size of your step.

Imagine that we graph our hypoth. funⁿ based on its fields θ_0 and θ_1 (actually we are graphing the cost funⁿ as a funⁿ of the para. estimates). We are not graphing x and y itself, but the para. range of our hypoth. funⁿ and the cost resulting from selecting a particular set of para.s.

We put θ_0 on the x-axis and θ_1 on the y-axis, w the cost funⁿ on the z-axis. The pt.s of our graph will be the result of the cost funⁿ using our cost funⁿ w those specific θ para.s

The graph below depicts such a setup



We will k/ that we have succeeded when the cost fun^n is at the very bottom of the pits in our graph i.e., when its val. is the min.
The blue arrow shows the min. pt.s in the graph.

The way we do this is by taking the derivative (the tangential line of a fun^n) of our cost fun^n.
The slope of the ~~derivative~~ tangent is the derivative at that pt. & it will give us a direc^n to move towards. We make steps down the cost fun^n in the direc^n w the steepest descent. The size of each step is determined by the para. α, wh. is called the Learning rate.

for ex., the diff. b/w each 'star' in the graph above represents a step determined by our para. 'α'. The smaller α would result in a smaller step and the larger α in a larger step.
(contd.)
The direc^n in wh. the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$.
Depending on where 1 starts graph, one could end up at diff. pt.s (local optima).

Gradient Descent Algo

repeat until convergence :

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

where

$j = 0, 1$ represents the feature index no.

$:=$ assignment operator

$a := b$ assign val. of b to a

$a := a + 1$ ✓

$=$ truth assert

$a = b$ assert that val. of a & b is equal

$a = b + 1$ ✗

α is called Learning Rate

it controls how big / small step we take downhill w creating descent.

$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ is a partial derivative term.

it determines the slope of our descent

also in grad. desc., the parameters $\theta_1, \theta_2, \ldots \theta_n$ are updated simultaneously (& not 1st θ_0 then θ_1)

$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ ✓

$temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$

$\theta_0 := temp0$

$\theta_1 := temp1$

$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ ✗

$\theta_0 := temp0$

$temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$

$\theta_1 := temp1$

bc this will change the val. of θ_2 for that respective θ_0

Gradient Descent Intuition

in math, the partial derivative $\frac{\partial}{\partial x}$ of a fun^n of several var.s is its derivative w.r.t. (only) 1 of those var.s, w the others held const. while in total derivative $\frac{d}{dx}$, all the var.s are allowed to vary.

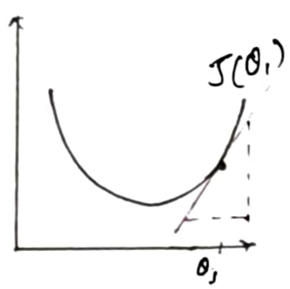
for intu^n, let us consider a fun^n of only 1 para θ_1.

Cost fun^n : $\min_{\theta_1} J(\theta_1)$

Grad. desc : repeat until convergence :

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

then,

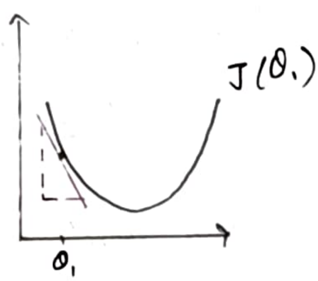


the slope of tangent is +ve

so, $\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$

$\theta_1 := \theta_1 - \alpha \cdot$ (~~α is~~ some +ve no.)

or, θ_1 is moved towards left to obtain local optimal min.



here, the slope of tangent is -ve

so, $\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$

$\theta_1 := \theta_1 - \alpha \cdot$ (some -ve no.)

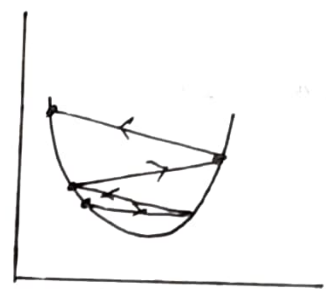
$\theta_1 := \theta_1 + \alpha \cdot$ (some +ve no.)

or, θ_1 moves towards right to obtain local optimal min.

we see that regardless of the slope's sign for $\frac{d}{d\theta_1} J(\theta_1)$, θ_1 eventually converges to it's min. val.

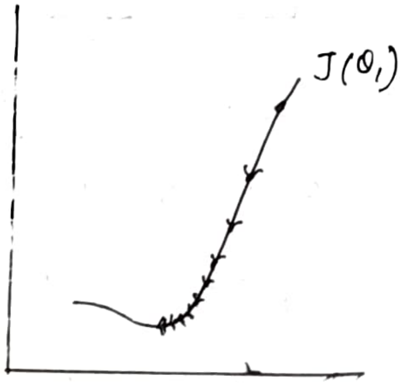
On a side note, we should ~~ensure~~ adjust 'α' to ensure that the grad. desc. converges in a reasonable time.

If α is too small, grad. desc. can be slow.



If α is too large., grad. desc. can overshoot the min. It may fail to converge or even diverge.

by def. local min. is when $\frac{d}{d\theta_1} J(\theta_1) = 0$ (slope = 0).



grad. desc. will converge even when α is fixed.

this is bc as we approach the local min, the slope gets steeper i.e., derivative automatically becomes smaller.

so, $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ is smaller and hence we do not need to dec. 'α'

Gradient Descent for Linear Regression

here we will put together grad. desc. w our cost funⁿ and that will give us an algo for Linear Regression, or putting a straight line to our data.

grad. desc. algo :

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 0$ & $j = 1$)

}

Linear Regression Model :

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Linear hypoth.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

sq.ed error cost funⁿ

Now, we need to calc. $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right)^2$$

$$= \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^i - y^i \right)^2$$

so, for

$$j=0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right)$$

(partial diff. w.r.t θ_j)

$$j=1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right) \cdot x^i$$

putting them in grad. desc. algo

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

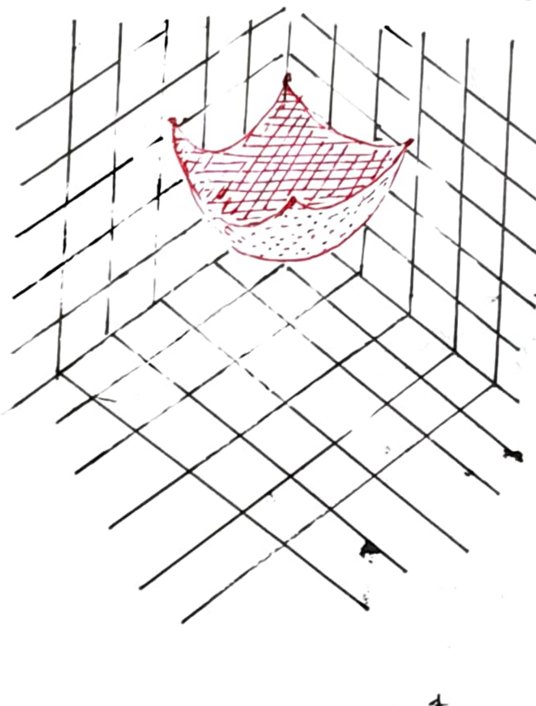
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

This algo is also k/a Batch grad. desc. (ig) where Batch means that every step of grad. desc. uses all training ex.s $\sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right)$

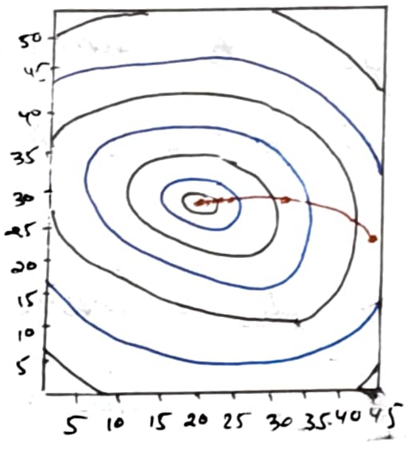
However, some algo.s also use subsets of training ex.s and not all of them.

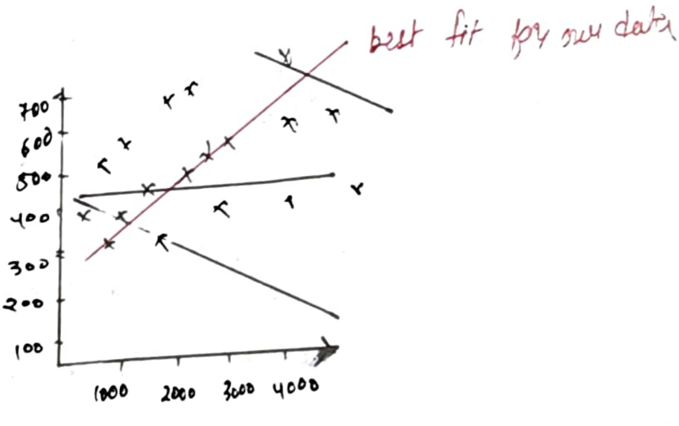
If we start w a guess for our hypoth. & then repeatedly apply grad. desc. eqn's, our hypoth. will become more accurate.

* the cost funn for Linear Regression is always going to be a Convex funn (bow-shaped funn) and so, it doesn't have the local optimum except the one global optimum.
so, the grad. desc. will always converge to the 1 local optimum.



$J(\theta_0, \theta_1)$ funn of paras θ_0, θ_1





Gradient Descent for Multiple Variables

hypoth : $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$

Para.s : $\theta_0, \theta_1, \theta_2, \ldots, \theta_n = \theta$

$\theta = [\theta_0, \ldots, \theta_n]$ is an $n+1$ dimension vec.

Cost fun^n : $J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

or, $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

Gradient Descent

for single feature / var,
for $n = 1$

Repeat

{

$$\theta_0 := \theta_0 - \alpha \overbrace{\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)}^{\frac{\partial}{\partial \theta_0} J(\theta_0)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x^{(i)}$$

$x_j^{(i)}$

(simultaneously update θ_0, θ_1)

}

The gradient descent equaⁿ itself is generally the same form, we just have to repeat it for our 'n' features.

$n \geq 1$.

repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

(simultaneously update θ_j for $j = 0, 1, \ldots, n$)

}

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_0^{(i)}$$

sim. to prev θ_0 (for $n = 1$) as for convenience, we have assumed $x_0 = 1$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_1^{(i)}$$

sim to prev θ_1 (for $n = 1$)

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_2^{(i)}$$

Gradient Descent in Practice I : Feature Scaling

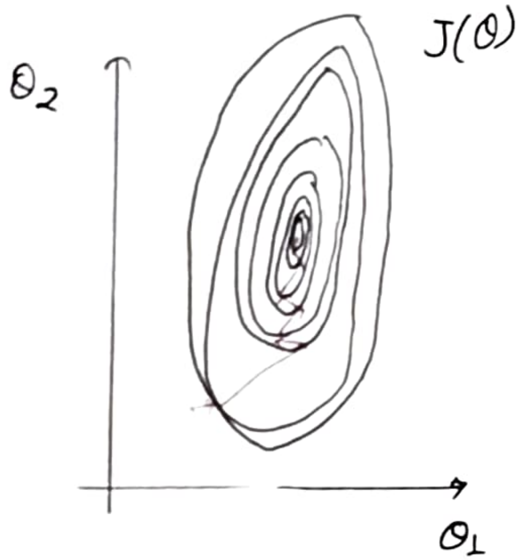
* ~~Idea~~ feature scaling can make grad. desc. run much faster and converge in a lot fewer iter's.

Idea: Put diff. features on a similar scale of val.s

ex:

x_1 = size (0 - 2000 ft^2)

x_2 = no. of bedrooms (1-5)

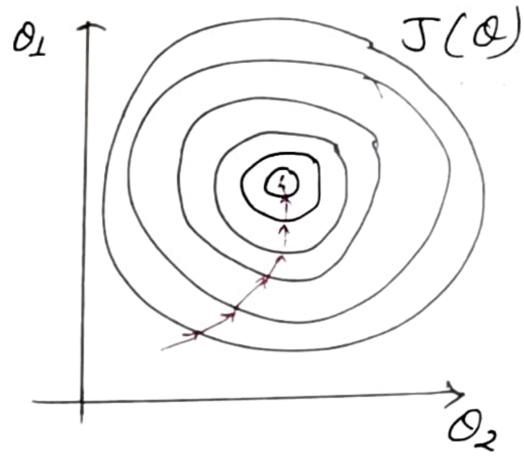


Here, contours of $J(\theta)$ are tall/ skinny/ skewed ellipses.
On running grad. desc., grad. desc. will oscillate back & forth until it gets to the local min. thus taking long time to reach the local min.

in these settings, a useful thing to do is scale the features

$$x_1 = \frac{\text{size } (ft^2)}{2000}$$

$$x_2 = \frac{\text{no. of bedrooms}}{5}$$



on scaling contours of $J(\theta)$ become much less skewed looking more like circles.
On running grad desc, grad desc takes a much more directed path rather than a convolutated path to get to local min.
so, by scaling the features so that there are consumer range of val.s (here $0 \leq x_1/x_2 \leq 0$), implementⁿ of grad. desc. converges much faster.

We can speed up grad. desc. by having each of our i/p val.s in roughly same range.
This is bc θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the var.s are very uneven. (as shown in ~~fet~~ left diag)

The way to prevent this is to modify the ranges of our i/p var.s so that they are all roughly the same. Ideally:

$$-1 \leq x_i \leq 1$$

not necessarily this range, but a range in gen. wh is scalable / compatible / not too big or small to plot

ex:

$$0 \leq x_1 \leq 3 \quad ✓$$

$$-2 \leq x_2 \leq 0.5 \quad ✓$$

$$-100 \leq x_3 \leq 100 \quad ✗$$

$$-0.0001 \leq x_4 \leq 0.0001 \quad ✗$$

* It's fine if features are not exactly in the same range (or scale) as long as they are close enough (to the grad. desc.)
The goal is to get all i/p var.s into roughly 1 of these ranges, give or take a few.

Feature Scaling involves dividing the i/p var.s by the range (max. val − min val) of the i/p var, resulting in a new range of just 1.

Mean Normalization.

Replace x_i w $x_i - \mu_i$ to make features have approx.ly zero mean

Implementing both feature scaling & mean normalizn :

$$\boxed{x_i := \frac{x_i - \mu_i}{s_i}}$$

where, μ_i = avg. val of feature (i)

s_i = range of val.s (max − min)

or standard devian for feature (i)

so, here,

$$x_1 \leftarrow \frac{x_1 - \mu_i}{s_i} = \frac{\text{size} - 1000}{2000} \quad \text{(say)} \rightarrow -0.5 \leq x_1 \leq 0.5$$

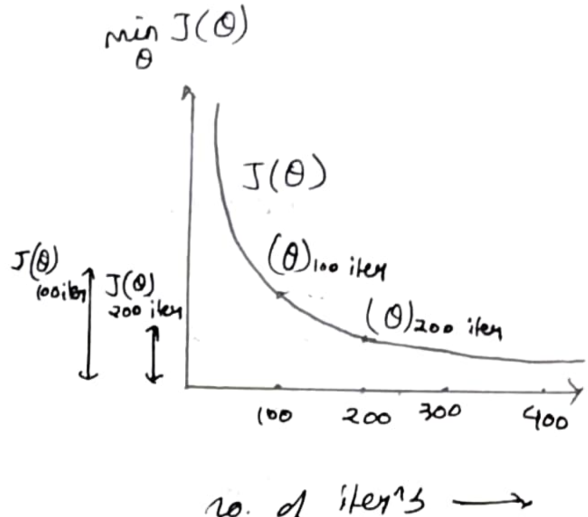
$$x_2 \leftarrow \frac{\text{no. of bedrooms} - 2}{54} \quad \text{(say)} \rightarrow -0.5 \leq x_2 \leq 0.75$$

Gradient Descent in Practice 2 : Learning Rate (α)

terminology : Debugging + how to make sure grad. desc. is working correctly.

→ how to choose learning rate α ?

→ If grad. desc. is working correctly, $J(\theta)$ should dec. after every iter^n.



→ When $J(\theta)$'s val. doesn't go down much after iter^n's (i.e. when the curves start flattening), it is safe to assume that grad. desc. has converged bc the cost fun^n isn't going down much.

→ The no. of iter^n's grad. desc. takes to converge for a phy. al applic^n can vary a lot (maybe 30 or 30000 ~~times~~. iter^n's) & its difficult to predict how many iter^n's grad. desc. would need to converge.
So it is often done w plotting such a graph b/w cost fun^n as we inc. no. of iter^n's.

→ Debugging gradient descent :

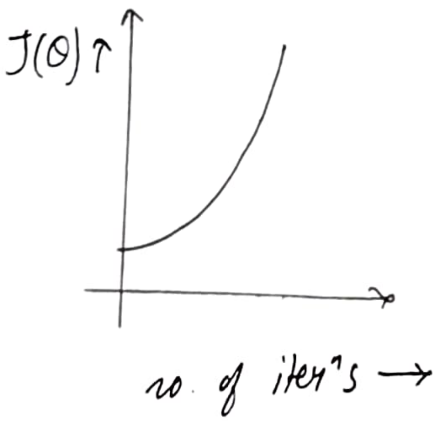
Make a plot w no. of iter's on X-axis. Now plot the cost function, $J(\theta)$ over the no. of iter's of gradient descent.
If $J(\theta)$ ever inc.es, then you probably need to dec. α.

→ Automatic Convergence Test :

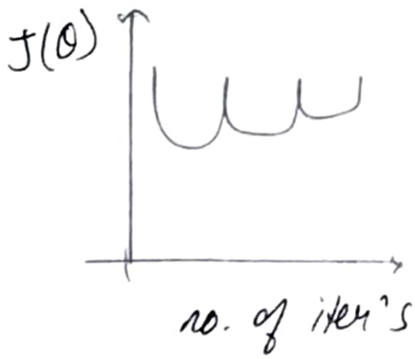
Declare convergence if $J(\theta)$ decreases by less than E in 1 itera^n, where E is some small val. (such as 10^{-3} or $\mathcal{E}$).
However in practice it's difficult to choose this threshold val.

this algo tells us if grad. desc. has converged.

→ Making sure grad. desc. is working correctly:



a plot like this clearly suggests grad. desc. isn't working correctly.
This is bc α is too large & it overshoots the min.
(dec. val. of α)

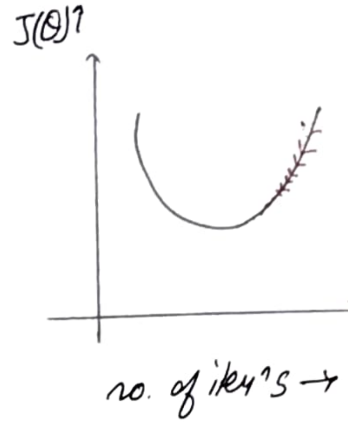
- 

a plot like this is also bc α is too large.

(dec. val. of α)

- for sufficiently small α, J(θ) should dec ∀ iter^n.

but if α is too small, grad. desc. can be too slow to converge.



→ Summary

- If α is too small : slow convergence
- If α is too large : J(θ) may not dec. on ∀ iter^n ; it may not even converge.

(slow converge is also possible but usually not the case).

To choose α, try (plotting for following vals of α until best fit is found).

..., 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, ...

Vectorization

Vectoriz^n is basically th art of getting rid of explicit 'for loops' from code.

a simple ex. would be multiplying 2 matrices using * operator instead of looping through all el.s

ex: $h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^T x$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$; $x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$

Unvectorized Implementation

```
prediction = 0.0;
for j = 1:n+1,
    prediction = prediction
                 + theta(j) * x(j);
end;
```

Vectorized Implement^n

```
prediction = theta' * x
```

Grad. Desc.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

$$\theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_0^{(i)}$$

$\vdots$

$$\theta_2 = \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_2^{(i)}$$

Vectorized Implementaⁿ:

$$\underbrace{\theta}_{\in \mathbb{R}^{n+1}} = \underbrace{\theta}_{\in \mathbb{R}^{n+1}} - \underbrace{\alpha}_{\substack{\in \mathbb{R} \\ \text{(is a no.)}}} \underbrace{\delta}_{\in \mathbb{R}^{n+1}}$$

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$$

where, $$\delta = \underbrace{\underbrace{\frac{1}{m}}_{\text{no.}} \sum_{i=1}^{m} \underbrace{\left(h_\theta(x^{(i)}) - y^{(i)}\right)}_{\in \mathbb{R}^n \text{ no.}} \underbrace{x^{(i)}}_{\text{vec.} \in \mathbb{R}^{n+1}}}_{\text{vec.}}$$

$$\delta = \begin{bmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \end{bmatrix}$$

$$\delta_0 = \frac{1}{m} \sum \left(h(x^{(i)}) - y^{(i)}\right) \cdot x_0^{(i)}$$

$$\delta = \left(h_\theta(x^{(1)}) - y^{(1)}\right) x^{(1)} + \left(h_\theta(x^{(2)}) - y^{(2)}\right) x^{(2)} + h_\theta(x^{(3)} - y^{(3)}) x^{(3)}$$