Gradient descent algorithm:

1. Full Batch gradient descent algorithm
2. Mini batch gradient descent algorithm
3. Stochastic gradient descent algorithm

In Full Batch gradient descent algorithm, we use whole data at once to compute the gradient, whereas in stochastic we use samples while calculate the gradient.

On the basis of different techniques:

1. First order differentiation
2. Second order differentiation

Challenges in executing gradient descent algorithm:

1. Data challenges
2. Gradient challenge
3. Implementation challenges

Data Challenges:

Variants of gradient descent algorithm:

1. Vanilla gradient descent algorithm.
2. Gradient descent momentum
3. ADAGRAD
4. xADAM

### Gradient Descent Procedure

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

coefficient = 0.0

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

cost = f(coefficient)

or

cost = evaluate(f(coefficient))

The derivative of the cost is calculated. The derivative is a concept from calculus and refers to the slope of the function at a given point. We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

delta = derivative(cost)

Now that we know from the derivative which direction is downhill, we can now update the coefficient values. A learning rate parameter (alpha) must be specified that controls how much the coefficients can change on each update.

coefficient = coefficient – (alpha \* delta)

This process is repeated until the cost of the coefficients (cost) is 0.0 or close enough to zero to be good enough.

You can see how simple gradient descent is. It does require you to know the gradient of your cost function or the function you are optimizing, but besides that, it’s very straightforward. Next we will see how we can use this in machine learning algorithms.



gradientR<-function(y, X, epsilon,eta, iters){

epsilon = 0.0001

X = as.matrix(data.frame(rep(1,length(y)),X))

N= dim(X)[1]

print("Initialize parameters...")

theta.init = as.matrix(rnorm(n=dim(X)[2], mean=0,sd = 1)) # Initialize theta

theta.init = t(theta.init)

e = t(y) - theta.init%\*%t(X)

grad.init = -(2/N)%\*%(e)%\*%X

theta = theta.init - eta\*(1/N)\*grad.init

l2loss = c()

for(i in 1:iters){

l2loss = c(l2loss,sqrt(sum((t(y) - theta%\*%t(X))^2)))

e = t(y) - theta%\*%t(X)

grad = -(2/N)%\*%e%\*%X

theta = theta - eta\*(2/N)\*grad

print(sqrt(sum(grad^2)))

if(sqrt(sum(grad^2)) <= epsilon){

break

}

}

print("Algorithm converged")

print(paste("Final gradient norm is",sqrt(sum(grad^2))))

values<-list("coef" = t(theta), "l2loss" = l2loss)

return(values)

}

ols <- solve(t(x)%\*%x)%\*%t(x)%\*%y

y = rnorm(n = 10000, mean = 0, sd = 1)

x1 = rnorm(n = 10000, mean = 0, sd = 1)

x2 = rnorm(n = 10000, mean = 10, sd = 1)

x3 = rnorm(n = 10000, mean = 0, sd = 10)

x4 = rnorm(n = 10000, mean = 200, sd = 100)

x5 = rnorm(n = 10000, mean = 5, sd = 5)

df = data.frame(x1,x2,x3, x4,x5)

gdec.eta1 = gradientR(y = y, X = df, eta = 100, iters = 1000)

**Gradient Descent with Math.**

This story I wanna talk about a famous machine learning algorithm called *Gradient Descent* which is used for optimizing the machine leaning algorithms and how it works including the math.

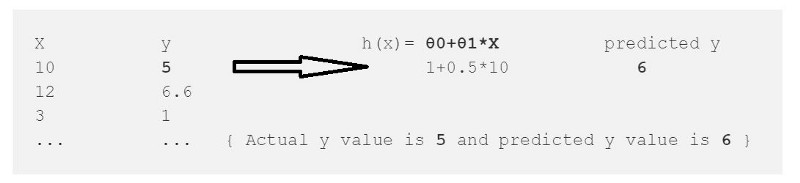
From [chapter 1](https://medium.com/@madhusanjeevi.ai/complete-linear-regression-with-math-edb05500e7ee) we know that we need to update *m* and *b* values, we call them **weights** in machine learning. Lets alias *b* and *m* as − *θ0* and *θ1* (theta 0 and theta 1 ) respectively.

First time we take random values for *θ0* and *θ1*, and we calculate y

**y = θ0+θ1\*X**   
In machine learning we say hypothesis so **h(X) = θ0+θ1\*X**

h(X)=y but this y is not actual value in our data-set, this is predicted y from our hypothesis.

For example lets say our data-set is something like below and we take random values which are **1** and **0.5** for **θ0** and **θ1** respectively.

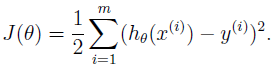


From this we calculate the error which is

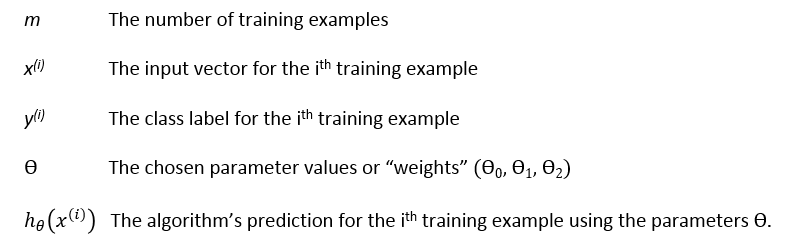
error = (h(x)-y)² --> (Predicted - Actual)²   
error = (6-5)² = 1

² is to get rid of negative values (what if Actual y=6 and Py=5)

we just calculated the error for one data point in our data-set , we need to repeat this for all data points in our data set and sum up the all errors to one error which is called ***Cost Function ‘J(*θ)’** in machine learning.



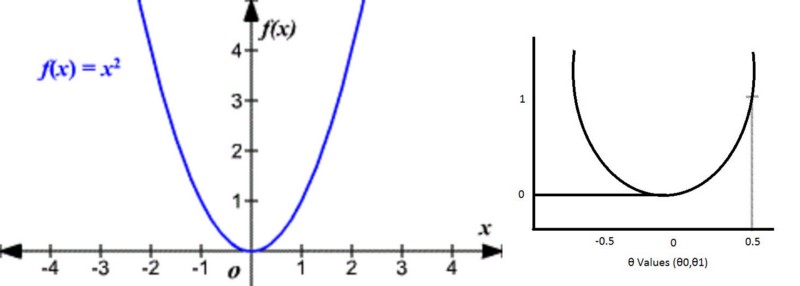
Cost Function.



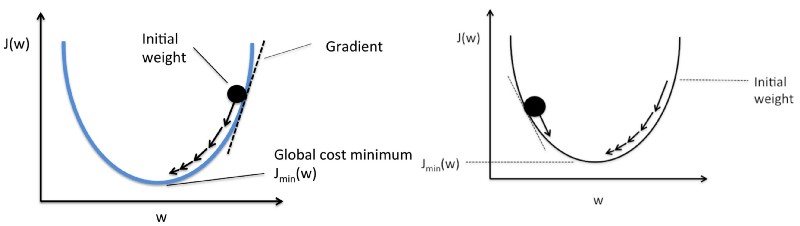
Our goal is to minimize the cost function (error) **we want our error close to zero Period.**

we have the error **1** for first data-point so lets treat that as whole error and reduce to zero for sake of understanding.

for (h(x)-y)² function we get always positive values and graph will look like this(Left) and lets plot the error graph.



Here is the gradient descent work comes into the picture.



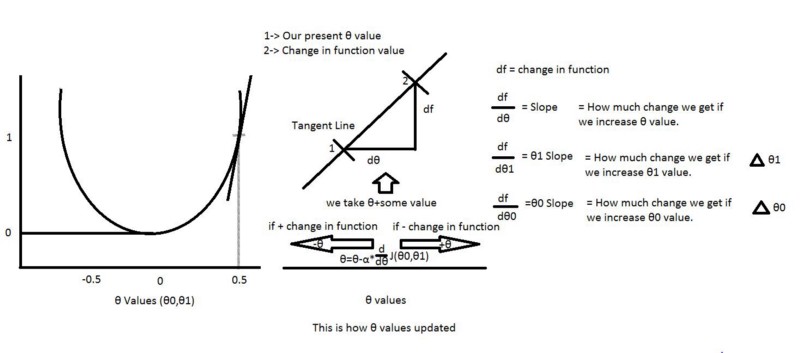
+**θ values (Left), -θ values(Right)**

By taking the little steps down to reach the minimum value (bottom of the curve) and changing the **θ** values in the process.

How does it know how much value it should go down???

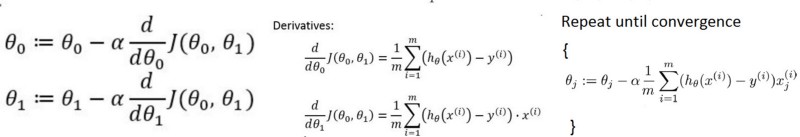
The answer is in Math.

1. It draws the line(Tangent) from the point.
2. It finds the slope of that line.
3. It identifies how much change is required by taking the partial derivative of the function with respective to **θ**
4. The change value will be multiplied with a variable called **alpha**(learning rate) *we provide the value for alpha usually 0.01*
5. It subtracts this change value from the earlier **θ** value to get new **θ** value .



From above picture we can define our **θ0 and θ1.**

And alpha here is a learning rate usually we give 0.01 but it depends, it tells how big the step-size is towards reaching the minimum value.

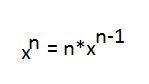


**θ0 and θ1 values(Left),more than two θ’s (Right)**

Again we know our J(**θ0,θ1)** so if we apply this to above equations for **θ0** and **θ1**, we get our new **θ0 and θ1** values**.**

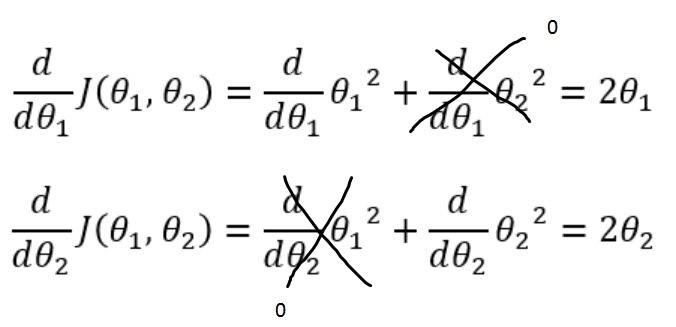
How to calculate the derivatives???

For example f(x) =x² → df/dx=2x How ???

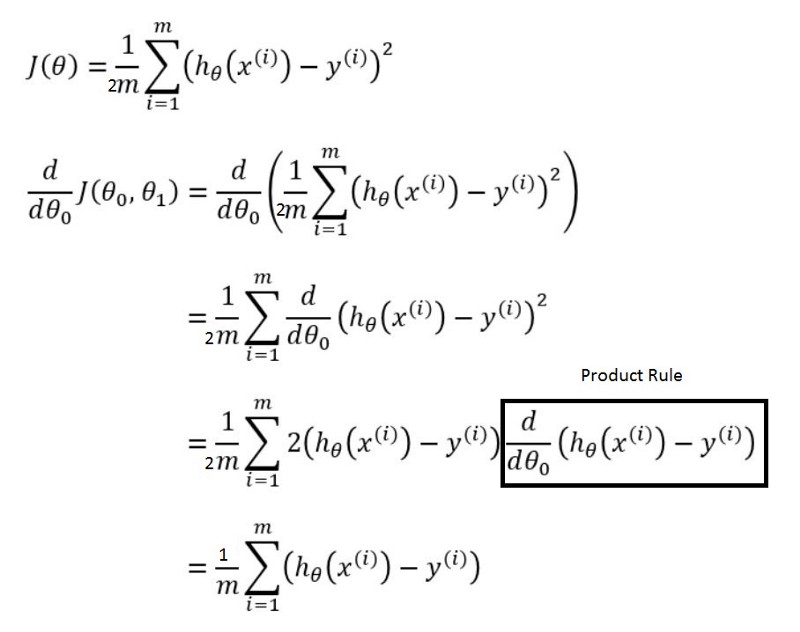


How to calculate the partial derivatives???

its same as calculating derivatives but here we calculate the derivative with respective to that value , others are constants (so d/dx(constant)=0)

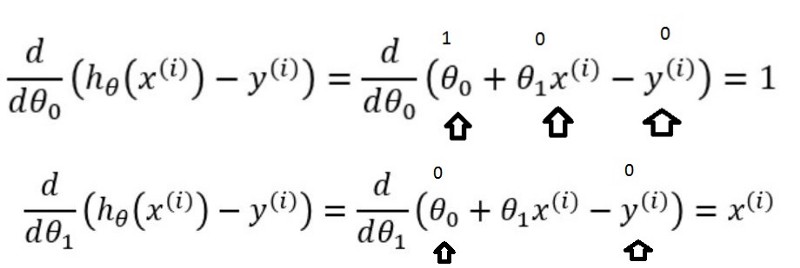


The same thing we can apply for calculating partial derivative with respective to **θ0** and **θ1**.



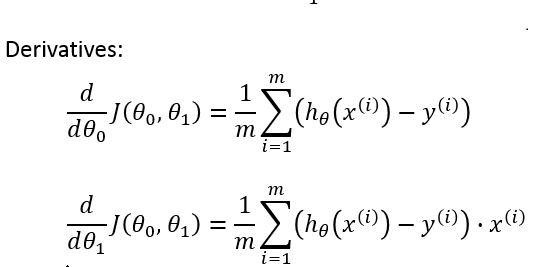
How come that box drawn disappeared in the next step above? just wait and see.

For calculating partial derivative with respective to **θ1** is also same as above except one little part is added



**θ0 box disappeared because value is 1 (Top)**

So Final picture is



Final **θ0** and **θ1**values

### **Stochastic Gradient Descent**

Gradient descent can often have slow convergence because each iteration requires calculation of the gradient for every single training example.

If we update the parameters each time by iterating through each training example, we can actually get excellent estimates despite the fact that we’ve done less work.

For stochastic gradient descent, thus:



Where i is each row of the data set.

This the stochastic gradient descent algorithm proceeds as follows for the case of linear regression:

#### Step 1

Randomly shuffle the data

#### Step 2

repeat {

for

*i*:=1,⋯,*N*{

*θ*:=*θ*−*η*∇*J*(*θ*)*i*

}

}

stoch\_gradDescent <- function(x, y, alpha, n){

# merge x and y to enable accurate random sampling

xy <- data.frame(cbind(y,x))

# theta MUST be 3 columns: must match width of x matrix

# set matrix theta and set all elements = 0 to start with

theta <- matrix(c(0, 0, 0), nrow = 1)

# Initialize a matrix to store values of theta for each iteration

thetaIter <- matrix(NA, nrow = n, ncol = 3)

# set seed value for random sampling

set.seed(42)

# now iterate using mini batches of randomly sampled data, updating theta each step

for (i in 1:n) {

# randomly sample 5 items from the combined xy data frame

xysamp <- as.matrix( xy[sample(nrow(xy), 5, replace = TRUE), ] )

# isolate 'x' component of random samples

xsamp <- as.matrix(xysamp[,2:4])

# isolate 'y' component of random samples

ysamp <- as.matrix(xysamp[,1])

# update theta using mini batches

# theta <- theta - 0.001 \* g\_part\_deriv(xsamp, ysamp, theta)

theta <- theta - alpha \* g\_part\_deriv(xsamp, ysamp, theta)

# save the theta values for iteration i to a matrix for future plotting

thetaIter[i,] <- theta

} # end for loop

return(thetaIter)

}

result1 <- stoch\_gradDescent(x, y, 0.001, 500)

