Name: Mrinmoy M. Dutta
Reg ID: SIRSS2103
Assignment No. 6

Q1. Calculate/ derive the gradients used to update the parameters in cost function optimization for simple linear regression.

Model Representation

The goal of most machine learning algorithms is to construct a model: a hypothesis that can be used to estimate Y based on X. The hypothesis, or model, maps *inputs* to *outputs.* So, for example, say I train a model based on a bunch of housing data that includes the size of the house and the sale price. By training a model, I can give you an estimate on how much you can sell your house for based on it's size. This is an example of a regression problem — given some input, we want to predict a continuous output.

The hypothesis is usually presented as:

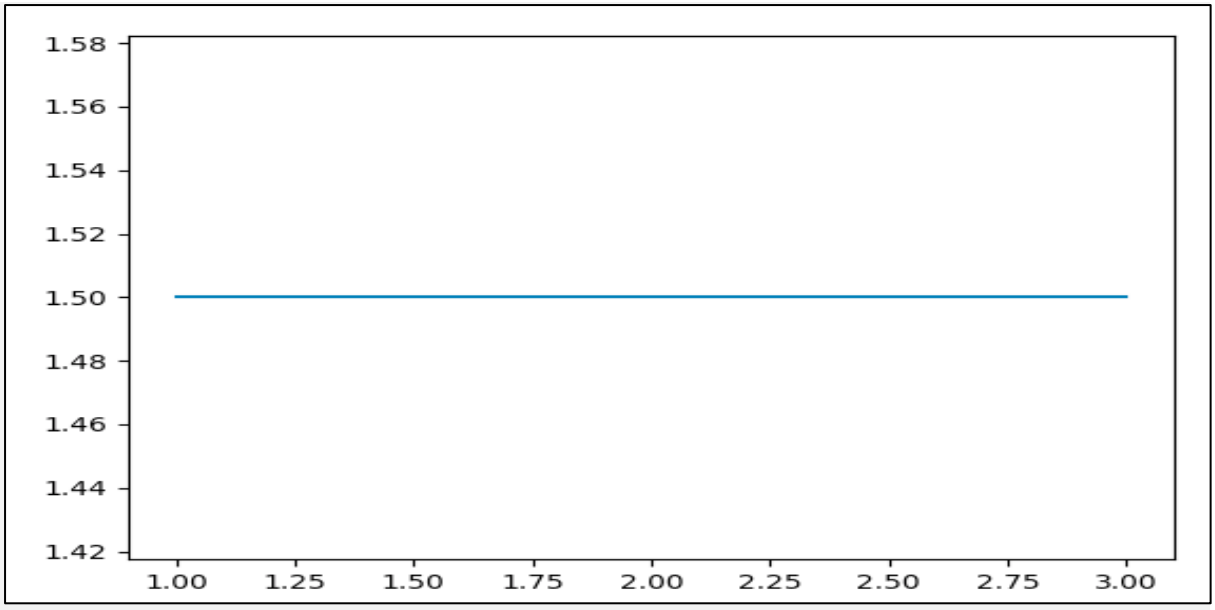
$$h_\theta(x) = \theta_0 + \theta_1 x$$

The theta values are the *parameters.*

Some quick examples of how we visualize the hypothesis:

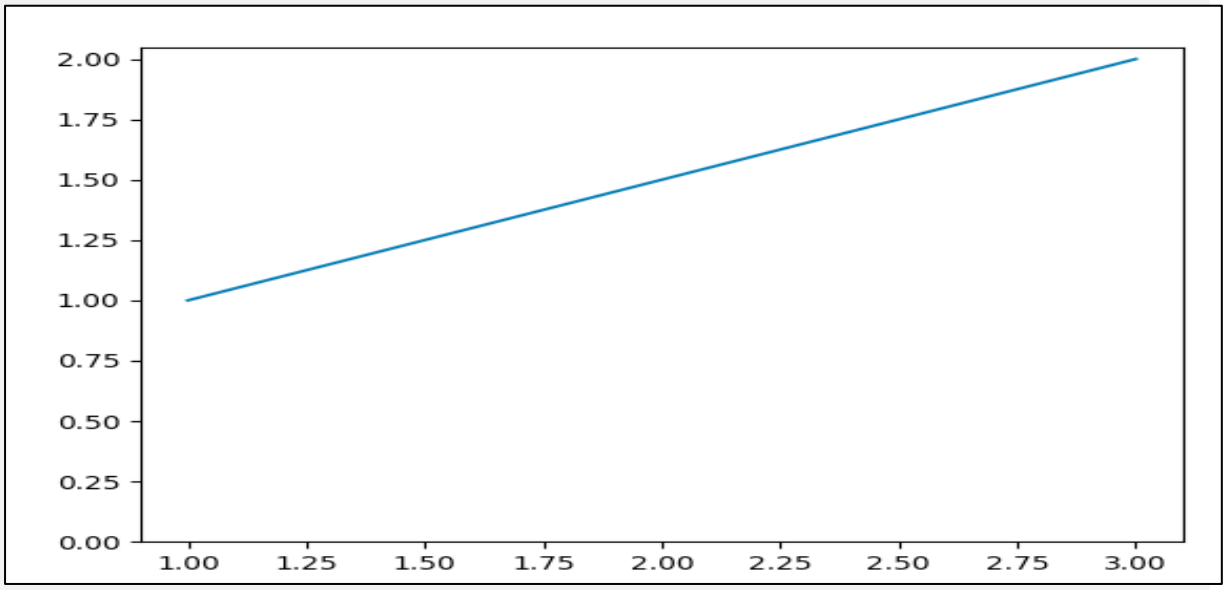
$$\theta_0 = 1.5$$
$$\theta_1 = 0$$

This yields h(x) = 1.5 + 0x. 0x means no slope, and y will always be the constant 1.5. This looks like:



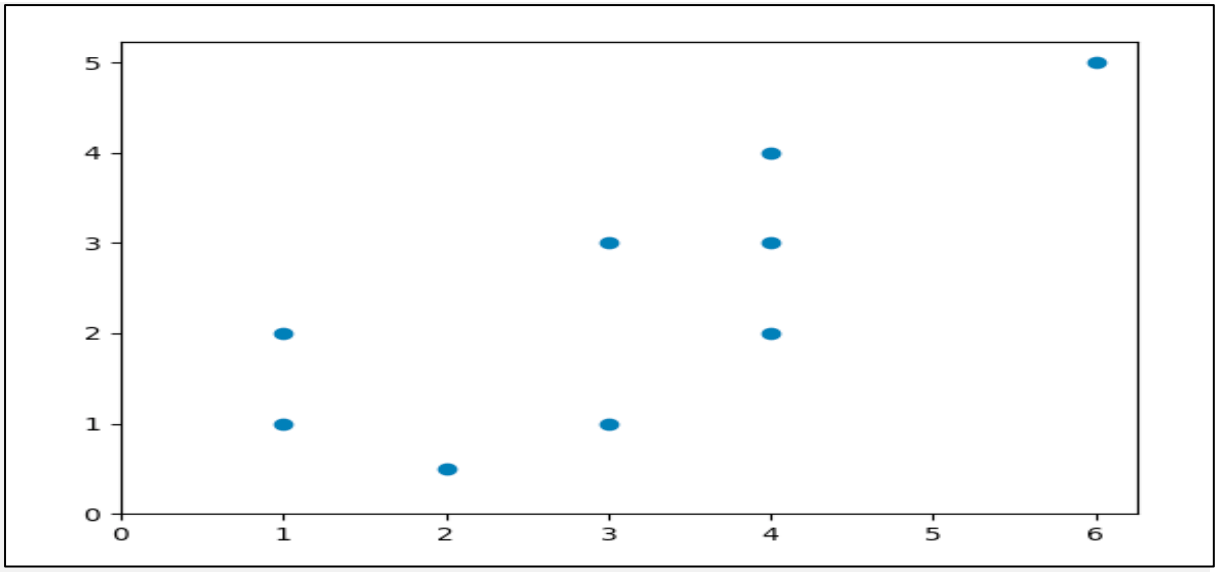
Now let's consider

$$\theta_0 = 1$$
$$\theta_1 = 0.5$$



The goal of creating a model is to choose parameters, or theta values, so that h(x) is close to y for the training data, x and y. So for the data below

x = [1, 1, 2, 3, 4, 3, 4, 6, 4]
y = [2, 1, 0.5, 1, 3, 3, 2, 5, 4]



Now we will try and find a best fit line for *linear regression.*

Cost Function

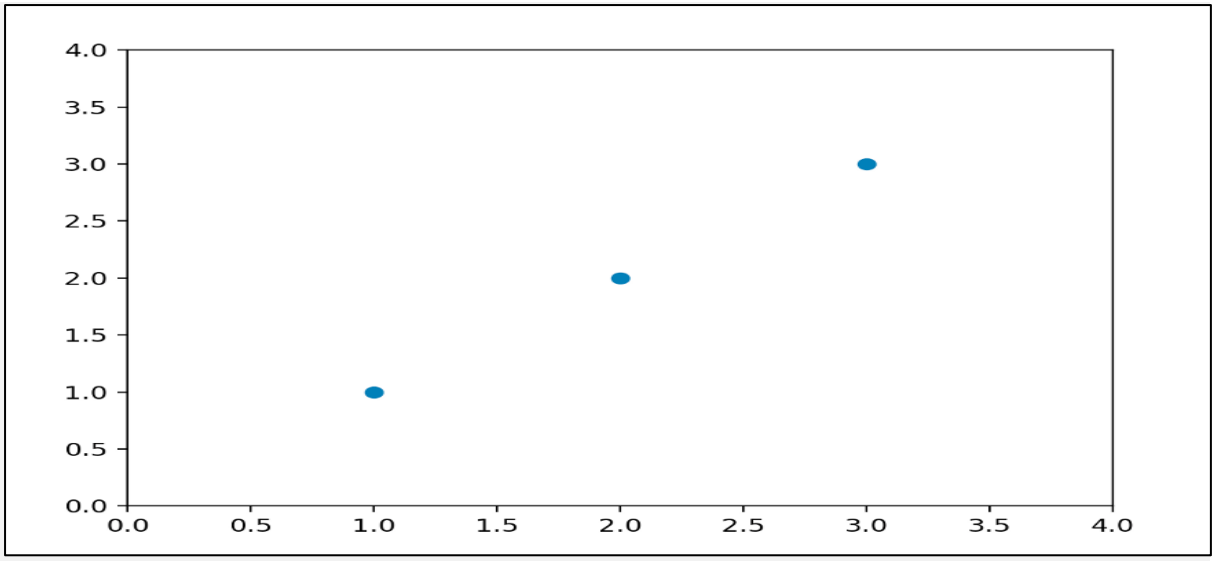
We need a function that will minimize the parameters over our dataset. One common function that is often used is mean squared error, which measure the difference between the estimator (the dataset) and the estimated value (the prediction). The formula is:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

It turns out we can adjust the equation a little to make the calculation down the track a little simpler. We get:

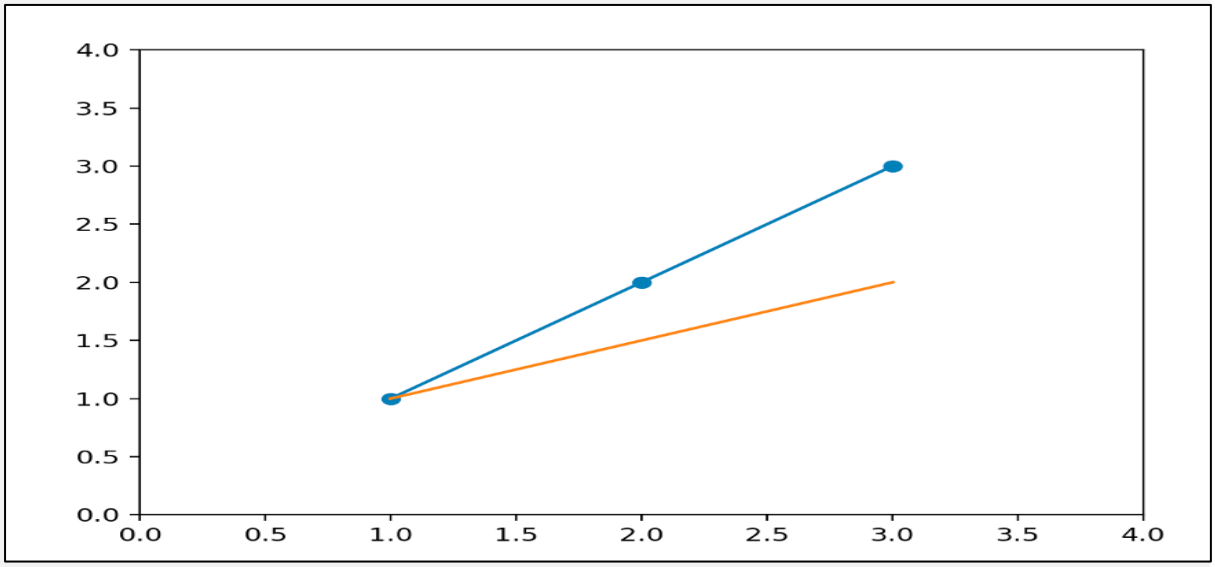
$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Let's apply this cost function to the follow data:



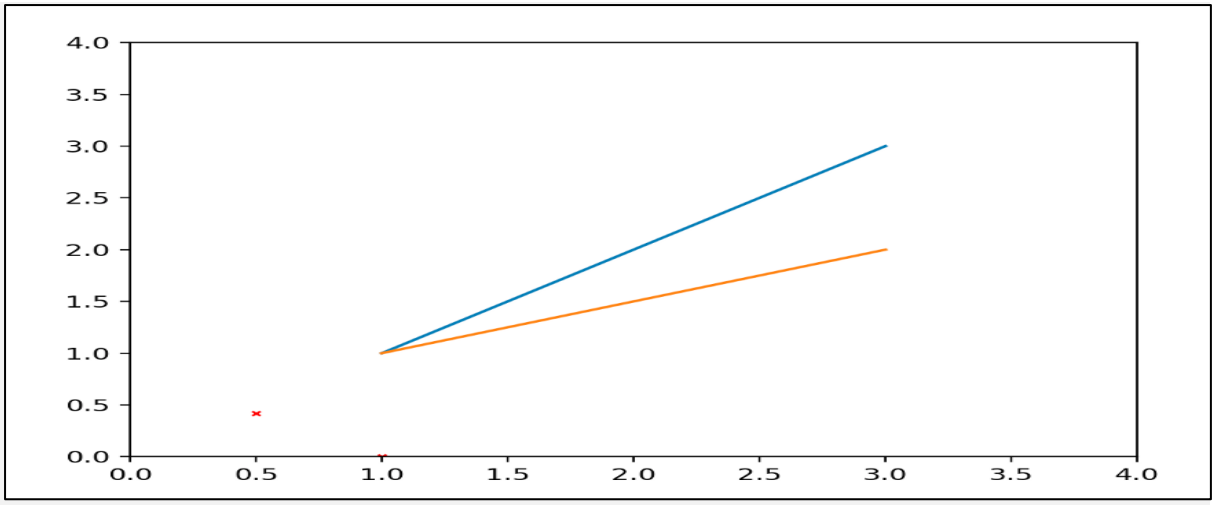
For now we will calculate some theta values, and plot the cost function by hand. Since this function passes through (0, 0), we are only looking at a single value of theta. From here on out, I'll refer to the cost function as J(Θ).

For J(1), we get 0. No surprise — a value of J(1) yields a straight line that fits the data perfectly. How about J(0.5)?

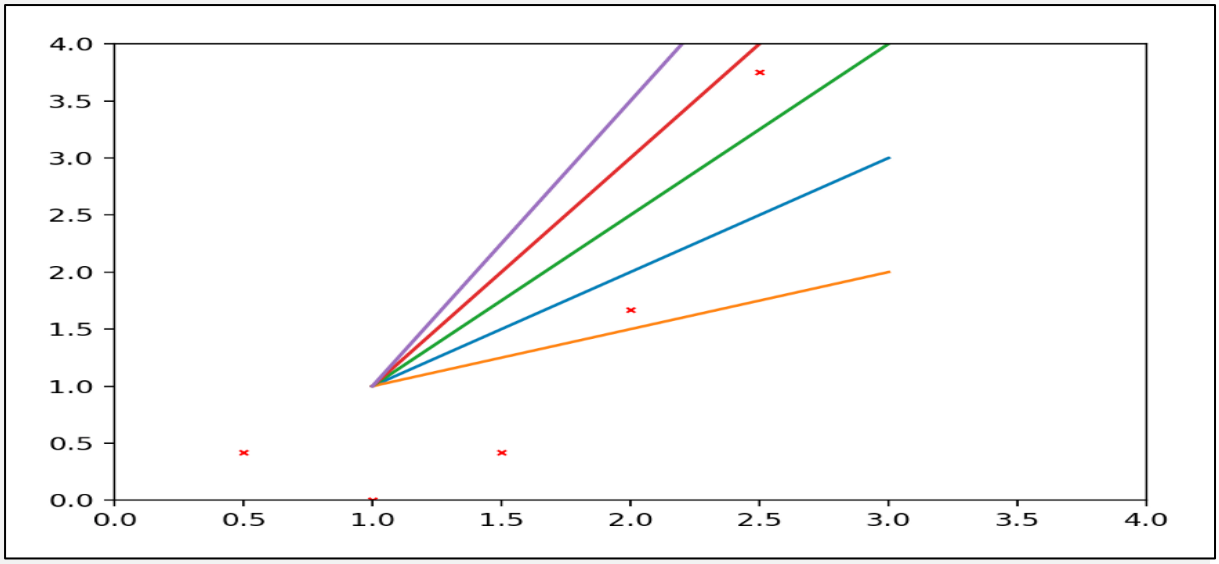


The MSE function gives us a value of 0.58. Let's plot both our values so far:

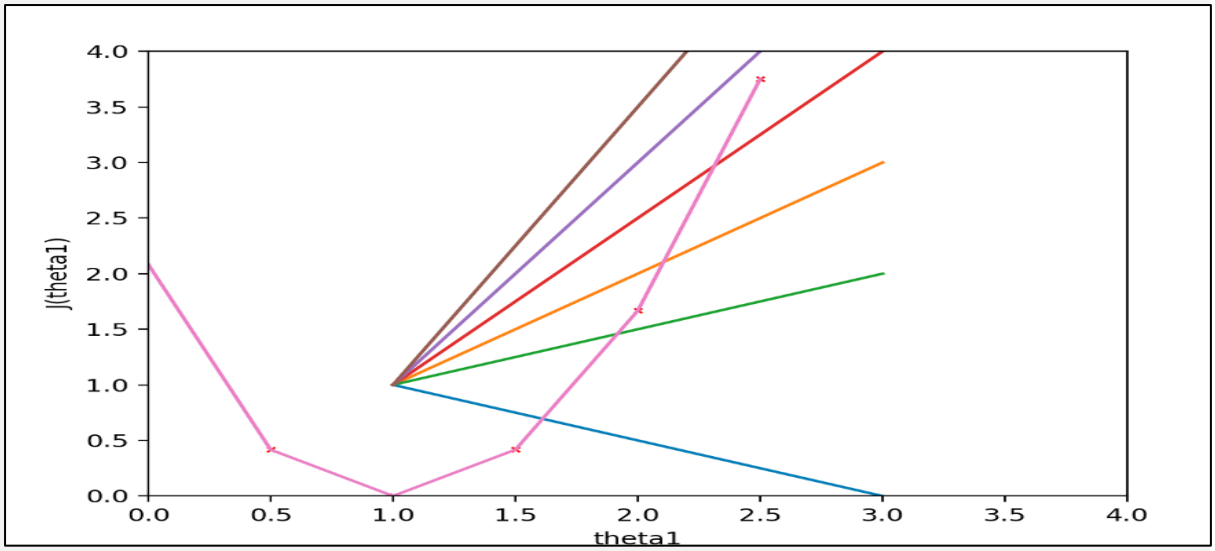
J(1) = 0, J(0.5) = 0.58



We will now calculate some more values of J(Θ).



And if we join the dots together nicely we get



We can see that the cost function is at a minimum when theta = 1. This makes sense — our initial data is a straight line with a slope of 1 (the orange line in the figure above).

Gradient Descent

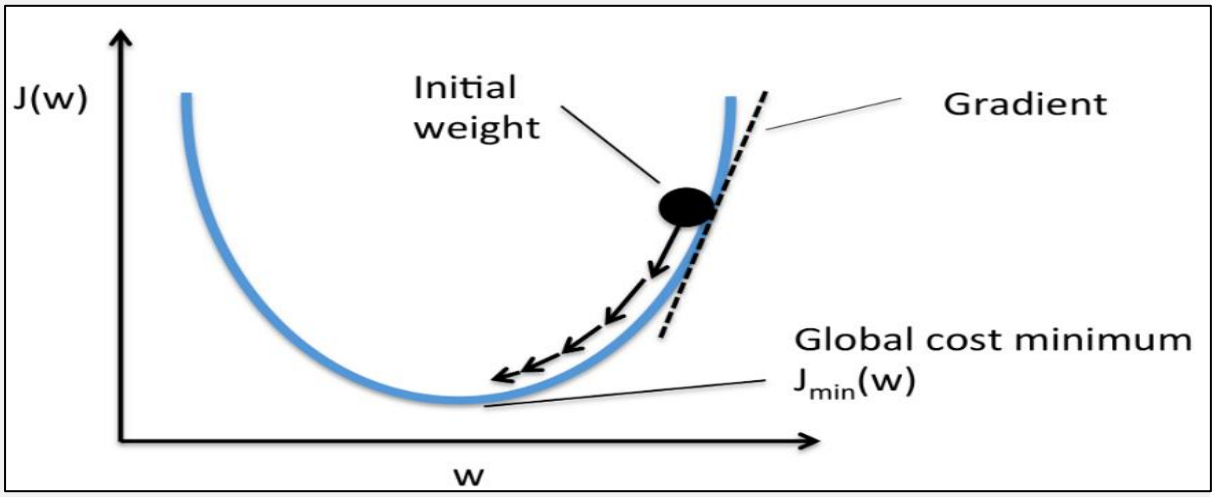
Gradient Descent is a general function for minimizing a function, in this case the Mean Squared Error cost function.

Gradient Descent basically just does what we were doing by hand — change the theta values, or parameters, bit by bit, until we *hopefully* arrived a minimum.

We start by initializing theta0 and theta1 to any two values, say 0 for both, and go from there. Formally, the algorithm is as follows:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad (\text{for } j = 0 \text{ and } j = 1)$$

where α, alpha, is the learning rate, or how quickly we want to move towards the minimum. If α is too large, however, we can overshoot.



Linear Regression

We have a hypothesis:

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

which we need fit to our training data. We can use a cost function such Mean Squared Error:

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

which we can minimize using gradient descent:

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1) \qquad (\text{for } j = 0 \text{ and } j = 1)$$

Which leads us to our first machine learning algorithm, linear regression. The last piece of the puzzle we need to solve to have a working linear regression model is the partial derivate of the cost function:

$$\frac{\partial}{\partial\theta_j}$$

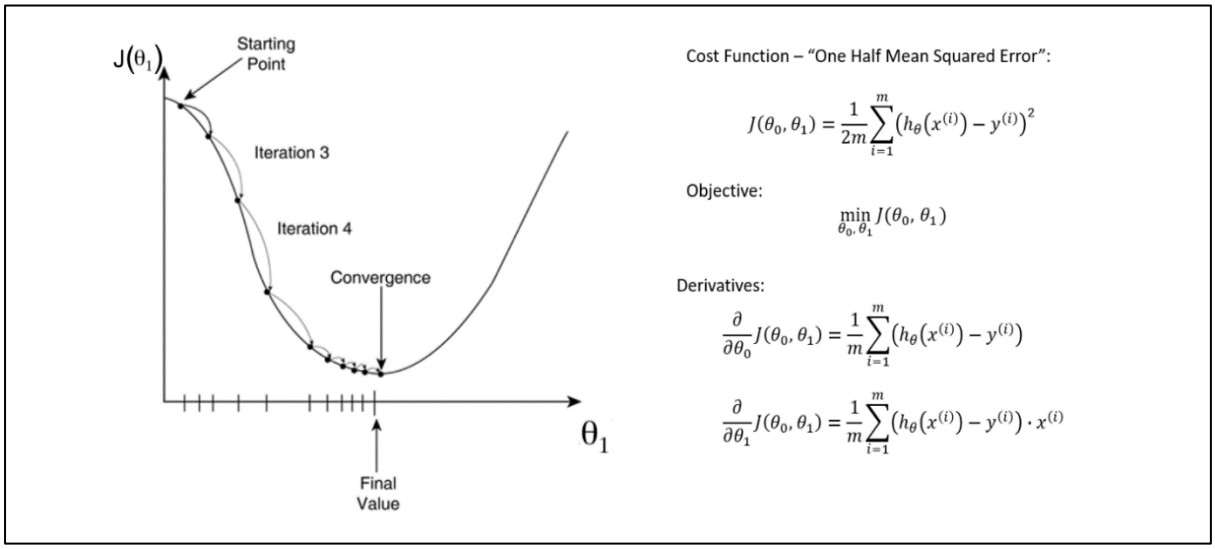
Which gives us linear regression!

$$\begin{aligned}&\text{repeat until convergence } \{\\ &\quad \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\\ &\quad \theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\cdot x^{(i)}\\ &\}\end{aligned}$$

Q2. What does the sign of gradient say about the relationship between the parameters and cost function?

Gradient descent is an efficient optimization algorithm that attempts to find a local or global minimum of a function.

Gradient Descent runs iteratively to find the optimal values of the parameters corresponding to the minimum value of the given cost function, using calculus. Mathematically, the technique of the '*derivative*' is extremely important to minimise the cost function because it helps get the minimum point. 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.



ϑ1 gradually converges towards a minimum value.

The derivative of a function (in our case, *J(ϑ)*) on each parameter (in our case weight *ϑ*) tells us the sensitivity of the function with respect to that variable or how changing the variable impacts the function value. Gradient descent, therefore, enables the learning process to make corrective updates to the learned estimates that move the model toward an optimal combination of parameters (*ϑ*). The cost is calculated for a machine learning algorithm over the entire training dataset for each iteration of the gradient descent algorithm. In Gradient Descent, one iteration of the algorithm is called one batch, which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration.

The step of the derivation

It would be better if you have some basic understanding of calculus because the technique of the partial derivative and the chain rule is being applied in this case.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \quad [1.0]$$

$$\frac{\partial}{\partial\theta_0}J(\theta_0,\theta_1) = \frac{\partial}{\partial\theta_0}\left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2\right) \quad [1.1]$$

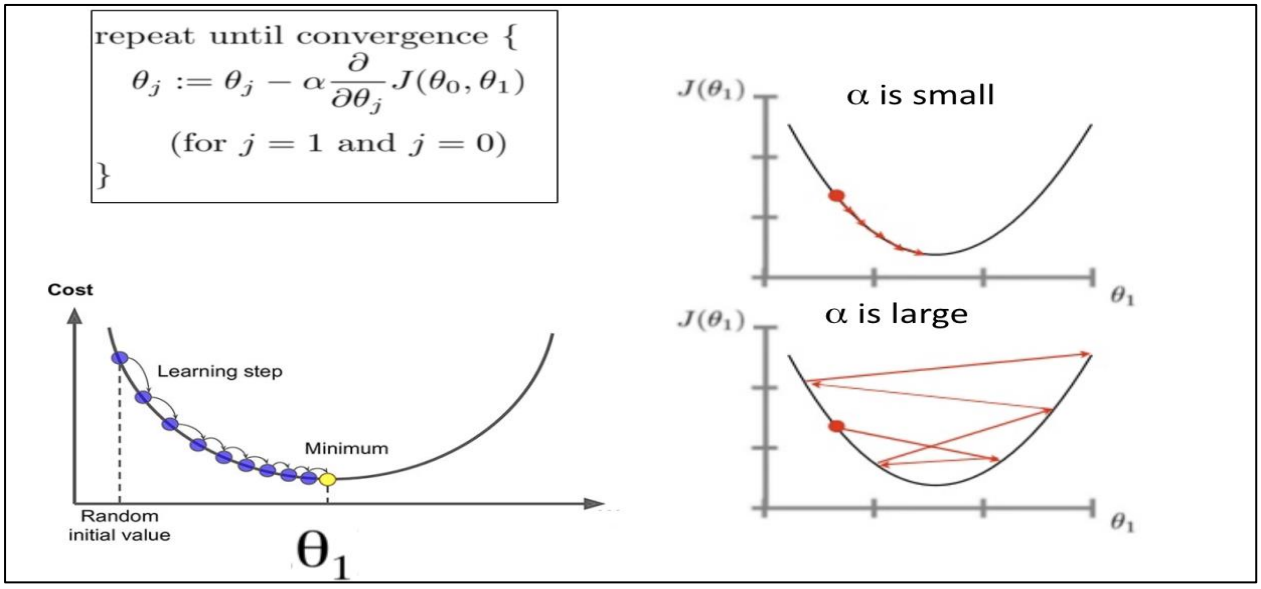
$$= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_0}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \quad [1.2]$$

$$= \frac{1}{m}\sum_{i=1}^{m}2\left(h_\theta(x^{(i)}) - y^{(i)}\right)\frac{\partial}{\partial\theta_0}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \quad [1.3]$$

$$= \frac{2}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \quad [1.4]$$

To solve for the gradient, we iterate through our data points using our new *weight 'ϑ0'* and *bias 'ϑ1'* values and compute the partial derivatives. This new gradient tells us the slope of our cost function at our current position (current parameter values) and the direction we should move to update our parameters. The size of our update is controlled by the learning rate.

Learning rate (α)



The size of these steps is called the **learning rate (α)** that gives us some additional control over how large of steps we make. With a large learning rate, we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom. The most commonly used rates are: *0.001, 0.003, 0.01, 0.03, 0.1, 0.3*.

Q3. Why Mean squared error is taken as the cost function for regression problems?

The error should decrease as we increase our sample data as the distribution of our data becomes more and more narrower (referring to normal distribution). The more data we have, the less is the error. Mean Squared Error is the best for this purpose. Its expression is:

$$\mathbf{L} = \frac{1}{N}[\Sigma(\hat{Y} - Y)^2]$$

We take the average or mean of Sum of Squared Errors. So more the data, lesser will be the aggregated error, MSE.

N	L	L/N
100	200	2
500	800	1.6
1000	1200	1.2

Here as you can see, the error is decreasing as our algorithm is gaining more and more experience. The Mean Squared Error is used as a default metric for evaluation of the performance of most regression algorithms.

Q4. What is the effect of learning rate on optimization, discuss all the cases.

Learning rate is used to scale the magnitude of parameter updates during gradient descent. The choice of the value for learning rate can impact two things: 1) how fast the algorithm learns and 2) whether the cost function is minimized or not.

For an optimal value of the learning rate, the cost function value is minimized in a few iterations (smaller time). This is represented by the blue line in the figure. If the learning rate used is lower than the optimal value, the number of iterations/epochs required to minimize the cost function is high (takes longer time).

If the learning rate is high, the cost function could saturate at a value higher than the minimum value. This is represented by the red line in the figure. If the learning rate selected is very high, the cost function could continue to increase with iterations/epochs. An optimal learning rate is not easy to find for a given problem. Though getting the right learning is always a challenge, there are some well-researched methods documented to figure out optimal learning rates. Some of these techniques are discussed in the following sections. In

all these techniques the fundamental idea is to vary the learning rate dynamically instead of using a constant learning rate.

Decaying Learning rate

In the decaying learning rate approach, the learning rate decreases with increase in epochs / iterations. The formula used for the decaying learning rate is shown below.

$$\alpha = \frac{\alpha_o}{1+\delta \times Epoch\#}$$

In the above equation, o is the initial learning rate, is the decay rate and is the learning rate at a given Epoch number.

Scheduled Drop Learning rate

Unlike the decay method, where the learning rate drops monotonously, in the drop learning rate method, the learning rate is dropped by a predetermined proportion at a predetermined frequency. The formula used to calculate the learning rate for a given epoch is shown in the below equation.

$$\alpha_n = \alpha_o \times D^{Quotient\left(\frac{n}{\rho}\right)}$$

In the above equation, o is the initial learning rate, '*n'* is the epoch/iteration number, '*D*' is a hyper-parameter which specifies by how much the learning rate has to drop, and ρ is another hyper-parameter which specifies the epoch-based frequency of dropping the learning rate.

The limitation with both the decay approach and the drop approach is that they do not evaluate if decreasing the learning rate is required or not. In both the methods, the learning rate decreases irrespective of difficulty involved in minimizing the cost function.

Adaptive Learning rate

In this approach, the learning rate increases or decreases based on the gradient value of the cost function. For higher gradient value, the learning rate will be smaller and for lower gradient value, the learning rate will be larger. Hence, the learning decelerates and accelerates respectively at steeper and shallower parts of the cost function curve. The formula used in this approach is shown in the below equation.

$$\alpha_n = \frac{\alpha_o}{\sqrt{S_n}}$$

In the above equation, o is the initial learning rate and '*sn'* is the momentum factor, which is calculated using below equation. '*n'* is the epoch/iteration number

$$S_n = \gamma S_{n-1} + (1-\gamma)\frac{\partial CF}{\partial \beta}]_n$$

In the above equation, γ is a hyperparameter whose value is typically between 0.7 and 0.9. Note that in the above equation, momentum factor *Sn* is an exponentially weighted average of gradients. So not only the value of the current gradient is considered, but also the values of gradients from the previous epochs are considered to calculate the momentum factor. Figure 5 shows the idea behind the gradient adapted learning rate. When the cost function curve is steep, the gradient is large, and the momentum factor 'Sn' is larger. Hence the learning rate is smaller. When the cost function curve is shallow, the gradient is small and the momentum factor 'Sn' is also small. The learning rate is larger.

The gradient adapted learning rate approach eliminates the limitation in the decay and the drop approaches by considering the gradient of the cost function to increase or decrease the learning rate. This approach is widely used in training deep neural nets with stochastic gradient descent.

Cycling Learning Rate

In this approach, the learning rate varies between a base rate and a maximum rate cyclically. It is reported that other forms such as sinusoidal or parabolic too yield similar results. The frequency of variation can be adjusted by setting the value of 'step size'. The formula used in this approach is shown below.

$$\alpha_E = \alpha_{E-1} + (\alpha_{max} - \alpha_{base}) \times (-1)^{quotient(\frac{E}{S}+1)} \rightarrow for\ E > S$$

In the above equation, E is the learning rates for a given epoch, *E* is the epoch number, max and base are respectively the maximum and the base learning rates. *S* is the step size. Note that the above equation valid when *E>S*. for E≤S below equation can be used

$$\alpha_E = \alpha_{max} - \frac{(\alpha_{max} - \alpha_{base})}{S} \times (S - E) \rightarrow for\ E\ S$$

An important step in making this approach work is identifying the right base and the maximum learning rates.

Reference:

1) Andrew NG's ML by Coursera

2) Chayan Kathuria (Towards Data Science)

3) Great Learning