Optimizers for Machine Learning

In our intro to ML we got an overview of what we expect from an ML model. And we learned that in order to build a model we are going to minimize a cost function with respect to its parameters in order to obtain the actual numeric value of the parameters .

In order to proceed with the same we will select a specific model which is simple enough for you to understand and still enables us to discuss intricacies associated with optimizers for ML.

It will help if you revised the discussion on cost function for regression before proceeding further .

Defining the optimization problem

For this discussion we'll consider regression with prediction model as a linear model , but the same idea will be applicable to any parametric model

Let's say we are trying to predict sales of a jewelery shop, considering gold price that day and temp that day as factors affecting sales . We can consider a linear model like this

$$sales_i = w_0 + w_1 * price_i + w_2 * temp_i$$

Here Ws are some constants. How do we determine, what values of Ws should we consider ?

`For linear models we can actually get closed form solution , shown towards the end of last chapter, but in this discussion we are trying to develop something which will work any complex parametric model , even deep neural networks`

As per the discussion that we have had about business expectation , we would desire Ws for which the cost function C is as low as possible for the given historical data given to us .

We can try out many different values of Ws and select the ones for which our cost function is coming out to be as low as possible. One way to find good Ws can be to simply compare the values of cost functions for each and pick the one with lowest cost function value.

Here are few steps of the process :

β_0	β_1	β_2	**Cost**	**Decision**
10	5	0	140	Start
-5	-2	4	160	Discard
1	3	7	150	Discard
4	7	02	135	Switch to theses Values
2	-2	10	130	Switch to theses Values
1	4	6	132	Discard
3	6	-5	141	Discard

We can keep on trying random values of Ws like that and switching to new values whenever we encounter a combination which gives us a new low for the cost function.

And we can stop randomly trying new values of Ws if we don't get any lower value for cost function for a long time .

This works in theory , given ; that we have infinite amount of time, resources and most important; patience!. We would want to have another method, which enables us to change our Ws in a such a way that cost function always goes down [instead of changing randomly and discarding most of them] .

Gradient Descent

C is a function with parameter W . If we change W by a small amount ΔW, Cost function changes by ΔC .

Rate of change of C with respect to W, will be $\frac{\Delta C}{\Delta W}$

We know for sufficiently small ΔW we can write :

$$\frac{\Delta C}{\Delta W} = \frac{\partial C}{\partial W}$$
$$\Delta C = \frac{\partial C}{\partial W} * \Delta W$$

if there were multiple parameters involved , say w_0 and $w_1 \cdots w_p$

then changing those parameters will individually contribute to changing the function , total change in C will be sum of these changes.

$$\Delta C = \frac{\partial C}{\partial w_0}\Delta w_0 + \frac{\partial C}{\partial w_1}\Delta w_1 \cdots + \frac{\partial C}{\partial w_p}\Delta w_p$$

this can be written as dot product between two vectors , which you will find to be gradient of C with respect to $W = [w_0 \quad w_1 \quad \cdots \quad w_p]$ and change vector for W

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_0} & \frac{\partial C}{\partial w_1} & \cdots & \frac{\partial C}{\partial w_p} \end{bmatrix}$$
$$\Delta W = [\Delta w_0 \quad \Delta w_1 \quad \cdots \Delta w_p]$$
$$\Delta C = \nabla C \cdot \Delta W$$

You must be wondering , why are we talking about all of this . This gives us an idea about how we can change our parameters such that cost function always goes down.

Consider

$$\Delta W = -\eta \nabla C$$

Where η is some small positive constant . if we put this back in $\Delta C = \nabla C \cdot \Delta W$, let's see what happens

$$\begin{aligned} \Delta C &= \nabla C \cdot \Delta W \\ &= -\eta \nabla C \cdot \nabla C \\ &= -\eta ||\nabla C||^2 \end{aligned}$$

this is an amazing result , it tells us that **if we changed our parameter , as per the suggestion , change in cost function will always be negative .** This gives us a consistent way of changing our parameters so that our cost function always goes down. Once we start to reach near the optimal value, gradient of the cost function ∇C will tend to zero and our parameter will stop to change .

Now we have consistent method for starting from random values of Ws and changing them in such a way that we eventually arrive at optimal values of Ws for given historical data. Let's see whether it really works or not with an example in context of linear regression .

Linear Regression Parameter Estimation example with Gradient Descent

Recall our discussion on cost function; for linear regression , it looks like this :

$$SSE = \sum {e_i}^2 = \sum (y_i - \hat{y_i})^2 = \sum (y_i - w_0 - w_1 * x_{i1} + \cdots + w_p * x_{ip})^2$$

If we consider entire X ,Y data and parameters βs to be written in matrix format like this

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & . & . & x_{1p} \\ 1 & x_{12} & x_{22} & . & . & x_{2p} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & x_{1n} & x_{n2} & . & . & x_{np} \end{bmatrix}$$

$$W = \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ w_p \end{bmatrix}$$

We can write :

predictions = XW [keep in mind that this will be a matrix multiplication]

errors = Y - predictions = $Y - X\beta$

cost $C = (Y - XW)^T(Y - XW)$ = $errors^T error$

I have taken liberty to extend the idea to more features which you see here as part of X matrix. x_{ij} represents j^{th} value of i^{th} variables/feature. All 1 column in X matrix represent multipliers of w_0.

Let's see what is the gradient of the loss function . Keep in mind that y_i and x_{ij} here are numbers/data points from the data. They are not parameters. They are observed values of the features.
We have seen this already without context in the last chapter.

gradient = $\nabla C = -2X^T(Y - X\beta)$

Lets simulate some data and put gradient descent to test

```
import pandas as pd
import numpy as np
```

```
x1=np.random.randint(low=1,high=20,size=20000)
x2=np.random.randint(low=1,high=20,size=20000)
```

```
y=3+2*x1-4*x2+np.random.random(20000)
```

you can see that we have generated data such that y is an approximate linear combination of x1 and x2, next we'll calculate optimal parameter values using gradient descent and compare them with results from sklearn and we'll see how good is the method.

```
x=pd.DataFrame({'intercept':np.ones(x1.shape[0]),'x1':x1,'x2':x2})
w=np.random.random(x.shape[1])
```

Lets write functions for predictions, error, cost and gradient that we discussed above

```
def myprediction(features,weights):
  predictions=np.dot(features,weights)
    return(predictions)

myprediction(x,w)
```

> array([3.83044239, 5.51284694, 8.82009525, ..., 4.82993684,
> 10.15713504, 2.49340259])

Note that , `np.dot` here is being used for matrix multiplication . Simple multiplication results to element wise multiplication , which is simply wrong in this context .

```
def myerror(target,features,weights):
    error=target-myprediction(features,weights)
    return(error)
myerror(y,x,w)
```

> array([-8.34075913, -26.3578085 , -53.60220599, ..., -39.73305801,
> -22.3456587 , -39.18996579])

```
def mycost(target,features,weights):
    error=myerror(target,features,weights)
    cost=np.dot(error.T,error)
    return(cost)

mycost(y,x,w)
```

```
23139076.992828812
```

```
def gradient(target,features,weights):

    error=myerror(target,features,weights)
    gradient=-np.dot(features.T,error)/features.shape[0]
    return(gradient)

gradient(y,x,w)
```

```
array([ 24.86385998, 212.05914008, 369.58961913])
```

Note that gradient here is vector of 3 values because there are 3 parameters . Also since this is being evaluated on the entire data, we scaled it down with number of observations . Do recall that , the approximation which led to the ultimate results was that change in parameters is small. We don't have any direct control over gradient , we can always chose a smaller value for η to ensure that change in parameter remains small. Also if we end up choosing too small value for η, we'll need to take larger number of steps to change in parameter in order to arrive at the optimal value of the parameters

Lets looks at the expected value for parameters from sklearn . Don't worry about the syntax here , we'll discuss that in detail, when we formally start with linear models in next module .


```
from sklearn.linear_model import LinearRegression
lr=LinearRegression()
lr.fit(x.iloc[:,1:],y)
sk_estimates=([lr.intercept_]+list(lr.coef_))
```

```
sk_estimates
```

```
[3.4963871364502594, 2.0000361062367253, -3.999828135637543]
```

When you run the same , these might be different for you, as we generated the data randomly .Now lets write our version of this , using gradient descent

```
def my_lr(target,features,learning_rate,num_steps,print_when):

    # start with random values of parameters
    weights=np.random.random(features.shape[1])
    # change parameter multiple times in sequence
    # using the cost function gradient which we discussed earlier
    for i in range(num_steps):
        weights -= learning_rate*gradient(target,features,weights)
```

```
    # this simply prints the cost function value every (print_when)th iteration
        if i%print_when==0:
            print(mycost(target,features,weights),weights)

    return(weights)


```

```
my_lr(y,x,.0001,500,100)
```

```
result after 100th iteration : 29897491.344063997 [0.86957516 0.90950957 0.35280286]
result after 200th iteration : 5141497.711763546 [ 0.75555271  0.26200844 -1.71416641]
result after 300th iteration : 2703652.3603685475 [ 0.74787632  0.6560246
-2.37514143]
result after 400th iteration : 1480112.0586781118 [ 0.75044815  1.03096429
-2.77924166]
result after 500th iteration : 815173.9884761975 [ 0.75398277  1.31567619 -3.06931226]
```

```
final weights after 500 iterations: array([ 0.75755246,  1.52465372, -3.28073366])
```

you can see that if we take too few steps , we did not reach to the optimal value

Lets increase the learning rate η

```
my_lr(y,x,.01,500,100)
```

```
result after 100th iteration : 40621523.14640909 [ 0.2279513  -1.98371006 -3.60180851]
result after 200th iteration : 8.399918899418999e+30 [-8.28197980e+10 -9.56164546e+11
-9.47578310e+11]
result after 300th iteration : 2.0273152258521542e+54 [-4.06871492e+22 -4.69738039e+23
-4.65519851e+23]
result after 400th iteration : 4.892912746164989e+77 [-1.99885070e+34 -2.30769721e+35
-2.28697438e+35]
result after 500th iteration : 1.1809014620072595e+101 [-9.81981827e+45
-1.13370985e+47 -1.12352928e+47]
```

```
final weights after 500 iterations:array([3.68564341e+57, 4.25511972e+58,
4.21690928e+58])
```

You can see that because of high learning rate , change in parameters is huge and we end up missing the optimal point , cost function values , as well as parameter values ended up exploding. Now lets run with low learning rate and higher number of steps

```
my_lr(y,x,.0004,100000,10000)
```

```
result after 10000th iteration : 30786777.19050019 [0.17828571 0.48834937 0.70073856]
result after 20000th iteration : 12603.561197114073 [ 1.44895395  2.0897855
-3.91144197]
result after 30000th iteration : 5543.348390660241 [ 2.27839978  2.05342668
-3.94724853]
result after 40000th iteration : 3044.822379129174 [ 2.77182469  2.03179736
-3.96854931]
result after 50000th iteration : 2160.623502773709 [ 3.06535578  2.0189304
-3.98122083]
result after 60000th iteration : 1847.715952746199 [ 3.23997303  2.01127604
-3.98875893]
result after 70000th iteration : 1736.981662664487 [ 3.34385023  2.00672257
-3.99324323]
result after 80000th iteration : 1697.7941039148957 [ 3.40564521  2.00401379
-3.99591087]
result after 90000th iteration : 1683.926089232013 [ 3.44240612  2.00240237
-3.99749782]
result after 100000th iteration : 1679.018362458989 [ 3.46427464  2.00144376
-3.99844186]
```

```
final result after 100000th iteration :
array([ 3.4772829 ,  2.00087354, -3.99900342])
```

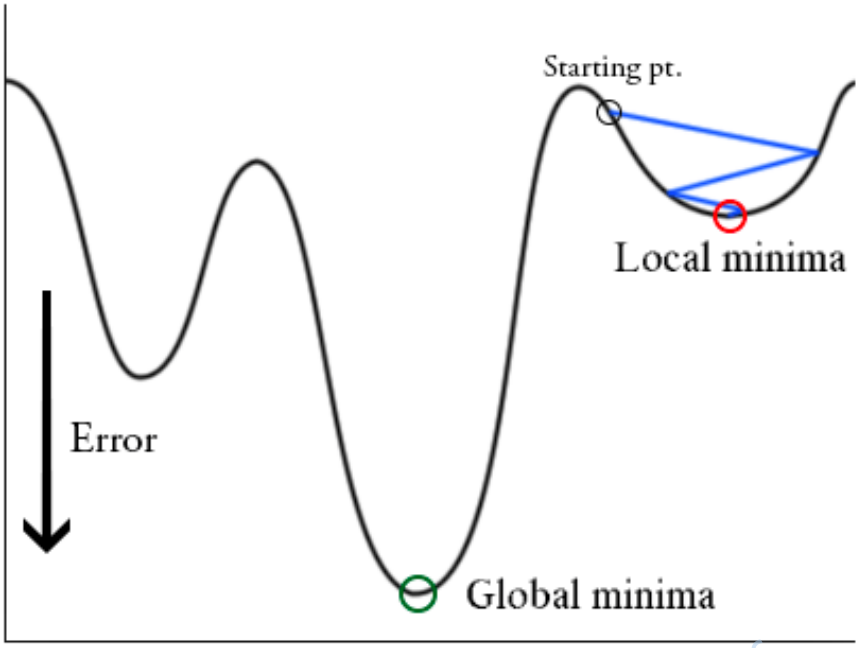
We can see here that we ended up with pretty good estimates for Ws , as good as from sklearn .

```
sk_estimates
```

```
[3.4963871364502594, 2.0000361062367253, -3.999828135637543]
```

There are a couple of issues here :

- We are using entire data for each iteration, while this is ok here with just two features , this can very fast lead to becoming a bottleneck with much larger data. In fact when we start working with data such as images where each individual observation in itself needs a lot of memory , this will become intractable
- We are going through 10000 iterations , which is a lot and can be improved upon a lot .
- Gradient calculated on the entire data is absolute . In the sense, if it becomes zero, due to arriving at an optimal point , it wont change, since the data it is being calculated for doesn't change. This is absolutely fine for simple cost function that we considered here. However cost functions become complex with introduction of more creative algorithms necessary for more complex data. They end up having lot of shallow local minimas, causing gradient decent to fail by getting stuck in these local optimas instead of reaching to better optimum.



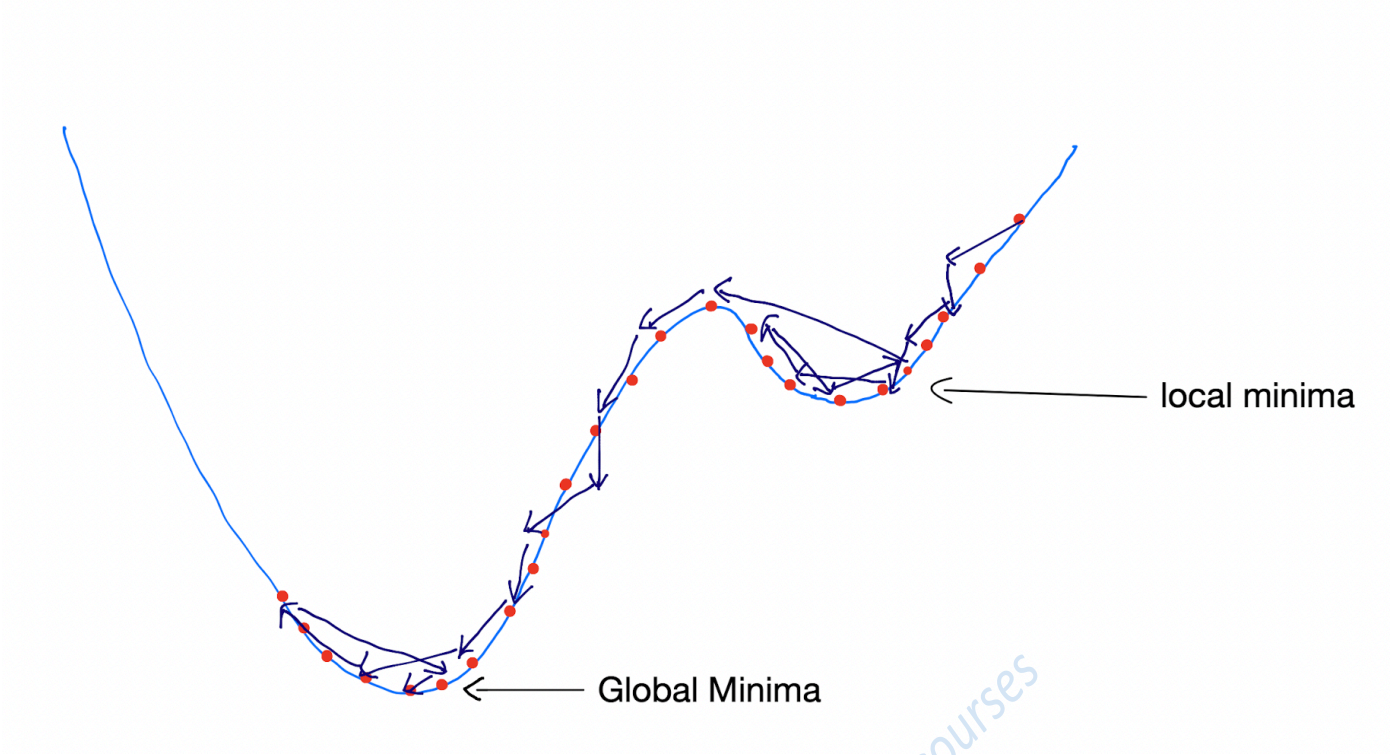
Further discussions on modification of gradient descent has these issues being resolved at their core . Requiring less data per iteration , less number of iterations and less prone to getting stuck in local minimas.

Stochastic Gradient Descent

Let me set the intuition with a simple example . Consider that we are trying to estimate true mean of some population. When we take a sample of data and calculate its mean, it wont be exactly equal to population mean, however it will be close to it. And every time we resample the data, it will somewhat different that the earlier result.

Same idea is behind stochastic gradient descent also. Instead of using entire data to get true gradient, we instead take a sample and use estimate of the gradient. This of course can cause our updates to move in a slightly imperfect direction for individual iteration , however overall direction will be similar to what we would have achieved with true gradient .

The added benefit here is that this estimated gradient will not be exactly zero at optimal points , which gives us the opportunity to jump out from local optimas .



As you can see that following this estimated gradient , we can still get stuck in minimas but there would be some sample gradient which will give us chance to jump out of shallow minimas . There is no fool proof mathematical idea which is guaranteed to always work w.r.t. the problem of getting stuck in local minimas however stochastic gradient descent is the best and simplest idea which almost always works in practice . It even works as a mild form of regularization as random sampling has benefit of averaging out the noise to some extent . And the performance remains as good as gradient descent if not better .

Its addressing of the first issue mentioned above makes it a de-facto default choice for almost all practical application. So much so that when you hear people talking about gradient descent in context of deep learning , it implies stochastic gradient descent by default .

Lets see an example on the same data. We are going to make some modification to code for visualizing all optimizer progress.

Here is a look at modified code for gradient descent .

```
def my_lr_gd(target,features,learning_rate,num_steps):

    weights=np.random.random(features.shape[1])
    cost=[]

    for i in range(num_steps):

        weights = weights-  learning_rate*gradient(target,features,weights)
        cost.append(mycost(target,features,weights))

    return(cost,weights)
```

for comparison we have already estimated weights from sklearn . here they are

```
w_sklearn
```

```
[5.4994489822121295, 1.9997995007450478, 2.9999404537853738]
```

Now lets see how simple gradient descent does with limited number of iterations

```
cost_gd,w_gd= my_lr_gd(y,x,.001,1000)
```

```
w_gd
```

```
array([1.12511216, 2.12604002, 3.1263241 ])
```

A good way to compare them with sklearn estimate is to take ratio with sklearn estimates . It should ideally be close to 1 for each of these estimates as we approach towards them .

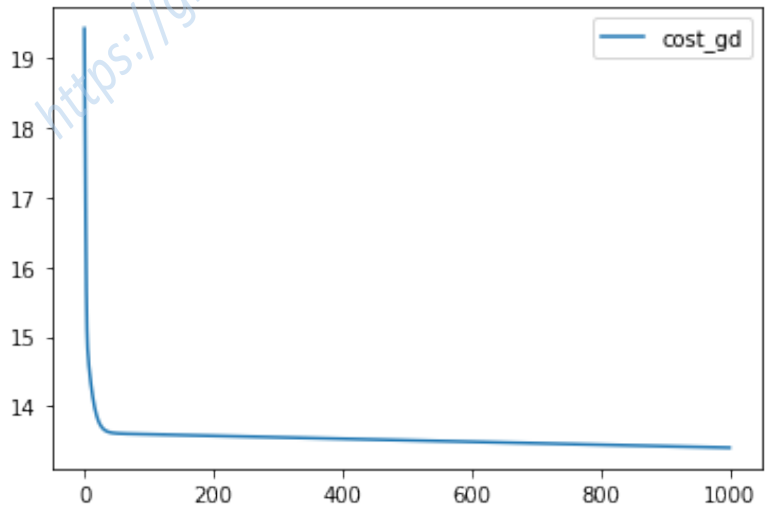
```
w_sklearn/w_gd
```

```
[4.88791177, 0.94062176, 0.95957436]
```

You can see that gradient descent with this few iterations is doing a terrible job at estimating w_0 . there is a good scope for improvement in w_1 and w_2 values as well.

We will plot the cost function descent in log scale .

```
np.log(pd.DataFrame({'cost_gd':cost_gd})).plot()
```



This in itself is not very informative . that steep fall essentially shows how cost function falls sharply initially , but slows down to a crawl once its near the optimal value because of gradients becoming too close to zero. As we move forward in our discussion of improved optimizers , we will be able to see how they behave relative to this . And thats going to be a fun ride !

Lets do some modifications to gradient descent code to include random sampling of observations, instead of entire data being used .

```
def my_lr_sgd(target,features,learning_rate,num_steps):
    cost=[]
    weights=np.random.random(features.shape[1])

    for i in np.arange(num_steps):

        rand_ind=np.random.choice(range(features.shape[0]),10)
        target_sub=target[rand_ind]
        features_sub=features.iloc[rand_ind,:]

        weights -= learning_rate*gradient(target_sub,features_sub,weights)
        cost.append(mycost(target,features,weights))

    return(cost,weights)
```

Here this small snippet :

```
rand_ind=np.random.choice(range(features.shape[0]),10)
target_sub=target[rand_ind]
features_sub=features.iloc[rand_ind,:]
```

is creating random subset of target and corresponding feature observation to make use of later in the code for calculating and using estimated gradient. Notice that instead of using all 200,000 observations , it makes use of just 10 each iteration. The effect this has on speed is not going to be noticeable here but for larger datasets , this gives phenomenal feasibility and speed to the whole process. Lets see the results .

```
cost_sgd,w_sgd=my_lr_sgd(y,x,.001,1000)
w_sklearn/w_sgd
```

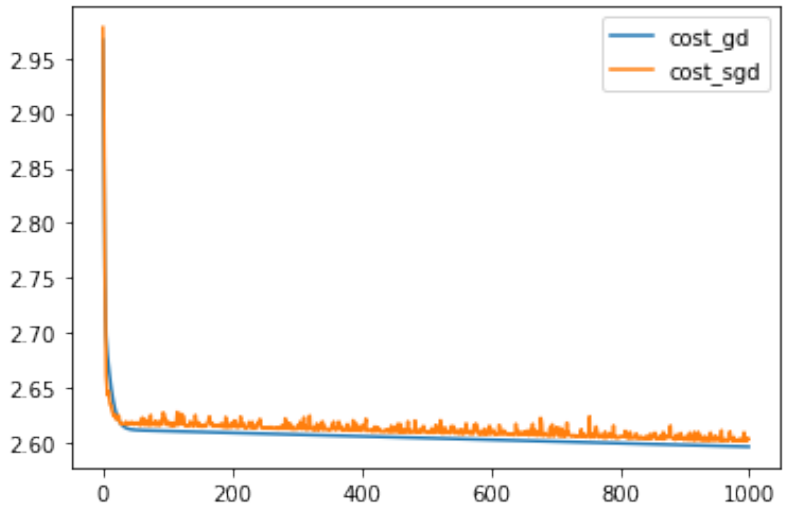
```
[5.80133537, 0.93643659, 0.95350591]
```

you can see that the performance is similar to gradient descent despite using very small subset of the data each iteration .

Note : it will not always be this small fraction , the pattern here is very simple and easy to extract. Usually more complex pattern will require much bigger samples but still fraction of the full data set.

Understandably , path to optimality with stochastic gradient descent is going to be much rougher as evident in the figure below.

```
np.log(
       pd.DataFrame({'cost_gd':cost_gd,
                     'cost_sgd':cost_sgd})
      ).plot()
```



Ok, so we are able to achieve similar performance with added benefit of using less data per iteration and being less prone to getting stuck in local minima [which is not displayed in this graph as it simply represents how cost function is decreasing with iterations]

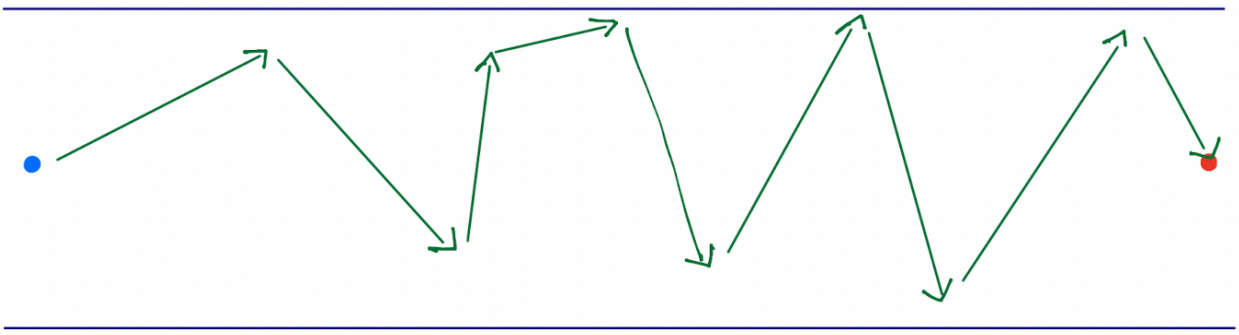
Notice that a large number of iterations are being spent in making very little improvement and this results in eventually not reaching to ideal optima [we are still far from our benchmark estimates that we obtained from sklearn]. Not lets improve our ability to do more in less iterations.

SGD with momentum

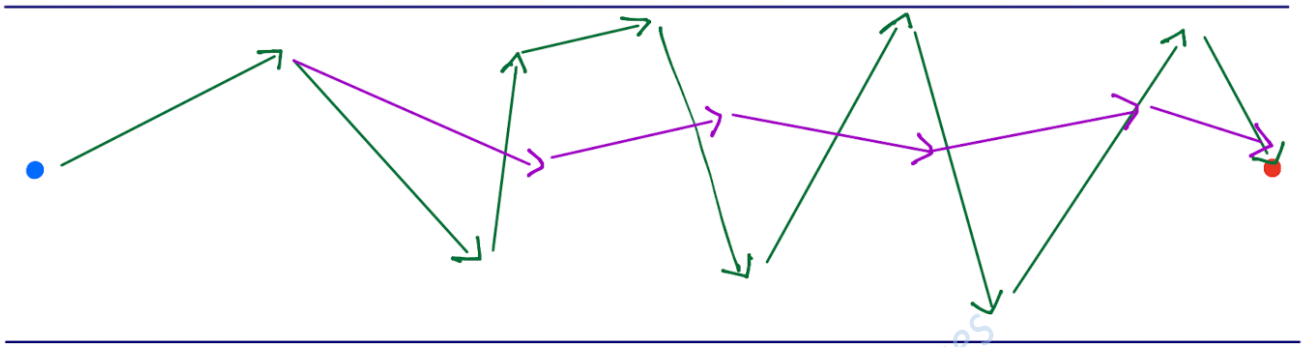
Lets discuss the issue with stochastic gradient descent which gets dealt with next to bring in the improvement

Due to estimated gradient coming from random samples , individual iterations very often move in a very suboptimal direction, which makes the approach to optima rough and requires more iterations .

Lets take a simpler analogy to understand this . Consider yourself to be starting at this blue dot on a street . You eventually have to reach at the red dot . You move in short burst , each time you start to move you receive a total random direction in comparison to your earlier direction.



you can see that you eventually reach to the spot but it would have been much quicker if direction of your travel were not changing so dramatically every iteration . We will still get new directions from random samples , what we can do is to not take them into account in their entirety . We keep the earlier direction and update it slightly with input from new sample. Essentially keeping a sort of exponential average .



Currently we use raw estimated gradient for the direction that is

$$\nabla C_i = \frac{\partial C}{\partial W}$$

now we'll use V_i as our gradient for i^{th} iteration . Where

$$V_i = \gamma V_{i-1} + (1-\gamma)\nabla C_i$$
$$\text{given } V_0 = 0$$

This V_i is now exponential average of sample gradients that we will follow . γ parameter here takes value in $(0, 1)$ and is known as momentum . Closer to 1 the value of γ is , less we update our direction as feedback from each sample. Making γ here 0 will take us back to simple SGD [stochastic gradient descent]

There is one problem though here , if you consider V_1 for instance with typical value of γ which is 0.9 , it comes out to be

$$V_1 = 0.1 * \nabla C_1$$

and V_2 will be

$$V_2 = 0.9 * 0.1 * \nabla C_1 + 0.1 * \nabla C_2$$

Remember that scale of gradient also determines the step size . So this SGD with momentum is going to be taking considerably smaller steps initially in comparison to original SGD and thus making initial approach to optima slower. We need to introduce some way of inflating this initial step size which should taper off as we move to higher iteration steps .

We can work with an adjust $V_{i_{adjusted}}$ defined as :

$$V_{i_{adjusted}} = \frac{V_i}{1-\gamma^i}$$

This will make

$$V_{1_{adjusted}} = \frac{0.1 * \nabla C_1}{0.1} = \nabla C_1$$

it will also inflate subsequent step sizes to bring them being equivalent to simple SGD. and as the number of iterations increase the inflating factor will reach to 1 and will stop to inflate the step size .

$$\text{since } \gamma < 1 \text{ , } \gamma^i \rightarrow 0 \text{ as i increases}$$

Earlier the parameters were updated using raw gradients using following idea :

$$\Delta w_i = -\eta \nabla C_i$$

Now instead of using ∇C_i , we will be using adjusted exponential averaged gradients with momentum $V_{i_{adjusted}}$

$$\Delta w_i = -\eta V_{i_{adjusted}}$$

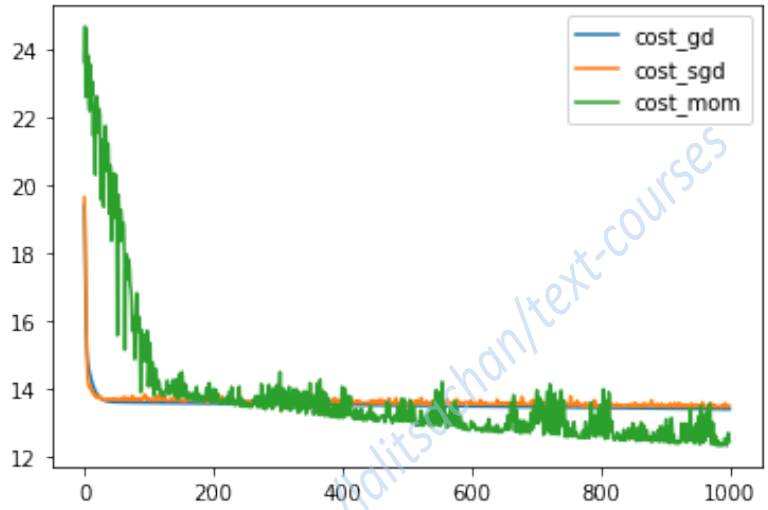
Lets see this implemented in code

```
def my_lr_mom(target,features,learning_rate,num_steps):
    cost=[]

    weights=np.random.random(features.shape[1])

    vw=np.zeros(features.shape[1])
    gamma=0.9

    for i in np.arange(num_steps):

        rand_ind=np.random.choice(range(features.shape[0]),10)
        target_sub=target[rand_ind]
        features_sub=features.iloc[rand_ind,:]

        vw=gamma*vw+(1-gamma)*gradient(target_sub,features_sub,weights)

        vw_a=vw/(1-gamma**(i+1))

        weights -= learning_rate*vw_a

        cost.append(mycost(target,features,weights))

    return(cost,weights)
cost_mom,w_mom=my_lr_mom(y,x,.01,1000)
w_sklearn/w_mom
```

```
[1.42942865, 0.97034651, 0.98032564]
```

you can see that our approach to optima has improved across the board [all ratios are approaching one , ratio for w_0 has improved from ≈ 4.9 to ≈ 1.4. this would reflect in decrease in cost function also , in same amount of iteration. We are also able to use relatively higher learning rate since our direction is much more streamlined and risk of off-shooting away from optima is lot less .

```
np.log(
   pd.DataFrame({'cost_gd':cost_gd,
                 'cost_sgd':cost_sgd,
                 'cost_mom':cost_mom})
   ).plot()
```



RMSPROP

We are able to do a lot in fewer iterations now . However you can notice that there is some discrepancy in terms of how much we reach to optima for one parameter Vs another . In our case here , performance for w_0 has been abysmal to say the least . This happens because we are possibly taking larger steps towards optimal values of w_1 and w_2 in comparison to w_0 . This comes from the fact that gradients for one parameter can be smaller in comparison to the other. Remember that here we are concerned with absolute values of the gradient not their sign .

RMSPROP tried to address this problem by normalizing sample gradients with absolute value of exponentially averaged gradient . The idea is to scale small gradients towards the optima to larger values so that approach to optima can be faster and scale back the larger gradients to avoid missing the optima.

First lets calculate exponential averaged squares of gradients [since we want absolute values free of signs]

$$S_i = \gamma S_{i-1} + (1-\gamma)\nabla C_i{}^2$$
$$\text{given } S_0 = 0$$

Here $\nabla C_i{}^2$ is element wise multiplication between the elements of vector ∇C_i

S_i also goes through similar adjustment as we saw earlier with V_i

$$S_{i_{adjusted}} = \frac{S_i}{1-\gamma^i}$$

Now we divide the raw sample gradient with square root of these $S_{i_{adjusted}}$ and use that instead of raw gradients while updating the weights .

$$\Delta w_i = -\eta \frac{\nabla C_i}{\sqrt{S_{i_{adjusted}}} + \epsilon}$$

While dividing ; ϵ , a small positive quantity is added to avoid division by zero issues. Here the raw gradients which are larger, end up getting divided by larger absolute values thus the approach towards their optima slows down, where as smaller raw gradients are divided by smaller absolute values and their approach towards the optima is accelerated. Due to smoothing of larger gradients , we will be able to use even larger learning rate with rmsprop which in turn result in making more progress in fewer iterations.

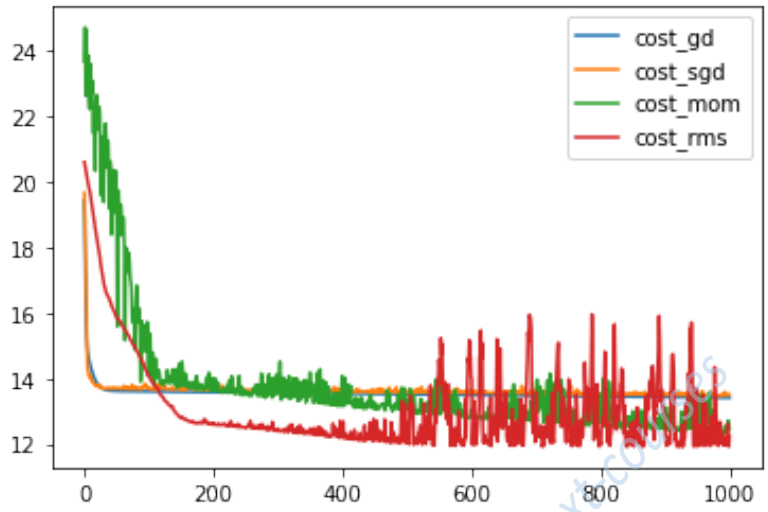
Lets see this implemented in the code

```
def my_lr_rms(target,features,learning_rate,num_steps):
    cost=[]

    weights=np.random.random(features.shape[1])
    sw=np.zeros(features.shape[1])

    gamma=0.99

    for i in np.arange(num_steps):
        rand_ind=np.random.choice(range(features.shape[0]),10)
        target_sub=target[rand_ind]
        features_sub=features.iloc[rand_ind,:]

        gd=gradient(target_sub,features_sub,weights)

        sw=gamma*sw+(1-gamma)*(gd**2)
        sw_a=sw/(1-gamma**(i+1))

        weights -= learning_rate*(gd/(np.sqrt(sw_a)+1e-15))
        cost.append(mycost(target,features,weights))

    return(cost,weights)
cost_rms,w_rms=my_lr_rms(y,x,.1,1000)
w_sklearn/w_rms
```

```
[0.99670036, 1.00812969, 1.00597921]
```

Wow , we have done pretty well with this improvement . Lets see how the cost function decrease looks like through the iterations

```
np.log(
       pd.DataFrame({'cost_gd':cost_gd,
                     'cost_sgd':cost_sgd,
                     'cost_mom':cost_mom,
                     'cost_rms':cost_rms})
                    ).plot()
```



You can see that rmsprop with higher learning rate is clearly able to make much more progress in same number of iterations . But as we approach very close to optimal point , due to using raw gradients , we see very erratic behavior. In fact some of the iterations even result in cost bouncing to worse values than all other variants that we have seen. If we stopped the descent at those iterations , we would have ended up with far worse weights estimates .

So RMSPROP approaches optima faster and evenly for all the parameters but has issue of becoming erratic when close to optima, Momentum approaches optima rather smoothly. Our next candidate combines both the ideas.

ADAM (Adaptive Moment Estimation)

We can combine the idea of using exponential averaged gradient instead of raw gradient and normalizing them to get something which approaches to optima faster as well as smoother .

Weight updating becomes

$$\Delta w_i = -\eta \frac{V_{i_{adjusted}}}{\sqrt{S_{i_{adjusted}}} + \epsilon}$$

Where $V_{i_{adjusted}}$ and $S_{i_{adjusted}}$ are the same ideas discussed in SGD with momentum and RMSPROP above.

The next bit of code looks intimidating but its just combining the above two ideas we discussed

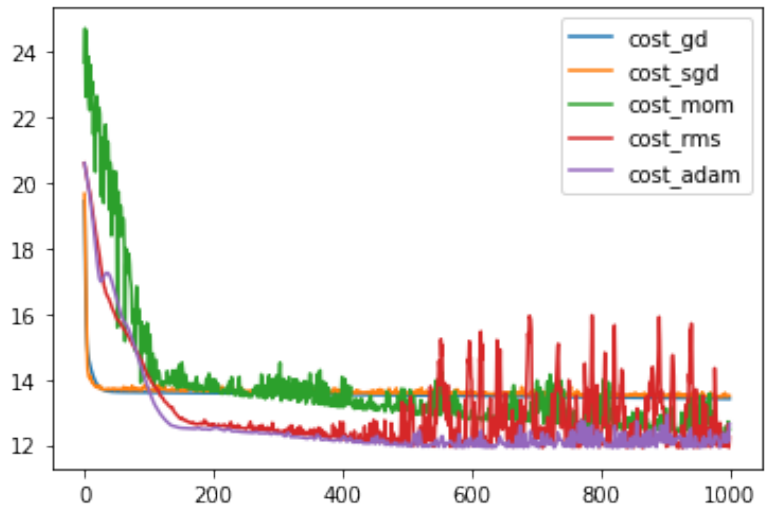
```
def my_lr_adam(target,features,learning_rate,num_steps):
    cost=[]

    weights=np.random.random(features.shape[1])
    sw=np.zeros(features.shape[1])
    vw=np.zeros(features.shape[1])
```

```
    gamma1=0.9
    gamma2=0.99

    for i in np.arange(num_steps):
        rand_ind=np.random.choice(range(features.shape[0]),10)
        target_sub=target[rand_ind]
        features_sub=features.iloc[rand_ind,:]

        gd=gradient(target_sub,features_sub,weights)

        vw=gamma1*vw+(1-gamma1)*gd
        sw=gamma2*sw+(1-gamma2)*(gd**2)

        vw_a=vw/(1-gamma1**(i+1))
        sw_a=sw/(1-gamma2**(i+1))

        weights -= learning_rate*(vw_a/(np.sqrt(sw_a)+1e-15))
        cost.append(mycost(target,features,weights))

    return(cost,weights)
cost_adam,w_adam=my_lr_adam(y,x,.1,1000)
w_sklearn/w_adam
```

```
[0.96120784, 1.01646743, 0.97955848]
```

Results are as good as RMSPROP with same learning rate and number of iterations . At the same time if you look at cost function descent through iterations , its much more smoother .

```
np.log(
       pd.DataFrame({'cost_gd':cost_gd,
                     'cost_sgd':cost_sgd,
                     'cost_mom':cost_mom,
                     'cost_rms':cost_rms,
                    'cost_adam':cost_adam})
                    ).plot()
```



Few important points to note

- You will never ever have to code these optimizers like this for real while building any of the machine learning solutions. These are implemented in sklearn, keras and we will be using those functions directly . The discussion here was done to make you understand the inner workings
- Improvements post SGD are targeted at achieving better results faster . However if we have enough time for higher number of iterations , you can achieve equivalent performance with just SGD as well. Many teams working in deep neural networks with billions of parameters with huge amount of resources and gigantic networks , simply use SGD for its simplicity

Optimization with constraints

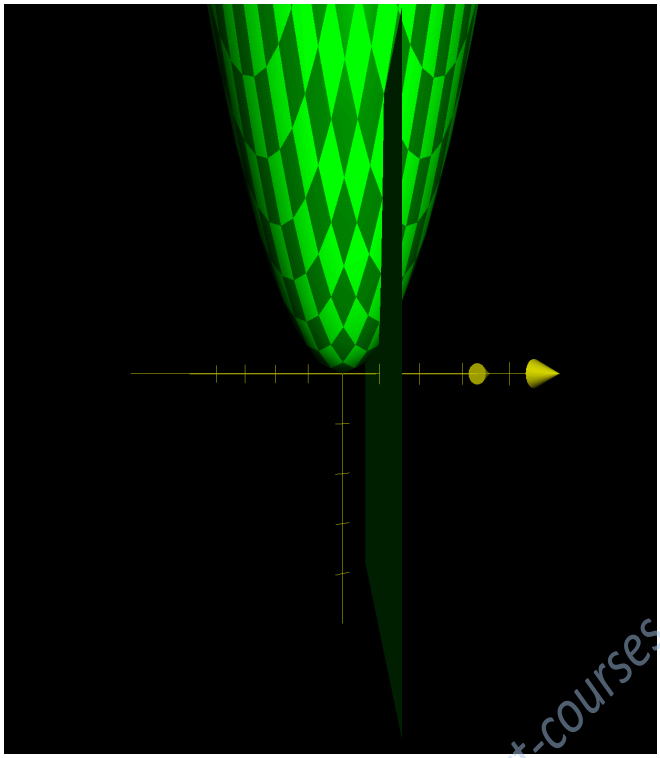
Although optimization in machine learning will mostly be about minimizing cost functions without any constraints, but sometimes you might come across scenarios where optimization include some kind of constraints, especially in unsupervised machine learning use cases . In this section we will talk about that.

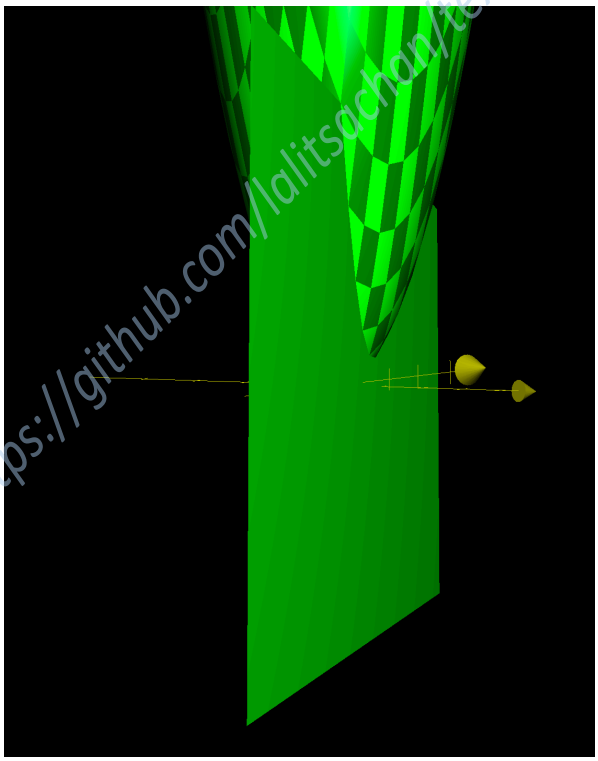
What is optimization with constraints a simple example

consider function $f(x) = x^2 + y^2$. If you were asked to find its minimum value. you will calculate its derivation $\begin{bmatrix} 2x & 2y \end{bmatrix}$, equate it to zero and find that the function has an optima at $x = 0, y = 0$, looking at the second derivative would have told you that its a minima.

But you need to take into account, this is only true if the parameters x, y in the function are allowed to take values in the interval which includes $x = 0, y = 0$. Now if the optimization problem given above was stated as below, it will have very different result.

$$\text{find minimum value of } f(x) = x^2 + y^2$$
$$\text{given that } x + y - 1 = 0$$





you can see that once x, y the parameters of the function are restricted to be on that plane $x + y - 1 = 0$ by the constraint, the optimal point changes, its at the bottom of the parabola formed by intersection of the function surface and the constraint; instead of being at $(0, 0)$. How do we go about finding the optimal point now?

Lagrange's Multiplier

In order to fully grasp the idea here , we first need to understand some fundamental ideas about how we look at the functions and how they and their derivatives behave .

1. Level Curves

as such a function g is not constrained, its free to take whatever values it can, given its formulation and parameter values with their full ranges.

a level curve is where a function enters into an equality with a constant c

$$g(w_0, w_1, w_2, \cdots, w_k) = c$$

Now in this context, only those values of the parameter W will be considered where the combination $[w_0 \quad w_1 \quad \cdots \quad w_k]$ is such that $g(w_0, w_1, w_2, \cdots, w_k) = c$

change vectors for level curves are parallel to the tangent on the level curves

the tangent vector at any particular value of W is the direction in which the function $g(W)$ remains constant, and any small movement along the curve can be represented by the vector $[\Delta w_0 \quad \Delta w_1 \quad \cdots \Delta w_k]$which is parallel to the tangent vector.

gradient of the function is perpendicular to the level curves

consider the fact that level curves are equal to a constant for all values of the parameters, so for any change vector $\Delta W = [\Delta w_0 \quad \Delta w_1 \quad \dots \quad \Delta w_k]$, the total change for the function itself will need to be 0 as long as we are on the level curve.

$$\Delta g = \frac{\partial g}{\partial w_0}\Delta w_0 + \frac{\partial g}{\partial w_1}\Delta w_1 + \cdots + \frac{\partial g}{\partial w_k}\Delta w_k = 0$$

$$[\frac{\partial g}{\partial w_0} \quad \frac{\partial g}{\partial w_1} \quad \cdots \quad \frac{\partial g}{\partial w_1}] \cdot [\Delta w_0 \quad \Delta w_1 \quad \dots \quad \Delta w_k] = 0$$

$$\nabla g \cdot \Delta W = 0$$

Since we have already established the fact that parameter change vector ΔW is parallel to the tangent; the expression $\nabla g \cdot \Delta W = 0$ implies that gradient of the function is perpendicular to the level curve.

Another intuitive way to think about this is that the tangent of the level curve essentially represents the direction along the curve , and there is zero change in the function along that direction since function is constant on the level curve . Gradient of the function on the other hand represents the direct of steepest change in the function, its natural that direction of no change and steepest change will be perpendicular to each other.

2. Proportionality of gradients for objective and constraints at the optimal point

Now consider that function that you need to optimize $f(W)$, where $W = [w_0 \quad w_1 \quad \cdots \quad w_k]$ under the constraint $g(W) = 0$.

we know that if we change W such that change vector ΔW is parallel to ∇g, it also implies that you can move along [by changing W] the level curve $g(W) = 0$ thus maintaining the constraint.

Now in order to find optimal value of f , you will need to move along ∇f, in order to most efficiently increase or decrease the function to reach optimal point.

these two movement directions will need to be parallel to each other, otherwise you will break the constraint defined by $g(w) = 0$.

hence we arrive at the condition for optimal point as

$$\nabla f = \lambda \nabla g$$

λ here is known as lagrange's multiplier . the above equation along with $g(W) = 0$ provides us with enough equations to solve for all $\begin{bmatrix} w_0 & w_1 & \dots & w_k \end{bmatrix}$ and λ at the optimal point.

Solving a constrained optimization problem with lagrange's multiplier method

find optimal value of $f(x, y) = x^2 + y^2$ s.t. $x + y - 1 = 0$.

here

$$\begin{aligned} W =& \begin{bmatrix} x & y \end{bmatrix} \\ f(W) =& x^2 + y^2 \\ g(W) =& x + y - 1 \end{aligned}$$

lets use $\nabla f = \lambda \nabla g$ for this :

$$\begin{aligned} \nabla f(x, y) &= \begin{bmatrix} 2x \\ 2y \end{bmatrix} \\ \nabla g(x, y) &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ \nabla f(x, y) &= \lambda \nabla g(x, y) \\ \begin{bmatrix} 2x \\ 2y \end{bmatrix} &= \begin{bmatrix} \lambda \\ \lambda \end{bmatrix} \\ x = y &= \frac{\lambda}{2} \end{aligned}$$

putting this back in the constraint $g(x, y) = 0$, we get $\lambda = 1$, which gives us the optimal point at $x = \frac{1}{2}, y = \frac{1}{2}$, with $f(\frac{1}{2}, \frac{1}{2}) = \frac{1}{2}$ as the optimal value.