Gradient Descent for Machine Learning

- Gradient descent
- Gradient descent algorithm - machine learning
- Gradient descent algorithm – some math
- Gradient descent algorithm – implementation in Python
- Illustration

https://medium.com/code-heroku/gradient-descent-for-machine-learning-3d871fa48b4c

Gradient Descent Algorithm (GDA)

Gradient descrescator; Gradient descendent; Coborârea gradientului; Coborârea pe gradient; Gradient de coborâre; **Coborârea pantei**

Picture this.
You want to be **the best basketball player** in the world. That means you want to score every time you shoot, have perfect passes, and always be in the right position at the right time for your teammates to pass to you.

Basically, you want to **reduce as much error as possible**. So, what to do?
You **train**. Much like perfecting basketball, gradient descent is an algorithm meant to minimize a certain cost function (room for error), so that the output is the most accurate it can be.

But before you start training, you need to have all your equipment. Who can play basketball without a ball? So, you need to know the function that you're trying to minimize (the cost function), its derivatives, and its current inputs, weight, and bias so you can get what you want: the most accurate output possible.

GDA is an algorithm used in almost every ML model.
The Gradient Descent serves to find the minimum of the cost function — basically the lowest point or deepest valley.

Gradient Descent Algorithm (GDA) - Analogy

A person is stuck in the mountains and is trying to get down (i.e. trying to find the global minimum). There is heavy fog such that visibility is extremely low. Therefore, the path down the mountain is not visible, so they **must use local information** to find the minimum.
They can use the method of **gradient descent**, which involves looking at the steepness of the hill at their current position, then proceeding in the direction with the **steepest descent** (i.e. downhill).
If they were trying to find the top of the mountain (i.e. the maximum), then they would proceed in the direction of steepest ascent (i.e. uphill).
Using this method, they would eventually find their way down the mountain or possibly get stuck in some hole (i.e., local minimum or saddle point), like a mountain lake.
However, assume also that the steepness of the hill is not immediately obvious with simple observation, but rather it requires a sophisticated instrument to measure, which the person happens to have now. It takes quite some time to measure the steepness of the hill with the instrument, thus they should minimize their use of the instrument if they wanted to get down the mountain before sunset. The difficulty then is choosing the frequency at which they should measure the steepness of the hill so not to go off track.

In this analogy, the person represents the algorithm, and the path taken down the mountain represents the sequence of parameter settings that the algorithm will explore.
The steepness of the hill represents the slope of the error surface at that point. The instrument used to measure steepness is differentiation (the slope of the error surface can be calculated by taking the derivative of the squared error function at that point). The direction they choose to travel in aligns with the gradient of the error surface at that point. The amount of time they travel before taking another measurement is **the learning rate of the algorithm**.

[https://en.wikipedia.org/wiki/Gradient_descent#An_analogy_for_understanding_gradient_descent]

Simple Linear Regression - revisited

Linear regression: a trend line that best fits the data

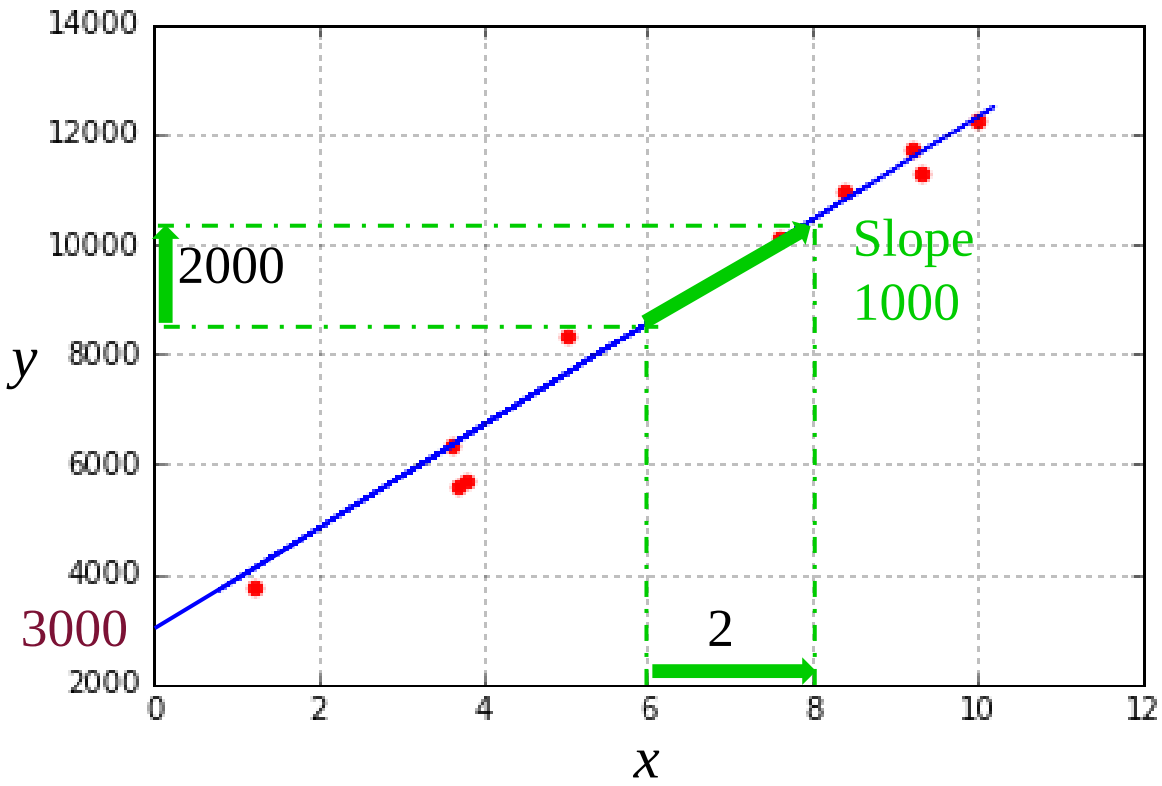
Red dots – facts; blue line – best fits the facts (the data) – linear regression

$$\hat{y} = ax + b$$

$$\hat{y} = 1000x + 3000$$

How would the computer know the right value of ***a*** and ***b*** for drawing the regression line with the minimum error?

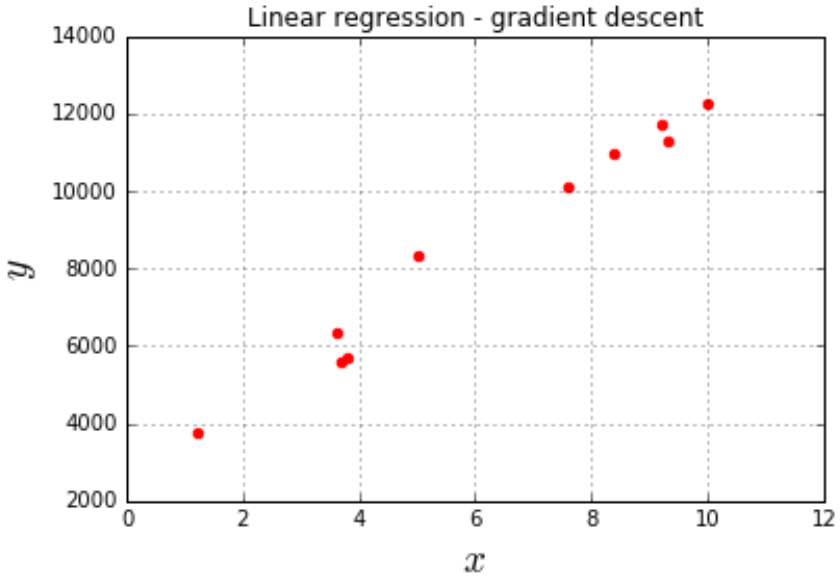
Gradient Descent Algorithm (GDA)



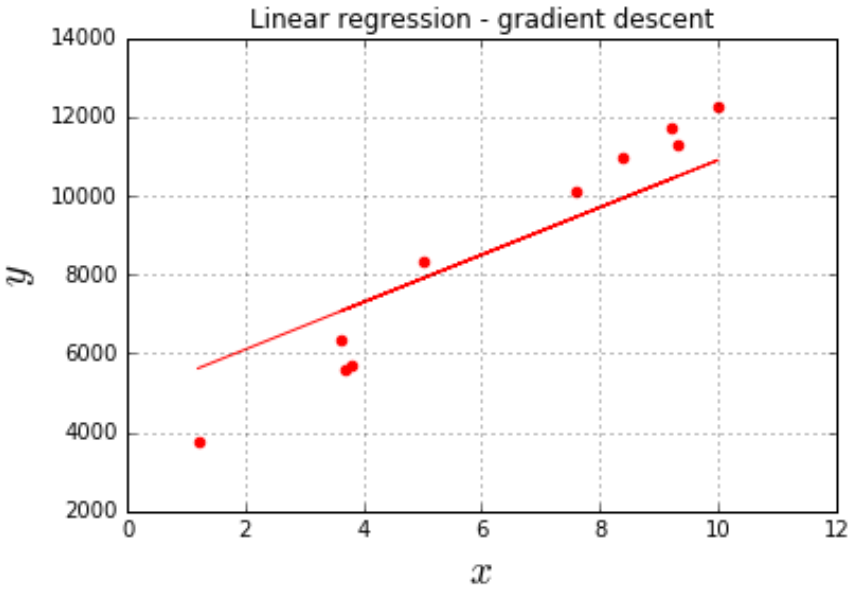
Gradient descent algorithm

- A trial-and-error method
- Iteratively give us different values of ***a*** and ***b*** to try

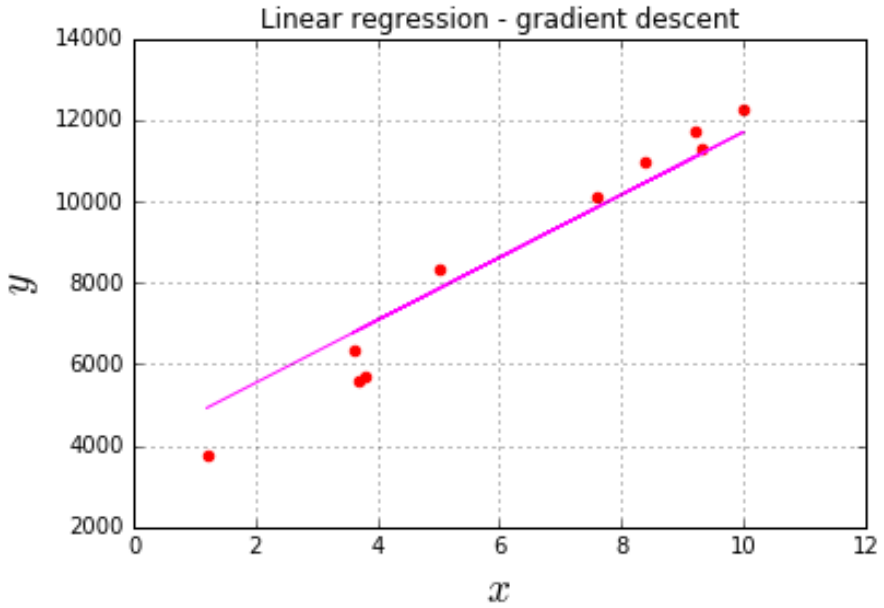
- In each iteration,
  - draw a regression line using ***a*** and ***b***
  - calculate the error for this model
  - adjust ***a*** and ***b*** to minimize the error
- Continue until we get ***a*** and ***b*** such that the **error is minimum**.



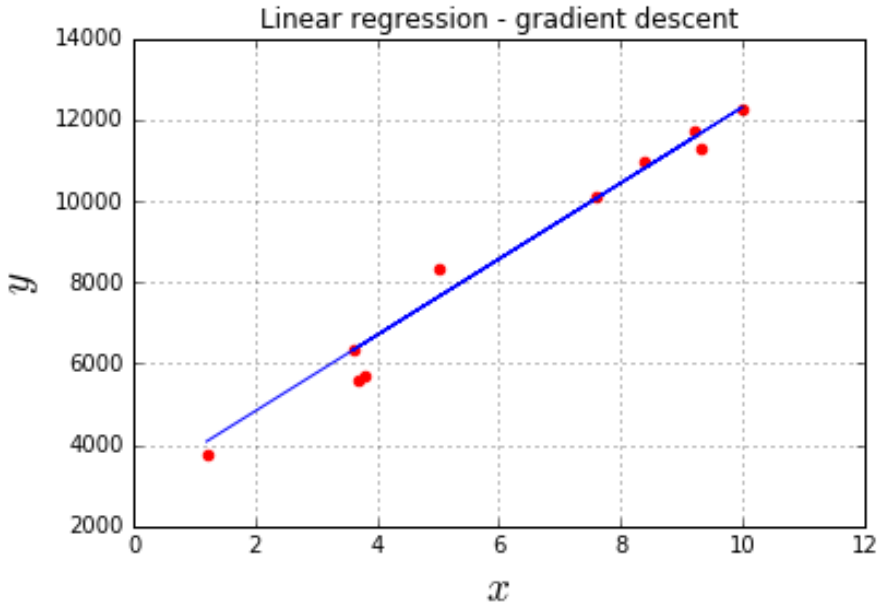
Step 1: Start with random values of ***a*** and ***b***



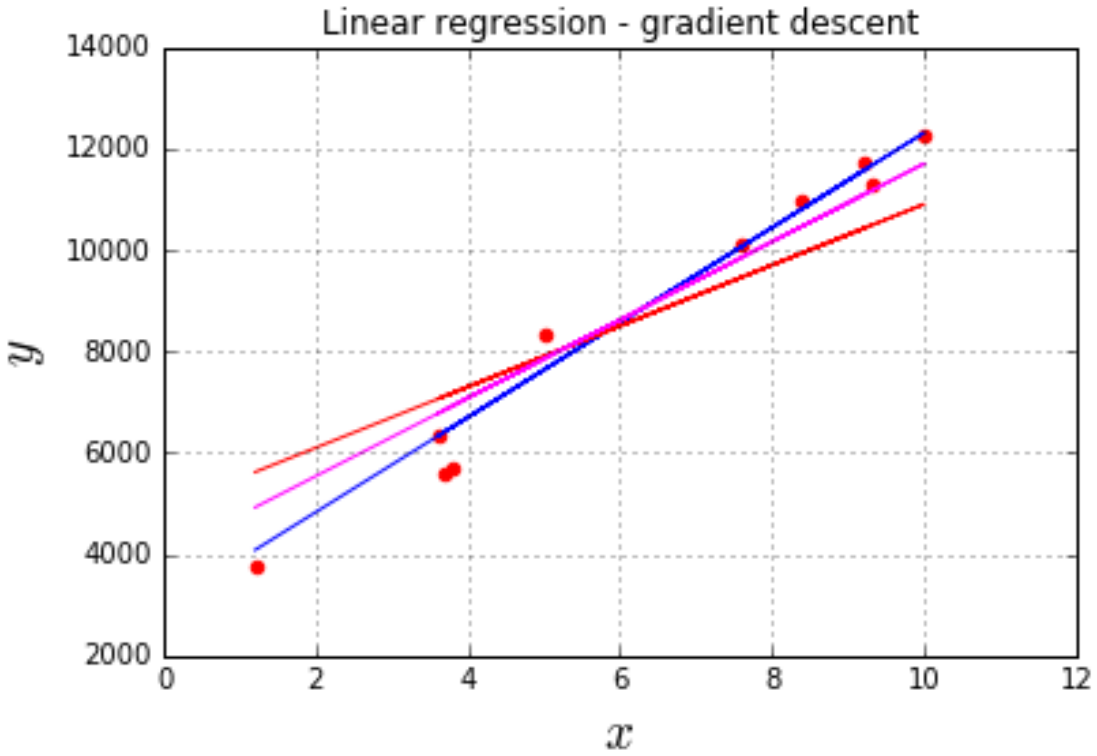
Step 2: Adjust ***a*** and ***b*** to reduce errors



Step 3: Repeat until converge to best approximation



Gradient descent algorithm



Algorithm evolution:

initial guess intermediate solution final-best approximation

Gradient descent is not limited to regression problems only.
It is an optimization algorithm which can be applied to any ML problem in general – including neuronal networks, deep learning

https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e

Cost Function

A Cost Function / Loss Function evaluate the performance of any machine learning algorithm.

- **Loss function** computes the error for a **single training example**
- **Cost function** usually is the average of the loss functions for **all the training examples**.

A cost function basically tells us ‘ how good’ our model is at making predictions for given values of ***a*** and ***b***

Cost function *J*: Mean squared-error

$$J = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

i^{th} example, $i = 1, \ldots, m$

$\hat{y}^{(i)}$ predicted values (estimated value)

$y^{(i)}$ reference values (ground truth, target, original, observed)

Minimizing the cost function

$$J = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(e^{(i)}\right)^2$$

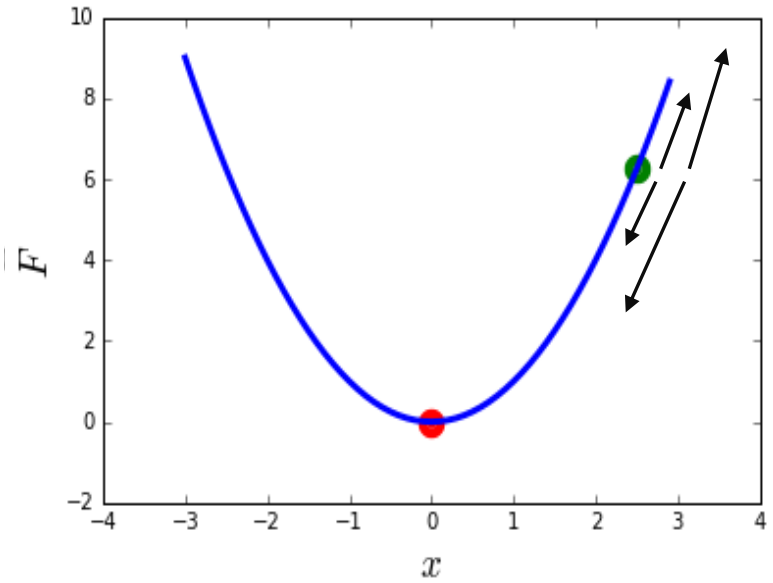
we want those ***a*** and ***b*** which give the smallest possible error.
The cost function J can be seen in fact as a simple squared function F:

$$\boldsymbol{F = x^2}$$

Currently: the 'green' dot.

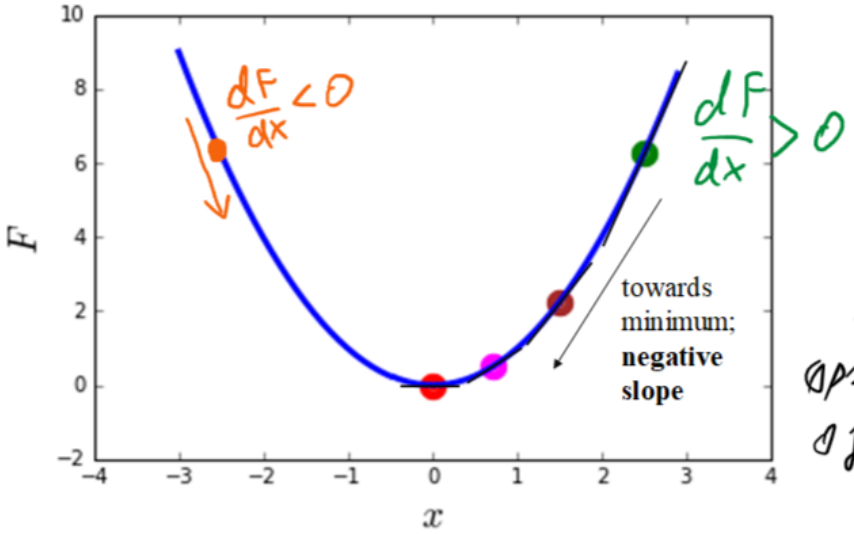
The aim is to reach the minimum (of the cost function) i.e the 'red' dot, but you don't know where it is (can't see it)

Possible action:
upward / **downward**
small / big step ?



Gradient Descent Algorithm makes these decisions (direction, step size) efficiently and effectively with the use of derivatives.

Derivative: the slope of the graph at a particular point.
The tangent gives a sense of the steepness of the slope



update x to find F minimum

$x := x - \eta \frac{dF}{dx}$

move on the opposite direction of the gradient

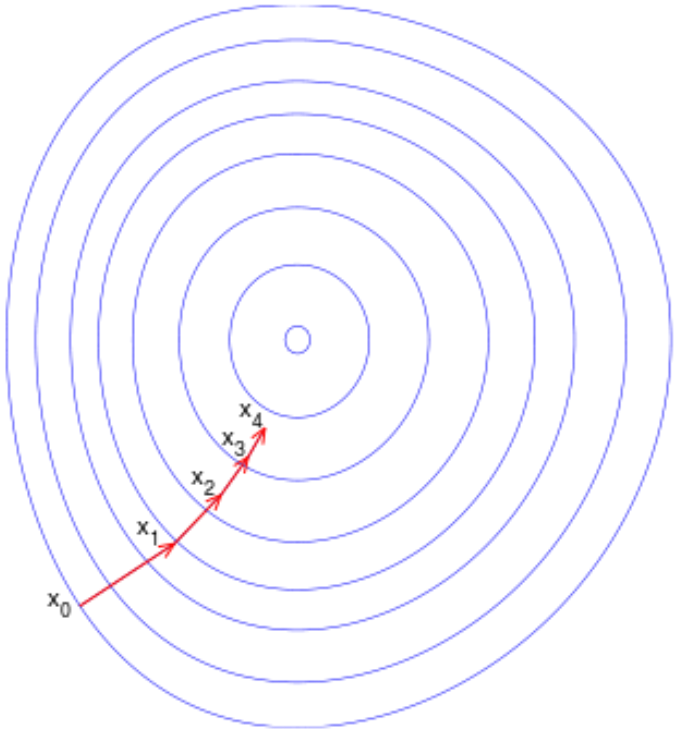
learning rate (control the step size)

$\frac{dF}{dx} > 0$; $x = x - \eta \frac{dF}{dx}$; $x \downarrow$; $F \downarrow$

$\frac{dF}{dx} < 0$; $x = x - \eta \frac{dF}{dx}$; $x \uparrow$; $F \downarrow$

The slope at the brown point is less steep than that at the green point; It will take smaller steps to reach the minimum from the brown point than from the green point.

Alternative view of GDA dynamic



https://en.wikipedia.org/wiki/Gradient_descent

The F function is assumed to be defined on the plane, and that its graph has a bowl shape.

The blue curves are the contour lines, that is, the regions on which the value of F is constant.

A red arrow originating at a point shows the direction of the negative gradient at that point.

The (negative) gradient at a point is orthogonal to the contour line going through that point.

Gradient descent leads to the bottom of the bowl, that is, to the point where the value of the function F is minimal.

BGD vs. SGD

Cost function $$J = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(e^{(i)}\right)^2$$

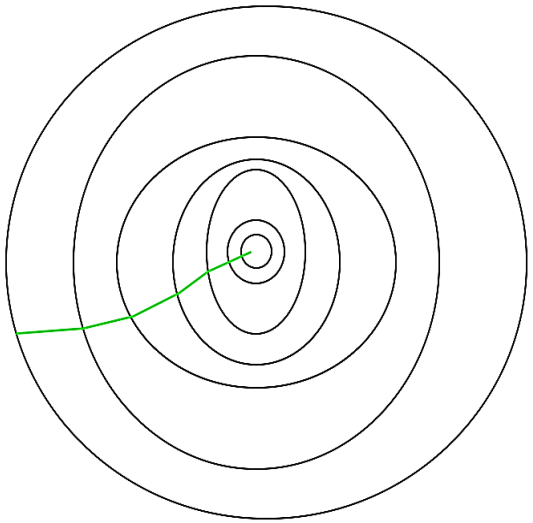
The summation part is important, especially with the concept of batch gradient descent (BGD) vs. stochastic gradient descent (SGD).

In **Batch Gradient Descent**, all the training data is taken into consideration to take a single step (one training epoch). We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters.

BGD is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.

The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

[https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a]



Path taken by BGD

[https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/?ref=rp]

In BGD we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge?

Deep learning models crave for data.
The more the data the more chances of a model to be good.

Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way.

To tackle this problem, we have

Stochastic Gradient Descent.

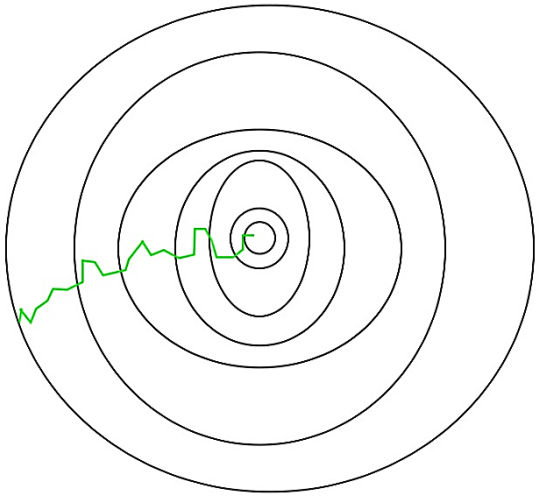
In **SGD**, we consider just one example at a time to take a single step (one training epoch).
[https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a]

The example is randomly shuffled and selected for performing the training epoch.

In SGD, since only one example from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than the one for the BGD. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with significantly shorter training time.

BGD vs. SGD – cont.

One thing to be noted is that, as SGD is generally noisier than BGD, it usually took a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than BGD, it is still computationally much less expensive than BGD.



Path taken by SGD

[https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/?ref=rp]

BGD vs. SGD – cont.

BGD can be used for smoother curves.
SGD can be used when the dataset is large.

BGD converges directly to minima.
SGD converges faster for larger datasets.

But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of BGD and SGD is used:

Mini Batch Gradient Descent

Neither we use all the dataset all at once nor we use the single example at a time.

We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both BGD and SGD.

We take the average of the gradients of the training examples in the mini-batch and then use that mean gradient to update the parameters.

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.

So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations

[https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a]

Gradient Descent Algorithm (GDA)

$$J = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(e^{(i)}\right)^2$$

Just to keep things simple, we will assume that we are looking at each error one at a time (SGD like) – ***algorithm intuition***

$\hat{y} = ax + b$ $\quad e = e(a,b)$ $\quad e$ is a function of a and b

$J = J(e) = J(a,b)$ $\quad J$ is a function of a and b

For simplicity, let's get rid of the summation sign and division by m, and use only the simplest version of the equivalent loss function F

$F = e^2$ $\quad F = F(a,b)$ $\quad F$ is a function of a and b

How does F depend on a and b, in general, but also, in each particular point?

Calculate partial derivatives of F w.r.t. a and b

$$\frac{\partial F}{\partial a} = \frac{\partial(e^2)}{\partial a} = 2e\frac{\partial e}{\partial a} \qquad \frac{\partial F}{\partial b} = \frac{\partial(e^2)}{\partial b} = 2e\frac{\partial e}{\partial b}$$

Chain rule applied

➢ Chain rule

If a variable *z* depends on the variable *y*, which itself depends on the variable *x* (i.e., *y* and *z* are dependent variables), then *z*, via the intermediate variable of *y*, depends on *x* as well.

$$z\big(y(\mathrm{x})\big)$$

The chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

$$\frac{\partial F}{\partial a} = \frac{\partial (e^2)}{\partial a} = 2e\frac{\partial e}{\partial a} \qquad \frac{\partial F}{\partial b} = \frac{\partial (e^2)}{\partial b} = 2e\frac{\partial e}{\partial b}$$

$$e = \hat{y} - y$$

$$\hat{y} = ax + b$$

$$e = ax + b - y$$

$$\frac{\partial e}{\partial a} = x \qquad \frac{\partial e}{\partial b} = 1$$

$$\frac{\partial F}{\partial a} = 2ex \qquad \frac{\partial F}{\partial b} = 2e$$

The variation of F w.r.t. a

- e
- x

The variation of F w.r.t. b

- e

$$\frac{\partial F}{\partial a} = 2ex$$

error input

$$\frac{\partial F}{\partial b} = 2e$$

error

Our cost (objective) function

$$F = e^2 = (ax + b - y)^2$$

will be minimized by updating the values for *a* and *b*

$$a := a - \eta \frac{\partial F}{\partial a} \qquad b := b - \eta \frac{\partial F}{\partial b}$$

✓ η - learning rate (constant) – controls the step size to reach the minimum of the cost function

$$\begin{aligned} a &:= a - \eta ex \\ b &:= b - \eta e \\ e &= \hat{y} - y = ax + b - y \end{aligned}$$

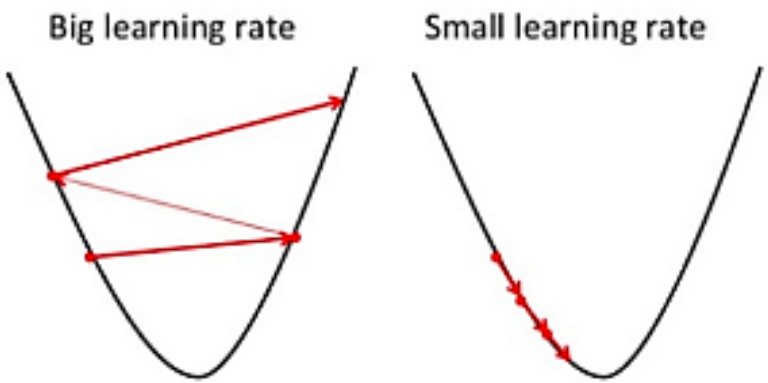
For linear regression $\hat{y} = ax + b$

Gradient descent is applied as

$$a := a - \eta \frac{\partial F}{\partial a} = a - \eta e x$$

$$b := b - \eta \frac{\partial F}{\partial b} = b - \eta e$$

One can cover more area with larger steps/higher learning rate but are at the risk of overshooting the minima and diverge.



On the other hand, small steps/smaller learning rates will consume a lot of time to reach the lowest point.

https://medium.com/code-heroku/gradient-descent-for-machine-learning-3d871fa48b4c

GDA for LR- overview

1) A random point is chosen initially by choosing random values of a and b.

2) Direction of the slope of that point is found by finding $\frac{\partial F}{\partial a}$ and $\frac{\partial F}{\partial b}$.

3) Since we want to move in the opposite direction of the slope, we will multiply -1 with both $\frac{\partial F}{\partial a}$ and $\frac{\partial F}{\partial b}$.

4) Since $\frac{\partial F}{\partial a}$ and $\frac{\partial F}{\partial b}$ gives only the direction, we need to multiply both with the learning rate (η) to specify the step size of each epoch.

5) Update the values of a and b such that the error (cost function) is reduced.

6) Repeat steps 2 to 5 until we converge at the minimum point of the cost function

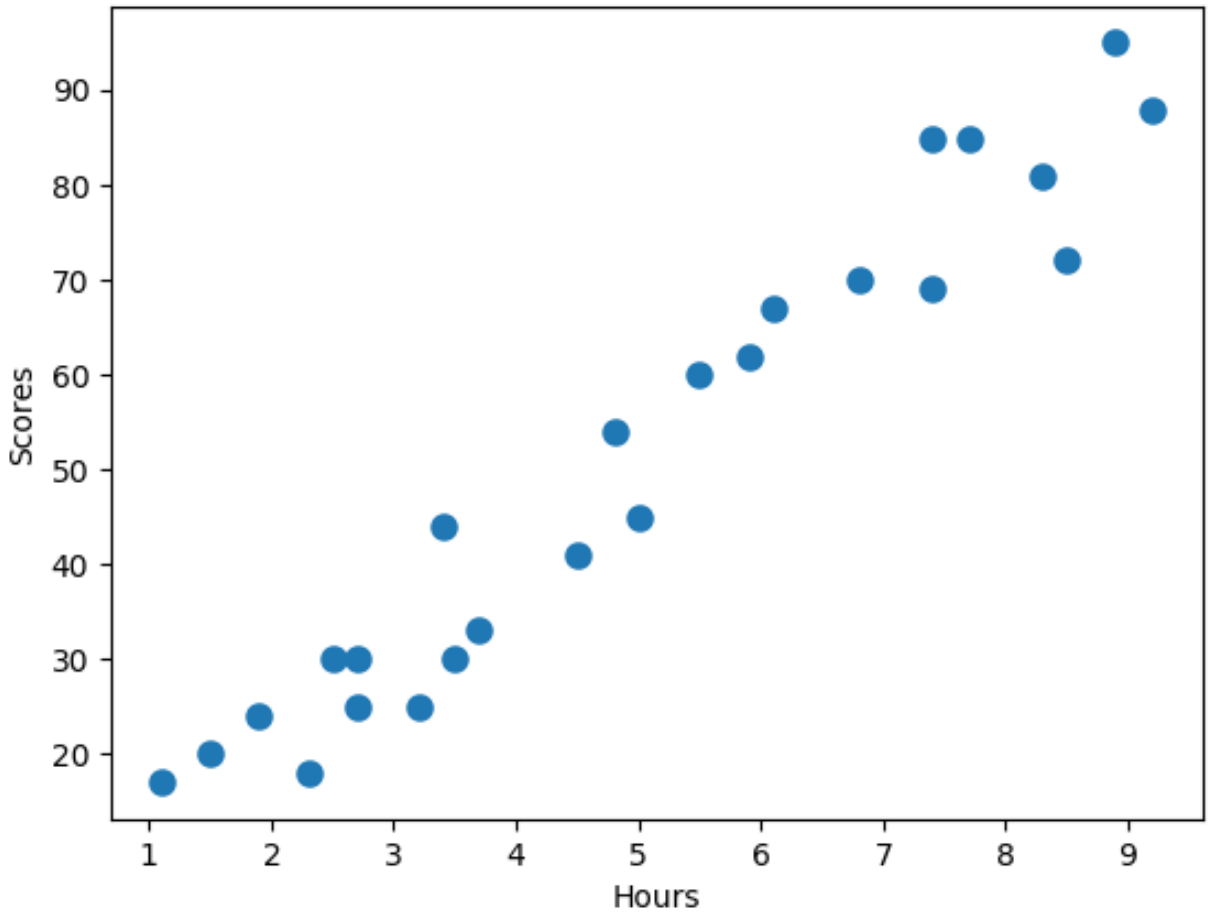
Case study – Python implementation

Problem:

Predict the score of a student based on number of hours studied

Data set: 25examples

Hours - Scores



$$\widehat{Scores} = a \cdot Hours + b$$

$a =?$ $b =?$

Python implementation

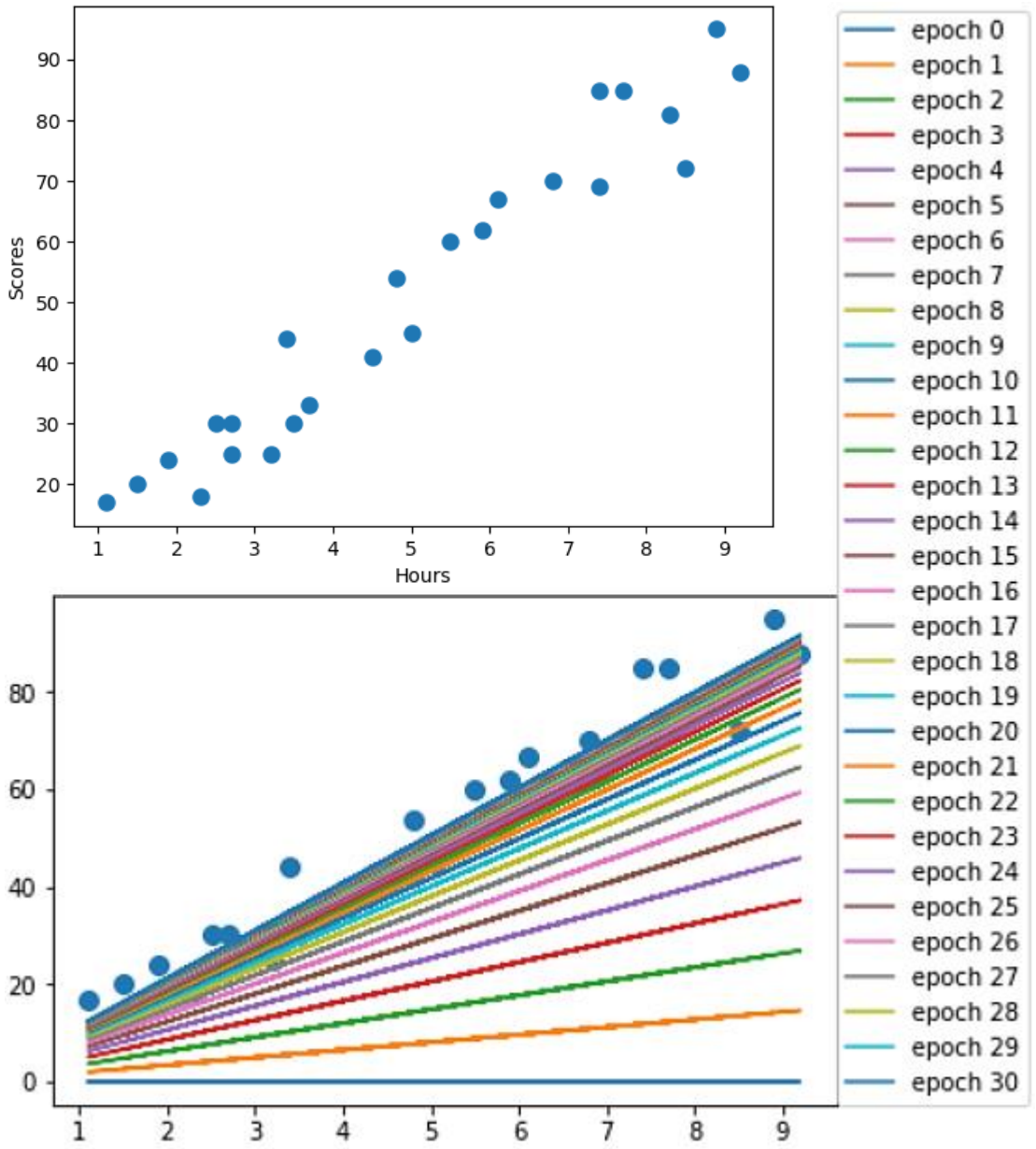
Colab notebook

https://colab.research.google.com/drive/14z1Xc6x-oXANp5Y2fLVq_I5SuHwmrPcB?usp=sharing

Results for BGD

eta = 0.005 # learning rate

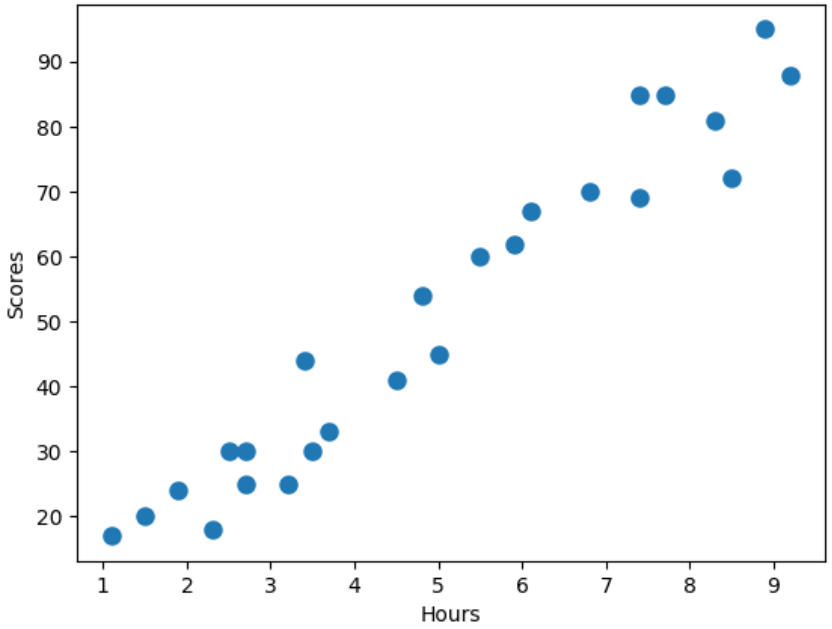
```
*****   BGD   *****
epochs     a          b           mse

  0   0.000000   0.000000 3189.760000
  1   1.559720   0.254000 2269.950225
  2   2.872441   0.467893 1618.378817
  3   3.977276   0.648030 1156.821083
  4   4.907142   0.799755 829.864511
  5   5.689747   0.927569 598.256198
  6   6.348408   1.035256 434.190321
  7   6.902753   1.126005 317.969899
  8   7.369299   1.202496 235.642058
  9   7.761950   1.266988 177.322914
 10   8.092407   1.321380 136.010960
 11   8.370517   1.367273 106.746494
 12   8.604571   1.406010  86.016183
 13   8.801545   1.438727  71.331263
 14   8.967309   1.466374  60.928755
 15   9.106807   1.489757  53.559805
 16   9.224197   1.509548  48.339756
 17   9.322980   1.526318  44.641938
 18   9.406103   1.540544  42.022431
 19   9.476045   1.552630  40.166775
 20   9.534894   1.562913  38.852212
 21   9.584406   1.571679  37.920948
 22   9.626060   1.579169  37.261202
 23   9.661100   1.585585  36.793795
 24   9.690573   1.591095  36.462635
 25   9.715361   1.595845  36.227990
 26   9.736206   1.599953  36.061714
 27   9.753732   1.603522  35.943869
 28   9.768465   1.606636  35.860331
 29   9.780847   1.609368  35.801096
 30   9.791251   1.611778  35.759077
```

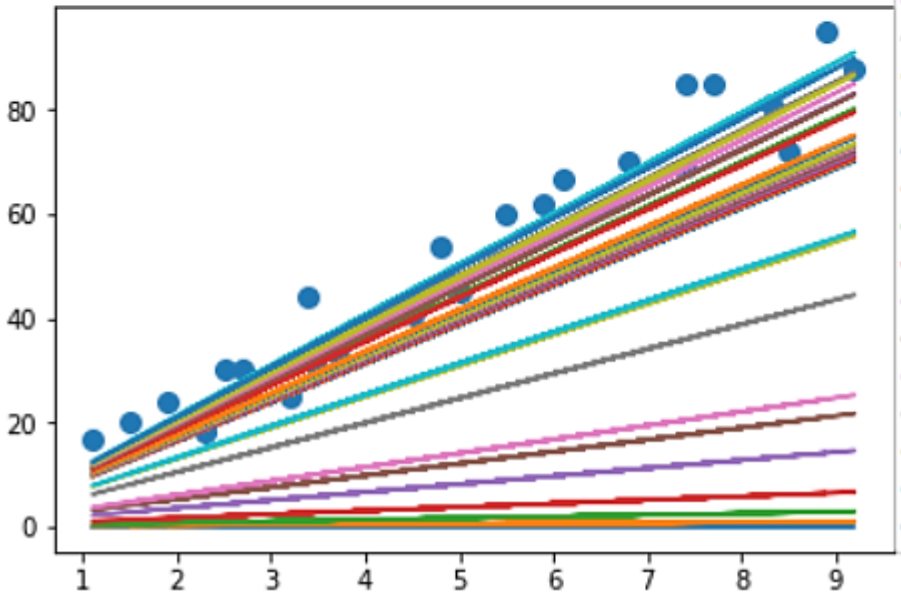


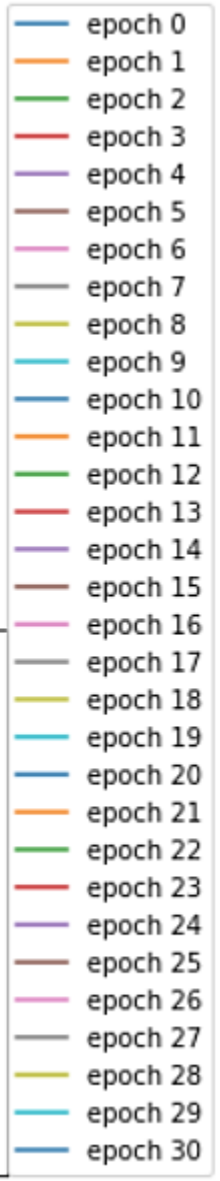
Results for SGD

eta = 0.005 # learning rate

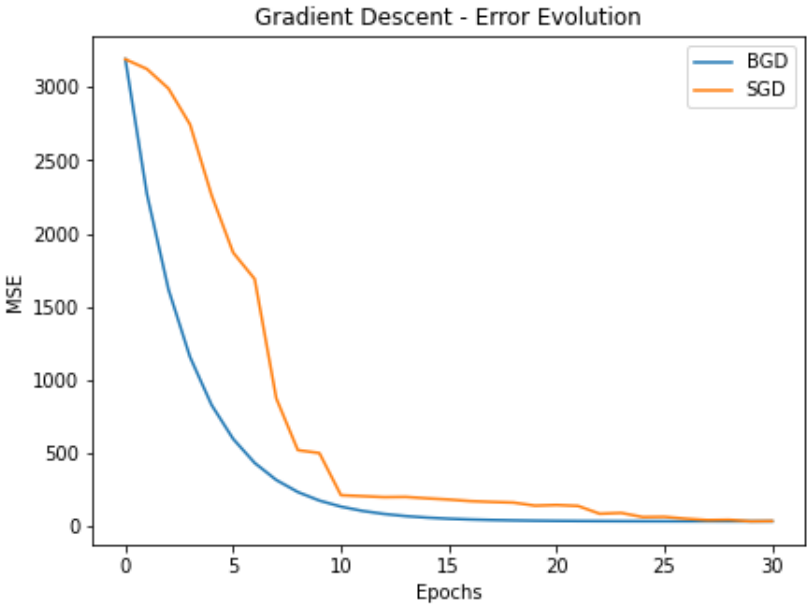
```
*****   SGD   *****
epochs     a         b          mse

  0   0.000000   0.000000 3189.760000
  1   0.093500   0.085000 3123.146650
  2   0.297049   0.173500 2990.073416
  3   0.688880   0.318622 2744.537210
  4   1.534462   0.506529 2261.627334
  5   2.290201   0.674471 1870.112544
  6   2.663122   0.781020 1689.118963
  7   4.727876   1.023932 875.162959
  8   5.948498   1.188881 521.636334
  9   6.022661   1.238323 501.706781
 10   7.472838   1.434293 214.024162
 11   7.515027   1.446347 207.532242
 12   7.555106   1.482783 200.891413
 13   7.545222   1.478485 202.406889
 14   7.611899   1.496506 192.567917
 15   7.665041   1.531934 184.472395
 16   7.740134   1.571457 173.623504
 17   7.778163   1.606029 167.981104
 18   7.810468   1.617993 163.638347
 19   7.980172   1.637958 142.820156
 20   7.945380   1.627086 147.067996
 21   7.993791   1.659360 140.859538
 22   8.532966   1.724321  87.524882
 23   8.468489   1.704172  93.092398
 24   8.854233   1.746101  64.779889
 25   8.828801   1.741014  66.377319
 26   9.060366   1.766184  53.567209
 27   9.291415   1.808193  44.109021
 28   9.232466   1.792261  46.221913
 29   9.699000   1.852850  35.974505
 30   9.572765   1.813402  37.425302
```

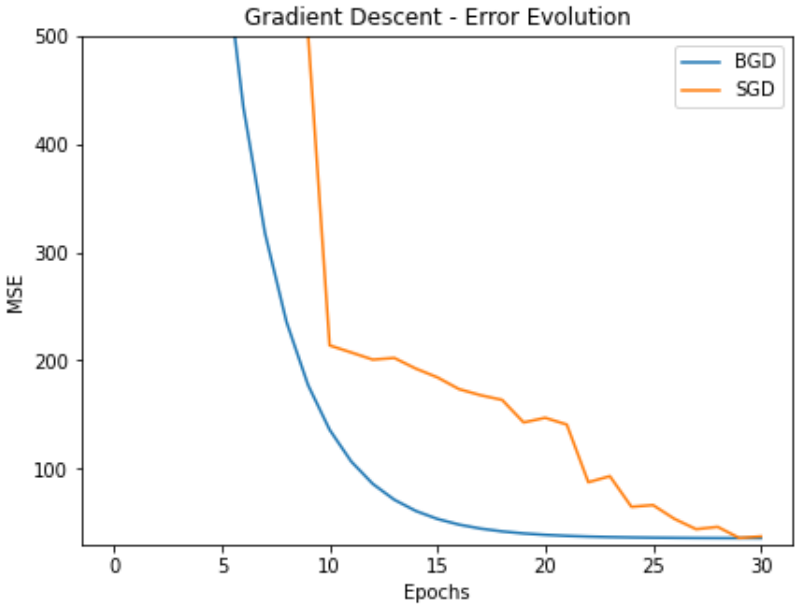


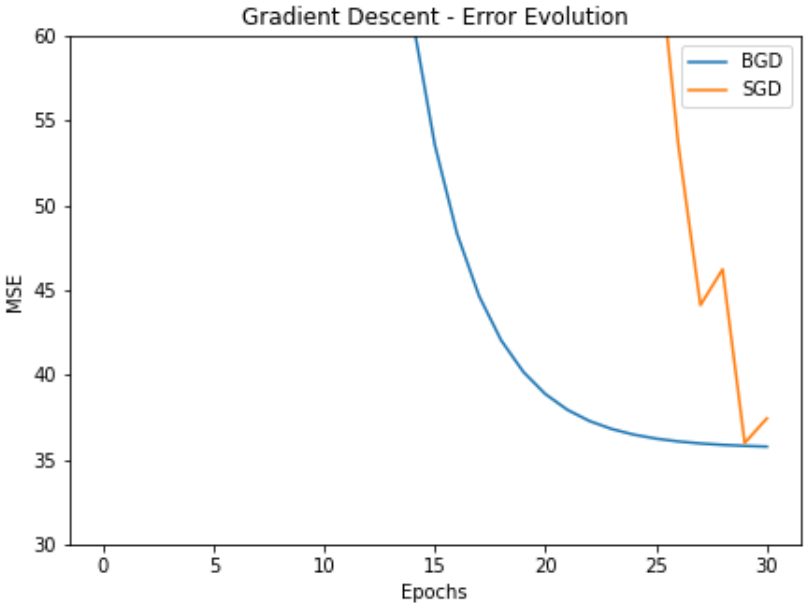




Results: BGD vs. SGD







For the data set of only 25 examples

BGD execution time for 1000 epochs:
0:00:01.555489 [s]

SGD execution time for 1000 epochs:
0:00:00.801469 [s]

Synthesis

$$\hat{y} = ax + b$$

where

$\hat{y}$ is the predicted value,

a is the slope (weight), b is the intercept (bias).

The objective of linear regression is to find the parameters a and b that minimize a cost function, typically the **Mean Squared Error (MSE),** defined as

$$J(a,b) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

$\hat{y}^{(i)}$ is the i-th predicted value,

$y^{(i)}$ is the i-th target value,

m is the number of examples (size of the dataset).

Synthesis

At each iteration the parameters a and b are updated:

$$a := a - \eta \frac{\partial J}{\partial a} \qquad a := a - \eta e x$$

$$b := b - \eta \frac{\partial J}{\partial b} \qquad b := b - \eta e$$

where

η is the **learning rate**, which controls the size of the steps we take toward the minimum of the cost function

$e = \hat{y} - y$ is the error

Synthesis

Gradient Descent Algorithm Steps for Linear Regression

1. **Initialize the parameters** a and b with some random values.
2. **Compute the predictions** $\hat{y}^{(i)} = ax^{(i)} + b$
3. **Compute the cost function** $J(a, b)$,
4. **Update the parameters** a and b using the gradient descent update rules.
5. **Repeat** steps 2-4 until the cost function converges (i.e., changes very little between iterations) or until a fixed number of iterations is reached.

Synthesis

For BGD (batch gradient descent)

Uses the entire dataset for every update (in each training epoch)

$$ex = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x^{(i)}$$

$$e = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

$$a := a - \eta ex = a - \eta\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x^{(i)}$$

$$b := b - \eta e = b - \eta\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

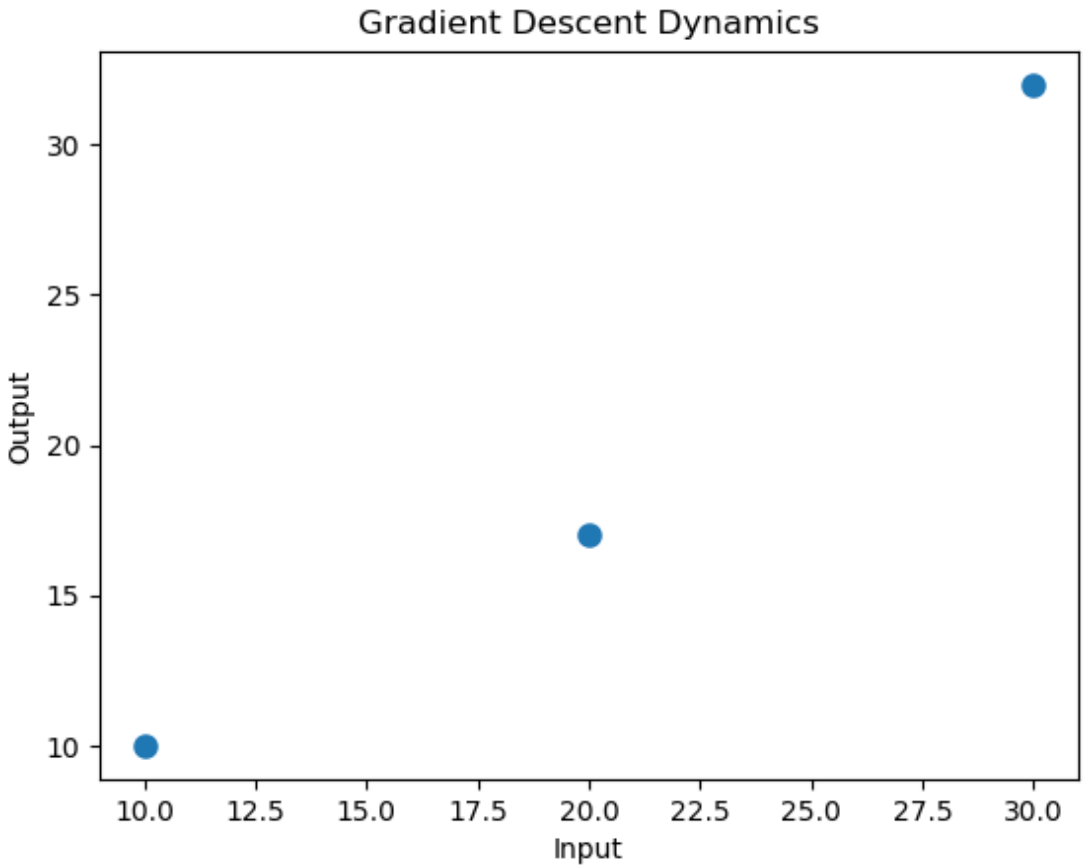
Synthesis

For SGD (Stochastic gradient descent)

In each training epoch, **only one training example** $\left(x^{(j)},\ y^{(j)}\right)$, **randomly selected** from the training set is used to update the parameters.

$$a := a - \eta e x = a - \eta(\hat{y}^{(j)} - y^{(j)})x^{(j)}$$

$$b := b - \eta e = b - \eta(\hat{y}^{(i)} - y^{(i)})$$



Problem

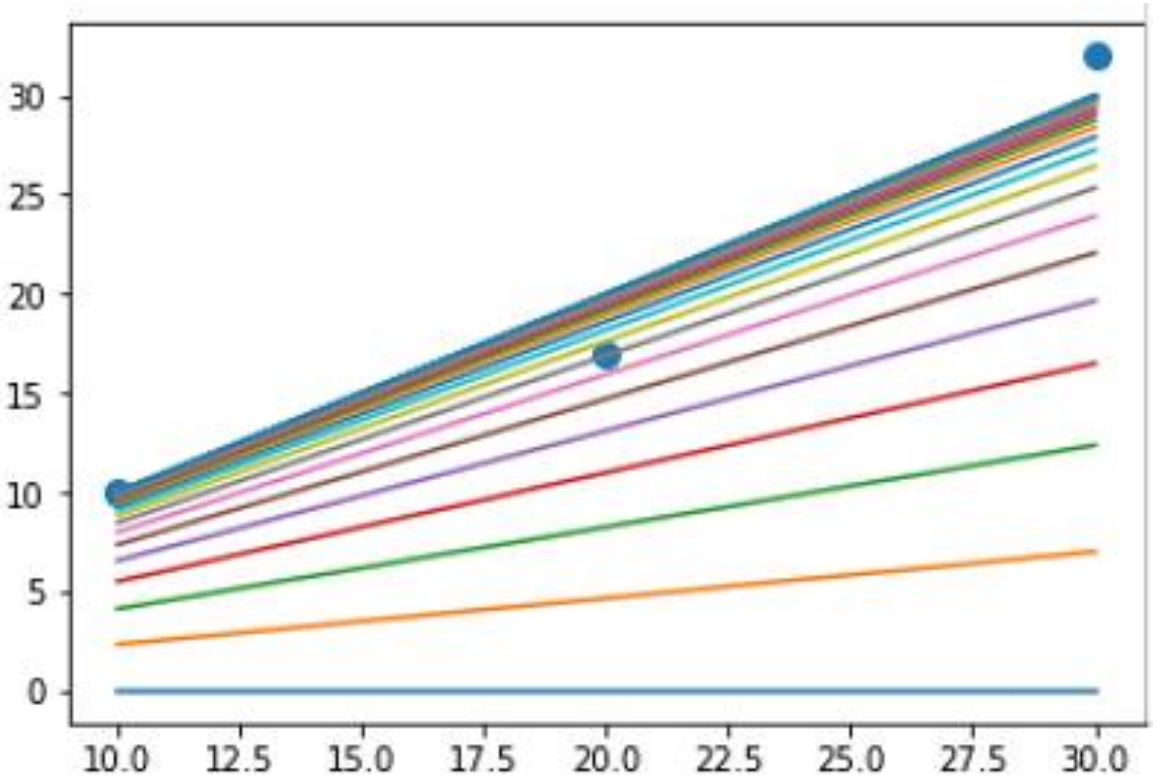
$$\widehat{Output} = a \cdot Input + b$$

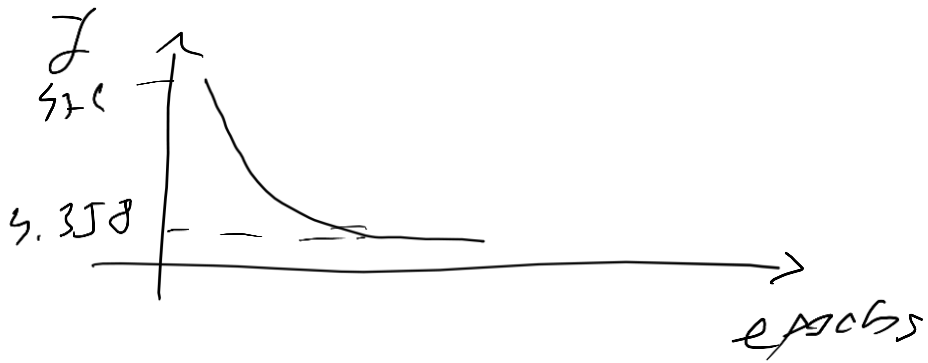
$a = ?$ $b = ?$

a) Perform the operations for the 1st training epoch of GDA considering the starting point $a = 0$, $b = 0$, $\eta = 0.0005$, in both cases: BGD and SGD.
b) What are the values of the cost function (MSE) in the starting point and after the 1st training epoch in both cases?

Solution BGD

```
*****   BGD   *****
epochs     a          b           mse

  0    0.000000   0.000000 471.000000
  1    0.233333   0.009833 278.334726
  2    0.412124   0.017328 165.216924
  3    0.549122   0.023032  98.803083
  4    0.654096   0.027362  59.810100
  5    0.734534   0.030641  36.916468
  6    0.796169   0.033114  23.475098
  7    0.843399   0.034969  15.583345
  8    0.879589   0.036351  10.949886
  9    0.907322   0.037370   8.229440
 10    0.928573   0.038111   6.632162
 11    0.944858   0.038640   5.694320
 12    0.957338   0.039005   5.143646
 13    0.966903   0.039246   4.820287
 14    0.974233   0.039391   4.630389
 15    0.979851   0.039462   4.518848
 16    0.984158   0.039477   4.453313
 17    0.987460   0.039449   4.414788
 18    0.989991   0.039388   4.392123
 19    0.991933   0.039302   4.378768
 20    0.993422   0.039196   4.370879
 21    0.994565   0.039076   4.366201
 22    0.995442   0.038944   4.363406
 23    0.996116   0.038803   4.361718
 24    0.996635   0.038656   4.360680
 25    0.997033   0.038504   4.360023
 26    0.997340   0.038347   4.359590
 27    0.997578   0.038188   4.359289
 28    0.997761   0.038027   4.359064
 29    0.997903   0.037863   4.358885
 30    0.998014   0.037699   4.358733
```





Solution SGD

```
*****   SGD   *****
epochs      a           b           mse
 0    0.000000    0.000000  471.000000
 1    0.170000    0.008500  325.543539
 2    0.305915    0.015296  228.737630
 3    0.648024    0.026699   61.789987
 4    0.836013    0.032966   16.689696
 5    0.838480    0.033089   16.317373
 6    0.846391    0.033880   15.160240
 7    0.945007    0.037167    5.689051
 8    0.947571    0.037424    5.563990
 9    0.950005    0.037667    5.450970
10    1.001938    0.039398    4.365957
11    1.030475    0.040349    4.844446
12    0.993976    0.038525    4.368151
13    1.026109    0.039596    4.720771
14    0.990491    0.037815    4.387784
15    1.024203    0.038938    4.671872
16    0.988973    0.037177    4.399846
17    0.989338    0.037214    4.396702
18    0.961099    0.035802    5.008987
19    0.938521    0.034673    6.036235
20    0.920470    0.033770    7.201234
21    0.924278    0.034151    6.929635
22    0.987840    0.036270    4.410187
23    0.959910    0.034873    5.051918
24    0.961740    0.035056    4.987409
25    0.963478    0.035230    4.929076
26    0.940430    0.034077    5.932034
27    0.996725    0.035954    4.358890
28    0.967021    0.034469    4.819596
29    0.943272    0.033281    5.782883
30    0.998300    0.035116    4.356938
```