Gradient Descent

Gradient descent

- **Gradient descent** is a **function** that measures the performance of a model for any given data
- GD is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function.
- This method is commonly used in *machine learning* (ML) and *deep learning*(DL) to minimise a cost/loss function (e.g. in a linear regression).
- its use is not limited to ML/DL only, it’s being widely used also in areas like:
  - control engineering (robotics, chemical, etc.)
  - computer games
  - mechanical engineering

Gradient descent

- Let’s say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?
- The best way is to observe the ground and find where the land descends.
- From that position, take a step in the descending direction and iterate this process until we reach the lowest point.

What are Local Minima and Global Minima in Gradient Descent?

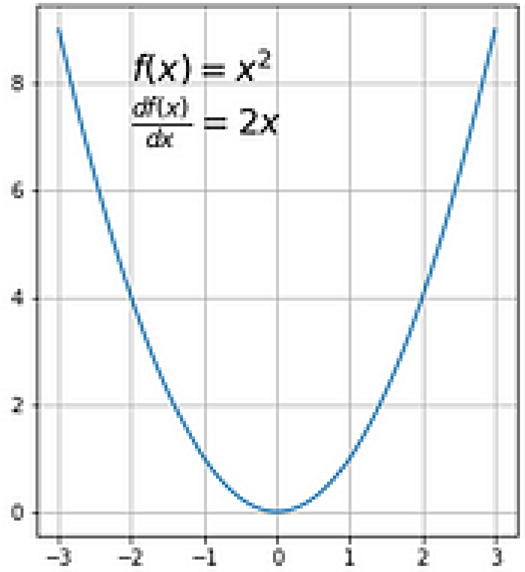
- **Local minima:**
  - The point in a curve which is minimum when compared to its preceding and succeeding points is called local minima.
- **Global minima:**
  - The point in a curve which is minimum when compared to all points in the curve is called Global Minima.
  - For a curve there can be more than one local minima, but it does have only one global minima.
- In gradient descent we use this local and global minima in order to decrease the loss functions.

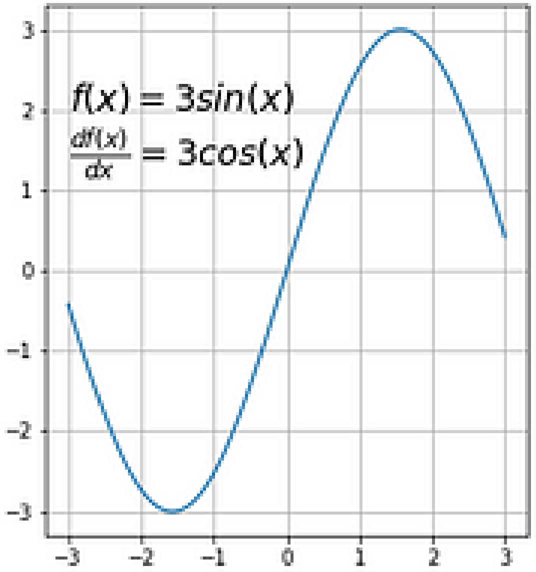
Function requirements

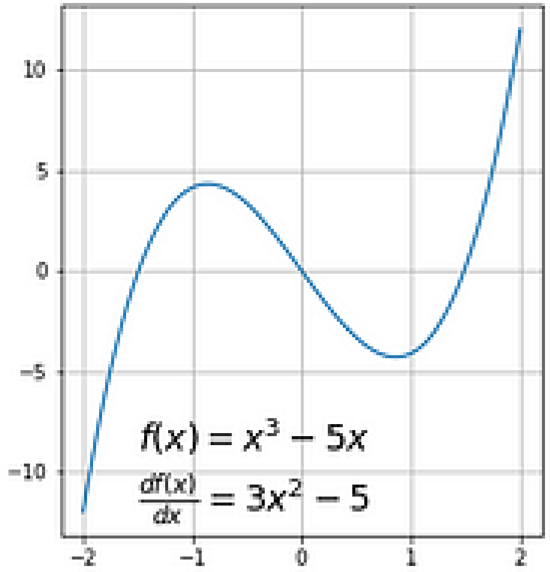
- Gradient descent algorithm does not work for all functions.
- A function has to be:
  - **differentiable**
  - **convex**
- First, what does it mean it has to be **differentiable**
- If a function is differentiable it has a derivative for each point in its domain — not all functions meet these criteria.

Differentiable functions

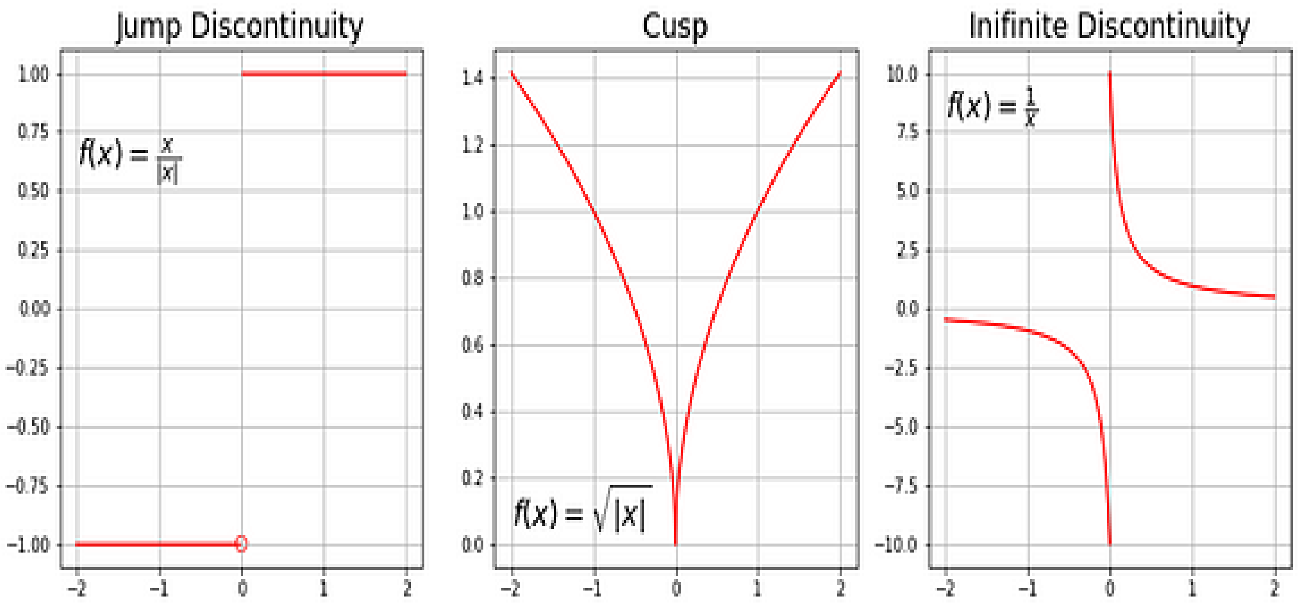
Some examples of functions meeting criterion - derivative for each point in its domain







Non-differentiable functions



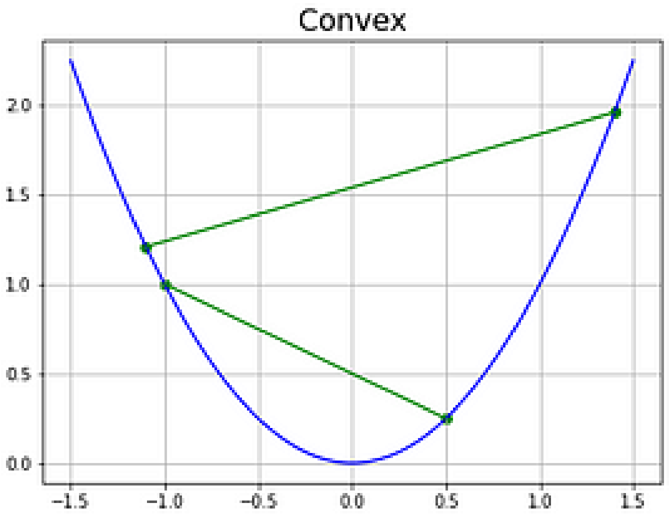
Typical non-differentiable functions have a step a cusp or a discontinuity

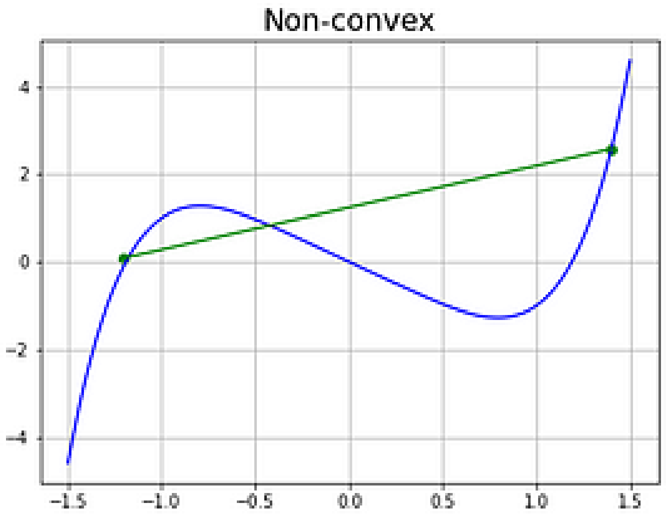
Convex function

- Next requirement — **function has to be convex**.
- Convex function :
- For a univariate function, the line segment connecting two function's points lays on or above its curve (it does not cross it).
- If it does it means that it has a local minimum which is not a global one.

Convex function

Two functions with exemplary section lines.





Convex function

- To check mathematically
- if a univariate function is convex :
  - calculate the second derivative and check if its value is always bigger than 0.

$$\frac{d^2 f(x)}{dx^2} > 0$$

Convex function -Example

Consider a simple quadratic function

$$f(x) = x^2 - x + 3$$

Its first and second derivative are:

$$\frac{df(x)}{dx} = 2x - 1, \qquad \frac{d^2 f(x)}{dx^2} = 2$$

second derivative is always bigger than 0, our function is strictly convex

Quasi Convex function -Example

- It is also possible to use **quasi-convex functions** with a gradient descent algorithm.
- They have so-called **saddle points** (called also minimax points) where the algorithm can get stuck

$$f(x) = x^4 - 2x^3 + 2$$

$$\frac{df(x)}{dx} = 4x^3 - 6x^2 = x^2(4x - 6)$$

- First derivative equal to zero at x=0 and x=1.5.
- These places are candidates for function’s extrema (minimum or maximum)
- Slope is zero at these places

Quasi Convex function -Example

Second derivative is,

$$\frac{d^2 f(x)}{dx^2} = 12x^2 + 12x = 12x(x-1)$$

- Value of this expression is zero for *x=0* and *x=1*.
- These locations are called an inflexion point

  — a place where the curvature changes sign

  — meaning it changes from convex to concave or vice-versa

By analysing this equation we conclude that :

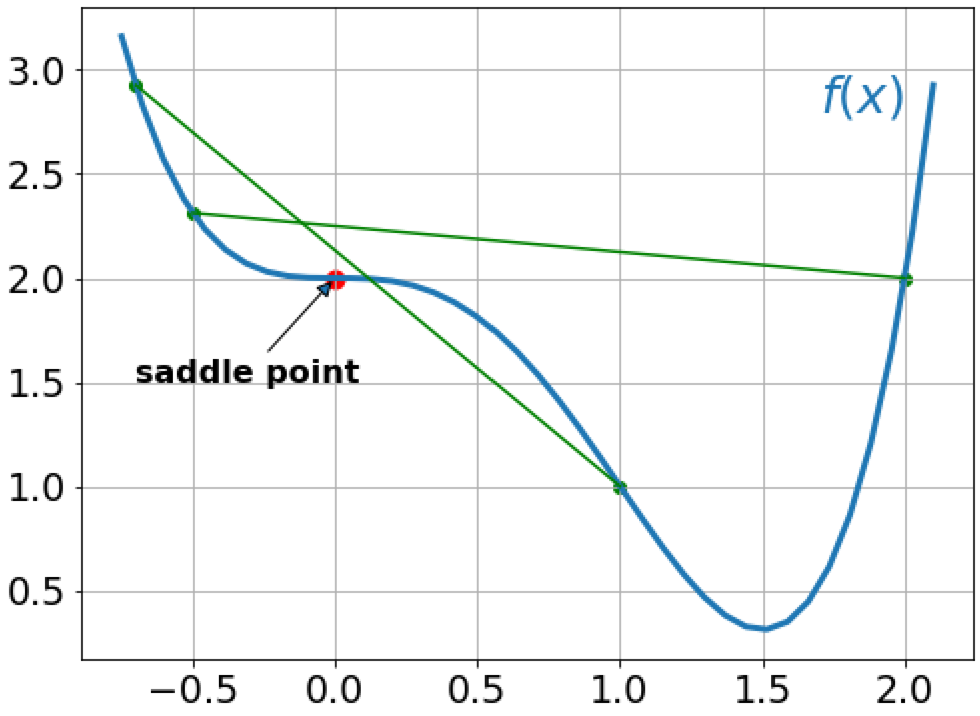
•for x<0: function is convex

•for 0<x<1: function is concave (the 2nd derivative < 0)

•for x>1: function is convex again

- Now we see that point *x=0* has both first and second derivative equal to zero meaning this is a saddle point
- point x=1.5 is a global minimum.

Saddle point -Example



Gradient

- What is a gradient?
  - Intuitively it is a slope of a curve at a given point in a specified direction.
  - In the case of **a univariate function**, it is simply the **first derivative at a selected point**.
  - In the case of **a multivariate function**, it is a **vector of derivatives** in each main direction (along variable axes).
    - We are interested only in a slope along one axis and don’t care about others - hence these derivatives are called **partial derivatives**
    - A gradient for an n-dimensional function f(x) at a given point p is defined as follows:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

Gradient

$$f(x) = 0.5x^2 + y^2$$

Let’s assume we are interested in a gradient at point p(10,10):

$$\frac{\partial f(x,y)}{\partial x} = x, \quad \frac{\partial f(x,y)}{\partial y} = 2y$$

So consequently:

$$\nabla f(x,y) = \begin{bmatrix} x \\ 2y \end{bmatrix}$$

$$\nabla f(10,10) = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

By looking at these values we conclude that the slope is twice steeper along the y axis.

Gradient Descent Algorithm

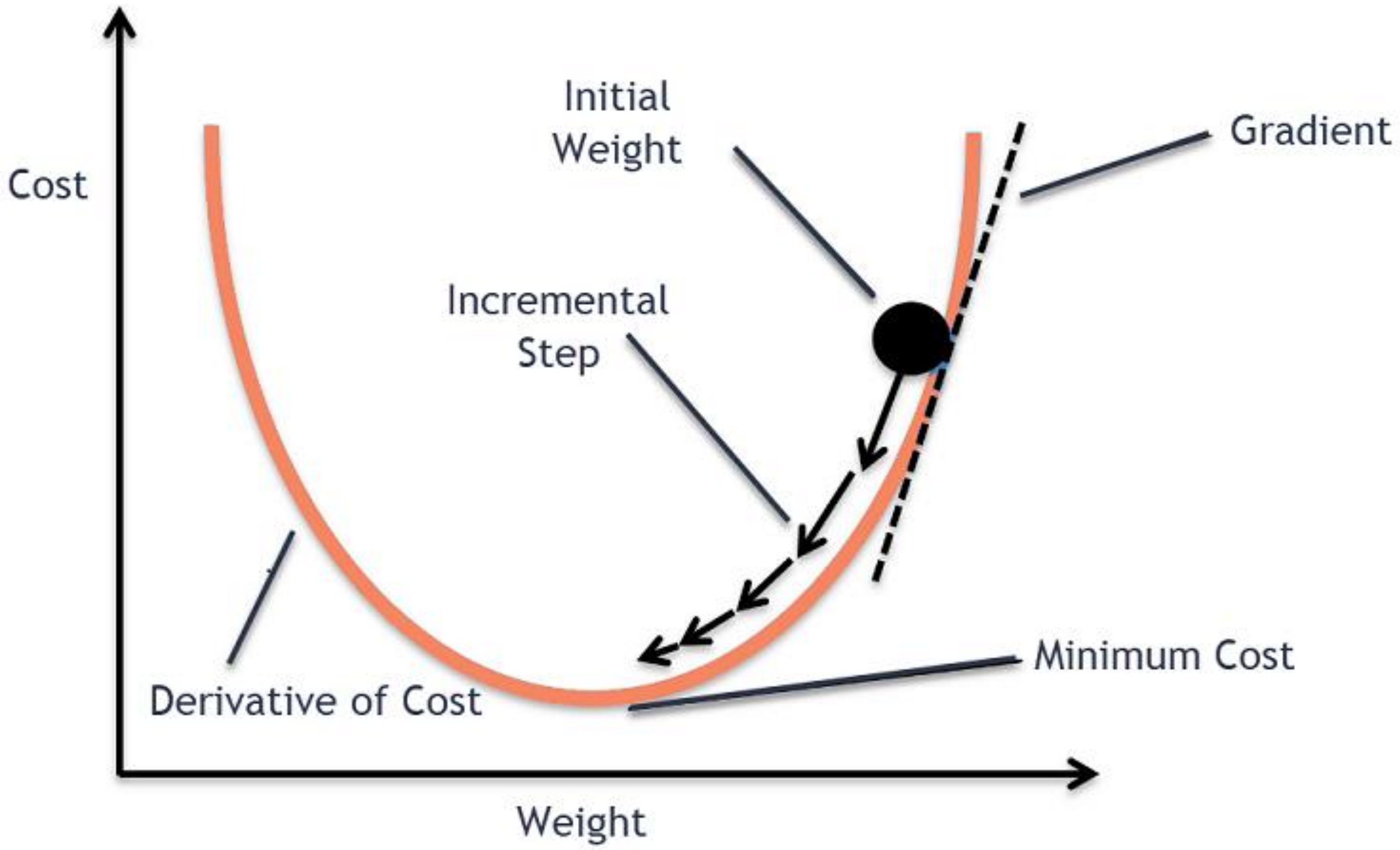
- Gradient Descent method's steps are:

  1. choose a starting point (random initialisation)
  2. calculate gradient at this point
  3. make a scaled step in the opposite direction to the gradient (objective: minimise)
  4. repeat points 2 and 3 until one of the below criteria is met:
     - maximum number of iterations reached
     - step size is smaller than the tolerance (due to scaling or a small gradient).

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

Parameter **η** which scales the gradient and thus controls the step size - called **learning rate**

Gradient Descent Algorithm



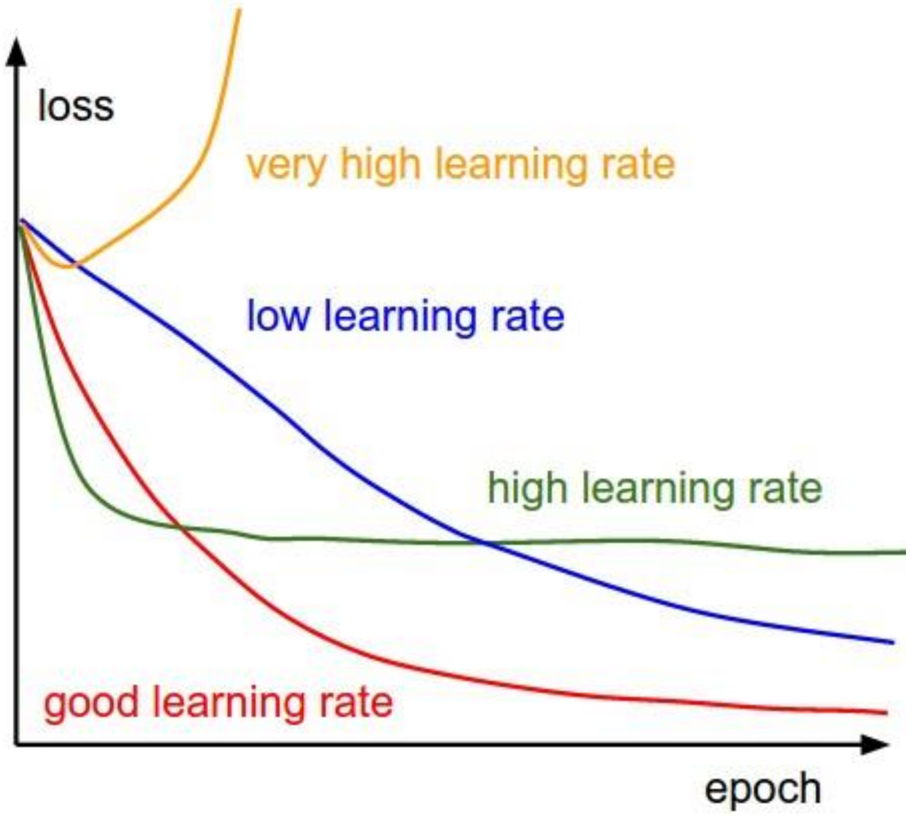
Gradient Descent Algorithm

- **Smaller learning rate**
  - Convergence of GD takes longer time
  - May reach maximum iteration before reaching the optimum point

- **Bigger learning rate**
  - GD algorithm may not converge to the optimal point (jump around or bounce) or even to diverge completely.

GD – Learning rate

- a) Learning rate is optimal, model converges to the minimum
- b) Learning rate is too small, it takes more time but converges to the minimum
- c) Learning rate is higher than the optimal value, it overshoots but converges ($1/C < \eta < 2/C$) where C-optimal value
- d) Learning rate is very large, it overshoots and diverges, moves away from the minima, performance decreases on learning



Source:Researchgate

Implementation of the Gradient Descent algorithm (with steps tracking):

```
import numpy as np
def gradient_descent(start, gradient, learn_rate,
max_iter, tol=0.01):
steps = [start] # history tracking

x = start

for _ in range(max_iter):
diff = learn_rate*gradient(x)
if np.abs(diff)<tol:
break
x = x - diff
steps.append(x) # history tracing

return steps, x
```

This function takes 5 parameters:

1. **starting point** - in our case, we define it manually but in practice, it is often a random initialisation
2. **gradient function** - has to be specified before-hand
3. **learning rate** - scaling factor for step sizes
4. maximum number of iterations
5. tolerance to conditionally stop the algorithm (in this case a default value is 0.01)

Example 1 — a quadratic function (GD algorithm)

Consider a simple quadratic equation

$$f(x) = x^2 - 4x + 1$$

It is an univariate function a gradient function is:

$$\frac{df(x)}{dx} = 2x - 4$$

these functions in Python

```
def func1(x):

    return x**2-4*x+1

def gradient_func1(x):
     return 2*x - 4
```

Example 1 — a quadratic function (GD algorithm)

- By taking:
  - a learning rate of 0.1 and
  - starting point at x=9
- Calculate each step by hand. Let’s do it for the first 3 steps:

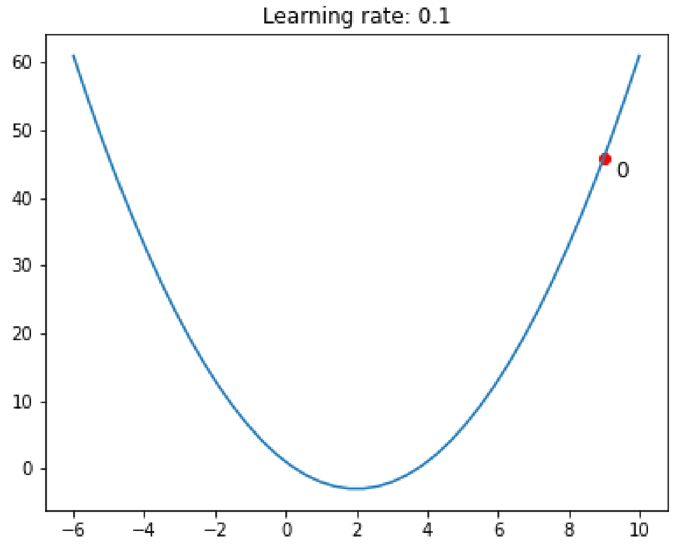
$$x_1 = 9 - 0.1 \cdot (2 \cdot 9 - 4) = 7.6$$
$$x_2 = 7.6 - 0.1 \cdot (2 \cdot 7.6 - 4) = 6.48$$
$$x_3 = 6.48 - 0.1 \cdot (2 \cdot 6.48 - 4) = 5.584$$

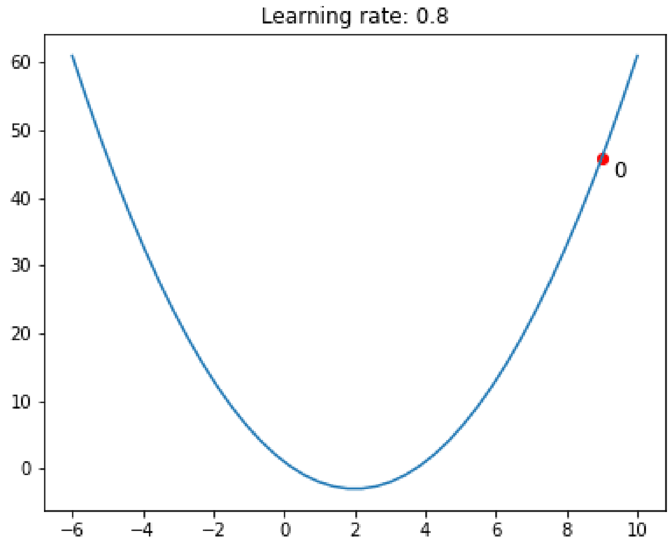
The python function is called by:

```
history, result = gradient_descent(9, gradient_func1, 0.1, 100)
```

Example 1 — a quadratic function (GD algorithm)

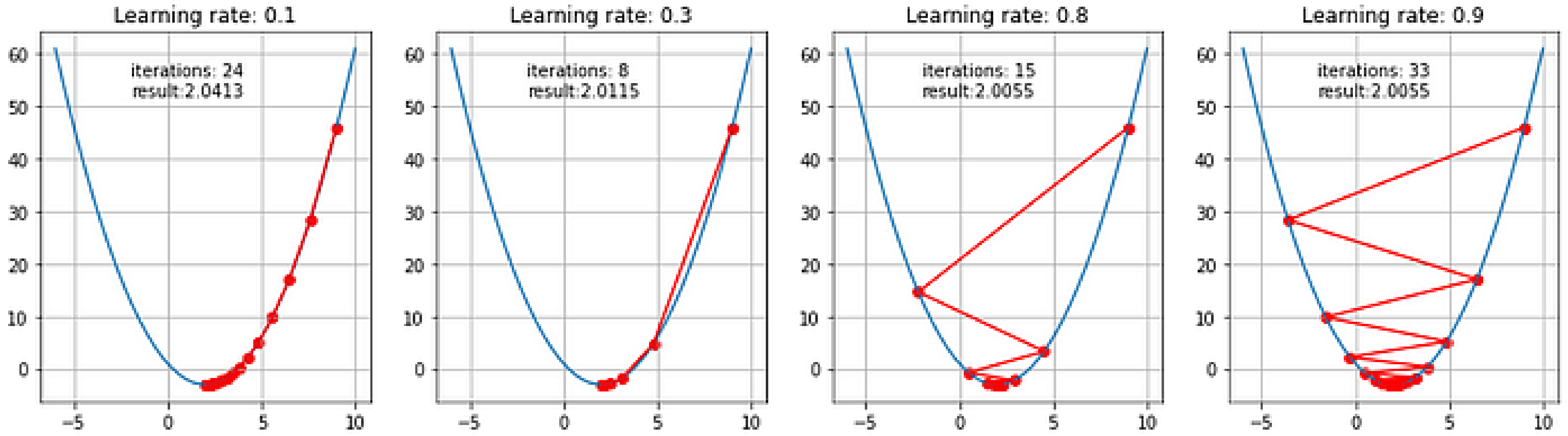
- The animation below shows steps taken by the GD algorithm for learning rates of 0.1 and 0.8.
- As you see, for the smaller learning rate, as the algorithm approaches the minimum the steps are getting gradually smaller.
- For a bigger learning rate, it is jumping from one side to another before converging.





Example 1 — a quadratic function (GD algorithm)

Trajectories, number of iterations and the final converged result (within tolerance) for various learning rates are shown below

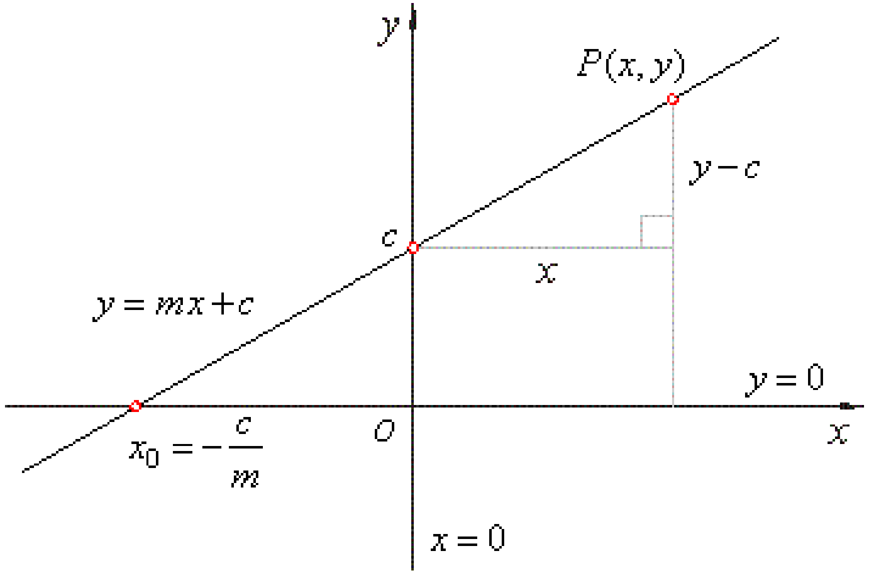


Linear Regression

- A linear approach to modelling the relationship between a dependent variable and one or more independent variables.
- Let **X** be the independent variable and **Y** be the dependent variable.
- A linear relationship between these two variables as:

$$Y=mX+c$$

Linear Regression



Y=mX+c

m is the slope of the line

c is the y intercept.

We use this equation to train our model with a given dataset and predict the value of **Y** for any given value of **X.**

Our challenge is to determine the value of **m** and **c**, such that the line corresponding to those values is the best fitting line or gives the minimum error.

Linear Regression

- **Loss Function**
- The loss is the error in our predicted value of **m** and **c**.
- Our goal is to minimize this error to obtain the most accurate value of **m** and **c**.
- We will use the Mean Squared Error function to calculate the loss. There are three steps in this function:
  - Find the difference between the actual y and predicted y value(y = mx + c), for a given x.
  - Square this difference.
  - Find the mean of the squares for every value in X.

$$E = \frac{1}{n} \sum_{i=0}^{n} (y_i - \bar{y}_i)^2$$

y_i is the actual value and $\bar{y}_i$ is the predicted value

Linear Regression

Substituting the value of $\bar{y}_i$:

$$E = \frac{1}{n} \sum_{i=0}^{n} (y_i - (mx_i + c))^2$$

Minimizing it and finding **m** and **c- Gradient descent algorithm**

Linear Regression - Applying Gradient Descent

- Let’s try applying gradient descent to **m** and **c** and approach it step by step:
- Initially let m = 0 and c = 0. Let L be our learning rate. This controls how much the value of **m** changes with each step. L could be a small value like 0.0001 for good accuracy.
- Calculate the partial derivative of the loss function with respect to m, and plug in the current values of x, y, m and c in it to obtain the derivative value **D**.

$$D_m = \frac{1}{n}\sum_{i=0}^{n} 2(y_i - (mx_i + c))(-x_i)$$

$$D_m = \frac{-2}{n}\sum_{i=0}^{n} x_i(y_i - \bar{y}_i)$$

D_m is the value of the partial derivative with respect to **m**

Linear Regression - Applying Gradient Descent

The partial derivative with respect to **c**, Dc :

$$D_c = \frac{-2}{n} \sum_{i=0}^{n} (y_i - \bar{y}_i)$$

3. Now we update the current value of **m** and **c** using the following equation:

$$m_{n+1} = m_n - L * D_m$$
$$c_{n+1} = c_n - L * D_c$$

4. We repeat this process until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy).
The value of **m** and **c** that we are left with now will be the optimum values.

Analogy – Reaching mountain valley

m can be considered the current position of the person.
D is equivalent to the steepness of the slope and
L can be the speed with which he moves.
Now the new value of **m** that we calculate using the above equation will be his next position, and
L×D will be the size of the steps he will take.
When the slope is more steep (**D** is more) he takes longer steps and
when it is less steep (**D** is less), he takes smaller steps.
Finally he arrives at the bottom of the valley which corresponds to our loss = 0.
Now with the optimum value of **m** and **c** our model is ready to make predictions !

Stochastic Gradient Descent (SGD)

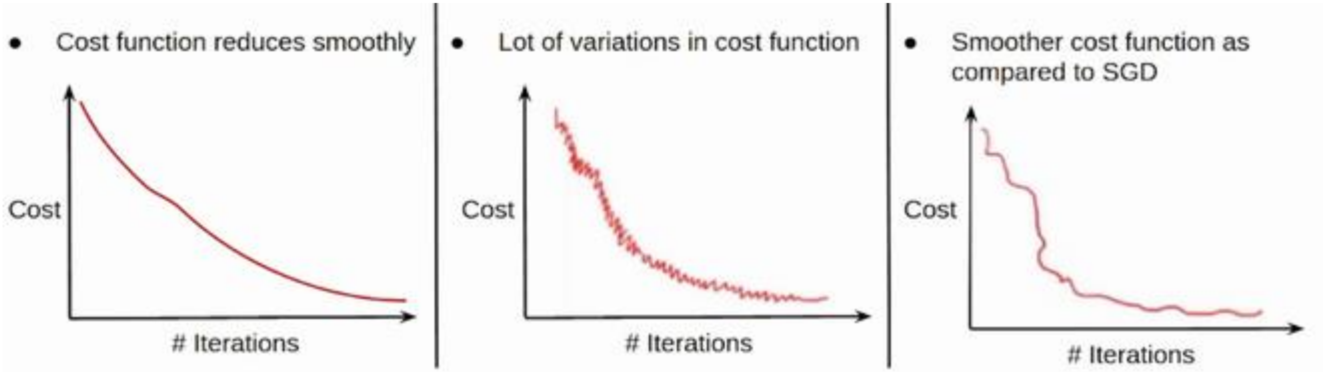
- A variant of the Gradient Descent algorithm used for optimizing machine learning models.
- In this variant, only one random training example is used to calculate the gradient and update the parameters at each iteration.
- **Advantages**:
- **Speed:** SGD is faster than other variants of Gradient Descent such as Batch Gradient Descent and Mini-Batch Gradient Descent since it uses only one example to update the parameters.
- **Memory Efficiency:** Since SGD updates the parameters for each training example one at a time, it is memory-efficient and can handle large datasets that cannot fit into memory.
- **Avoidance of Local Minima:** Due to the noisy updates in SGD, it has the ability to escape from local minima and converge to a global minimum.

Stochastic Gradient Descent (SGD)

- **Disadvantages:**
- **Noisy updates:** The updates in SGD are noisy and have a high variance, which can make the optimization process less stable and lead to oscillations around the minimum.
- **Slow Convergence:** SGD may require more iterations to converge to the minimum since it updates the parameters for each training example one at a time.
- **Sensitivity to Learning Rate:** The choice of learning rate can be critical in SGD since using a high learning rate can cause the algorithm to overshoot the minimum, while a low learning rate can make the algorithm converge slowly.
- **Less Accurate:** Due to the noisy updates, SGD may not converge to the exact global minimum and can result in a suboptimal solution. This can be mitigated by using techniques such as learning rate scheduling and momentum-based updates

Minibatch Gradient

- Parameters are updated after computing the gradient of the error with respect to a subset of the training set



Batch Gradient Descent	Stochastic Gradient Descent	Mini-Batch Gradient Descent
Since the entire training data is considered before taking a step in the direction of gradient, therefore it takes a lot of time for making a single update.	Since only a single training example is considered before taking a step in the direction of gradient, we are forced to loop over the training set and thus cannot exploit the speed associated with vectorizing the code.	Since a subset of training examples is considered, it can make quick updates in the model parameters and can also exploit the speed associated with vectorizing the code.
It makes smooth updates in the model parameters	It makes very noisy updates in the parameters	Depending upon the batch size, the updates can be made less noisy – greater the batch size less noisy is the update