**Gradient Descent**

Table of Contents

[Intuition 2](#_Toc106494886)

[Formulae 3](#_Toc106494887)

[Learning Rate 4](#_Toc106494888)

[Generic Functions 6](#_Toc106494889)

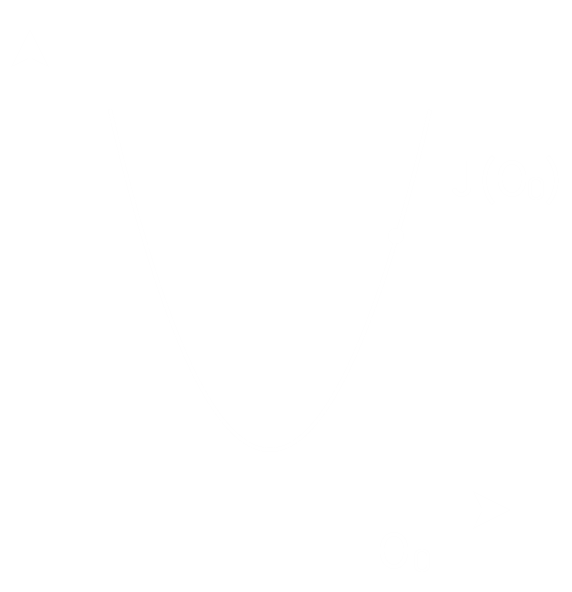
[Number of Iterations 7](#_Toc106494890)

[Derivatives 7](#_Toc106494891)

[Batch vs Stochastic vs Mini-Batch Gradient Descent 8](#_Toc106494892)

As a reminder, our objective is to find . Although we are currently using **gradient descent** for just two parameters, it can be used with any number of parameters as well.

## Intuition



Consider that we have a **convex graph** in a **2D space**, such as the one above. For such a graph, if we have a point that is on the **right-side** of the graph, we need to **decrease** to reach the minima point, and if we have a point that is on the **left-side** of the graph, we need to **increase** to reach the minima point.

Another way of looking at this is that we need to subtract the gradient, multiplied by some constant, to find the value of for which we reach the minima point. The **gradient** on the **right side** of this graph is **positive**, while the gradient on the **left side** of this graph is **negative**, so subtracting the gradient **decreases** and **increases** the value of respectively, which is exactly what we need. If we are already at the minima, the gradient will be , so subtracting it will make no difference.

This is essentially what happens in gradient descent, albeit in a multi-dimensional space.

## Formulae

Firstly, we pick a random point . Normally, the point picked is .

Next, we apply the following equations.

We repeatedly apply the above formula until we reach **convergence**, i.e. the values of and stop changing.

There’s a whole lot to unpack here, so lets go step-by-step.

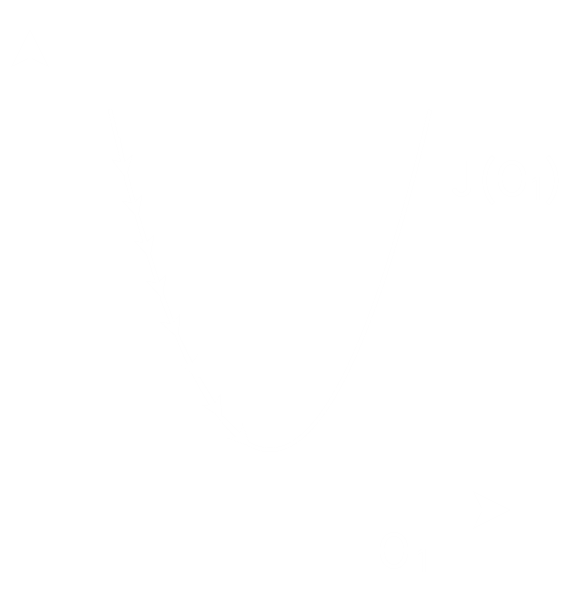
Firstly, notice that the **assignment** operator, , is being used instead of a normal sign. This is because we will be using the sign to show that two values are **equivalent**.

Secondly, the here is called the **step size** or the **learning rate**. This is a value that we specify. We will get back to the details and significance of this later on.

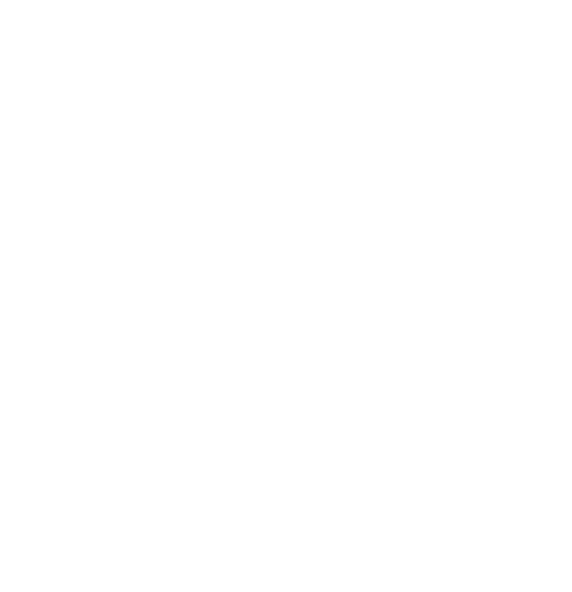
Finally, we are doing things a little weirdly in the equations above. Why did we take the values of the actual equations and store them in **temporary variables**, only to go and assign them to and respectively in just the next step? The reason is, at the first equation, if we **directly set** the result to , then we will end up using the **new value** of to calculate the **new value** of in the next equation. What we need however, is to use the **old value** of . To deal with that, we have to store the results in temporary variables. What we did above is called a **simultaneous update**. The actual equation mathematically (ignoring the problems of directly assigning the value) would look like this:

## Learning Rate

Coming back to , the **learning rate**, this value determines how large the **change** from one value of to the next value is.



If we set the value to a **very small number**, then we will need to change many times before we reach the point of convergence.

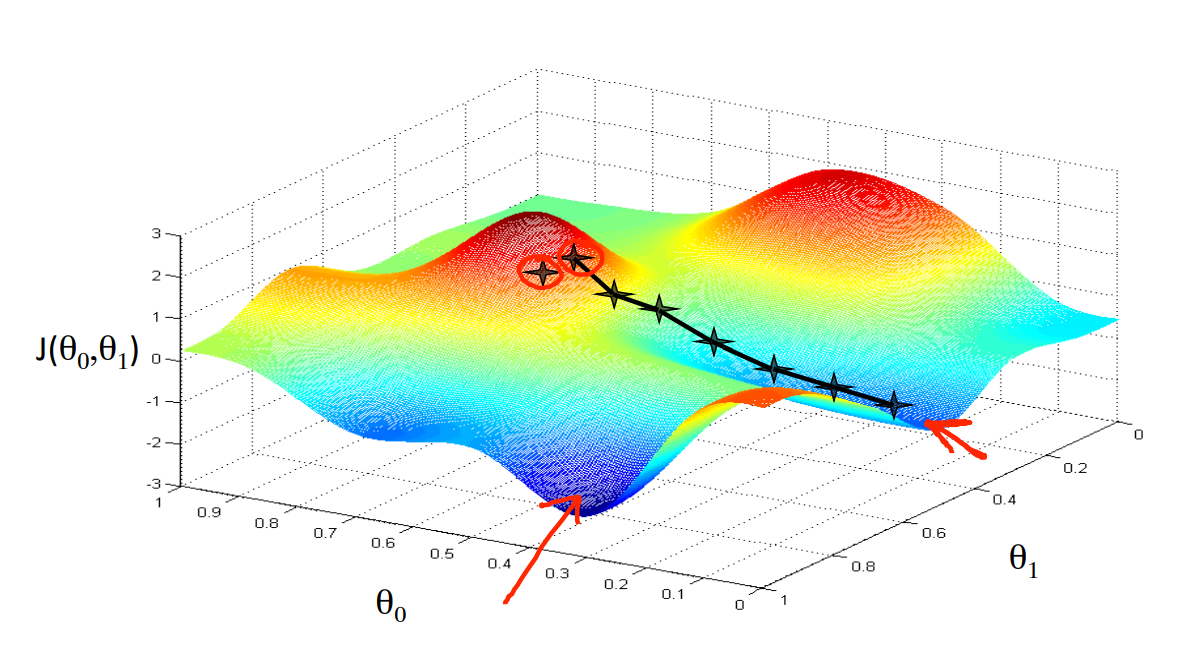


If we set the value to be **too large**, then when we get close to the point of convergence, we might **overshoot** it. Worse still, for **extremely large values**, we might even **diverge** instead of converging.

Thus, it would be best if we could somehow have a **large value** of towards the **start**, and gradually **decrease** it the closer we get to the point of convergence. We can actually do this by setting , where is the number of **iterations**.

Normally though, we set the value of to a **constant** . This is because as we get close to the point of convergence, the **gradient** becomes **smaller**, which in turn makes the changes in automatically smaller without having to modify .

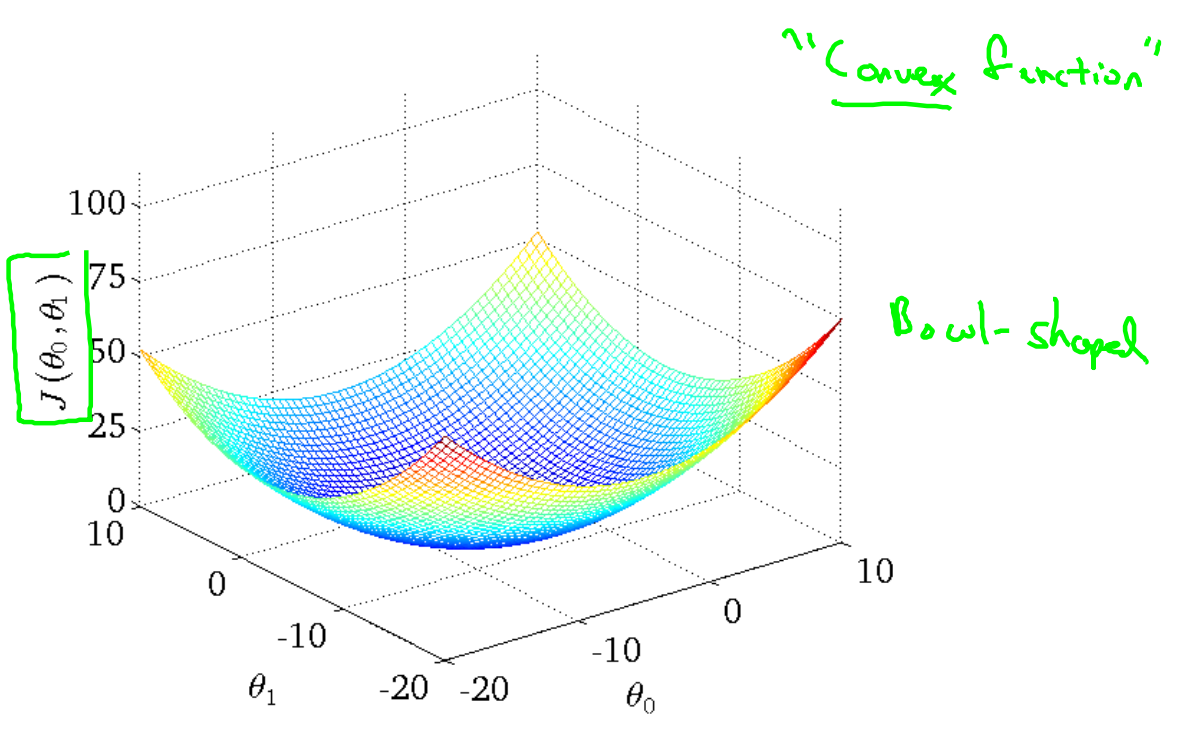
## Generic Functions



For a random function , we can have **multiple local minima**. Depending on our starting point , ), we could end up at **different minima**. So how do we tell if the minima we are at is the **global minima**?

One solution could be to **repeatedly run** the algorithm for different starting values and then take the result which is **least**.

Luckily for us though, the **optimization problems** we will be working with are guaranteed to always be a **convex shape**, as shown below. We will look into the proof for this soon.



## Number of Iterations

What if we end up repeatedly changing by extremely small values and we never fully converge? How long do we keep going?

One way to deal with this could be to set a **maximum iteration count**, say , but even this may be too large.

Another way could be to specify a certain **error threshold**, say . If the change in the value of the cost function, , is less than this value, then we stop. This is the level of imperfection in our algorithm that we are willing to accept.

## Derivatives

For ,

For ,

If is **multi-dimensional**, meaning ,

and so on.

Thus,

given that .

## Batch vs Stochastic vs Mini-Batch Gradient Descent

The details we have covered above fall specifically into the category of **batch gradient descent**. Under this method, we consider all of the data at each step (which is the part of us taking the sum of the values). This method is inconvenient when we have a very large amount of data.

Another option is **stochastic gradient descent**, in which we consider just one of the training examples at each step. By doing this, we are making things more convenient for us, but at the same time, the results will be **less accurate** and noisy. This is because we are optimizing the algorithm with respect to just one training example. We will still reach the solution eventually, but it will be **slower**, due to the noise. However, for each step, this process is faster.

We also have **mini-batch gradient descent**, in which we take a subset of the samples, . This is the mid-point between batch gradient descent and stochastic gradient descent. It is the best method, even though we cannot prove it theoretically.