GRADIENT DESCENT

Umarani Jayaraman

Gradient Descent - Introduction

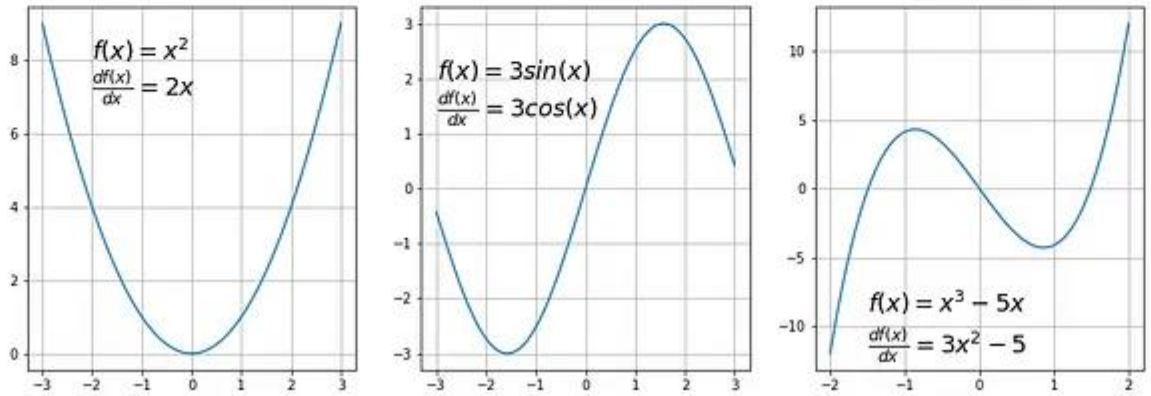
- **Gradient descent** (GD) is an iterative first-order optimization algorithm, used to find a local minimum/maximum of a given function.
- This method is commonly used in *machine learning* (ML) and *deep learning* (DL) to minimize a cost/loss function (e.g. in a linear regression).
- This method was proposed long before the era of modern computers by Augustin-Louis Cauchy in 1847.

Function requirements

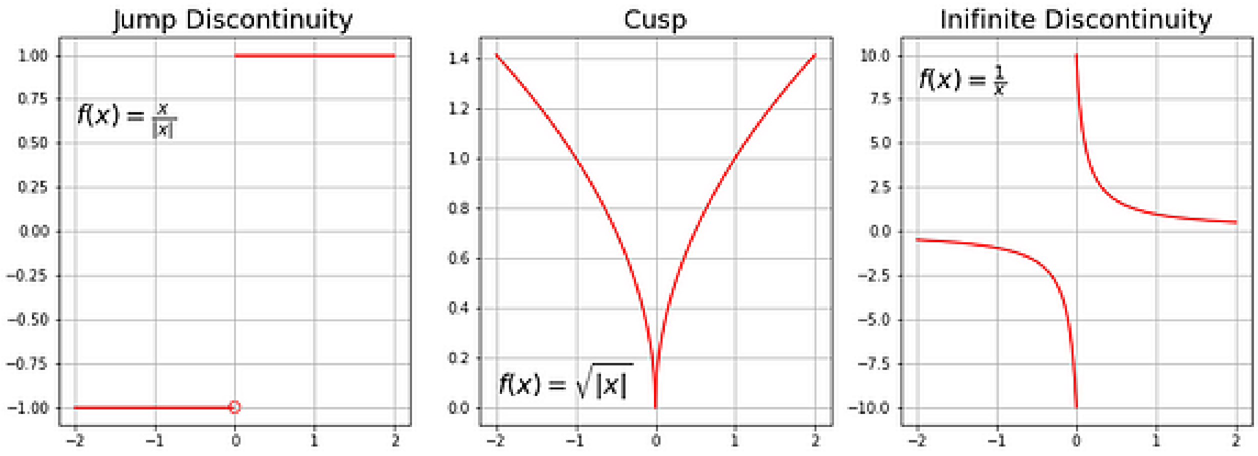
- Gradient descent algorithm does not work for all functions. There are two specific requirements. A function has to be:
- **differentiable**
- **convex**

First requirement - **differentiable**

- 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. First, let's see some examples of functions meeting this criterion:



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

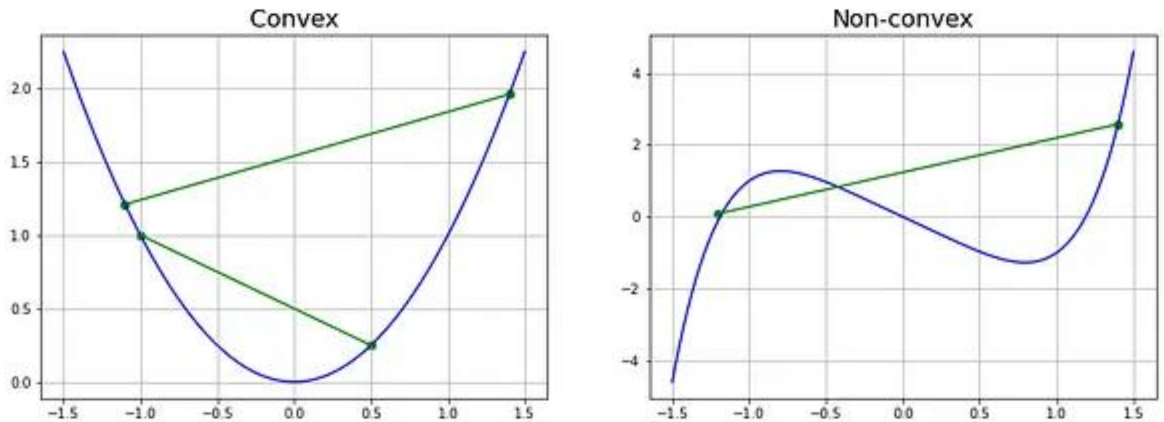


Next requirement - **function has to be convex**

- Next requirement — **function has to be convex.**
- For a univariate function, this means that the line segment connecting two function's points lays on or above its curve (it does not cross it).
- If it crosses it has a local minimum which is not a global one.
- Mathematically, for two points x_1, x_2 laying on the function's curve this condition is expressed as:

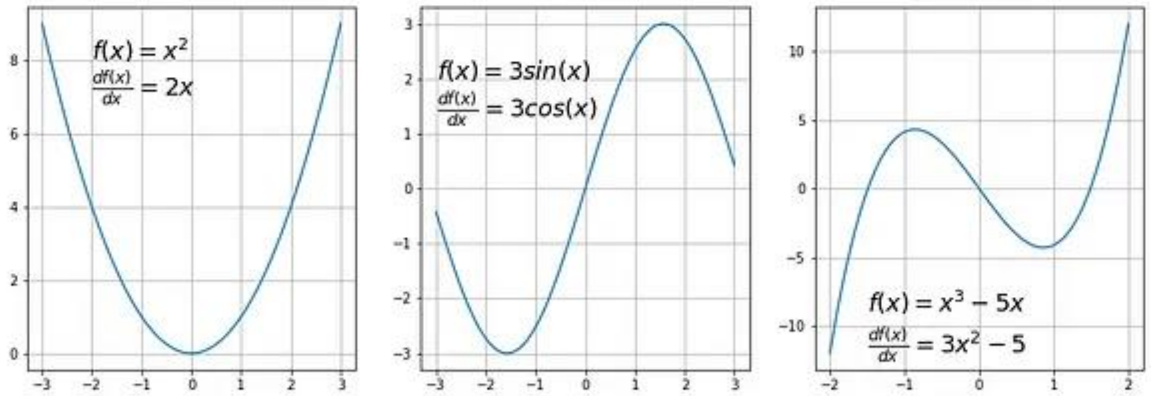
$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$$

- where λ denotes a point's location on a section line and its value has to be between 0 (left point) and 1 (right point),
- e.g. λ=0.5 means a location in the middle.
- Below there are two functions with exemplary section lines.



Caution: First requirement – differentiable, what about second requirement?

- 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
- Second and third function is not convex



- Another way to check mathematically if a univariate function is convex is to calculate the second derivative and check if its value is **always bigger than 0.**

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

Let's do a simple example

Let's investigate a simple quadratic function given by:

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

Its first and second derivative are:

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

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

saddle points

- It is also possible to use **quasi-convex functions** with a gradient descent algorithm.
- However, often they have so-called **saddle points** (called also *minimax* points) where the algorithm can get stuck
- An example of a quasi-convex function is:

- First order derivative

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

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

Let's stop here for a moment. We see that the first derivative equal zero at x=0 and x=1.5. This places are candidates for function's extrema (minimum or maximum)— the slope is zero there. But first we have to check the second derivative first.

Second order derivative

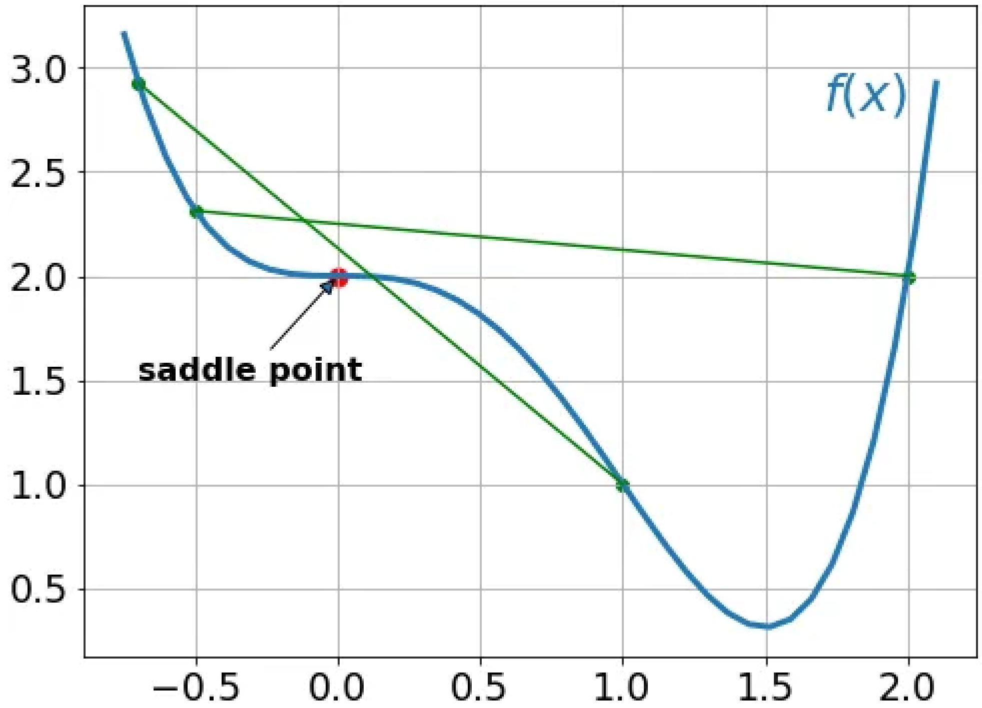
$$\frac{d^2 f(x)}{dx^2} = 12x^2 + 12x = 12x(x-1)$$

The 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 and point x=1.5 is a global minimum.

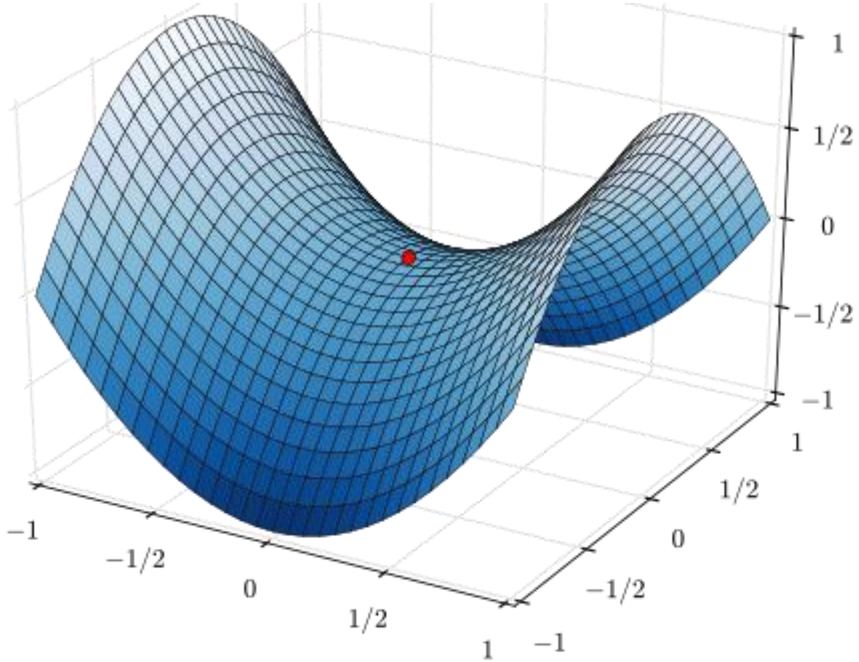
Let's look at the graph of this function. As calculated before a saddle point is at $x=0$ and minimum at $x=1.5$.



Semi-convex function with a saddle point; Image by author

- Example of a saddle point in a bivariate function is show below.

$z = x^2 - y^2$



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) (i.e) **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 Descent Procedure

- In summary, Gradient Descent method’s steps are:
- 1. choose a starting point (initialization)
- 2. calculate gradient at this point
- 3. make a scaled step in the opposite direction to the gradient (objective: minimize)
- 4. repeat points 2 and 3 until one of the criteria is met:
  - maximum number of iterations reached
  - step size is smaller than the tolerance (due to scaling or a small gradient).

Gradient Descent: sample code

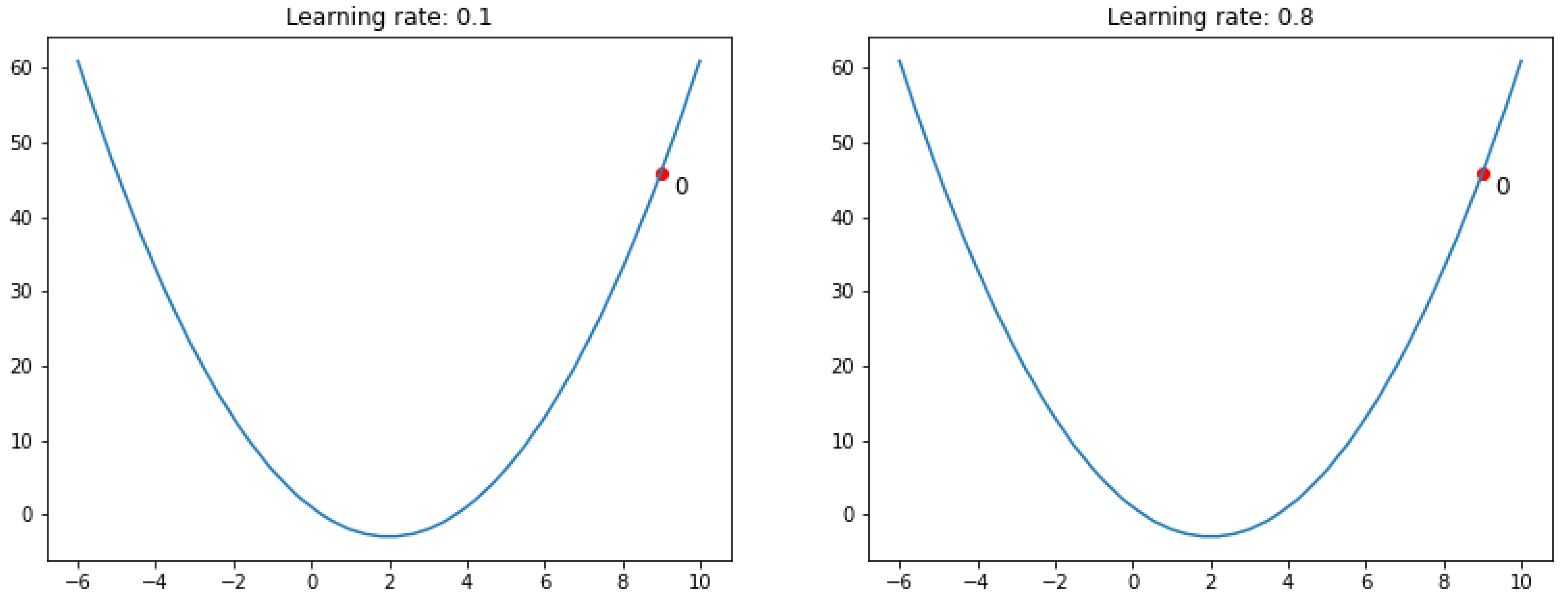
```python
import numpy as np
from typing import Callable


def gradient_descent(start: float, gradient: Callable[[float], float],
                     learn_rate: float, max_iter: int, tol: float = 0.01):
    x = start
    steps = [start]  # history tracking

    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
```

VanillaGD.py hosted with ❤ by GitHub   view raw

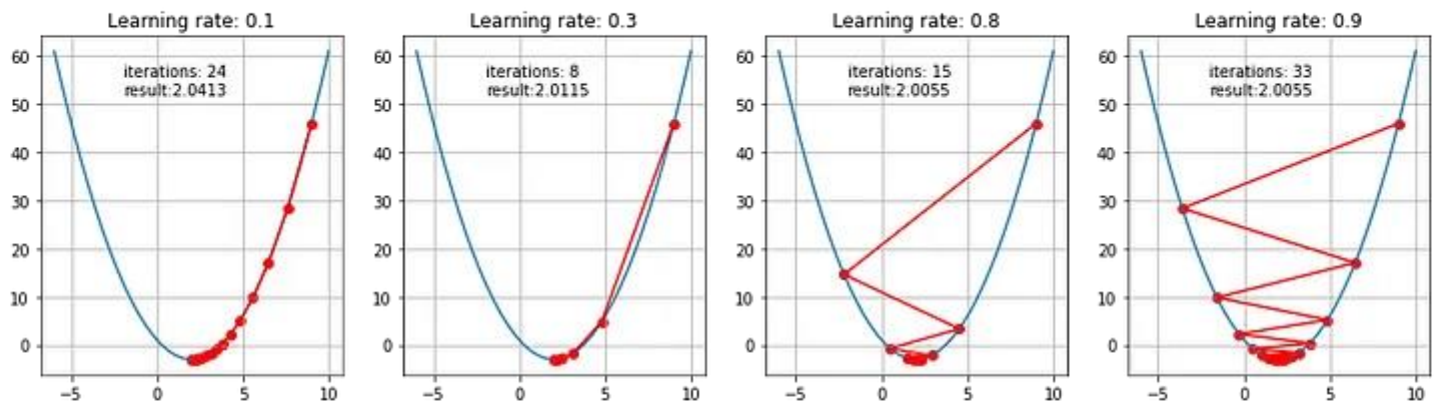
- This function takes 5 parameters:
- 1. **starting point** [float] - in our case, we define it manually but in practice, it is often a random initialisation
- 2. **gradient function** [object] - function calculating gradient which has to be specified before-hand and passed to the GD function
- 3. **learning rate** [float] - scaling factor for step sizes
- 4. **maximum number of iterations** [int]
- 5. **tolerance** [float] to conditionally stop the algorithm (in this case a default value is 0.01)

Effect of different learning rate

- The animation below shows steps taken by the GD algorithm for learning rates of 0.1 and 0.8.



Results of various learning rate



Gradient - summary

- The gradient is a fundamental concept in calculus and optimization technique
- The gradient of a function, denoted by ∇ (nabla), is a vector that points in the direction of the steepest increase of the function at a given point.
- Mathematically, for a function $f(x1,x2,\ldots,xn)$, the gradient is given by:
- $\nabla f=(\partial f/\partial x1,\ \partial f/\ \partial x2\ ,\ldots,\ \partial f/\ \partial xn\ )$
- Each component of the gradient represents the partial derivative of the function with respect to one of its input variables.

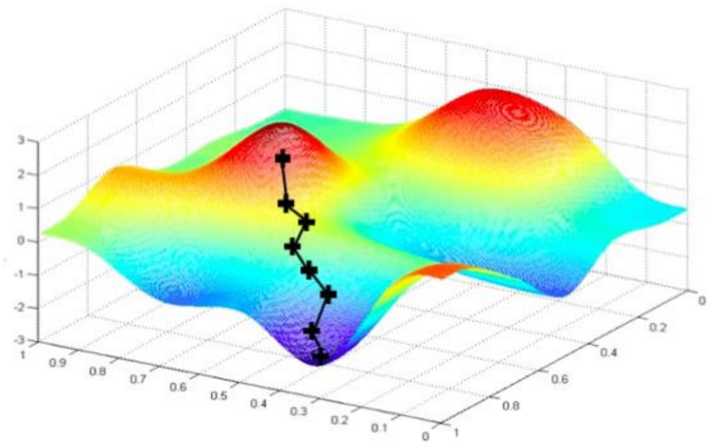
Significance in Optimization:

- In the context of optimization problems, the goal is often to find the minimum or maximum of a function.
- The gradient provides crucial information about the **direction and rate of change** of the function.
- The **negative gradient** points in the direction of the steepest decrease of the function.
- Therefore, moving in the opposite direction of the gradient helps in descending towards the **minimum of the function.**

Gradient Descent Algorithm

Algorithm

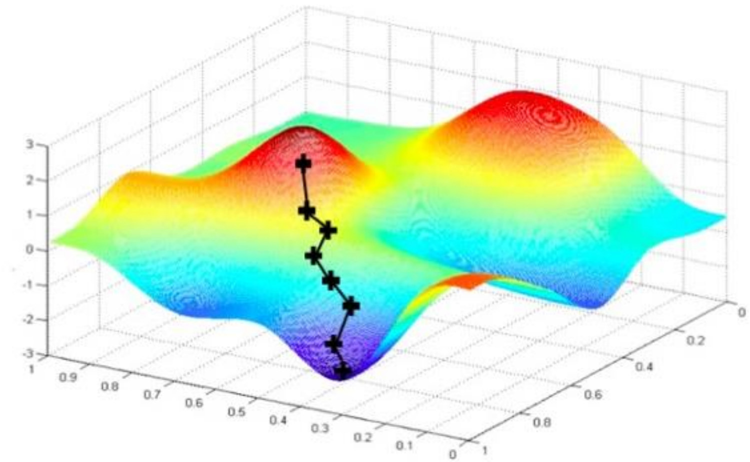
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



Batch Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\boxed{\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

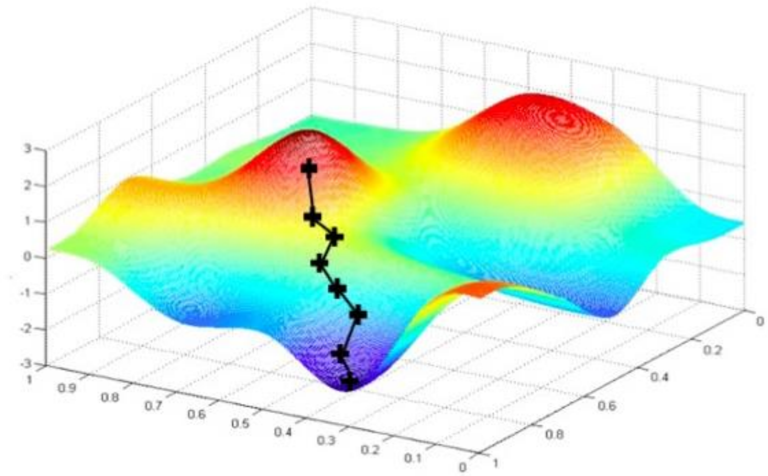


It can be computationally intensive to compute

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\boxed{\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

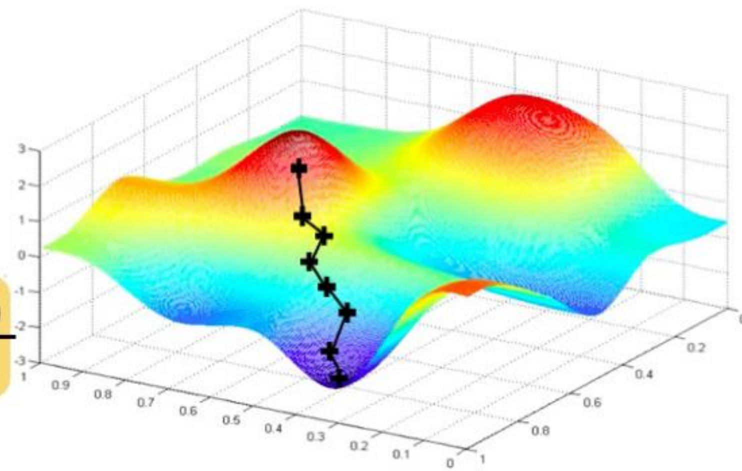


Easy to compute but very noisy

Mini-batch Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

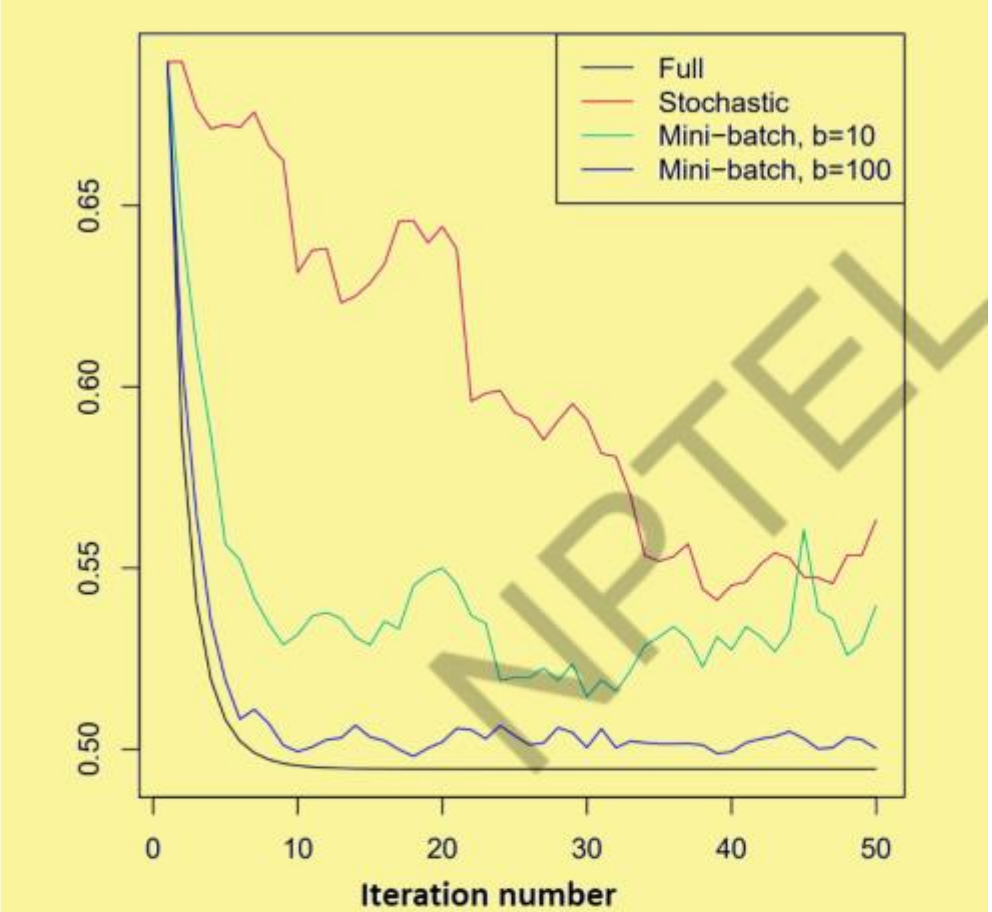


Fast to compute and a much better estimate of the true gradient

Mini-batches while training

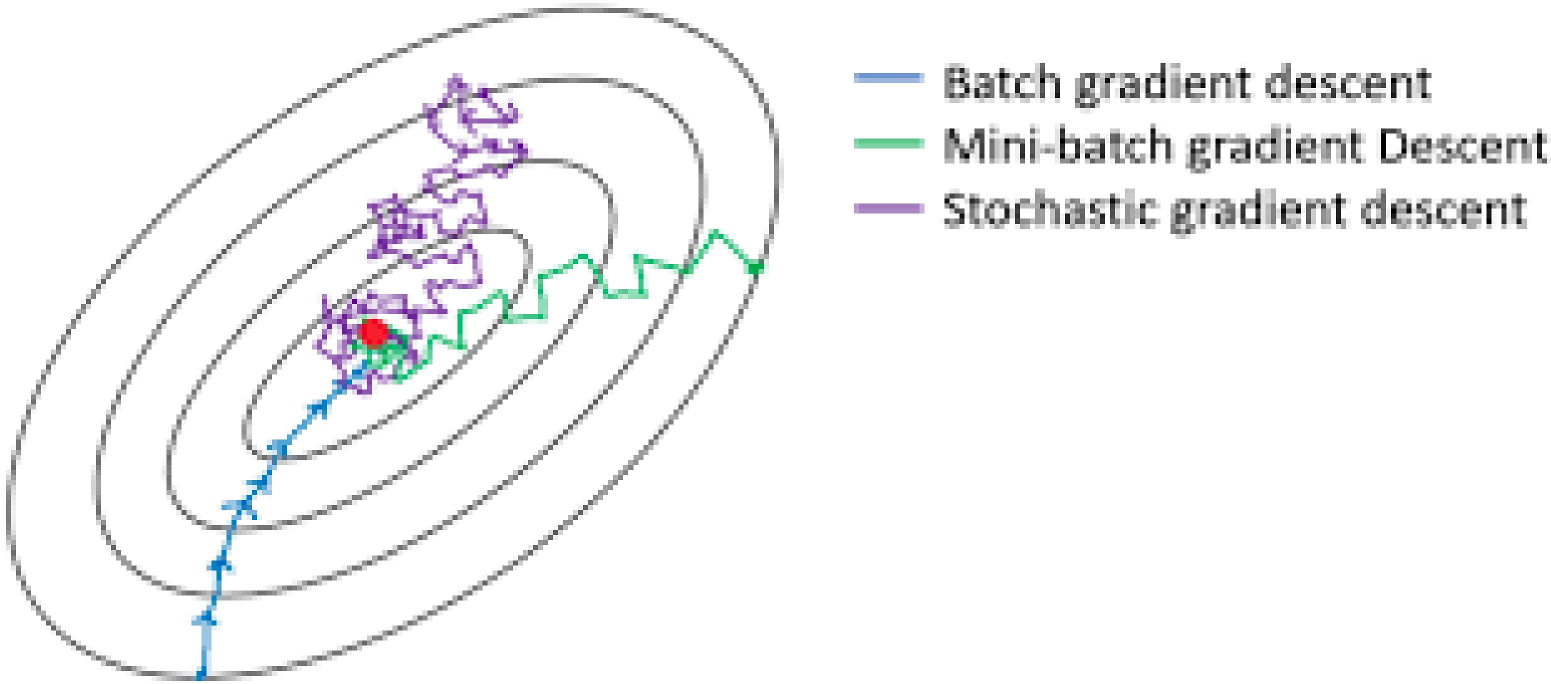
- More accurate estimation of gradient
- Smoother convergence
- Allows for larger leaning rates
- Mini-batches lead to fast training

Error minimization with iterations



Gradient Descent- Variants

- Batch
- Stochastic
- Mini-batch



Thank you