Optimization in Machine Learning

MET CS Generative AI

Reza Rawassizadeh

Outline

- **Mathematic Concepts**
- What is Optimization?
- Gradient Descent
- Newton Method

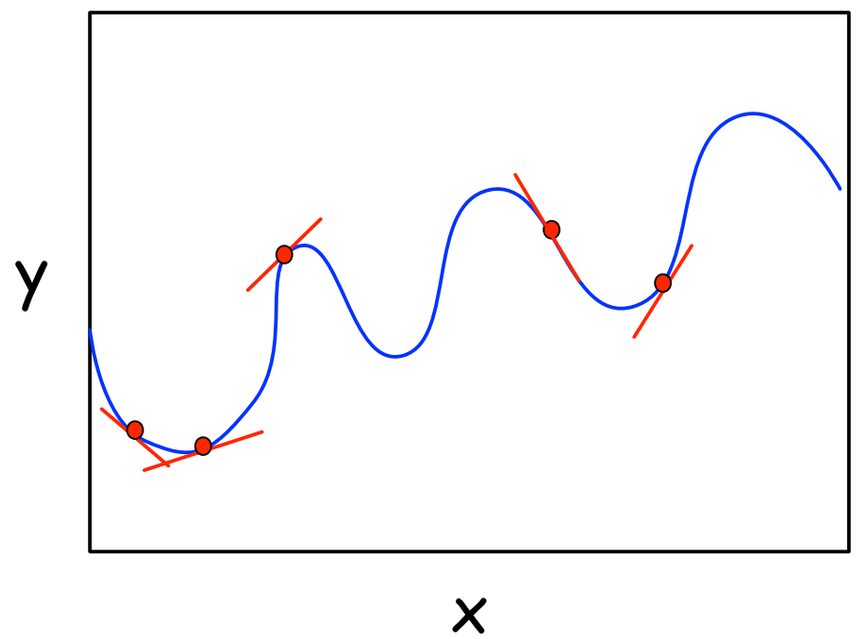
Derivative

- A function is something that gets an input variable x, does something with it, and produces the output, i.e. $f(x)$ or y.

- The derivative is a variable that measures the sensitivity of changes in the output variable with respect to changes in the input variable.

- The derivative can be written as a ratio of output changes over the ratio of input changes:

$$Derivative = \frac{Output\ Changes}{Input\ Changes}$$

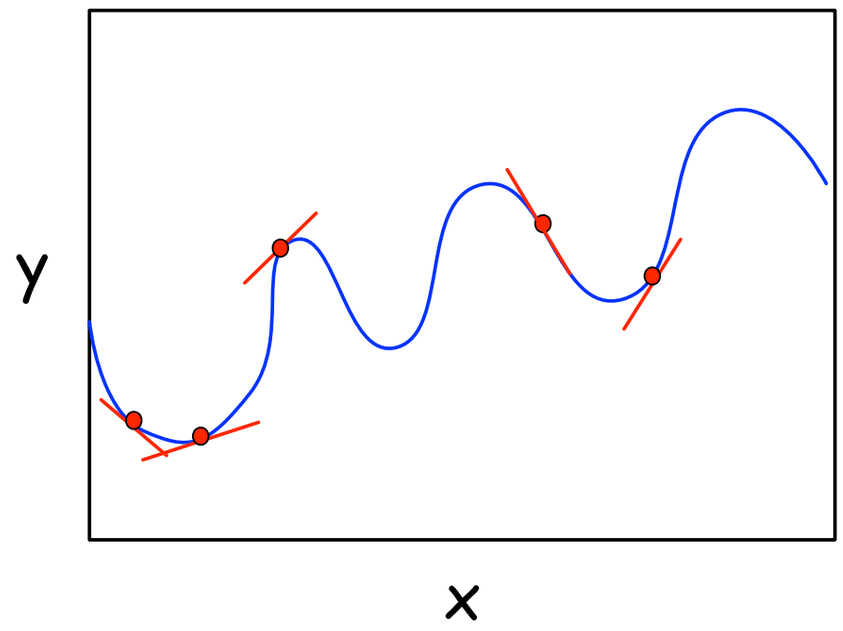


Derivative

- A line that “just touches” the curve at each arbitrary point is called a *tangent line*. The particular point that connects the tangent line to the curve is called the *point of tangency.*

- *The derivative is responsible for specifying the slope for the given tangent line of the function.*

- $$slope = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

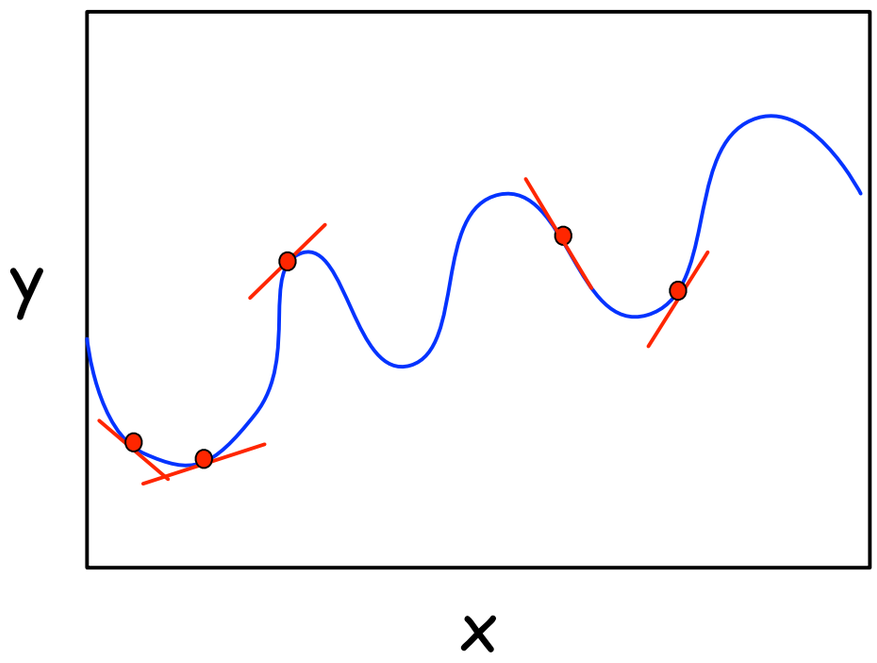


Derivative

- If the function is a single straight line, the derivative is a constant variable. If the function is a horizontal line parallel to X axis, it means there is no slope and thus its derivative will be zero. If the function is changing at different paces, such as in the Figure, the derivative will be a function itself.

- The process of finding the derivative is called **differentiation**.

- A constant derivative means that the function is growing constantly and a derivative of a straight line is a constant value.



Derivative

- A derivative of a function $f(x)$ is written as $f'(x)$, which is the notation introduced by Lagrange.

- Another version of writing derivative that is widely in use is called Leibniz notation, and it is written as follows:

$$f'(x) = \frac{d\,f(x)}{dx}$$

Derivative

$$slope = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

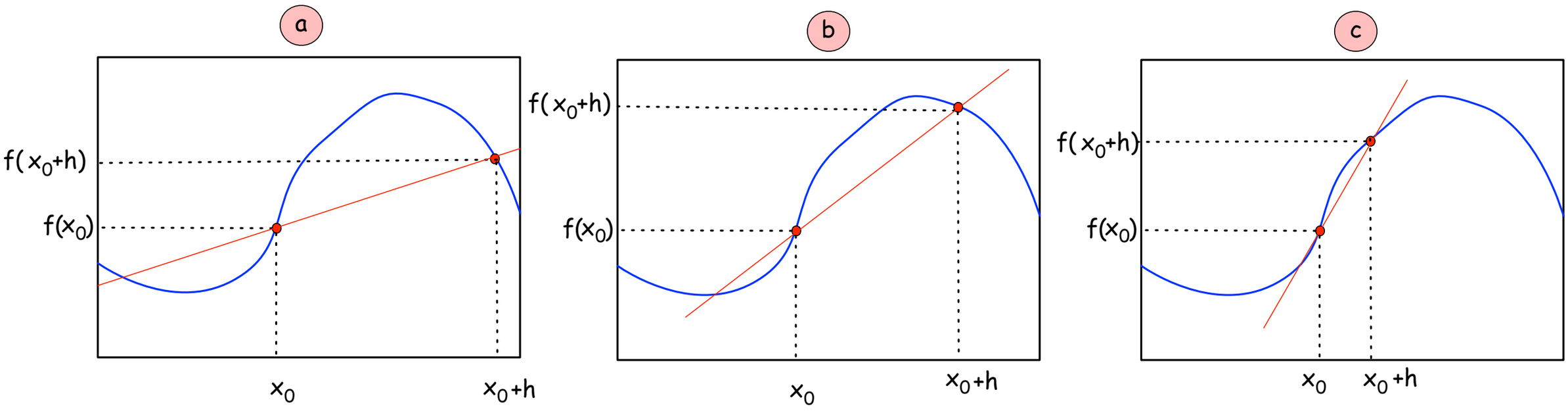
- To calculate the slope, we should select two points from the curve and by using them we can calculate the slope of the line connecting them.

- Let's say the first data point has coordinates $(x_0, f(x_0))$, the second data point is in *h* distance to the first data point, thus it has a coordinate of $(x_0 + h, f(x_0 + h))$. We can rewrite the slope formula as follows:

$$slope = \frac{f(x_0 + h) - f(x_0)}{(x_0 + h) - x_0} = \frac{f(x_0 + h) - f(x_0)}{h}$$

Derivative

the smaller h leads to having a more accurate slope of that particular curve, but why?



We can conclude from this Figure, that as the *h* distance is getting smaller the slope line to the tangent line is getting more accurate. Therefore, we can write the derivation function of x is the **limit** as h **is approaching zero**:

$$f'(x) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Derivative Rules

- There are some basic rules for derivative that we need to know, assuming n is a constant number, including:

$$f(x) = x^n \rightarrow f'(x) = nx^{(n-1)},$$
$$f(x) = nx \rightarrow f'(x) = n,$$
$$f(x) = ln(x) \rightarrow f'(x) = 1/x,$$
$$f(x) = x \rightarrow f'(x) = 1,$$
$$f(x) = n \rightarrow f'(x) = 0,$$
$$\dots$$

- In other words, we can say the function x^n has the slope of nx, the function nx has the slope of n, etc.
- What we have explained is a first-order derivative. It is also possible to make a derivative from a derivative, i.e., **second-order derivative,** and it is written as $f''(x)$, e.g. $f(x) = x^2 \rightarrow f'(x) = 2x \rightarrow f''(x) = 2$

Derivative Rules

- To calculate the derivative we should use some rules specified for the derivative including the “chain rule”, “quotient rule”, and “product rule" .

Reciprocal rule: $(\frac{1}{x})' = \frac{-1}{x^2}$

Product rule: $(f(x) \,.\, g(x))' = f(x) \,.\, g(x)' + f(x)' \,.\, g(x)$

Quotient rule: $(\frac{f(x)}{g(x)})' = \frac{g(x)f(x)' - g(x)'f(x)}{g(x)^2}$

Difference rule: $(f(x) - g(x))' = f'(x) - g'(x)$

Chain rule (outside-inside rule): $[f(g(x))]' = f'(g(x))g'(x)$

f(g(h(x))) = df/dg x dg/dh x dh/dx

- The Chain rule is useful to find a derivative of a function when we have nested functions (a function inside another function).

Partial Derivative

- Sometimes in an equation, we need to deal with more than one variable, e.g., $f(x, y) = x^2 + 4y$.

- To write a derivative of such a function once we need to write derivative for x and consider the other variable (y) as constant, then we need to write derivative for y and consider the variable x as constant. This is called a partial derivative.

- To write a partial derivative for variable x and variable y we write the following:

- $\frac{\partial}{\partial x}[f(x, y)] = 2x + 0$ and $\frac{\partial}{\partial y}[f'(y)] = 0 + 4$

- Instead of d that we use to refer to the derivative here, we use ∂ sign for partial derivative.

Gradient

- We can see that the derivative is defined in two-dimensional space (it can be shown as $\mathbb{R}^2$) or a *univariate* function.

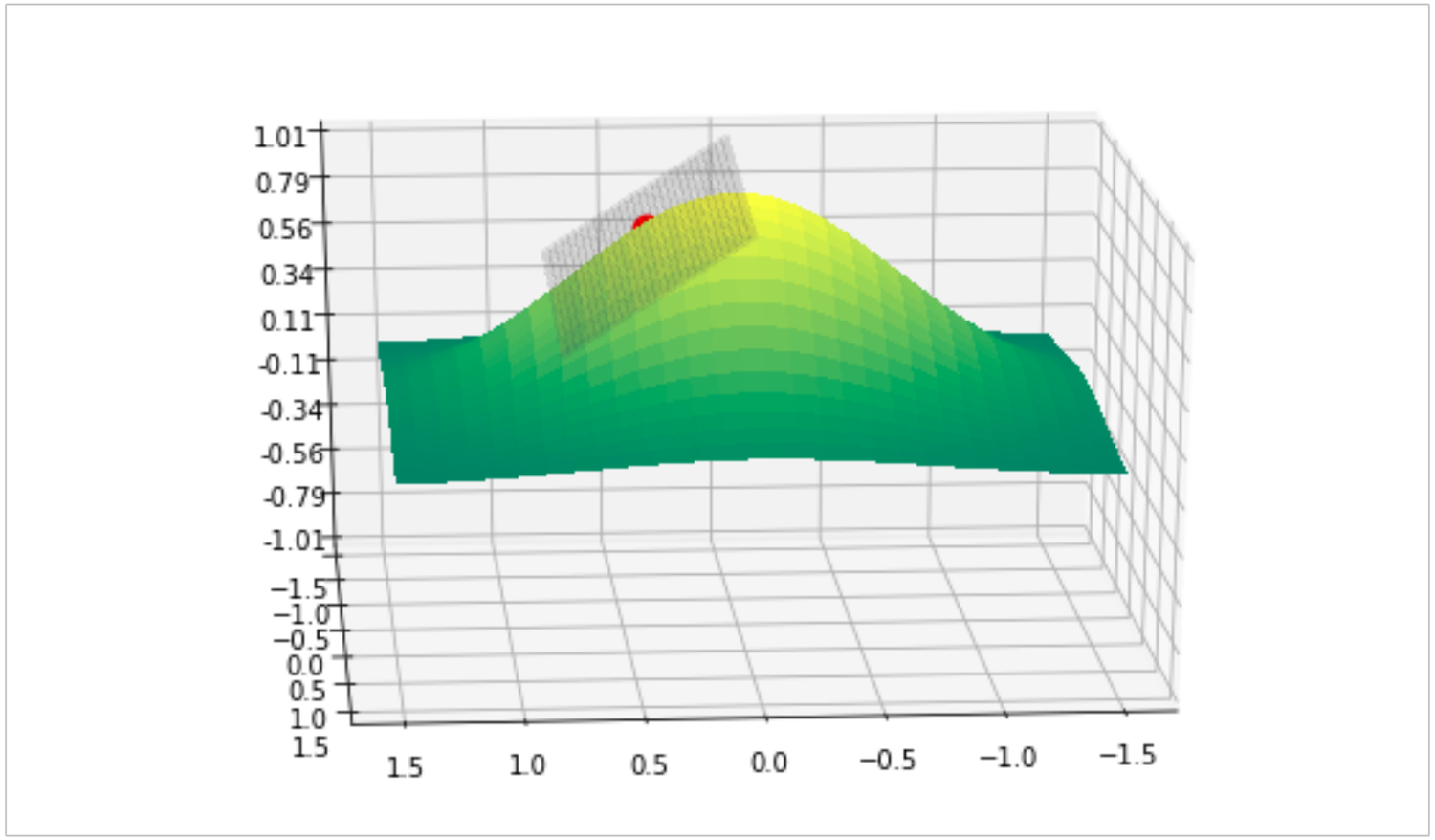
- How about having a *multivariate* function, e.g., $\mathbb{R}^3$?

- The generalization of derivatives for a multivariate function is called **Gradient**.

- In the context of machine learning, we can say derivative is used for vectors, but Gradient is the derivative for matrices and tensors.

Gradient

- Here we deal with Tangent-plane (or hyperplane) instead of Tangent Line



Gradient

A Gradient of a function $f(X)$, assuming $X = \{x_1, x_2, \ldots x_n\}$can be written as:

$$\nabla f(X) = \left[\frac{\partial f(X)}{\partial x_1}, \frac{\partial f(X)}{\partial x_2}, \ldots, \frac{\partial f(X)}{\partial x_n}\right].$$

For example for our $f(x_1, x_2) = ax_1^2 + bx_2 + c$ function, the derivative of this function with respect to x_1 is written as $\frac{\partial f(x_1, x_2)}{\partial x_1}$. Its gradient will be written as follows :

$$\frac{\partial f(x_1, x_2)}{\partial x_1} = 2ax_1 + 0 + 0\ ,\ \frac{\partial f(x_1, x_2)}{\partial x_2} = 0 + b + 0.$$

$$\nabla f(x_1, x_2) = \left[\frac{\partial f(x_1, x_2)}{\partial x_1}, \frac{\partial f(x_1, x_2)}{\partial x_2}\right] = \left[2ax_1, b\right]$$

Jacobian

- When we stack the gradient of two functions, as a vector of variables, into a matrix, it is called a **Jacobian matrix**.

- For example, assume we have $\nabla f(x, y)$ and $\nabla g(x, y)$. The Jacobian matrix is written as follows:

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x, y)}{\partial x} & \frac{\partial f(x, y)}{\partial y} \\ \frac{\partial g(x, y)}{\partial x} & \frac{\partial g(x, y)}{\partial y} \end{bmatrix}$$

- For example, if we have $f(x, y) = 4x^2y$ and $g(x, y) = 2x + y^3$, then their Jacobian matrix is written as follows:

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} 8xy & 4x^2 \\ 2 & 3y^2 \end{bmatrix}$$

Hessian

- Gradient is the first-order derivative of a multivariate function and can be written as a vector of variables.

- The second order of gradient (a derivative of derivative) is called Hessian (read it as heshian) and it is written as a matrix.

$$\nabla^2 f \quad or \quad H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Hessian includes all possible second-order partial derivatives, or we can say it includes every possible pairing of the n variables.

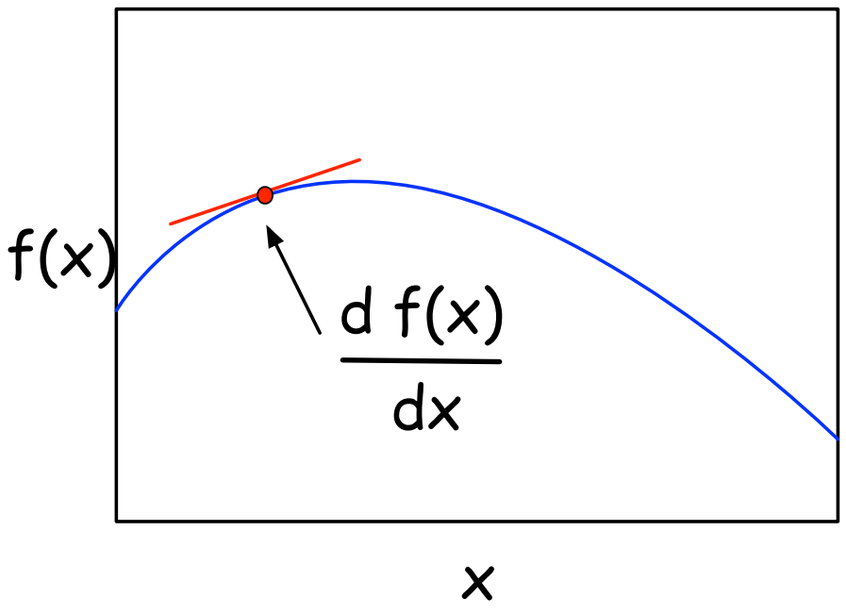
Integral

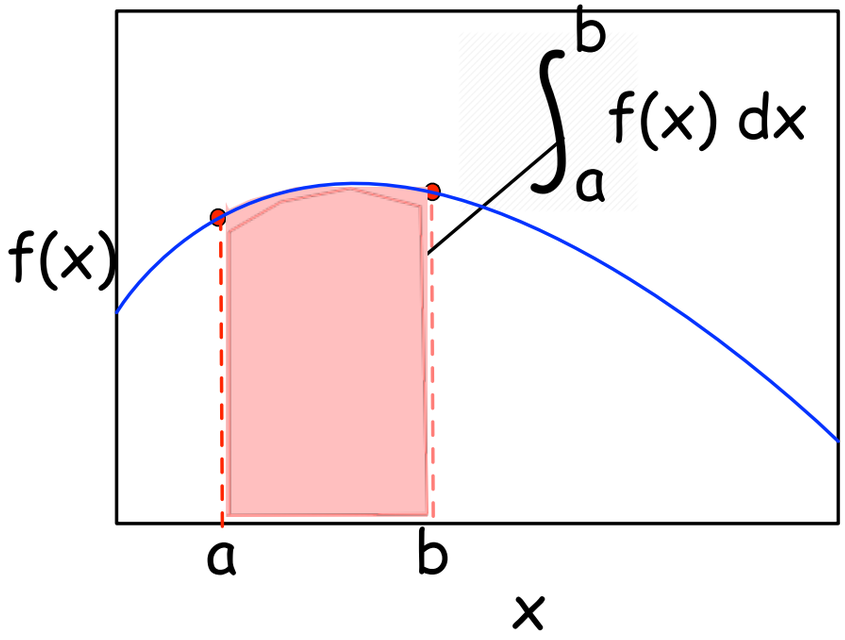
- Many physical and non-physical entities, such as acceleration, force, and velocity, are defined as a rate of change. *The derivation is used to quantify the rate of change* in mathematics.

- Integral has many physical and non-physical entities, such as specifying a region's weight with respect to the overall weight of a function.

- The integral of a function is the *area under the curve,* and we use it to *specify the weight of the function in a certain range*.

- Assume we are filling a tank with water from a tap. The pace of water coming out of the tap is derivative and the amount of water inside the tank is integral. For example, if the tap is open and gives a constant amount of water every second, e.g., 1 *ml*, the water inside the tank will increase with the constant pace, i.e. $f(x) = x$. If the tap is open and each second its pace of pumping water increases $2x$ (which is a derivative of x^2) times, then the amount of water inside the tank will increase with $f(x) = x^2$.





Taylor Series

- A **series** in mathematics is referred to as a *group of several terms that are all about the same thing*. For example, the following is a series:$\frac{1}{1}+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots.$

- Taylor series or Taylor expansion is a function that has an infinite sum of terms. These terms, however, were calculated based on the repeated derivate at a single point. In mathematics, any function can be represented with a Taylor series. For example, the Taylor series of functions e^x and $sin(x)$ are written as follows:

$$e^x = 1 + x + \frac{x^2}{2!} + +\frac{x^3}{3!} + \ldots$$

$$sin(x) = x - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

- Given *a* as a variable of the function *f(x)*, a Taylor series for *f(x)* will be calculated by using the following equation:

$$f(x) \approx f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots \approx \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Taylor Series

- Taylor series is used in the approximation function.

- Approximation functions try to approximate a target function that is either hard to calculate or unknown, by a simpler function that is easy to calculate or known function.

- Linear approximation refers to the first-order Taylor approximation and quadratic approximation refers to the second-order Taylor approximation.

Outline

- Mathematic Concepts
- **What is Optimization?**
- Gradient Descent
- Newton Method

Optimization

- Any repetitive process that is performed by humans is associated with optimization.

- For example, as a human grows she optimizes her eating attitude by using spoon, fork, or chopsticks, and while the human is an infant she experiments with her first eating performance by using her hand and face.

- It means we learn a process and the more we learn, the more we optimize the process.

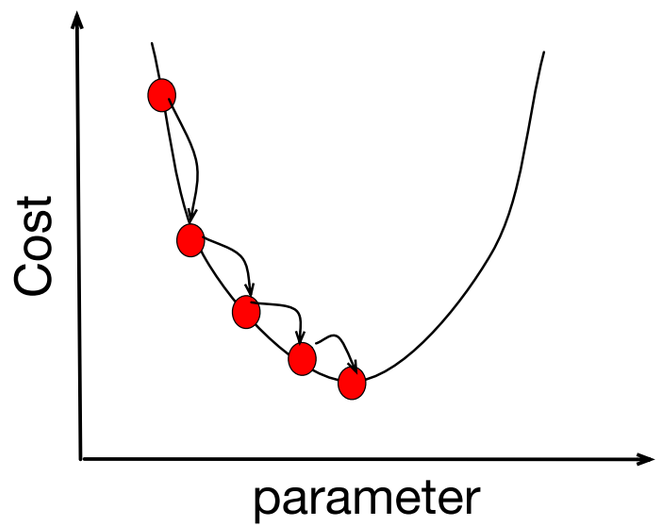
- Identifying parameters of an unknown mathematical equation, which is the objective of many machine learning algorithms, consume lots of computational resources.

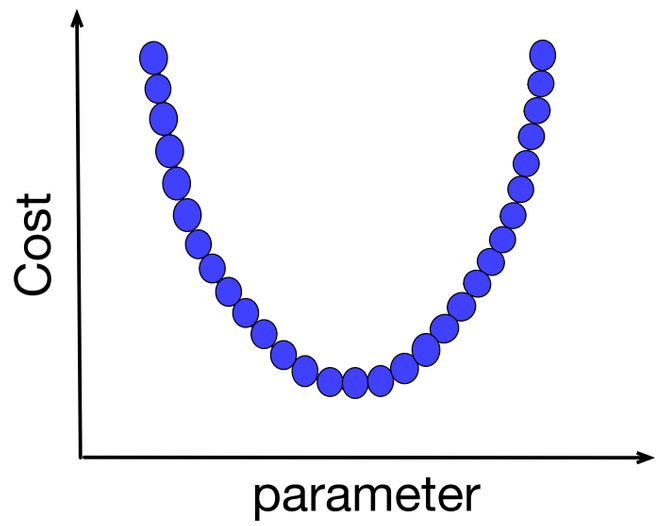
- Since resources are limited the optimization process helps the algorithm to search for an appropriate parameter in the environment with limited resources.
—> we use optimization to estimate model parameters while reducing the resource required to search for the best model parameters.

Optimization

- *Optimization is the process to change and tweak model parameters to minimize the cost function.*

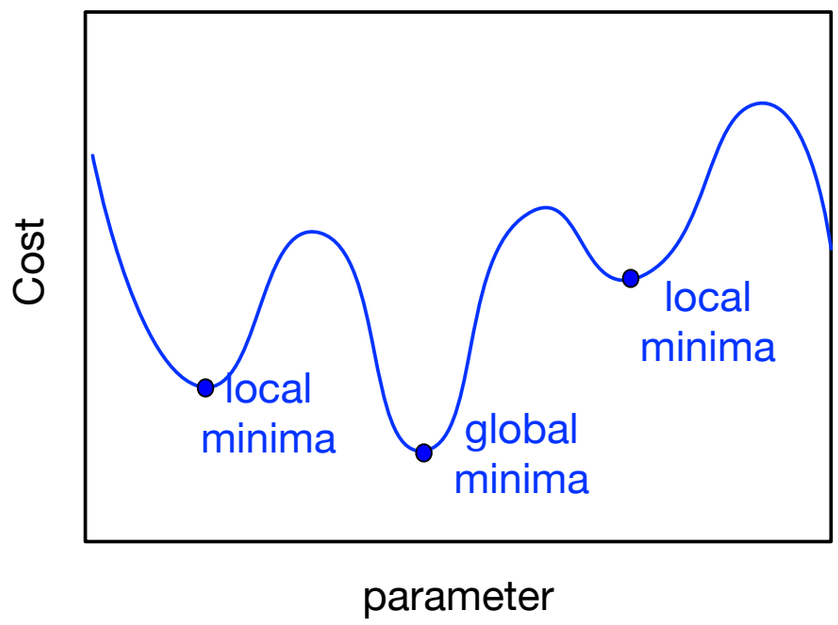
- Why do we need optimization and why not experiment with all possible parameters until we get the best result?





Optimization in Non-Convex Function

- The ultimate goal of optimization is to find the global minimum.
- In reality, if we deal with non-convex shape optimization, it is not always possible to find the optimal point (global minima), and thus, we seek to identify strong local minima.
- If we have a convex shape, the local minima could be the global minima as well, but still, it is not guaranteed that the optimization algorithm will find it.



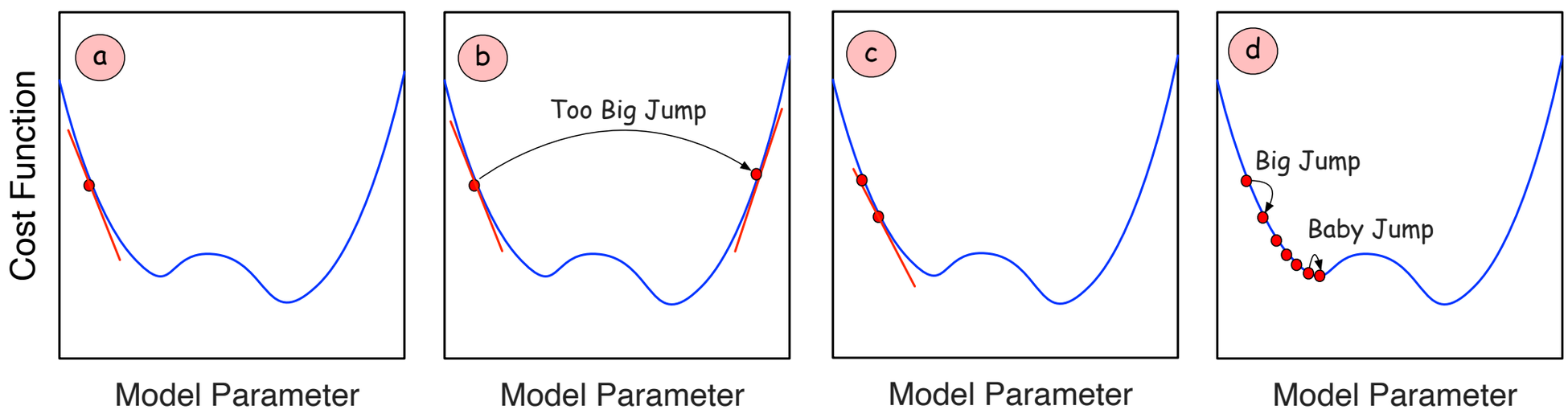
Local Minima Conditions

- Mathematically there are two conditions required to claim a datapoint as a local minima:
  - (i) its derivative is zero, i.e., $f'(x) = 0$
  - (ii) its second-order derivative is not negative $f''(x) \geq 0$.
- *The process of optimization is focused on reducing the cost or loss function by choosing the right values for the model parameters.*
- In other words, our goal in building a model is to find the best model to describe the dataset. Therefore, identifying the optimal value for model parameters reduces the error, and thus we can have the optimal model.

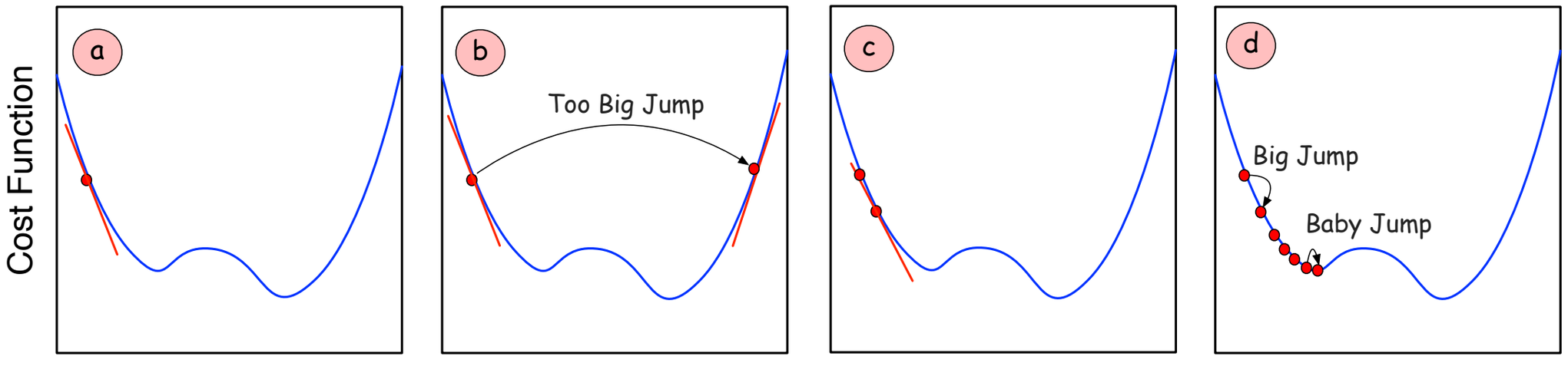
Gradient Descent

- Gradient descent is a first-order iterative optimization algorithm that focuses on *minimizing a function by moving its direction toward the steepest descent* (negative of gradient).

- Gradient descent stops either (i) a specific threshold (maximum epoch) reaches, (ii) the step size is getting very small (derivative is getting too small and close to zero), which means we reach the minima.



Gradient Descent

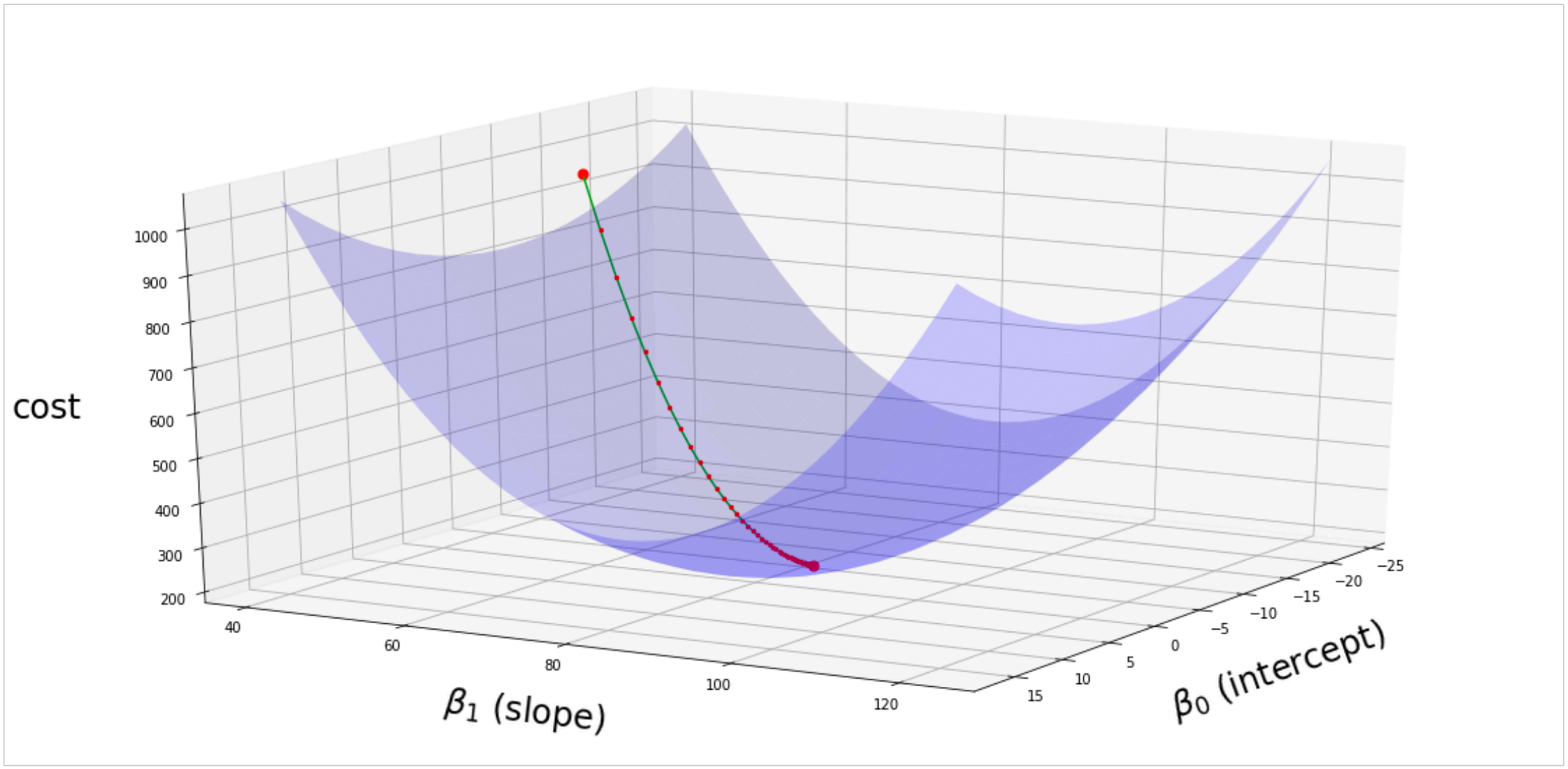


the sign of the derivative changes, thus it the algorithm realized that it was not a correct jump. It is referred to as Overshooting

the absolute value of the derivative is smaller than the previous point it is a good choice and the algorithm realized that it was a correct jump

Why do we not take the step at the beginning small enough to avoid jumps like Figure b?

Gradient Descent Example for Linear Regression



Gradient Descent Visualization (1/3)

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy import stats

from sklearn.datasets.samples_generator import make_regression
```

```
x, y = make_regression(n_samples = 100,
                       n_features=1,
                       n_informative=1,
                       noise=20,
                       random_state=2017)
x = x.flatten()
slope, intercept, _,_,_ = stats.linregress(x,y)
best_fit = np.vectorize(lambda x: x * slope + intercept)
plt.plot(x,y, 'o', alpha=0.5)
grid = np.arange(-3,3,0.1)
plt.plot(grid,best_fit(grid), '.')
```

Gradient Descent Visualization (2/3)

```
def gradient_descent(x, y, theta_init, step=0.001, maxsteps=0, precision=0.001, ):
    costs = []
    m = y.size # number of data points
    theta = theta_init
    history = [] # to store all thetas
    preds = []
    counter = 0
    oldcost = 0
    pred = np.dot(x, theta)
    error = pred - y
    currentcost = np.sum(error ** 2) / (2 * m)
    preds.append(pred)
    costs.append(currentcost)
    history.append(theta)
    counter+=1
    while abs(currentcost - oldcost) > precision:
        oldcost=currentcost
        gradient = x.T.dot(error)/m
        theta = theta - step * gradient  # update
        history.append(theta)
        pred = np.dot(x, theta)
        error = pred - y
        currentcost = np.sum(error ** 2) / (2 * m)
        costs.append(currentcost)
        if counter % 25 == 0: preds.append(pred)
        counter+=1
        if maxsteps:
            if counter == maxsteps:
                break
return history, costs, preds, counter
```

Gradient Descent Visualization (3/3)

```
xaug = np.c_[np.ones(x.shape[0]), x]
theta_i = [-15, 40] + np.random.rand(2)
history, cost, preds, iters = gradient_descent(xaug, y, theta_i, step=0.1)
theta = history[-1]
print("Gradient Descent: {:.2f}, {:.2f} {:d}".format(theta[0], theta[1], iters))
print("Least Squares: {:.2f}, {:.2f}".format(intercept, slope))
from mpl_toolkits.mplot3d import Axes3D
def error(X, Y, THETA):
    return np.sum((X.dot(THETA) - Y)**2)/(2*Y.size)
ms = np.linspace(theta[0] - 20 , theta[0] + 20, 20)
bs = np.linspace(theta[1] - 40 , theta[1] + 40, 40)
M, B = np.meshgrid(ms, bs)
zs = np.array([error(xaug, y, theta)
               for theta in zip(np.ravel(M), np.ravel(B))])
Z = zs.reshape(M.shape)
fig = plt.figure(figsize=(20, 10))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(M, B, Z, rstride=1, cstride=1, color='b', alpha=0.2)
ax.set_xlabel(r'$ \beta_0$ (intercept)', labelpad=30, fontsize=24)
ax.set_ylabel(r'$ \beta_1$ (slope)', labelpad=30, fontsize=24)
ax.set_zlabel('cost', labelpad=30, fontsize=24)
ax.view_init(elev=20., azim=30)
ax.plot([theta[0]], [theta[1]], [cost[-1]] , markerfacecolor='r', markeredgecolor='r', marker='o',
markersize=7)
ax.plot([history[0][0]], [history[0][1]], [cost[0]] , markerfacecolor='r', markeredgecolor='r',
marker='o', markersize=7)

ax.plot([t[0] for t in history], [t[1] for t in history], cost , markerfacecolor='r',
markeredgecolor='r', marker='.', markersize=5)
plt.savefig("Minimization_image.png")
```

Learning Rate

- Gradient descent **step sizes** were calculated by a parameter called **learning rate** times **slope** of that particular point for each parameter, i.e.:

Next point = current point - learning rate × slope (a derivative of that particular parameter).

- In particular, *the learning rate specifies the step in each iteration.* A reasonable learning rate is usually a constant and is given as a hyperparameter.

- Usually, the gradient descent algorithm starts with a small learning rate of α=0.001 and improves it as they experiment more with the dataset.

Gradient Descent Steps

(1) The objective is to select a cost function for the given parameter values. For example, we chose to calculate RMSE for the given β_0 and β_1.

(2) A random value for each parameter will be selected and the cost for these parameters will be calculated.

(3) The gradient (if the model has only one parameter, then its derivative, otherwise gradient), which gives us the slope of this particular cost value (from step 2) will be calculated.

(4) The algorithm calculates the next data point by using the following equation: $new\_point = current\_point - learning\_rate \times slope$. As the new data point will be specified, the algorithm uses this data point. The slope is equal to the cost function's gradient at the current point. Assuming the α is the learning rate, we can write the next point will be calculated as follows: $x_{t+1} = x_t - \alpha_t \,.\, \nabla f(x_t)$

(5) The optimization algorithm checks whether a given threshold for iteration has been reached or the slope size of this new data point is zero (it means we reach the minima), or both values are converged (β_0 and β_1) and not changing anymore. If none of these two conditions were true, then again, it goes to step 3 and continues to step 4 and then 5. Otherwise, it stops.

Gradient Descent Example

Each time one training set is analyzed by gradient descent we say one epoch passed.

As a formalized example, let's say the linear regression has the following equation: $y = \beta_0 x + \beta_1$ we intend to use the Sum of Squared Residuals (SSR) error as a cost function.

We do not know the values for βs and both parameters should be identified to minimize the, which is written as $l = \sum_{i=1}^{n} (\hat{y}_i - (\beta_0 + \beta_1 x_i))^2$.

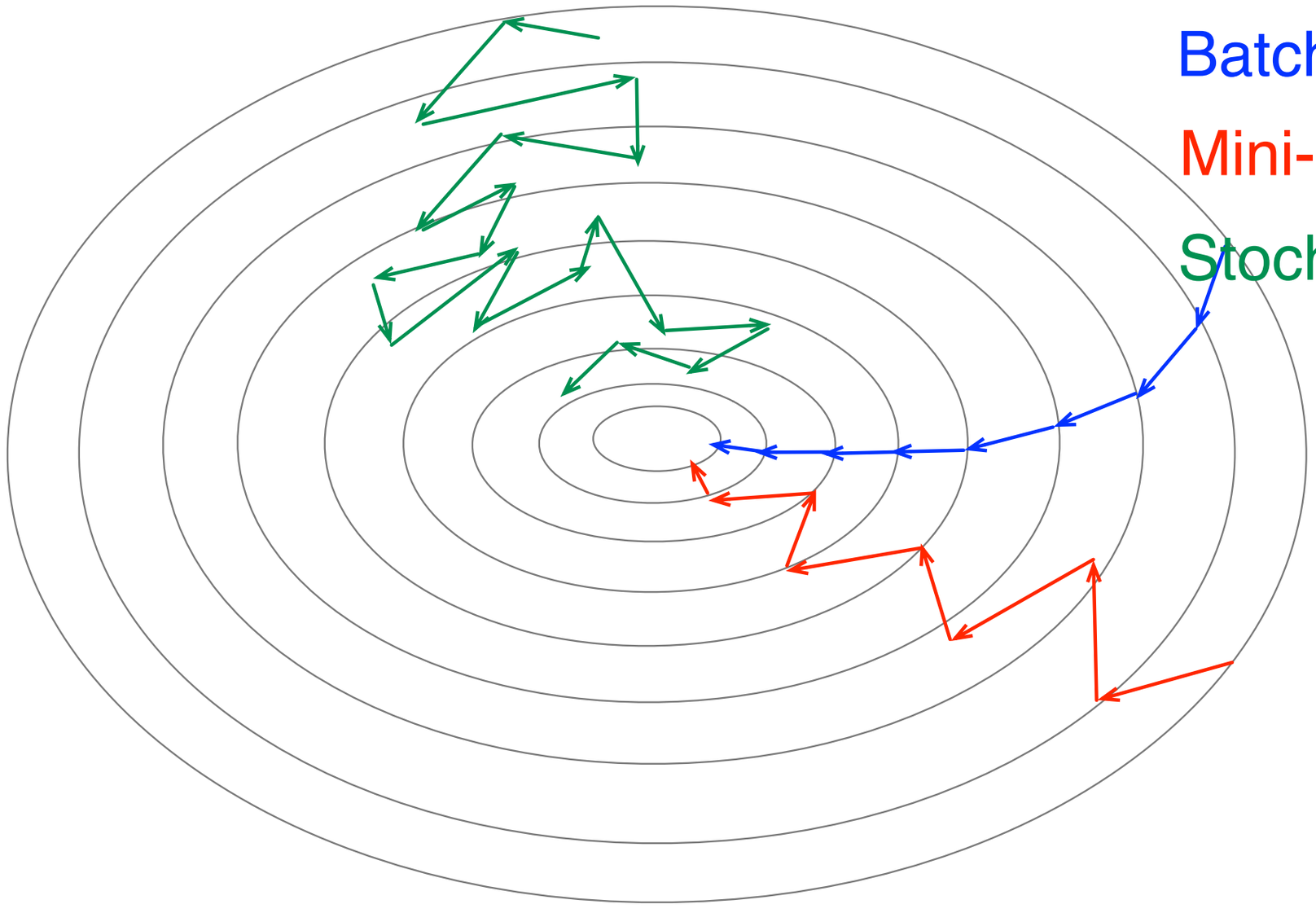
Gradient descent starts with calculating a partial derivative for every parameter based on the chain rule: $\frac{\partial l}{\partial \beta_0} = -2(\hat{y}_i - (\beta_0 + \beta_1 x_i))$ and $\frac{\partial l}{\partial \beta_1} = -2x_i(\hat{y}_i - (\beta_0 + \beta_1 x_i))$

Instead of SSR we can use RMSE (Quadratic cost function) or other cost functions as well, including cross entropy cost, which is used for classification, and its equation is written as follows:

$$l = -\sum_{j} (\hat{y}_j log(y_j) + (1 - \hat{y}_j) log(1 - y_j))$$

Types of Gradient Descent

- Batch Gradient Descent (BGD)
- Stochastic Gradient Descent (SGD)
- Mini Batch Gradient Descent (miniBGD)



Batch Gradient Descent

Mini-batch Gradient Descent

Stochastic Gradient Descent

Batch Gradient Descent (BGD)

- A training dataset can be divided into smaller segments referred to as **batches.** When all training datasets are collected in one single batch, and we use gradient descent for optimization, it is called Batch Gradient Descent.

- Batch Gradient Descent is the simplest form of gradient descent (but the most accurate one) because it uses the entire training set to compute the gradients at every step.

- This means, in each iteration, *all training datasets* will be used, and then the gradient will be calculated to decide the next step.

- This approach is useful when we have a convex function such as RSME in linear regression because, with enough iterations, the algorithm can easily reach the global minima. Nevertheless, it uses the entire training set; when the training set is large, the batch gradient descent is not resource-efficient.

Stochastic Gradient Descent (SGD):

- SGD picks one random sample from the training set, instead of using the entire dataset, and calculates its gradient in the first step. Then, again for the next step, the algorithm takes another single random sample from the training set and again calculates their gradient. These iterations will continue until it reaches zero gradients or reaches the maximum threshold for iteration.

- Using random samples from the training set (not the entire training set) makes the algorithm very fast and efficient, especially when the training set is too large to load into the memory.

Stochastic Gradient Descent (SGD):

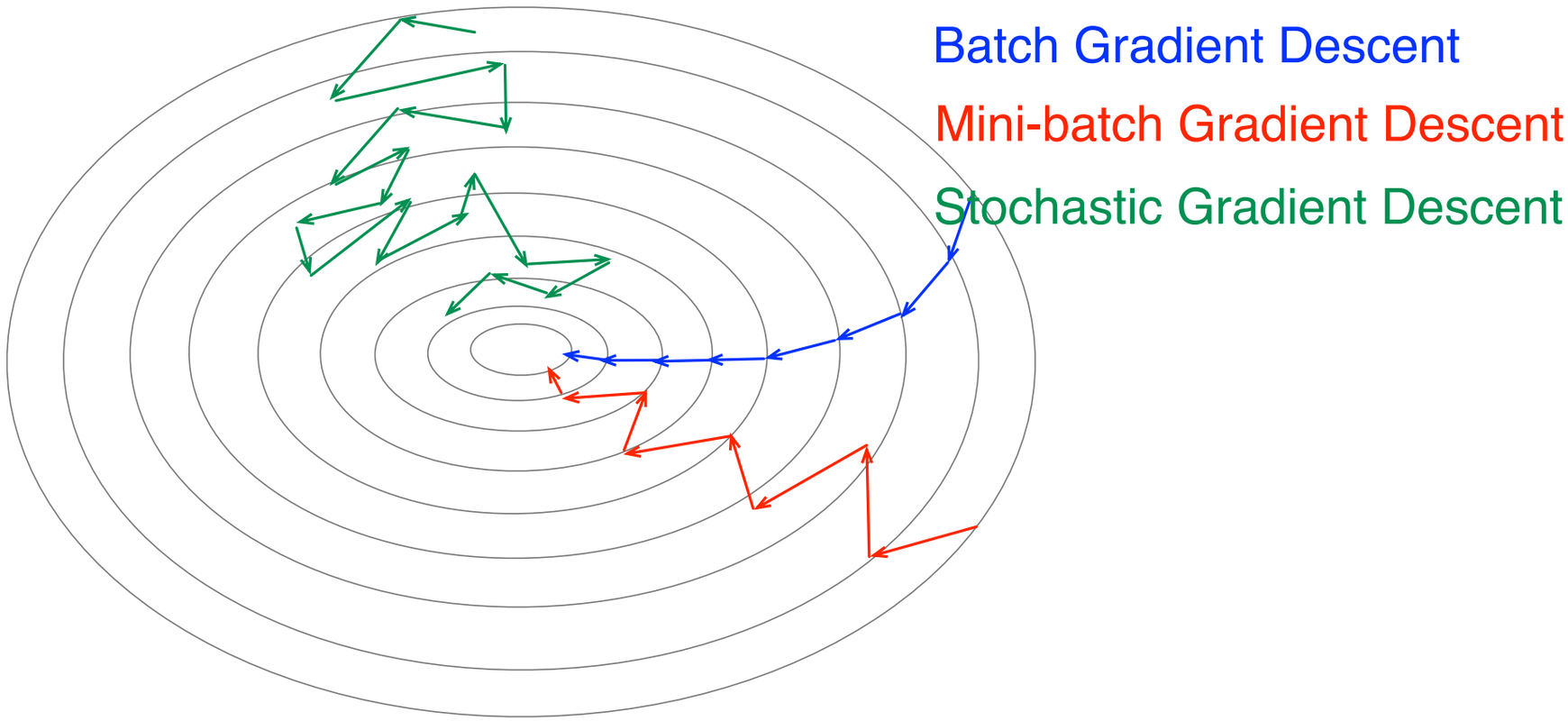
- Nevertheless, it is bouncing around the minima and unlike BGD the SGD is not moving straightly toward the minima.

- Due to its bouncing behavior, if the cost function is non-convex, it will rarely approach the global minima.

- If the cost function is convex, BGD can approach global minima easily. On the other hand, if the cost function is non-convex BGD is stuck in local minima, but SGD can jump out of the local minima, because of its irregular bouncing behavior.

Mini Batch Gradient Descent

- Mini Batch Gradient Descent updates model parameters using small random subsets of the training data in each iteration.



Bouncing Problem in SGD

- SGD is bouncing around the minima, but BGD moves toward the minima. However, there is no guarantee that this minimum is the global minima and it could be a weak local minima.

- Even sometimes when the step size is too large the gradient descent can jump out of the minima (as we have seen earlier). This problem is called **overshooting**.

Reducing Learning Rate

- One solution to enforce SGD moves toward minima (even in cone shape functions) is reducing the learning rate slowly and cautiously, which helps the algorithm to settle at the global minima.

- The process of slowly reducing the learning rate to reach global minima is called simulated annealing and the function that determines a value for the learning rate is called learning schedule.

- If the learning rate reduces too fast we might stuck in local minima, if it reduces too slow, we might jump to global minima and end up in a sub-optimal solution. Therefore, implementing a good learning schedule is not easy.

Summary of Gradient Descent

	Advantages	DisAdvantages
Batch Gardient Descent	- no bouncing and no variance - can identify global minima in convex function	- very slow - not suitable for large dataset - can stuck in local minima
Mini-Batch Gardient Descent	- fast - suitable for large dataset - reduce the variance overhead of SGD	- sensitive to the learning schedule - can stuck in local minima but better than BGD
Stochastic Gardient Descent	- very fast - suitable for large dataset - very unlikely to stuck in local minima	- sensitive to deal learning schedule - very high variance that might miss a good minima

- Batch Gradient Descent: Batch size = Training set size
- Stochastic Gradient Descent: Batch size = 1
- Mini-Batch Gradient Descent: 1 < Batch size < Training set size

Gradient Descent (1/3)

```
#-------------- Gradient descent algorithm ----------------
def gradient_descent(derivative, bounds, n_iter, step_size):
    # track all solutions
    solutions, scores = list(), list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # run the gradient descent
    for i in range(n_iter):
    # calculate gradient
    gradient = derivative(solution)
    # take a step
    solution = solution - step_size * gradient
    # evaluate candidate point
    solution_eval = objective(solution)
    # store solution
    solutions.append(solution)
    scores.append(solution_eval)
    # report progress
    print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
return [solutions, scores]
```

Gradient Descent (2/3)

```
#-------- Create a sample Objective Function & plot it-------
from numpy import *
import matplotlib.pyplot as plt

# objective function
def myobjective(x):
    return x**2.0

# derivative of objective function
def derivative(x):
    return x * 2.0

# define range for input
r_min, r_max = -1.0, 1.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max+0.1, 0.1)
# compute targets
results = myobjective(inputs)
# create a line plot of input vs result
plt.plot(inputs, results)
# show the plot
plt.show()
```

Gradient Descent (3/3)

```
#------- Apply gradient descent on the myObjective function ----

# example of gradient descent for a one-dimensional function
from numpy import asarray
from numpy.random import rand
import matplotlib.pyplot as plt

# define range for input
bounds = asarray([[-1.0, 1.0]])
# define the total iterations
n_iter = 30
# define the step size
step_size = 0.1 # TODO: experiment with different step size
# perform the gradient descent search
solutions, scores = gradient_descent(derivative, bounds, n_iter, step_size)
print('Done!')
# compute targets
results = myobjective(inputs)
# create a line plot of input vs result
plt.plot(inputs, results)
# plot the solutions found
plt.plot(solutions, scores, '.-', color='red')
# show the plot
plt.show()
```

- We will get more examples with Gradient Descent but not details of its implementation.

Newton Method

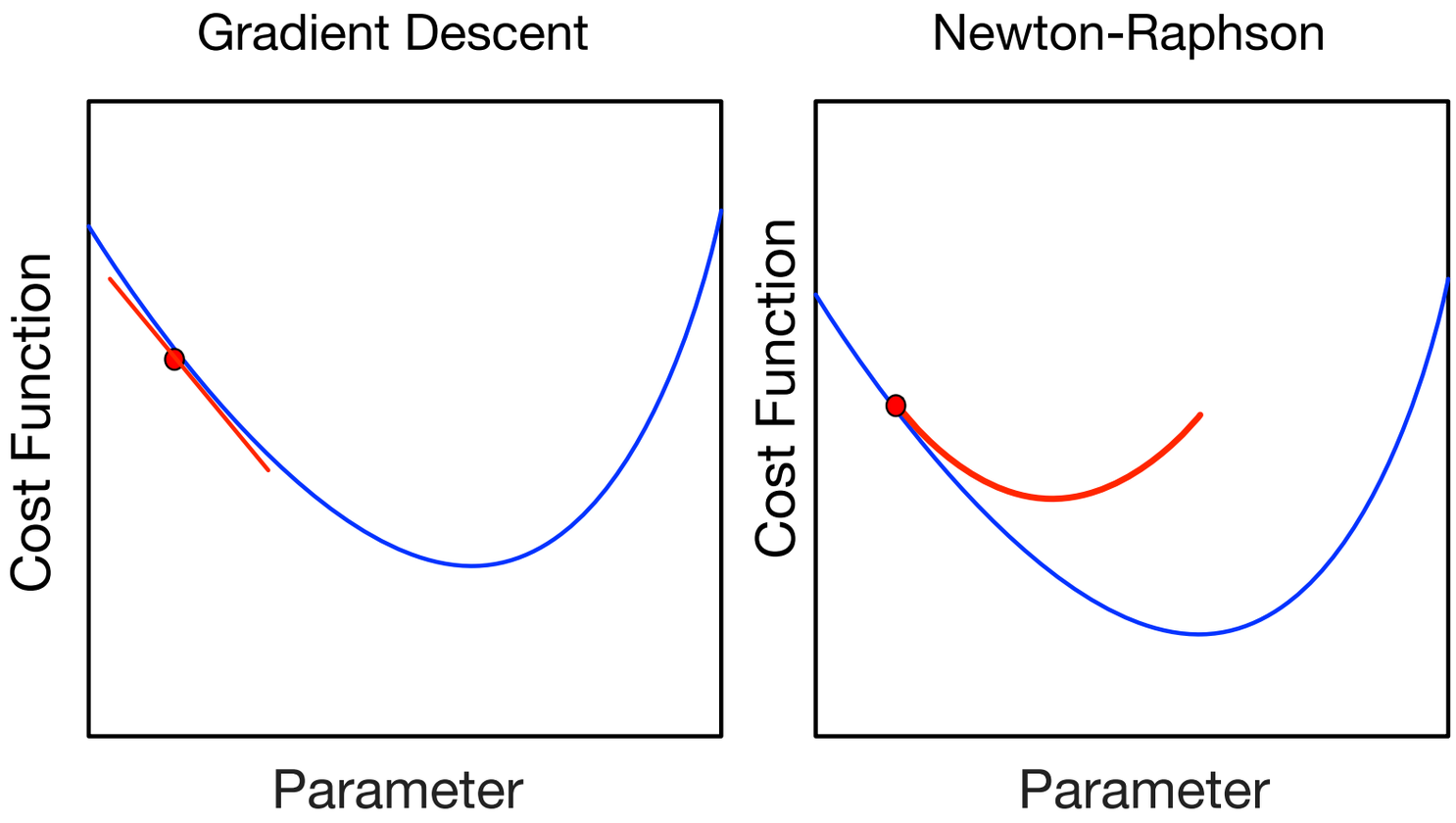
- Gradient descent algorithms and other first-order iterative algorithms (first-order means using the first derivative) can determine the direction to move toward a minima (either good local or global).

- However, a limitation of first-order optimization algorithms are that *they cannot determine the right step size*.

- Second-order optimization algorithms, e.g. Newton-Raphson or Newton, is another iterative method that allows us to determine the *approximate step size* toward a minima.

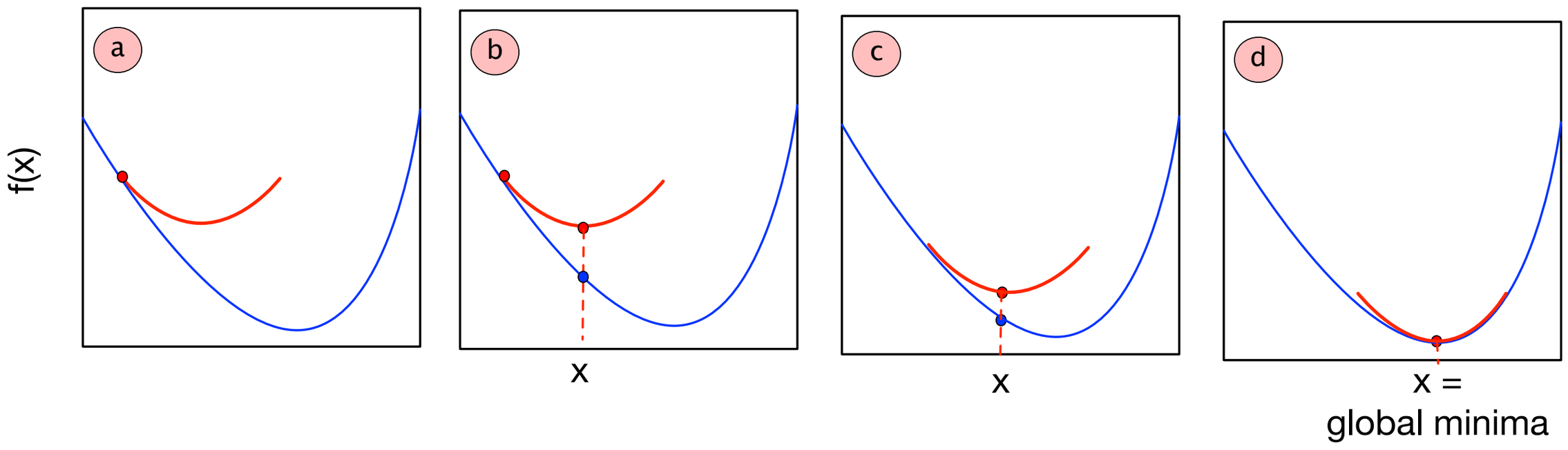
Newton Method

- Gradient descent draws a tangent line (at the given random point), which is identified by a derivative. Newton-Raphson's method draws a parabola (at the given random point), which is identified by the second-order derivative.



Newton Method

- Since the parabola is a convex function, it is very easy to find the single minimum in the parabola.
- The minimum of parabola represents $f(x)$ and by having $f(x)$ its associate x will be calculated as well.
- The next point to continue the iterative search for minima will be the recently identified x (which is extracted from the center of the recently created parabola).



Mathematical concepts of Newton Optimization

Assuming the current point is x_t how did we calculate the next point, $x_t + 1$ in gradient descent? If you remember we use this equation:

$$x_{t+1} = x_t - \alpha_t \,.\, \nabla f(x_t)$$

Based on the Taylor series, a (quadratic) approximation function is being used to define the Newton-Raphson method. You do not need to remember the Taylor series for that, but for your information, $q(x)$ is the quadratic approximation for the Taylor series:

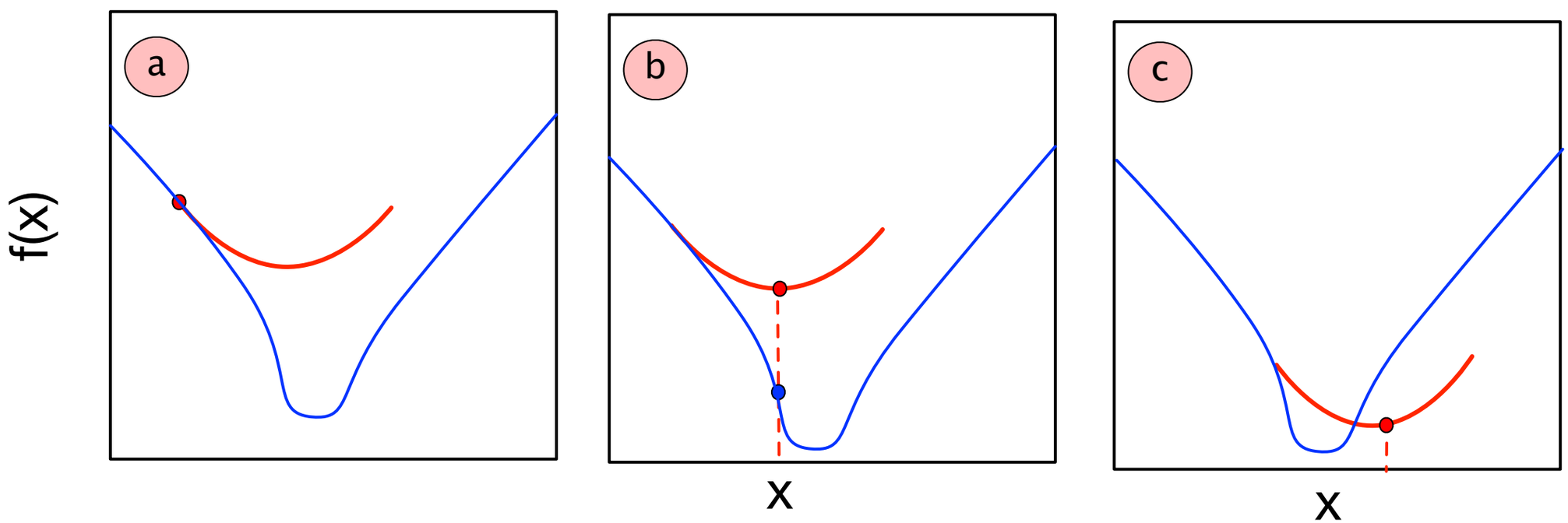
$$q(x) = f(x^{(k)}) + (x - x^{(k)}) \,.\, f'(x^{(k)}) + \frac{x - x^{(k)}}{2} \,.\, f''(x^{(k)})$$

To calculate the next point instead of using the **slope** we substitute it with the **Hessian** of that point. Therefore, we will have $x_{t+1} = x_t - Hessian^{-1} \,.\, Gradient.$

Or we can write it as follows: $x_{t+1} = x_t - [\nabla^2 f(x_t)]^{-1} \,.\, \nabla f(x_t)$

Newton Problem

- Sometimes the cost function is fairly flat and the parabola takes a too large step that falls outside the function. This is a big drawback of the Newton-Raphson method. For example, in the Figure we can see the the parabola is too larger than the mimina of function
- Therefore, the Newton-Raphson method is working when we are close to the minimum.



Newton-Raphson Example to Solve an Equation

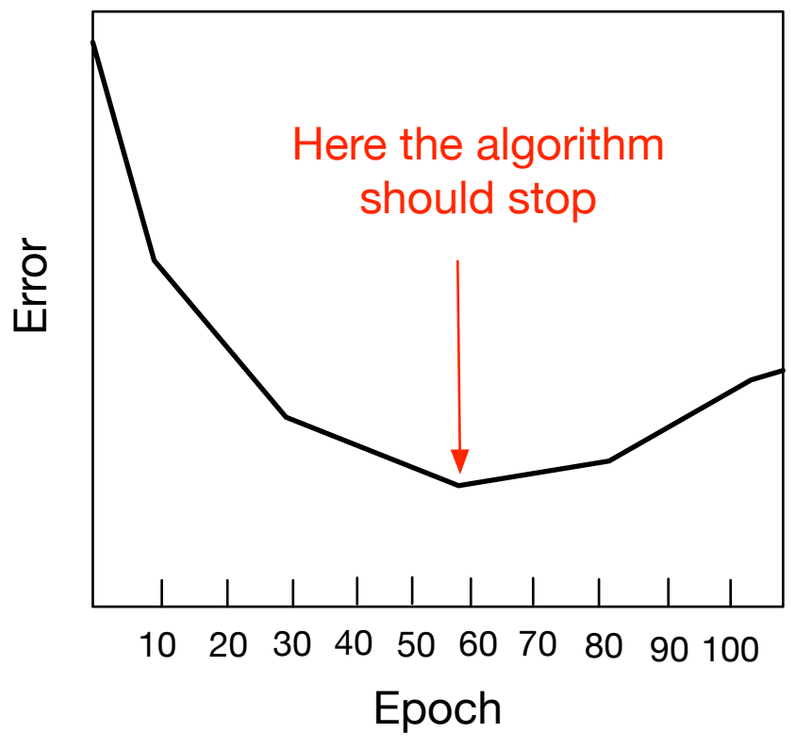
```
#-------- Using Newton-Raphson from Scipy to solve Equation---------
from scipy.optimize import newton

#----- Random Complex function
def f(x):
    return (1/4)*x**3+(3/4)*x**2-(3/2)*x-2
x = 4
x0 = newton(f, x, fprime=None, args=(), tol=1.48e-08, maxiter=10,
fprime2=None)

print('x: ', x)
print('x0: ', x0)
print("f(x0) = ", ((1/4)*x**3+(3/4)*x**2-(3/2)*x-2 ))
```

Early Stopping

- In an iterative learning optimization algorithm, usually in each epoch, the error gets reduced, until a specific point, and then again it starts to increase.

- Before finishing all requested number of epochs when the error rate starts to get higher we should stop, This phenomenon is called early stopping.

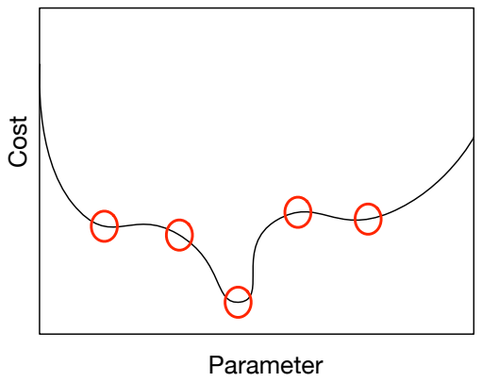
- Most implementation of gradient descent objective function, enables the user to specify having or not having an early stopping as a hyperparameter.



Some Notes

- The minima data point is where the gradient is zero, but a zero-gradient does not imply necessary optimality.

- **Inflection points** are when the curve of the function goes from downward to upward or vice versa. Where the derivative is zero, but there are not necessarily local or global minima points. The following Figure presents examples of inflection points that are not minima or maxima.



- A model is composed of a set of features, and while using gradient descent, we should ensure that all features have a similar scale.

- Gradient descent for linear regression easily gets to the global minima, because the cost function of linear regression is convex and it has a bowl shape. Therefore, there is no challenge of being stuck in local minima, never reaching global minima, etc.

- SGD is good for use in non-convex cost functions and reduces the chance of getting stuck in local minima.

- Gradient descent optimization algorithms assume the training dataset data objects are **independent and identically distributed** (i.i.d). Therefore, to ensure they are not used by the algorithm in any sorted or ordered approach, it is recommended to shuffle the dataset before feeding it into the gradient descent algorithm.

- Gradient descent is a type of “hill climbing” algorithm, a popular optimization algorithm. However, gradient descent uses the slope in the local neighbors and then chooses the point that has the steepest slope; hill climbing look uses a cost function in the local neighbors and chooses the point that has the lowest cost score.