**Gradient Descent**

Gradient descent (GD) is an algorithm that minimizes (or maximizes) functions. To apply, start at an initial set of a function’s parameter values and iteratively move toward a set of parameter values that minimize the function. Iterative minimization is achieved using calculus by taking steps in the negative direction of the function’s gradient. GD is important because optimization is a big part of machine learning. Also, GD is easy to implement, generic, and efficient (fast). GD concepts with examples include:

1. Simple function minimization (and maximization)
2. Sigmoid function minimization (and maximization)
3. Euclidean distance minimization controlling for step size
4. Stabilizing Euclidean distance minimization with Monte Carlo simulation
5. Substituting a NumPy method to hasten Euclidean distance minimization
6. Stochastic gradient descent minimization and maximization

*Simple Function Minimization (and Maximization)*

GD is a 1st order iterative optimization algorithm for finding the minimum of a function f. A function can be denoted as f or f(x). Simply, GD finds the minimum error by minimizing (or maximizing) a cost function. A cost function is something that you want to minimize.

Let’s begin with a minimization example. To find the local minimum of f, take steps proportional to the negative of the gradient of f at the current point. The gradient is the derivative (rate of change) of f. The only weakness of GD is that it finds the local minimum rather than the minimum for the whole function.

The power rule is used to differentiate functions of the form f(x) = xr:

xn = nxn-1

So, the derivative of xn equals nxn-1. Simply, the derivative is the product of the exponent times x with the exponent reduced by 1. To minimize f(x) = x4 – 3x3 + 2 find the derivative, which is f'(x) = 4x3 – 9x2. So, the 1st step is always to find the derivative f'(x). The 2nd step is to plot the original function to get an idea of its shape. The 3rd step is to run GD. The 4th step is to plot the local minimum.

The 1st example finds the local minimum of f(x). It displays f(x), f'(x), and minimum in a subplot.

import matplotlib.pyplot as plt, numpy as np

def f(x):

return x\*\*4 - 3 \* x\*\*3 + 2

def df(x):

return 4 \* x\*\*3 - 9 \* x\*\*2

if \_\_name\_\_ == "\_\_main\_\_":

x = np.arange(-5, 5, 0.2)

y, y\_dx = f(x), df(x)

f, axarr = plt.subplots(3, sharex=True)

axarr[0].plot(x, y, color='mediumspringgreen')

axarr[0].set\_xlabel('x')

axarr[0].set\_ylabel('f(x)')

axarr[0].set\_title('f(x)')

axarr[1].plot(x, y\_dx, color='coral')

axarr[1].set\_xlabel('x')

axarr[1].set\_ylabel('dy/dx(x)')

axarr[1].set\_title('derivative of f(x)')

axarr[2].set\_xlabel('x')

axarr[2].set\_ylabel('GD')

axarr[2].set\_title('local minimum')

iterations, cur\_x, gamma, precision = 0, 6, 0.01, 0.00001

previous\_step\_size = cur\_x

while previous\_step\_size > precision:

prev\_x = cur\_x

cur\_x += -gamma \* df(prev\_x)

previous\_step\_size = abs(cur\_x - prev\_x)

iterations += 1

axarr[2].plot(prev\_x, cur\_x, "o")

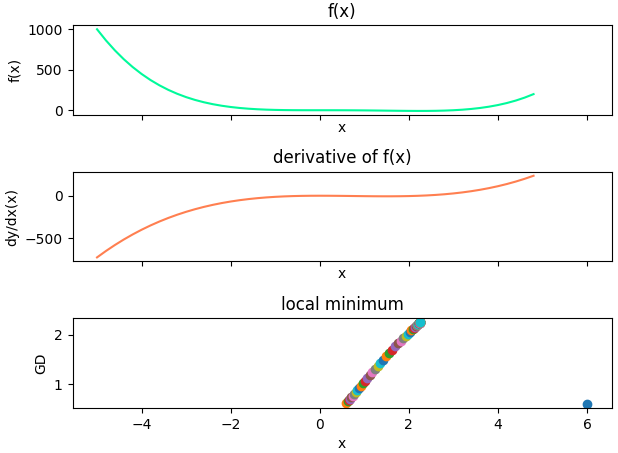
f.subplots\_adjust(hspace=0.3)

f.tight\_layout()

plt.show()

print ('minimum:', cur\_x, '\niterations:', iterations)

Output:





The code example begins by importing matplotlib and numpy. It continues with function f(x) used to plot the original function and function df(x) used to plot the derivative. The main block begins by creating values for f(x). It continues by creating a subplot. GD begins by initializing variables. Variable cur\_x is the starting point for the simulation. Variable gamma is the step size. Variable precision is the tolerance. Smaller tolerance translates into more precision, but requires more iterations (resources). The simulation continues until previous\_step\_size is greater than precision. Each iteration multiplies -gamma (step\_size) by the gradient (derivative) at the current point to move it to the local minimum. Variable previous\_step\_size is then assigned the difference between cur\_x and prev\_x. Each point is plotted. The minimum for f(x) solving for x is approximately 2.25. I know this result is correct because I calculated it by hand. Check out [**http://www.dummies.com/education/math/calculus/how-to-find-local-extrema-with-the-first-derivative-test/**](http://www.dummies.com/education/math/calculus/how-to-find-local-extrema-with-the-first-derivative-test/) for a nice lesson on how to calculate by hand.

The 2nd example finds the local minimum and maximum of f(x) = x3 – 6x2 + 9x + 15. First find f'(x), which is 3x2 – 12x + 9. Next, find the local minimum, plot, local maximum, and plot. I don’t use a subplot in this case because the visualization is not as rich. That is, it is much easier to see the approximate local minimum and maximum by looking at a plot of f(x), and easier to see how the GD process works its magic.  
  
import matplotlib.pyplot as plt, numpy as np

def f(x):

return x\*\*3 - 6 \* x\*\*2 + 9 \* x + 15

def df(x):

return 3 \* x\*\*2 - 12 \* x + 9

if \_\_name\_\_ == "\_\_main\_\_":

x = np.arange(-0.5, 5, 0.2)

y = f(x)

plt.figure('f(x)')

plt.xlabel('x')

plt.ylabel('f(x)')

plt.title('f(x)')

plt.plot(x, y, color='blueviolet')

plt.figure('local minimum')

plt.xlabel('x')

plt.ylabel('GD')

plt.title('local minimum')

iterations, cur\_x, gamma, precision = 0, 6, 0.01, 0.00001

previous\_step\_size = cur\_x

while previous\_step\_size > precision:

prev\_x = cur\_x

cur\_x += -gamma \* df(prev\_x)

previous\_step\_size = abs(cur\_x - prev\_x)

iterations += 1

plt.plot(prev\_x, cur\_x, "o")

local\_min = cur\_x

print ('minimum:', local\_min, 'iterations:', iterations)

plt.figure('local maximum')

plt.xlabel('x')

plt.ylabel('GD')

plt.title('local maximum')

iterations, cur\_x, gamma, precision = 0, 0.5, 0.01, 0.00001

previous\_step\_size = cur\_x

while previous\_step\_size > precision:

prev\_x = cur\_x

cur\_x += -gamma \* -df(prev\_x)

previous\_step\_size = abs(cur\_x - prev\_x)

iterations += 1

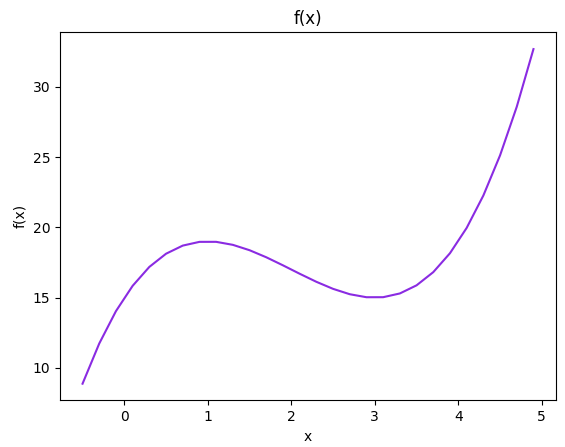
plt.plot(prev\_x, cur\_x, "o")

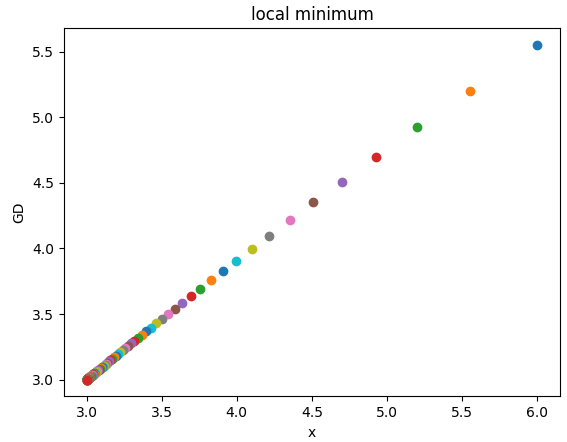
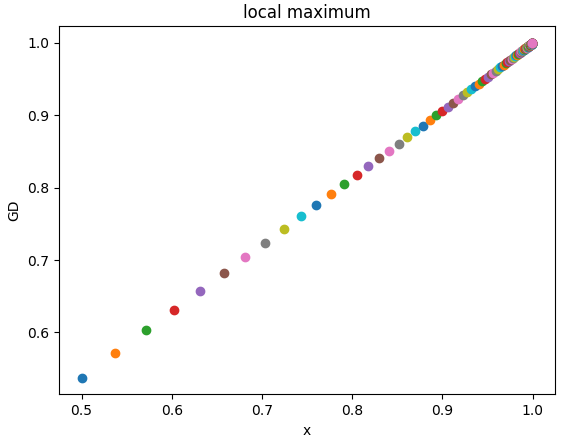
local\_max = cur\_x

print ('maximum:', local\_max, 'iterations:', iterations)

plt.show()

Output:  
  

The code begins by importing matplotlib and numpy libraries. It continues with functions f(x) and df(x), which represent the original function and its derivative algorithmically. The main block begins by creating data for f(x) and plotting it. It continues by finding the local minimum and maximum, and plotting them. Notice the cur\_x (the beginning point) for local minimum is 6, while it is 0.5 for local maximum. This is where data science is more of an art than a science because I found these points by trial and error. Also notice that GD for the local maximum is the negation of the derivative. Again, I know that the results are correct because I calculated both local minimum and maximum by hand. The main reason that I used separate plots rather than a subplot for this example is to demonstrate why it is so important to plot f(x). Just by looking at the plot, you can tell that the local maximum of x for f(x) is close to one, and the local minimum of x for f(x) is close to 3. In addition, you can see that the function has an overall maximum that is greater than 1 from this plot.

*Sigmoid Function Minimization (and Maximization)*

A sigmoid function is a mathematical function with a S-shaped or sigmoid curve. It is very important in data science for several reasons. First, it is easily differentiable with respect to network parameters, which are pivotal in training neural networks. Second, the cumulative distribution functions for many common probability distributions are sigmoidal. Third, many natural processes (e.g., complex learning curves) follow a sigmoidal curve over time. So, a sigmoid function is often used if no specific mathematical model is available.

The 1st example finds the local minimum of the sigmoid function:

import matplotlib.pyplot as plt, numpy as np

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def df(x):

return x \* (1-x)

if \_\_name\_\_ == "\_\_main\_\_":

x = np.arange(-10., 10., 0.2)

y, y\_dx = sigmoid(x), df(x)

f, axarr = plt.subplots(3, sharex=True)

axarr[0].plot(x, y, color='lime')

axarr[0].set\_xlabel('x')

axarr[0].set\_ylabel('f(x)')

axarr[0].set\_title('Sigmoid Function')

axarr[1].plot(x, y\_dx, color='coral')

axarr[1].set\_xlabel('x')

axarr[1].set\_ylabel('dy/dx(x)')

axarr[1].set\_title('Derivative of f(x)')

axarr[2].set\_xlabel('x')

axarr[2].set\_ylabel('GD')

axarr[2].set\_title('local minimum')

iterations, cur\_x, gamma, precision = 0, 0.01, 0.01, 0.00001

previous\_step\_size = cur\_x

while previous\_step\_size > precision:

prev\_x = cur\_x

cur\_x += -gamma \* df(prev\_x)

previous\_step\_size = abs(cur\_x - prev\_x)

iterations += 1

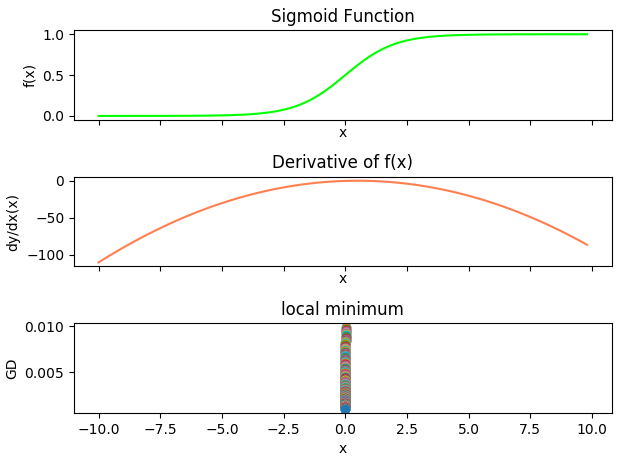
plt.plot(prev\_x, cur\_x, "o")

f.subplots\_adjust(hspace=0.3)

f.tight\_layout()

print ('minimum:', cur\_x, '\niterations:', iterations)

plt.show()  
  
Output:  
  

The code begins by importing matplotlib and numpy. It continues with functions sigmoid(x) and df(x), which represent the sigmoid function and its derivative algorithmically. The main block begins by creating data for f(x) and f'(x). It continues by creating subplots for f(x), f'(x), and the local minimum. In this case, using subplots was fine for visualization. It is easy to see from the f(x) and f'(x) plots that the local minimum is close to 0. Next, the code runs GD to find the local minimum and plots it. Again, the starting point for GD, cur\_x, was found by trial and error. If you start cur\_x further from the local minimum (you can estimate this by looking at the subplot of f'(x)), the number of iterations increases because it takes longer for the GD algorithm to converge on the local minimum. As expected, the local minimum is approximately 0.

The 2nd example finds the local maximum of the sigmoid function:

import matplotlib.pyplot as plt, numpy as np

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def df(x):

return x \* (1-x)

if \_\_name\_\_ == "\_\_main\_\_":

x = np.arange(-10., 10., 0.2)

y, y\_dx = sigmoid(x), df(x)

f, axarr = plt.subplots(3, sharex=True)

axarr[0].plot(x, y, color='lime')

axarr[0].set\_xlabel('x')

axarr[0].set\_ylabel('f(x)')

axarr[0].set\_title('Sigmoid Function')

axarr[1].plot(x, y\_dx, color='coral')

axarr[1].set\_xlabel('x')

axarr[1].set\_ylabel('dy/dx(x)')

axarr[1].set\_title('Derivative of f(x)')

axarr[2].set\_xlabel('x')

axarr[2].set\_ylabel('GD')

axarr[2].set\_title('local maximum')

iterations, cur\_x, gamma, precision = 0, 0.01, 0.01, 0.00001

previous\_step\_size = cur\_x

while previous\_step\_size > precision:

prev\_x = cur\_x

cur\_x += -gamma \* -df(prev\_x)

previous\_step\_size = abs(cur\_x - prev\_x)

iterations += 1

plt.plot(prev\_x, cur\_x, "o")

f.subplots\_adjust(hspace=0.3)

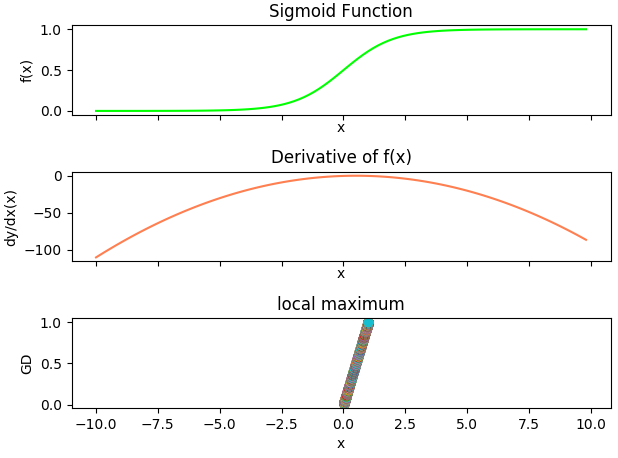
f.tight\_layout()

print ('maximum:', cur\_x, '\niterations:', iterations)

plt.show()

Output:





The code begins by importing matplotlib and numpy. It continues with functions sigmoid(x) and df(x), which represent the sigmoid function and its derivative algorithmically. The main block begins by creating data for f(x) and f'(x). It continues by creating subplots for f(x), f'(x), and the local maximum. It is easy to see from the f(x) plot that the local maximum is close to 1. Next, the code runs GD to find the local maximum and plots it. Again, the starting point for GD, cur\_x, was found by trial and error. If you start cur\_x further from the local maximum (you can estimate this by looking at the subplot of f(x)), the number of iterations increases because it takes longer for the GD algorithm to converge on the local maximum. As expected, the local maximum is approximately 1.

*Euclidean Distance Minimization Controlling for Step Size*

Euclidean distance is the ordinary straight-line distance between two points in Euclidean space. With this distance, Euclidean space becomes a metric space. The associated norm is the Euclidean norm (EN). The EN assigns each vector the length of its arrow. So, EN is really just the magnitude of a vector. A vector space on which a norm is defined is the normed vector space.

To find the local minimum of f(x) in 3-dimensional (3D) space, the 1st step is to find the minimum for all 3D vectors. The 2nd step is to create a random 3D vector [x, y, z]. The 3rd step is to pick a random starting point, and then take tiny steps in the opposite direction of the gradient f'(x) until a point is reached where the gradient is very small. Each tiny step (from the current vector to the next vector) is measured with the ED metric. The ED metric is the distance between two points in Euclidean space. The metric is required because we need to know how to move for each tiny step. So, the ED metric supplements GD to find the local minimum in 3D space.

The code example finds the local minimum of the sigmoid function in 3-dimensional space:

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import random, numpy as np

from scipy.spatial import distance

def step(v, direction, step\_size):

return [v\_i + step\_size \* direction\_i

for v\_i, direction\_i in zip(v, direction)]

def sigmoid\_gradient(v):

return [v\_i \* (1-v\_i) for v\_i in v]

def mod\_vector(v):

for i, v\_i in enumerate(v):

if v\_i == float("inf") or v\_i == float("-inf"):

v[i] = random.randint(-1, 1)

return v

if \_\_name\_\_ == "\_\_main\_\_":

v = [random.randint(-10, 10) for i in range(3)]

tolerance = 0.0000001

iterations = 1

fig = plt.figure('Euclidean')

ax = fig.add\_subplot(111, projection='3d')

while True:

gradient = sigmoid\_gradient(v)

next\_v = step(v, gradient, -0.01)

xs = gradient[0]

ys = gradient[1]

zs = gradient[2]

ax.scatter(xs, ys, zs, c='lime', marker='o')

v = mod\_vector(v)

next\_v = mod\_vector(next\_v)

test\_v = distance.euclidean(v, next\_v)

if test\_v < tolerance:

break

v = next\_v

iterations += 1

print ('minimum:', test\_v, '\niterations:', iterations)

ax.set\_xlabel('X axis')

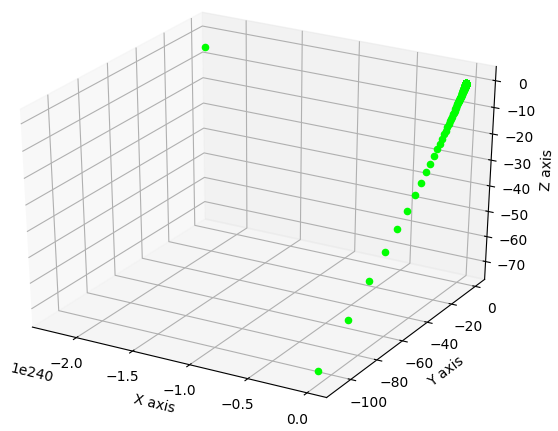
ax.set\_ylabel('Y axis')

ax.set\_zlabel('Z axis')

plt.tight\_layout()

plt.show()  
  
Output:





The code begins by importing matplotlib, mpl\_toolkits, random, numpy, and scipy libraries. Function step() moves a vector in a direction (based on the gradient), by a step size. Function sigmoid\_gradient() is the f'(sigmoid) returned as a point in 3D space. Function mod\_vector() ensures that an erroneous vector generated by the simulation is handled properly. The main block begins by creating a randomly generated 3D vector [x, y, z] as a starting point for the simulation. It continues by creating a tolerance (precision). A smaller tolerance results in a more accurate result. A subplot is created to hold a 3D rendering of the local minimum. The GD simulation creates a set of 3D vectors influenced by the sigmoid gradient until the gradient is very small. The size (magnitude) of the gradient is calculated by the ED metric. The local minimum, as expected is close to 0.

*Stabilizing Euclidean Distance Minimization with Monte Carlo Simulation*

The Euclidean distance experiment in the previous example is anchored by a stochastic process. Namely, the starting vector v is stochastically generated by randomint(). As a result, each run of the GD experiment generates a different result for number of iterations. From chapter 2, we already know that Monte Carlo simulation (MCS) efficiently models stochastic (random) processes. However, MCS can also stabilize stochastic experiments.

The code example first wraps the GD experiment in a loop that runs n number of simulations. With n simulations, an average number of iterations is calculated. The resultant code is then wrapped in another loop that runs m trials. With m trials, an average gap between each average number of iterations, is calculated. Gap is calculated by subtracting the minimum from the maximum average iteration. The smaller the gap, the more stable (accurate) the result. To increase accuracy, increase simulations (n). The only limitation is computing power. That is, running 1000 simulations takes a lot more computing power than 100. Stable (accurate) results allow comparison to alternative experiments.

import random, numpy as np

from scipy.spatial import distance

def step(v, direction, step\_size):

return [v\_i + step\_size \* direction\_i

for v\_i, direction\_i in zip(v, direction)]

def sigmoid\_gradient(v):

return [v\_i \* (1-v\_i) for v\_i in v]

def mod\_vector(v):

for i, v\_i in enumerate(v):

if v\_i == float("inf") or v\_i == float("-inf"):

v[i] = random.randint(-1, 1)

return v

if \_\_name\_\_ == "\_\_main\_\_":

trials= 10

sims = 10

avg\_its = []

for \_ in range(trials):

its = []

for \_ in range(sims):

v = [random.randint(-10, 10) for i in range(3)]

tolerance = 0.0000001

iterations = 0

while True:

gradient = sigmoid\_gradient(v)

next\_v = step(v, gradient, -0.01)

v = mod\_vector(v)

next\_v = mod\_vector(next\_v)

test\_v = distance.euclidean(v, next\_v)

if test\_v < tolerance:

break

v = next\_v

iterations += 1

its.append(iterations)

a = round(np.mean(its))

avg\_its.append(a)

gap = np.max(avg\_its) - np.min(avg\_its)

print (trials, 'trials with', sims, 'simulations each:')

print ('gap', gap)

print ('avg iterations', round(np.mean(avg\_its)))

Output:







Output is for 10, 100, and 1000 simulations. By running 1000 simulations ten times (trials), the gap is down to 13. So, confidence is high that the number of iterations required to minimize the function is close to 1,089 . We can further stabilize by wrapping the code in another loop to decrease variation in gap and number of iterations. However, computer processing time becomes an issue. Leveraging MCS for this type of experiment makes a strong case for cloud computing. It may be tough to get your head around this application of MCS, but it is a very powerful tool for working with and solving data science problems.

*Substituting a NumPy Method to Hasten Euclidean Distance Minimization*

Since numpy arrays are faster than Python lists, it follows that using a numpy method would be more efficient for calculating Euclidean distance. The code example substitutes np.linalg.norm() for distance.euclidean() to calculate Euclidean distance for the GD experiment.

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import random, numpy as np

def step(v, direction, step\_size):

return [v\_i + step\_size \* direction\_i

for v\_i, direction\_i in zip(v, direction)]

def sigmoid\_gradient(v):

return [v\_i \* (1-v\_i) for v\_i in v]

def round\_v(v):

return np.round(v, decimals=3)

if \_\_name\_\_ == "\_\_main\_\_":

v = [random.randint(-10, 10) for i in range(3)]

tolerance = 0.0000001

iterations = 1

fig = plt.figure('norm')

ax = fig.add\_subplot(111, projection='3d')

while True:

gradient = sigmoid\_gradient(v)

next\_v = step(v, gradient, -0.01)

round\_gradient = round\_v(gradient)

xs = round\_gradient[0]

ys = round\_gradient[1]

zs = round\_gradient[2]

ax.scatter(xs, ys, zs, c='lime', marker='o')

norm\_v = np.linalg.norm(v)

norm\_next\_v = np.linalg.norm(next\_v)

test\_v = norm\_v - norm\_next\_v

if test\_v < tolerance:

break

v = next\_v

iterations += 1

print ('minimum:', test\_v, '\niterations:', iterations)

ax.set\_xlabel('X axis')

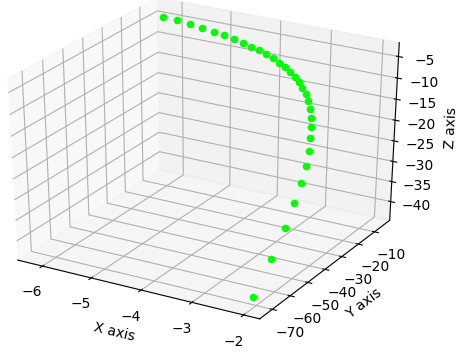
ax.set\_ylabel('Y axis')

ax.set\_zlabel('Z axis')

plt.show()

Output:





The number of iterations is much lower at 31. However, given that the GD experiment is stochastic, we can use MCS for objective comparison.

Using the same MCS methodology, the code example first wraps the GD experiment in a loop that runs n number of simulations. The resultant code is then wrapped in another loop that runs m trials.

import random, numpy as np

def step(v, direction, step\_size):

return [v\_i + step\_size \* direction\_i

for v\_i, direction\_i in zip(v, direction)]

def sigmoid\_gradient(v):

return [v\_i \* (1-v\_i) for v\_i in v]

def round\_v(v):

return np.round(v, decimals=3)

if \_\_name\_\_ == "\_\_main\_\_":

trials= 10

sims = 10

avg\_its = []

for \_ in range(trials):

its = []

for \_ in range(sims):

v = [random.randint(-10, 10) for i in range(3)]

tolerance = 0.0000001

iterations = 0

while True:

gradient = sigmoid\_gradient(v)

next\_v = step(v, gradient, -0.01)

norm\_v = np.linalg.norm(v)

norm\_next\_v = np.linalg.norm(next\_v)

test\_v = norm\_v - norm\_next\_v

if test\_v < tolerance:

break

v = next\_v

iterations += 1

its.append(iterations)

a = round(np.mean(its))

avg\_its.append(a)

gap = np.max(avg\_its) - np.min(avg\_its)

print (trials, 'trials with', sims, 'simulations each:')

print ('gap', gap)

print ('avg iterations', round(np.mean(avg\_its)))

Output:







Processing is much faster using numpy. The average number of iterations is close to 193. As such, using the numpy alternative for calculating Euclidean distance is more than 5 times faster!

*Stochastic Gradient Descent Minimization and Maximization*

Up to this point in the chapter, optimization experiments used batch GD. Batch GD computes the gradient using the whole dataset. Stochastic GD computes the gradient using a single sample, so it is computationally much faster. It is called stochastic GD because the gradient is randomly determined. However, unlike batch GD, stochastic GD is an approximation. If the exact gradient is required, stochastic GD is not optimal. Another issue with stochastic GD is that it can hover around the minimum forever without actually converging. So, it is important to plot progress of the simulation to see what is happening.

Let’s change direction and optimize another important function – residual sum of squares (RSS). A RSS function is a statistical technique that measures the amount of error (variance) remaining between the regression function and the data set. Regression analysis is an algorithm that estimates relationships between variables. It is widely used for prediction and forecasting. It is also a popular modeling and predictive algorithm for data science applications.  
  
The 1st code example generates a sample, runs the GD experiment n times, and processes the sample randomly:

import matplotlib.pyplot as plt

import random, numpy as np

def rnd():

return [random.randint(-10,10) for i in range(3)]

def random\_vectors(n):

ls = []

for v in range(n):

ls.append(rnd())

return ls

def sos(v):

return sum(v\_i \*\* 2 for v\_i in v)

def sos\_gradient(v):

return [2 \* v\_i for v\_i in v]

def in\_random\_order(data):

indexes = [i for i, \_ in enumerate(data)]

random.shuffle(indexes)

for i in indexes:

yield data[i]

if \_\_name\_\_ == "\_\_main\_\_":

v, x, y = rnd(), random\_vectors(3), random\_vectors(3)

data = list(zip(x, y))

theta = v

alpha, value = 0.01, 0

min\_theta, min\_value = None, float("inf")

iterations\_with\_no\_improvement = 0

n, x = 30, 1

for i, \_ in enumerate(range(n)):

y = np.linalg.norm(theta)

plt.scatter(x, y, c='r')

x = x + 1

s = []

for x\_i, y\_i in data:

s.extend([sos(theta), sos(x\_i), sos(y\_i)])

value = sum(s)

if value < min\_value:

min\_theta, min\_value = theta, value

iterations\_with\_no\_improvement = 0

alpha = 0.01

else:

iterations\_with\_no\_improvement += 1

alpha \*= 0.9

g = []

for x\_i, y\_i in in\_random\_order(data):

g.extend([sos\_gradient(theta), sos\_gradient(x\_i),

sos\_gradient(y\_i)])

for v in g:

theta = np.around(np.subtract(theta,alpha\*np.array(v)),3)

g = []

print ('minimum:', np.around(min\_theta, 4),

'with', i+1, 'iterations')

print ('iterations with no improvement:',

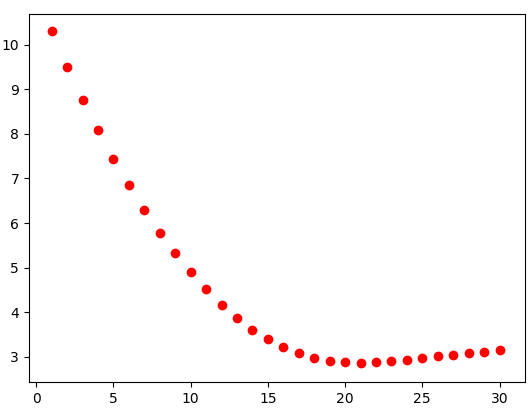
iterations\_with\_no\_improvement)

print ('magnitude of min vector:', np.linalg.norm(min\_theta))

plt.show()

Output:





The code begins by importing matplotlib, random, and numpy. It continues with function rnd(), which returns a list of random integers from -10 to 10. Function random\_vectors() generates a list (random sample) of n numbers. Function sos() returns the RSS for a vector. Function sos\_gradient() returns the derivative (gradient) of RSS for a vector. Function in\_random\_order() generates a list of randomly shuffled indexes. This function adds the stochastic flavor to the GD algorithm. The main block begins by generating a random vector v as the starting point for the simulation. It continues by creating a sample of x and y vectors of size 3. Next, the vector is assigned to theta, which is a common name for a vector of some general probability distribution. We can call the vector anything we want, but a common data science problem is to find the value(s) of theta. The code continues with a fixed step size alpha, minimum theta value, minimum ending value, iterations with no improvement, number of simulations n, and a plot value for the x-coordinate.

The simulation begins by assigning y the magnitude of theta. Next, it plots the current x and y coordinates. The x-coordinate is incremented by 1 to plot the convergence to the minimum for each y-coordinate. The next block of code finds the RSS for each theta, and the sample of x and y values. This value determines if the simulation is hovering around the local minimum rather than converging. The final part of the code traverses the sample data points in random (stochastic) order, finds the gradient of theta, x and y, places these three values in list g, and traverses this vector to find the next theta value.

Whew! This is not simple, but this is how stochastic GD operates. Notice that the minimum generated is 2.87, which is not the true minimum of 0. So, stochastic GD requires few iterations, but does not produce the true minimum.

The previous simulation can be refined by adjusting the algorithm for finding the next theta. In the previous example, the next theta is calculated for the gradient based on the current theta, x value, and y-value for each sample. However, the actual new theta is based on the 3rd data point in the sample. So, the 2nd example is refined by taking the minimum theta from the entire sample rather than the 3rd data point.

import matplotlib.pyplot as plt

import random, numpy as np

def rnd():

return [random.randint(-10,10) for i in range(3)]

def random\_vectors(n):

ls = []

for v in range(n):

ls.append([random.randint(-10,10) for i in range(3)])

return ls

def sos(v):

return sum(v\_i \*\* 2 for v\_i in v)

def sos\_gradient(v):

return [2 \* v\_i for v\_i in v]

def in\_random\_order(data):

indexes = [i for i, \_ in enumerate(data)]

random.shuffle(indexes)

for i in indexes:

yield data[i]

if \_\_name\_\_ == "\_\_main\_\_":

v, x, y = rnd(), random\_vectors(3), random\_vectors(3)

data = list(zip(x, y))

theta = v

alpha, value = 0.01, 0

min\_theta, min\_value = None, float("inf")

iterations\_with\_no\_improvement = 0

n, x = 60, 1

for i, \_ in enumerate(range(n)):

y = np.linalg.norm(theta)

plt.scatter(x, y, c='r')

x = x + 1

s = []

for x\_i, y\_i in data:

s.extend([sos(theta), sos(x\_i), sos(y\_i)])

value = sum(s)

if value < min\_value:

min\_theta, min\_value = theta, value

iterations\_with\_no\_improvement = 0

alpha = 0.01

else:

iterations\_with\_no\_improvement += 1

alpha \*= 0.9

g, t, m = [], [], []

for x\_i, y\_i in in\_random\_order(data):

g.extend([sos\_gradient(theta), sos\_gradient(x\_i),

sos\_gradient(y\_i)])

m = np.around([np.linalg.norm(x) for x in g], 2)

for v in g:

theta = np.around(np.subtract(theta,alpha\*np.array(v)),3)

t.append(np.around(theta,2))

mm = np.argmin(m)

theta = t[mm]

g, m, t = [], [], []

print ('minimum:', np.around(min\_theta, 4),

'with', i+1, 'iterations')

print ('iterations with no improvement:',

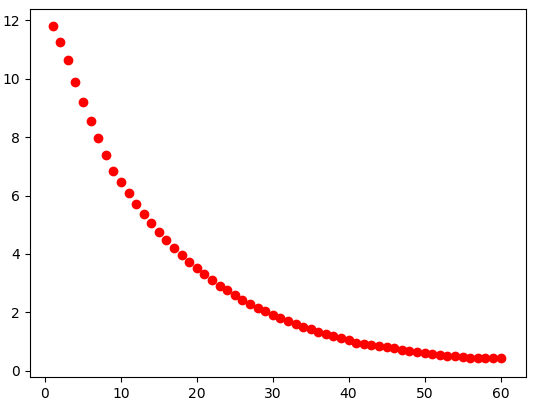
iterations\_with\_no\_improvement)

print ('magnitude of min vector:', np.linalg.norm(min\_theta))

plt.show()

Output:





The only difference in the code is toward the bottom where the minimum theta is calculated. Although it took 60 iterations, the minimum is much closer to 0 and much more stable. That is, the prior example deviates quite a bit more each time the experiment is run.

The 3rd example finds the maximum:

import matplotlib.pyplot as plt

import random, numpy as np

def rnd():

return [random.randint(-10,10) for i in range(3)]

def random\_vectors(n):

ls = []

for v in range(n):

ls.append([random.randint(-10,10) for i in range(3)])

return ls

def sos\_gradient(v):

return [2 \* v\_i for v\_i in v]

def negate(function):

def new\_function(\*args, \*\*kwargs):

return np.negative(function(\*args, \*\*kwargs))

return new\_function

def in\_random\_order(data):

indexes = [i for i, \_ in enumerate(data)]

random.shuffle(indexes)

for i in indexes:

yield data[i]

if \_\_name\_\_ == "\_\_main\_\_":

v, x, y = rnd(), random\_vectors(3), random\_vectors(3)

data = list(zip(x, y))

theta, alpha = v, 0.01

neg\_gradient = negate(sos\_gradient)

n, x = 100, 1

for i, row in enumerate(range(n)):

y = np.linalg.norm(theta)

plt.scatter(x, y, c='r')

x = x + 1

g = []

for x\_i, y\_i in in\_random\_order(data):

g.extend([neg\_gradient(theta), neg\_gradient(x\_i),

neg\_gradient(y\_i)])

for v in g:

theta = np.around(np.subtract(theta,alpha\*np.array(v)),3)

g = []

print ('maximum:', np.around(theta, 4),

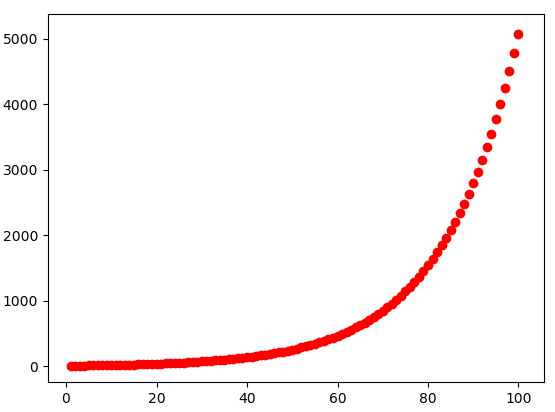
'with', i+1, 'iterations')

print ('magnitude of max vector:', np.linalg.norm(theta))

plt.show()

Output:





The only difference in the code from the 1st example is the negate() function, which negates the gradient to find the maximum. Since the maximum of RSS is infinity (we can tell by the visualization), we can stop at 100 iterations. Try 1000 iterations and see what happens.