MACHINE LEARNING MASTERY

Optimization for Machine Learning

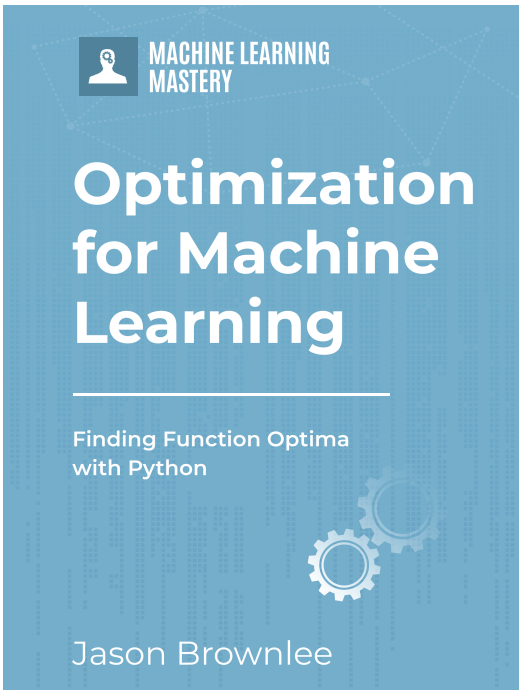
Finding Function Optima with Python

Jason Brownlee

This is Just a Sample

Thank-you for your interest in **Optimization for Machine Learning**.

This is just a sample of the full text. You can purchase the complete book online from: https://machinelearningmastery.com/optimization-for-machine-learning/



Contents

Introduction

Welcome to *Optimization for Machine Learning*.

You probably tried to shower in a hotel and turned the faucet left and right a couple times to get the right temperature of water. This is an example of optimization. We call this gradient descent. You probably also tried to visit all stores in your neighborhood to see where you can buy something the cheapest. This is another example of optimization, the naive way. We call this the exhaustive search. Indeed when we train a machine learning model, it is running optimization algorithm under the hood.

This book is to teach you step-by-step the basics of optimization algorithms that we use in machine learning, with executable examples in Python. We cover just enough to let you feel comfortable in doing your machine learning projects.

Who is this book for?

This book is for developers that may know some applied machine learning. Perhaps you have built some models and did some projects end-to-end, or modified from existing example code from popular tools to solve your own problem. Before you begin, this book assumes

- ▷ You know your way around basic Python for Programming
- ▷ You may know some basic NumPy for array manipulation
- ▷ You heard about gradient descent, simulated annealing, BFGS, or some other optimization algorithms and want to deepen your understanding

The following is to present the concept of function optimization and the numerical algorithms in doing so in a top-down and result-first approach, in the same style as you're familiar with MachineLearningMastery.com.

What to expect?

This book will teach you the basics of some optimization algorithms that you need to know as a machine learning practitioner. After reading and working through the book, you will know:

- ▷ What is function optimization and why it is relevant and important to machine learning

- ▷ The trade-off in applying optimization algorithms, and the trade-off in tuning the hyperparameters
- ▷ The difference between local optimal and global optimal
- ▷ How to visualize the progress and result of function optimization algorithms
- ▷ The stochastic nature of optimization algorithms
- ▷ Optimization by random search or grid search
- ▷ Carrying out local optimization by pattern search, quasi-Newton, least-square, and hill climbing methods
- ▷ Carrying out global optimization using evolution algorithms and simulated annealing
- ▷ The difference in various gradient descent algorithms, including momentum, AdaGrad, RMSProp, Adadelta, and Adam; and how to use them
- ▷ How to apply optimization to common machine learning tasks

This book is not to substitute the optimization or numerical methods course in undergraduate college. The textbooks for such courses will bring you deeper theoretical understanding, but this book could complement them with practical examples. For some examples of the textbooks and other resources on optimization, see the *Further Readings* section at the end of each chapter.

How to read this book?

This book was written to be read linearly, from start to finish. However, if you are already familiar with a topic, you should be able to skip a chapter without losing track. If you want to learn a particular topic, you can also flip straight to a particular section. The content of this book is created in a guidebook format. There are substantial amount of example codes in this book. Therefore, you are expected to have this book opened on your workstation with an editor side-by-side so you can try out the examples while you read them. You can get most from the content by extending and modifying the examples.

Optimization is a very broad and deep subject. The goal of this book is to give you intuitions for the bits and pieces you need to know, and how to get things done with function optimization. This book is divided into six parts:

- ▷ **Part I: Foundation**. A gentle introduction to function optimization and its relationship with machine learning.
- ▷ **Part II: Background**. To understand what function optimization can and cannot do, and what are the pitfalls. This part also gives an overview of how various optimization algorithms fall into broad categories.
- ▷ **Part III: Local Optimization**. Discover the optimization algorithms that optimize a function based on local information.
- ▷ **Part IV: Global Optimization**. Perform function optimization by exploring the solution space. This part covers evolution algorithms and simulated annealing as a better alternatives to grid search.

- ▷ **Part V: Gradient Descent**. Introduce the common gradient descent algorithms that we may encounter in, for example, neural network models. Examples are given on how they are implemented.
- ▷ **Part VI: Projects**. Four examples are given to show how the function optimization algorithms can be used to solve a real problem.

These are not designed to tell you everything, but to give you understanding of how they work and how to use them. This is to help you learn by doing so you can get the result the fastest.

How to run the examples?

All examples in this book are in Python. The examples in each chapter are complete and standalone. You should be able to run it successfully as-is without modification, given you have installed the required packages. No special IDE or notebooks are required. A command line execution environment is all it needed in most cases. A complete working example is always given at the end of each chapter. To avoid mistakes at copy-and-paste, all source code are also provided with this book. Please use them whenever possible for a better learning experience.

All code examples were tested on a POSIX-compatible machine with Python 3.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- ▷ Books and book chapters.
- ▷ API documentation.
- ▷ Articles and Webpages.

Wherever possible, links to the relevant API documentation are provided in each lesson. Books referenced are provided with links to Amazon so you can learn more about them. If you found some good references, feel free to let us know so we can update this book.

Gradient Descent Optimization from Scratch

21

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function. It is a simple and effective technique that can be implemented with just a few lines of code. It also provides the basis for many extensions and modifications that can result in better performance. The algorithm also provides the basis for the widely used extension called stochastic gradient descent, used to train deep learning neural networks.

In this tutorial, you will discover how to implement gradient descent optimization from scratch. After completing this tutorial, you will know:

- ▷ Gradient descent is a general procedure for optimizing a differentiable objective function.
- ▷ How to implement the gradient descent algorithm from scratch in Python.
- ▷ How to apply the gradient descent algorithm to an objective function.

Let's get started.

21.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Gradient Descent
2. Gradient Descent Algorithm
3. Gradient Descent Worked Example

21.2 Gradient Descent Optimization

Gradient descent[1] is an optimization algorithm. It is technically referred to as a first-order optimization algorithm as it explicitly makes use of the first-order derivative of the target objective function.

[1] https://en.wikipedia.org/wiki/Gradient_descent

> First-order methods rely on gradient information to help direct the search for a minimum ...
>
> — Page 69, *Algorithms for Optimization*, 2019.

The first-order derivative, or simply the "derivative[2]," is the rate of change or slope of the target function at a specific point, e.g. for a specific input. If the target function takes multiple input variables, it is referred to as a multivariate function and the input variables can be thought of as a vector. In turn, the derivative of a multivariate target function may also be taken as a vector and is referred to generally as the "gradient[3]."

▷ **Gradient**: First order derivative for a multivariate objective function.

The derivative or the gradient points in the direction of the steepest ascent of the target function for an input.

> The gradient points in the direction of steepest ascent of the tangent hyperplane ...
>
> — Page 21, *Algorithms for Optimization*, 2019.

Specifically, the sign of the gradient tells you if the target function is increasing or decreasing at that point.

▷ **Positive Gradient**: Function is increasing at that point.

▷ **Negative Gradient**: Function is decreasing at that point.

Gradient descent refers to a minimization optimization algorithm that follows the negative of the gradient downhill of the target function to locate the minimum of the function. Similarly, we may refer to gradient ascent for the maximization version of the optimization algorithm that follows the gradient uphill to the maximum of the target function.

▷ **Gradient Descent**: Minimization optimization that follows the negative of the gradient to the minimum of the target function.

▷ **Gradient Ascent**: Maximization optimization that follows the gradient to the maximum of the target function.

Central to gradient descent algorithms is the idea of following the gradient of the target function. By definition, the optimization algorithm is only appropriate for target functions where the derivative function is available and can be calculated for all input values. This does not apply to all target functions, only so-called differentiable functions[4]. The main benefit of the gradient descent algorithm is that it is easy to implement and effective on a wide range of optimization problems.

> Gradient methods are simple to implement and often perform well.
>
> — Page 115, *An Introduction to Optimization*, 2001.

Gradient descent refers to a family of algorithms that use the first-order derivative to navigate to the optima (minimum or maximum) of a target function. There are many extensions to the main approach that are typically named for the feature added to the algorithm, such as gradient descent with momentum, gradient descent with adaptive gradients, and so on.

[2] https://en.wikipedia.org/wiki/Derivative
[3] https://en.wikipedia.org/wiki/Gradient
[4] https://en.wikipedia.org/wiki/Differentiable_function

Gradient descent is also the basis for the optimization algorithm used to train deep learning neural networks, referred to as stochastic gradient descent, or SGD. In this variation, the target function is an error function and the function gradient is approximated from prediction error on samples from the problem domain.

Now that we are familiar with a high-level idea of gradient descent optimization, let's look at how we might implement the algorithm.

21.3 Gradient Descent Algorithm

In this section, we will take a closer look at the gradient descent algorithm. The gradient descent algorithm requires a target function that is being optimized and the derivative function for the target function. The target function $f()$ returns a score for a given set of inputs, and the derivative function $f'()$ gives the derivative of the target function for a given set of inputs.

- ▷ **Objective Function**: Calculates a score for a given set of input parameters.
- ▷ **Derivative Function**: Calculates derivative (gradient) of the objective function for a given set of inputs.

The gradient descent algorithm requires a starting point (x) in the problem, such as a randomly selected point in the input space. The derivative is then calculated and a step is taken in the input space that is expected to result in a downhill movement in the target function, assuming we are minimizing the target function. A downhill movement is made by first calculating how far to move in the input space, calculated as the step size (called α or the learning rate) multiplied by the gradient. This is then subtracted from the current point, ensuring we move against the gradient, or down the target function.

$$x_{\text{new}} = x - \alpha \times f'(x)$$

The steeper the objective function at a given point, the larger the magnitude of the gradient, and in turn, the larger the step taken in the search space. The size of the step taken is scaled using a step size hyperparameter.

- ▷ **Step Size (α)**: Hyperparameter that controls how far to move in the search space against the gradient each iteration of the algorithm.

If the step size is too small, the movement in the search space will be small and the search will take a long time. If the step size is too large, the search may bounce around the search space and skip over the optima.

> "We have the option of either taking very small steps and re-evaluating the gradient at every step, or we can take large steps each time. The first approach results in a laborious method of reaching the minimizer, whereas the second approach may result in a more zigzag path to the minimizer."
>
> — Page 114, *An Introduction to Optimization*, 2001.

Finding a good step size may take some trial and error for the specific target function. The difficulty of choosing the step size can make finding the exact optima of the target function hard. Many extensions involve adapting the learning rate over time to take smaller steps or different sized steps in different dimensions and so on to allow the algorithm to hone in on the function

optima. The process of calculating the derivative of a point and calculating a new point in the input space is repeated until some stop condition is met. This might be a fixed number of steps or target function evaluations, a lack of improvement in target function evaluation over some number of iterations, or the identification of a flat (stationary) area of the search space signified by a gradient of zero.

▷ **Stop Condition**: Decision when to end the search procedure.

Let's look at how we might implement the gradient descent algorithm in Python. First, we can define an initial point as a randomly selected point in the input space defined by a bounds. The bounds can be defined along with an objective function as an array with a min and max value for each dimension. The `rand()`[5] NumPy function can be used to generate a vector of random numbers in the range 0 to 1.

```
...
solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
```

Program 21.1: Generate an initial point

We can then calculate the derivative of the point using a function named `derivative()`.

```
...
gradient = derivative(solution)
```

Program 21.2: Calculate gradient

And take a step in the search space to a new point down the hill of the current point. The new position is calculated using the calculated gradient and the `step_size` hyperparameter.

```
...
solution = solution - step_size * gradient
```

Program 21.3: Take a step

We can then evaluate this point and report the performance.

```
...
solution_eval = objective(solution)
```

Program 21.4: Evaluate candidate point

This process can be repeated for a fixed number of iterations controlled via an `n_iter` hyperparameter.

```
...
for i in range(n_iter):
    # calculate gradient
    gradient = derivative(solution)
    # take a step
    solution = solution - step_size * gradient
    # evaluate candidate point
```

[5] https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html

```
    solution_eval = objective(solution)
    # report progress
    print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
```

Program 21.5: Run the gradient descent

We can tie all of this together into a function named `gradient_descent()`. The function takes the name of the objective and gradient functions, as well as the bounds on the inputs to the objective function, number of iterations and step size, then returns the solution and its evaluation at the end of the search. The complete gradient descent optimization algorithm implemented as a function is listed below.

```
def gradient_descent(objective, derivative, bounds, n_iter, step_size):
    # 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)
        # report progress
        print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]
```

Program 21.6: Gradient descent algorithm

Now that we are familiar with the gradient descent algorithm, let's look at a worked example.

21.4 Gradient Descent Worked Example

In this section, we will work through an example of applying gradient descent to a simple test optimization function. First, let's define an optimization function. We will use a simple one-dimensional function that squares the input and defines the range of valid inputs from -1.0 to 1.0.

The `objective()` function below implements this function.

```
def objective(x):
    return x**2.0
```

Program 21.7: Objective function

We can then sample all inputs in the range and calculate the objective function value for each.

```
...
# 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 = objective(inputs)
```

Program 21.8: Compute the objective function for all inputs in range

Finally, we can create a line plot of the inputs (x-axis) versus the objective function values (y-axis) to get an intuition for the shape of the objective function that we will be searching.

```
...
# create a line plot of input vs result
pyplot.plot(inputs, results)
# show the plot
pyplot.show()
```

Program 21.9: Plot the objective function input and result

The example below ties this together and provides an example of plotting the one-dimensional test function.

```
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(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 = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
# show the plot
pyplot.show()
```

Program 21.10: Plot of simple function

Running the example creates a line plot of the inputs to the function (x-axis) and the calculated output of the function (y-axis). We can see the familiar U-shaped called a parabola.

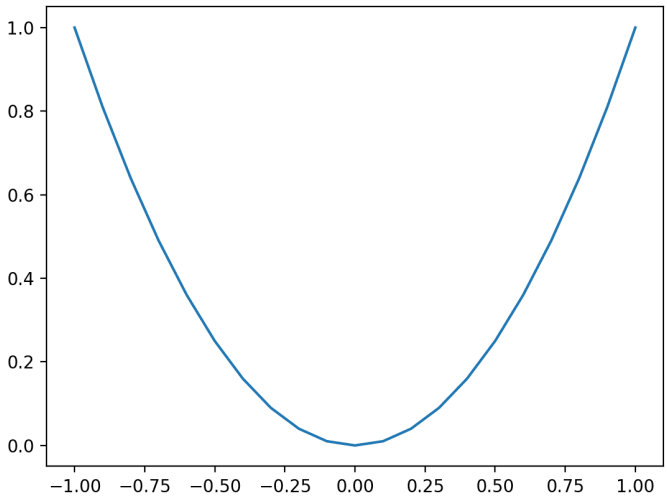


Figure 21.1: Line Plot of Simple One-Dimensional Function

Next, we can apply the gradient descent algorithm to the problem. First, we need a function that calculates the derivative for this function. The derivative of x^2 is $2x$ and the `derivative()` function implements this below.

```
def derivative(x):
    return x * 2.0
```

Program 21.11: Derivative of objective function

We can then define the bounds of the objective function, the step size, and the number of iterations for the algorithm. We will use a step size of 0.1 and 30 iterations, both found after a little experimentation.

```
...
# define range for input
bounds = asarray([[-1.0, 1.0]])
# define the total iterations
n_iter = 30
# define the maximum step size
step_size = 0.1
# perform the gradient descent search
best, score = gradient_descent(objective, derivative, bounds, n_iter, step_size)
```

Program 21.12: Perform gradient descent search

Tying this together, the complete example of applying gradient descent optimization to our one-dimensional test function is listed below.

```
from numpy import asarray
from numpy.random import rand

# objective function
```

```
def objective(x):
    return x**2.0

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

# gradient descent algorithm
def gradient_descent(objective, derivative, bounds, n_iter, step_size):
    # 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)
        # report progress
        print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]

# 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
# perform the gradient descent search
best, score = gradient_descent(objective, derivative, bounds, n_iter, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 21.13: Example of gradient descent for a one-dimensional function

Running the example starts with a random point in the search space then applies the gradient descent algorithm, reporting performance along the way.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that the algorithm finds a good solution after about 20–30 iterations with a function evaluation of about 0.0. Note the optima for this function is at $f(0.0) = 0.0$.

```
>0 f([-0.36308639]) = 0.13183
>1 f([-0.29046911]) = 0.08437
>2 f([-0.23237529]) = 0.05400
>3 f([-0.18590023]) = 0.03456
>4 f([-0.14872018]) = 0.02212
>5 f([-0.11897615]) = 0.01416
```

```
>6 f([-0.09518092]) = 0.00906
>7 f([-0.07614473]) = 0.00580
>8 f([-0.06091579]) = 0.00371
>9 f([-0.04873263]) = 0.00237
>10 f([-0.0389861]) = 0.00152
>11 f([-0.03118888]) = 0.00097
>12 f([-0.02495111]) = 0.00062
>13 f([-0.01996089]) = 0.00040
>14 f([-0.01596871]) = 0.00025
>15 f([-0.01277497]) = 0.00016
>16 f([-0.01021997]) = 0.00010
>17 f([-0.00817598]) = 0.00007
>18 f([-0.00654078]) = 0.00004
>19 f([-0.00523263]) = 0.00003
>20 f([-0.0041861]) = 0.00002
>21 f([-0.00334888]) = 0.00001
>22 f([-0.0026791]) = 0.00001
>23 f([-0.00214328]) = 0.00000
>24 f([-0.00171463]) = 0.00000
>25 f([-0.0013717]) = 0.00000
>26 f([-0.00109736]) = 0.00000
>27 f([-0.00087789]) = 0.00000
>28 f([-0.00070231]) = 0.00000
>29 f([-0.00056185]) = 0.00000
Done!
f([-0.00056185]) = 0.000000
```

Output 21.1: Result from Program 21.13

Now, let's get a feeling for the importance of good step size. Set the step size to a large value, such as 1.0, and re-run the search.

```
...
step_size = 1.0
```

Program 21.14: Define a larger step size

Run the example with the larger step size and inspect the results.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

We can see that the search does not find the optima, and instead bounces around the domain, in this case between the values 0.64820935 and −0.64820935.

```
...
>25 f([0.64820935]) = 0.42018
>26 f([-0.64820935]) = 0.42018
>27 f([0.64820935]) = 0.42018
>28 f([-0.64820935]) = 0.42018
>29 f([0.64820935]) = 0.42018
```

```
Done!
f([0.64820935]) = 0.420175
```

Output 21.2: Result from Program 21.13 with a larger step size

Now, try a much smaller step size, such as `1e-5`.

```
...
step_size = 1e-5
```

Program 21.15: Define a smaller step size

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Re-running the search, we can see that the algorithm moves very slowly down the slope of the objective function from the starting point.

```
...
>25 f([-0.87315153]) = 0.76239
>26 f([-0.87313407]) = 0.76236
>27 f([-0.8731166]) = 0.76233
>28 f([-0.87309914]) = 0.76230
>29 f([-0.87308168]) = 0.76227
Done!
f([-0.87308168]) = 0.762272
```

Output 21.3: Result from Program 21.13 with a smaller step size

These two quick examples highlight the problems in selecting a step size that is too large or too small and the general importance of testing many different step size values for a given objective function. Finally, we can change the learning rate back to 0.1 and visualize the progress of the search on a plot of the target function. First, we can update the `gradient_descent()` function to store all solutions and their score found during the optimization as lists and return them at the end of the search instead of the best solution found.

```
def gradient_descent(objective, 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]
```

Program 21.16: Gradient descent algorithm with the scores stored

The function can be called, and we can get the lists of the solutions and their scores found during the search.

```
...
solutions, scores = gradient_descent(objective, derivative, bounds, n_iter,
↪   step_size)
```

Program 21.17: Perform the gradient descent search and retrieve the scores

We can create a line plot of the objective function, as before.

```
...
# sample input range uniformly at 0.1 increments
inputs = arange(bounds[0,0], bounds[0,1]+0.1, 0.1)
# compute targets
results = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
```

Program 21.18: Plot the objective function

Finally, we can plot each solution found as a red dot and connect the dots with a line so we can see how the search moved downhill.

```
...
pyplot.plot(solutions, scores, '.-', color='red')
```

Program 21.19: Plot the solutions found on the objective function

Tying this all together, the complete example of plotting the result of the gradient descent search on the one-dimensional test function is listed below.

```
from numpy import asarray
from numpy import arange
from numpy.random import rand
from matplotlib import pyplot

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

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

```
# gradient descent algorithm
def gradient_descent(objective, 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]

# 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
# perform the gradient descent search
solutions, scores = gradient_descent(objective, derivative, bounds, n_iter,
    step_size)
# sample input range uniformly at 0.1 increments
inputs = arange(bounds[0,0], bounds[0,1]+0.1, 0.1)
# compute targets
results = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
# plot the solutions found
pyplot.plot(solutions, scores, '.-', color='red')
# show the plot
pyplot.show()
```

Program 21.20: Example of plotting a gradient descent search on a one-dimensional function

Running the example performs the gradient descent search on the objective function as before, except in this case, each point found during the search is plotted.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that the search started about halfway up the left part of the function and stepped downhill to the bottom of the basin. We can see that in the parts of the objective

function with the larger curve, the derivative (gradient) is larger, and in turn, larger steps are taken. Similarly, the gradient is smaller as we get closer to the optima, and in turn, smaller steps are taken. This highlights that the step size is used as a scale factor on the magnitude of the gradient (curvature) of the objective function.

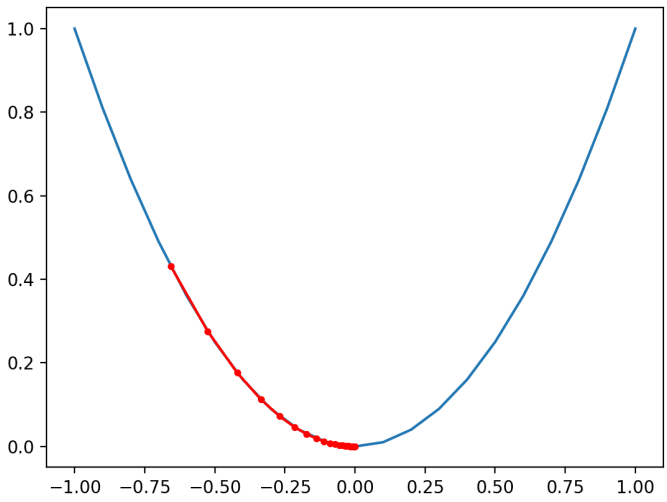


Figure 21.2: Plot of the Progress of Gradient Descent on a One Dimensional Objective Function

21.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J

Edwin K. P. Chong and Stanislaw H. Zak. *An Introduction to Optimization*. Wiley-Blackwell, 2001.
https://amzn.to/37S9WVs

APIs

`numpy.random.rand` *API*.
https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html

`numpy.asarray` *API*.
https://numpy.org/doc/stable/reference/generated/numpy.asarray.html

Matplotlib API.
https://matplotlib.org/api/pyplot_api.html

Articles

Gradient descent. Wikipedia.
https://en.wikipedia.org/wiki/Gradient_descent
Gradient. Wikipedia.
https://en.wikipedia.org/wiki/Gradient
Derivative. Wikipedia.
https://en.wikipedia.org/wiki/Derivative
Differentiable function. Wikipedia.
https://en.wikipedia.org/wiki/Differentiable_function

21.6 Summary

In this tutorial, you discovered how to implement gradient descent optimization from scratch. Specifically, you learned:

- ▷ Gradient descent is a general procedure for optimizing a differentiable objective function.
- ▷ How to implement the gradient descent algorithm from scratch in Python.
- ▷ How to apply the gradient descent algorithm to an objective function.

Next, we will learn about a strategy that can improve gradient descent.

This is Just a Sample

Thank-you for your interest in **Optimization for Machine Learning**.

This is just a sample of the full text. You can purchase the complete book online from: https://machinelearningmastery.com/optimization-for-machine-learning/