MACHINE LEARNING MASTERY

Optimization for Machine Learning

Finding Function Optima with Python

Jason Brownlee

Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Credits

Editor: Adrian Tam
Technical reviewers: Andrei Cheremskoy and Arun Koshy

Copyright

Contents

III Local Optimization

V   Gradient Descent   215

21 Gradient Descent Optimization from Scratch   216

22 Gradient Descent with Momentum   230

23 Gradient Descent with AdaGrad   248

24 Gradient Descent with RMSProp   264

25 Gradient Descent with Adadelta   281

VII Appendix

A Getting Help

B How to Setup Your Python Environment

How Far You Have Come

Preface

This book is to help machine learning practitioners to understand the optimization algorithms that we regularly encounter.

All machine learning models involve optimization. From the simplest model, such as linear regression, we assumed the data is in a linear relationship and then we work toward the values of the coefficients such that the difference between model prediction and the input data is minimized. Such examples are everywhere. However, what distinguishes the optimization in mathematics and the optimization in machine learning is that we almost always have to resort to numerical and computational methods, rather than algebraic methods, to optimize. While algebraic methods may give unique close-form solutions, there are numerous computational algorithms for optimization and all give approximate solutions in various degree. Not a single algorithm can be a silver bullet to all problems.

Therefore, as a machine learning practitioner, we need to know what the different optimization algorithms are about, as well as their strengths and weaknesses. We need to know that so we can pick the most suitable one for a particular problem. Likewise, we need to know what we can fine-tune in case the machine learning result needs improvement, or at least draw some conclusions on the nature of the problem if the machine learning result is not as good as we expect. To get the result first, you should probably try out some machine learning projects instead of reading this book. But once you finish a few projects and wonder what the computer was doing in "training" and why it took some noticeable time to give you the trained model, this book will give you some clues.

Just like any topic in the theoretical side of machine learning, optimization can be a very deep and broad subject. It is important to know a bit of everything so that it makes sense to you when you read the API documentation or other people's work. This book does not aim to be a comprehensive guide. Indeed, most practitioners do not need to, for example, evaluate the numerical stability of different optimization algorithm, nor know how to find the optima in a nonlinear function mathematically. If you wish to, you can go deeper with the more rigorous academic titles. The goal of this book is to provide you an overview and show you how the different optimization algorithms can be applied.

Following the top-down approach of our other books, we let you see how we can optimize a function of our choice with the different algorithm. We use Python in the examples. Hence you may reuse the code by simply replacing the objective function with your own, and help

you solve your own problems. This book is put together with the hope that, you will not see gradient descent and the names of other algorithms as magic that only appear in the machine learning libraries. Rather, you can find them as generic methods to find the optimum value of a numerical function, and it just happened that helping machine learning models to train is an example of such.

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.

Foundations I

What is Function Optimization? **1**

Function optimization is a foundational area of study and the techniques are used in almost every quantitative field. Importantly, function optimization is central to almost all machine learning algorithms, and predictive modeling projects. As such, it is critical to understand what function optimization is, the terminology used in the field, and the elements that constitute a function optimization problem. In this tutorial, you will discover a gentle introduction to function optimization. After completing this tutorial, you will know:

▷ The three elements of function optimization as candidate solutions, objective functions, and cost.

▷ The conceptualization of function optimization as navigating a search space and response surface.

▷ The difference between global optima and local optima when solving a function optimization problem.

Let's get started.

1.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Function Optimization
2. Candidate Solutions
3. Objective Functions
4. Evaluation Costs

1.2 Function Optimization

Function optimization is a subfield of mathematics, and in modern times is addressed using numerical computing methods. *Continuous function optimization* ("*function optimization*" here for short) belongs to a broader field of study called mathematical optimization. It is distinct from other types of optimization as it involves finding optimal candidate solutions composed of numeric input variables, as opposed to candidate solutions composed of sequences

or combinations (e.g. combinatorial optimization). Function optimization is a widely used tool bag of techniques employed in practically all scientific and engineering disciplines.

> *People optimize. Investors seek to create portfolios that avoid excessive risk while achieving a high rate of return. [...] Optimization is an important tool in decision science and in the analysis of physical systems.*
>
> — Page 2, *Numerical Optimization*, 2006.

It plays a central role in machine learning, as almost all machine learning algorithms use function optimization to fit a model to a training dataset. For example, fitting a line to a collection of points requires solving an optimization problem. As does fitting a linear regression or a neural network model on a training dataset. In this way, optimization provides a tool to adapt a general model to a specific situation. Learning is treated as an optimization or search problem. Practically, function optimization describes a class of problems for finding the input to a given function that results in the minimum or maximum output from the function.

> *The objective depends on certain characteristics of the system, called variables or unknowns. Our goal is to find values of the variables that optimize the objective.*
>
> — Page 2, *Numerical Optimization*, 2006.

Function Optimization involves three elements: the input to the function (e.g. x), the objective function itself (e.g. $f(x)$) and the output from the function (e.g. *cost, y*).

▷ **Input** x: The input to the function to be evaluated, i.e. a candidate solution.

▷ **Function** $f()$: The objective function or target function that evaluates inputs.

▷ **Cost** y: The result of evaluating a candidate solution with the objective function, minimized or maximized.

Let's take a closer look at each element in turn.

1.3   Candidate Solutions

A candidate solution is a single input to the objective function. The form of a candidate solution depends on the specifics of the objective function. It may be a single floating point number, a vector of numbers, a matrix of numbers, or as complex as needed for the specific problem domain. Most commonly, vectors of numbers. For a test problem, the vector represents the specific values of each input variable to the function ($x = [x_1, x_2, x_3, \ldots, x_n]$). For a machine learning model, the vector may represent model coefficients or weights.

> *Mathematically speaking, optimization is the minimization or maximization of a function subject to constraints on its variables.*
>
> — Page 2, *Numerical Optimization*, 2006.

There may be constraints imposed by the problem domain or the objective function on the candidate solutions. This might include aspects such as:

▷ The number of variables (1, 20, 1,000,000, etc.)

▷ The data type of variables (integer, binary, real-valued, etc.)

▷ The range of accepted values (between 0 and 1, etc.)

Importantly, candidate solutions are discrete and there are many of them. The universe of candidate solutions may be vast, too large to enumerate. Instead, the best we can do is sample candidate solutions in the search space. As a practitioner, we seek an optimization algorithm that makes the best use of the information available about the problem to effectively sample the search space and locate a good or best candidate solution.

▷ **Search Space**: Universe of candidate solutions defined by the number, type, and range of accepted inputs to the objective function.

Finally, candidate solutions can be rank-ordered based on their evaluation by the objective function, meaning that some are better than others.

1.4   Objective Functions

The objective function is specific to the problem domain. It may be a test function, e.g. a well-known equation with a specific number of input variables, the calculation of which returns the cost of the input. The optima of test functions are known, allowing algorithms to be compared based on their ability to navigate the search space efficiently.

In machine learning, the objective function may involve plugging the candidate solution into a model and evaluating it against a portion of the training dataset, and the cost may be an error score, often called the loss of the model. The objective function is easy to define, although expensive to evaluate. Efficiency in function optimization refers to minimizing the total number of function evaluations.

Although the objective function is easy to define, it may be challenging to optimize. The difficulty of an objective function may range from being able to analytically solve the function directly using calculus or linear algebra (easy), to using a local search algorithm (moderate), to using a global search algorithm (hard). The difficulty of an objective function is based on how much is known about the function. This often cannot be determined by simply reviewing the equation or code for evaluating candidate solutions. Instead, it refers to the structure of the response surface. The response surface (or search landscape) is the geometrical structure of the cost in relation to the search space of candidate solutions. For example, a smooth response surface suggests that small changes to the input (candidate solutions) result in small changes to the output (cost) from the objective function.

▷ **Response Surface**: Geometrical properties of the cost from the objective function in response to changes to the candidate solutions.

The response surface can be visualized in low dimensions, e.g. for candidate solutions with one or two input variables. A one-dimensional input can be plotted as a 2D scatter plot with input values on the x-axis and the cost on the y-axis. A two-dimensional input can be plotted as a 3D surface plot with input variables on the x- and y-axis, and the height of the surface representing the cost.

In a minimization problem, poor solutions would be represented as hills in the response surface and good solutions would be represented by valleys. This would be inverted for maximizing problems. The structure and shape of this response surface determine the difficulty an algorithm will have in navigating the search space to a solution. The complexity of real objective functions means we cannot analyze the surface analytically, and the high dimensionality

of the inputs and computational cost of function evaluations makes mapping and plotting real objective functions intractable.

1.5   Evaluation Costs

The cost of a candidate solution is almost always a single real-valued number. The scale of the cost values will vary depending on the specifics of the objective function. In general, the only meaningful comparison of cost values is to other cost values calculated by the same objective function. The minimum or maximum output from the function is called the optima of the function, typically simplified to simply the minimum. Any function we wish to maximize can be converted to minimizing by adding a negative sign to the front of the cost returned from the function.

> In global optimization, the true global solution of the optimization problem is found; the compromise is efficiency. The worst-case complexity of global optimization methods grows exponentially with the problem sizes . . .
>
> — Page 10, *Convex Optimization*, 2004.

An objective function may have a single best solution, referred to as the global optimum of the objective function. Alternatively, the objective function may have many global optima, in which case we may be interested in locating one or all of them.

> Many numerical optimization methods seek local minima. Local minima are locally optimal, but we do not generally know whether a local minimum is a global minimum.
>
> — Page 8, *Algorithms for Optimization*, 2019.

In addition to a global optima, a function may have local optima, which are good candidate solutions that may be relatively easy to locate, but not as good as the global optima. Local optima may appear to be global optima to a search algorithm, e.g. may be in a valley of the response surface, in which case we might refer to them as deceptive as the algorithm will easily locate them and get stuck, failing to locate the global optima.

 ▷ **Global Optima**: The candidate solution with the best cost from the objective function.

 ▷ **Local Optima**: Candidate solutions are good but not as good as the global optima.

The relative nature of cost values means that a baseline in performance on challenging problems can be established using a naive search algorithm (e.g. random) and "goodness" of optimal solutions found by more sophisticated search algorithms can be compared relative to the baseline. Candidate solutions are often very simple to describe and very easy to construct. The challenging part of function optimization is evaluating candidate solutions.

Solving a function optimization problem or objective function refers to finding the optima. The whole goal of the project is to locate a specific candidate solution with a good or best cost, give the time and resources available. In simple and moderate problems, we may be able to locate the optimal candidate solution exactly and have some confidence that we have done so.

> Many algorithms for nonlinear optimization problems seek only a local solution, a point at which the objective function is smaller than at all other feasible nearby

points. They do not always find the global solution, which is the point with lowest function value among all feasible points. Global solutions are needed in some applications, but for many problems they are difficult to recognize and even more difficult to locate.

"

— Page 6, *Algorithms for Optimization*, 2019.

On more challenging problems, we may be happy with a relatively good candidate solution (i.e. good enough) given the time available for the project.

1.6 Further Reading

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

Books

Jorge Nocedal and Stephen Wright. *Numerical Optimization*. 2nd ed. Springer, 2006.
https://amzn.to/3sbjF2t
Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge, 2004.
https://amzn.to/34mvCr1
Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J

Articles

Mathematical optimization. Wikipedia.
https://en.wikipedia.org/wiki/Mathematical_optimization

1.7 Summary

In this tutorial, you discovered a gentle introduction to function optimization. Specifically, you learned:

▷ The three elements of function optimization as candidate solutions, objective functions and cost.

▷ The conceptualization of function optimization as navigating a search space and response surface.

▷ The difference between global optima and local optima when solving a function optimization problem.

Next, you will see how function optimization can help machine learning.

Optimization and Machine Learning

2

Machine learning involves using an algorithm to learn and generalize from historical data in order to make predictions on new data. This problem can be described as approximating a function that maps examples of inputs to examples of outputs. Approximating a function can be solved by framing the problem as function optimization. This is where a machine learning algorithm defines a parameterized mapping function (e.g. a weighted sum of inputs) and an optimization algorithm is used to fund the values of the parameters (e.g. model coefficients) that minimize the error of the function when used to map inputs to outputs. This means that each time we fit a machine learning algorithm on a training dataset, we are solving an optimization problem. In this tutorial, you will discover the central role of optimization in machine learning.

After completing this tutorial, you will know:

▷ Machine learning algorithms perform function approximation, which is solved using function optimization.

▷ Function optimization is the reason why we minimize error, cost, or loss when fitting a machine learning algorithm.

▷ Optimization is also performed during data preparation, hyperparameter tuning, and model selection in a predictive modeling project.

Let's get started.

2.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Machine Learning and Optimization

2. Learning as Optimization

3. Optimization in a Machine Learning Project

    (a) Data Preparation as Optimization

    (b) Hyperparameter Tuning as Optimization

    (c) Model Selection as Optimization

2.2   Machine Learning and Optimization

Function optimization is the problem of finding the set of inputs to a target objective function that result in the minimum or maximum of the function. It can be a challenging problem as the function may have tens, hundreds, thousands, or even millions of inputs, and the structure of the function is unknown, and often non-differentiable and noisy.

 ▷ **Function Optimization**: Find the set of inputs that results in the minimum or maximum of an objective function.

Machine learning can be described as *function approximation.* That is, approximating the unknown underlying function that maps examples of inputs to outputs in order to make predictions on new data. It can be challenging as there is often a limited number of examples from which we can approximate the function, and the structure of the function that is being approximated is often nonlinear, noisy, and may even contain contradictions.

 ▷ **Function Approximation**: Generalize from specific examples to a reusable mapping function for making predictions on new examples.

Function optimization is often simpler than function approximation. Importantly, in machine learning, we often solve the problem of function approximation using function optimization. At the core of nearly all machine learning algorithms is an optimization algorithm. In addition, the process of working through a predictive modeling problem involves optimization at multiple steps in addition to learning a model, including:

 ▷ Choosing the hyperparameters of a model.

 ▷ Choosing the transforms to apply to the data prior to modeling

 ▷ Choosing the modeling pipeline to use as the final model.

Now that we know that optimization plays a central role in machine learning, let's look at some examples of learning algorithms and how they use optimization.

2.3   Learning as Optimization

Predictive modeling problems involve making a prediction from an example of input. A numeric quantity must be predicted in the case of a regression problem, whereas a class label must be predicted in the case of a classification problem. The problem of predictive modeling is sufficiently challenging that we cannot write code to make predictions. Instead, we must use a learning algorithm applied to historical data to learn a "*program*" called a predictive model that we can use to make predictions on new data.

In statistical learning, a statistical perspective on machine learning, the problem is framed as the learning of a mapping function (f) given examples of input data (X) and associated output data (y).

$$y = f(X)$$

Given new examples of input ($\hat{X}$), we must map each example onto the expected output value ($\hat{y}$) using our learned function ($\hat{f}$).

$$\hat{y} = \hat{f}(\hat{X})$$

The learned mapping will be imperfect. No model is perfect, and some prediction error is expected given the difficulty of the problem, noise in the observed data, and the choice of learning algorithm. Mathematically, learning algorithms solve the problem of approximating the mapping function by solving a function optimization problem. Specifically, given examples of inputs and outputs, find the set of inputs to the mapping function that results in the minimum loss, minimum cost, or minimum prediction error. The more biased or constrained the choice of mapping function, the easier the optimization is to solve.

Let's look at some examples to make this clear.

A linear regression (for regression problems) is a highly constrained model and can be solved analytically using linear algebra. The inputs to the mapping function are the coefficients of the model. We can use an optimization algorithm, like a quasi-Newton local search algorithm, but it will almost always be less efficient than the analytical solution.

▷ **Linear Regression**: Function inputs are model coefficients, optimization problems that can be solved analytically.

A logistic regression (for classification problems) is slightly less constrained and must be solved as an optimization problem, although something about the structure of the optimization function being solved is known given the constraints imposed by the model. This means a local search algorithm like a quasi-Newton method can be used. We could use a global search like stochastic gradient descent, but it will almost always be less efficient.

▷ **Logistic Regression**: Function inputs are model coefficients, optimization problems that require an iterative local search algorithm.

A neural network model is a very flexible learning algorithm that imposes few constraints. The inputs to the mapping function are the network weights. A local search algorithm cannot be used given the search space is multimodal and highly nonlinear; instead, a global search algorithm must be used.

A global optimization algorithm is commonly used, specifically stochastic gradient descent, and the updates are made in a way that is aware of the structure of the model (backpropagation and the chain rule). We could use a global search algorithm that is oblivious of the structure of the model, like a genetic algorithm, but it will almost always be less efficient.

▷ **Neural Network**: Function inputs are model weights, optimization problems that require an iterative global search algorithm.

We can see that each algorithm makes different assumptions about the form of the mapping function, which influences the type of optimization problem to be solved. We can also see that the default optimization algorithm used for each machine learning algorithm is not arbitrary; it represents the most efficient algorithm for solving the specific optimization problem framed by the algorithm, e.g. stochastic gradient descent for neural nets instead of a genetic algorithm. Deviating from these defaults requires a good reason.

Not all machine learning algorithms solve an optimization problem. A notable example is the k-nearest neighbors algorithm that stores the training dataset and does a lookup for the k best matches to each new example in order to make a prediction.

Now that we are familiar with learning in machine learning algorithms as optimization, let's look at some related examples of optimization in a machine learning project.

2.4 Optimization in a Machine Learning Project

Optimization plays an important part in a machine learning project in addition to fitting the learning algorithm on the training dataset. The step of preparing the data prior to fitting the model and the step of tuning a chosen model also can be framed as an optimization problem. In fact, an entire predictive modeling project can be thought of as one large optimization problem.

Let's take a closer look at each of these cases in turn.

2.4.1 Data Preparation as Optimization

Data preparation involves transforming raw data into a form that is most appropriate for the learning algorithms. This might involve scaling values, handling missing values, and changing the probability distribution of variables. Transforms can be made to change representation of the historical data to meet the expectations or requirements of specific learning algorithms. Yet, sometimes good or best results can be achieved when the expectations are violated or when an unrelated transform to the data is performed. We can think of choosing transforms to apply to the training data as a search or optimization problem of best exposing the unknown underlying structure of the data to the learning algorithm.

 ▷ **Data Preparation**: Function inputs are sequences of transforms, optimization problems that require an iterative global search algorithm.

This optimization problem is often performed manually with human-based trial and error. Nevertheless, it is possible to automate this task using a global optimization algorithm where the inputs to the function are the types and order of transforms applied to the training data. The number and permutations of data transforms are typically quite limited and it may be possible to perform an exhaustive search or a grid search of commonly used sequences.

2.4.2 Hyperparameter Tuning as Optimization

Machine learning algorithms have hyperparameters that can be configured to tailor the algorithm to a specific dataset. Although the dynamics of many hyperparameters are known, the specific effect they will have on the performance of the resulting model on a given dataset is not known. As such, it is a standard practice to test a suite of values for key algorithm hyperparameters for a chosen machine learning algorithm. This is called *hyperparameter tuning* or *hyperparameter optimization*. It is common to use a naive optimization algorithm for this purpose, such as a random search algorithm or a grid search algorithm.

 ▷ **Hyperparameter Tuning**: Function inputs are algorithm hyperparameters, optimization problems that require an iterative global search algorithm.

Nevertheless, it is becoming increasingly common to use an iterative global search algorithm for this optimization problem. A popular choice is a Bayesian optimization algorithm that is capable of simultaneously approximating the target function that is being optimized (using a surrogate function) while optimizing it.

This is desirable as evaluating a single combination of model hyperparameters is expensive, requiring fitting the model on the entire training dataset one or many times, depending on the choice of model evaluation procedure (e.g. repeated k-fold cross-validation).

2.4.3 Model Selection as Optimization

Model selection involves choosing one from among many candidate machine learning models for a predictive modeling problem. Really, it involves choosing the machine learning algorithm or machine learning pipeline that produces a model. This is then used to train a final model that can then be used in the desired application to make predictions on new data. This process of model selection is often a manual process performed by a machine learning practitioner involving tasks such as preparing data, evaluating candidate models, tuning well-performing models, and finally choosing the final model. This can be framed as an optimization problem that subsumes part of or the entire predictive modeling project.

> ▷ **Model Selection**: Function inputs are data transform, machine learning algorithm, and algorithm hyperparameters; optimization problem that requires an iterative global search algorithm.

Increasingly, this is the case with automated machine learning (AutoML) algorithms being used to choose an algorithm, an algorithm and hyperparameters, or data preparation, algorithm and hyperparameters, with very little user intervention.

Like hyperparameter tuning, it is common to use a global search algorithm that also approximates the objective function, such as Bayesian optimization, given that each function evaluation is expensive. This automated optimization approach to machine learning also underlies modern machine learning as a service (MLaaS) products provided by companies such as Google, Microsoft, and Amazon. Although fast and efficient, such approaches are still unable to outperform hand-crafted models prepared by highly skilled experts, such as those participating in machine learning competitions.

2.5   Further Reading

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

Articles

Mathematical optimization. Wikipedia.
   https://en.wikipedia.org/wiki/Mathematical_optimization
Function approximation. Wikipedia.
   https://en.wikipedia.org/wiki/Function_approximation
Least-squares function approximation. Wikipedia.
   https://en.wikipedia.org/wiki/Least-squares_function_approximation
Hyperparameter optimization. Wikipedia.
   https://en.wikipedia.org/wiki/Hyperparameter_optimization
Model selection. Wikipedia.
   https://en.wikipedia.org/wiki/Model_selection

2.6   Summary

In this tutorial, you discovered the central role of optimization in machine learning. Specifically, you learned:

▷ Machine learning algorithms perform function approximation, which is solved using function optimization.

▷ Function optimization is the reason why we minimize error, cost, or loss when fitting a machine learning algorithm.

▷ Optimization is also performed during data preparation, hyperparameter tuning, and model selection in a predictive modeling project.

Next, you will be introduced to the different optimization algorithms.

How to Choose an Optimization Algorithms

3

Optimization is the problem of finding a set of inputs to an objective function that results in a maximum or minimum function evaluation. It is the challenging problem that underlies many machine learning algorithms, from fitting logistic regression models to training artificial neural networks. There are perhaps hundreds of popular optimization algorithms, and perhaps tens of algorithms to choose from in popular scientific code libraries. This can make it challenging to know which algorithms to consider for a given optimization problem.

In this tutorial, you will discover a guided tour of different optimization algorithms. After completing this tutorial, you will know:

▷ Optimization algorithms may be grouped into those that use derivatives and those that do not.

▷ Classical algorithms use the first and sometimes second derivative of the objective function.

▷ Direct search and stochastic algorithms are designed for objective functions where function derivatives are unavailable.

Let's get started.

3.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Optimization Algorithms
2. Differentiable Objective Function
3. Non-Differential Objective Function

3.2   Optimization Algorithms

Optimization refers to a procedure for finding the input parameters or arguments to a function that result in the minimum or maximum output of the function. The most common type of optimization problems encountered in machine learning are *continuous function optimization*, where the input arguments to the function are real-valued numeric values, e.g. floating point

values. The output from the function is also a real-valued evaluation of the input values. We might refer to problems of this type as continuous function optimization, to distinguish from functions that take discrete variables and are referred to as combinatorial optimization problems.

There are many different types of optimization algorithms that can be used for continuous function optimization problems, and perhaps just as many ways to group and summarize them. One approach to grouping optimization algorithms is based on the amount of information available about the target function that is being optimized that, in turn, can be used and harnessed by the optimization algorithm. Generally, the more information that is available about the target function, the easier the function is to optimize if the information can effectively be used in the search.

Perhaps the major division in optimization algorithms is whether the objective function can be differentiated at a point or not. That is, whether the first derivative (gradient or slope) of the function can be calculated for a given candidate solution or not. This partitions algorithms into those that can make use of the calculated gradient information and those that do not.

▷ Differentiable Target Function?

  ○ Algorithms that use derivative information.

  ○ Algorithms that do not use derivative information.

We will use this as the major division for grouping optimization algorithms in this tutorial and look at algorithms for differentiable and non-differentiable objective functions.

Note: this is not an exhaustive coverage of algorithms for continuous function optimization, although it does cover the major methods that you are likely to encounter as a regular practitioner.

3.3 Differentiable Objective Function

A *differentiable function*[1] is a function where the derivative can be calculated for any given point in the input space. The derivative of a function for a value is the rate or amount of change in the function at that point. It is often called the slope.

▷ **First-Order Derivative**: Slope or rate of change of an objective function at a given point.

The derivative of the function with more than one input variable (e.g. multivariate inputs) is commonly referred to as the gradient.

▷ **Gradient**: Derivative of a multivariate continuous objective function.

A derivative for a multivariate objective function is a vector, and each element in the vector is called a partial derivative, or the rate of change for a given variable at the point assuming all other variables are held constant.

▷ **Partial Derivative**: Element of a derivative of a multivariate objective function.

---

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

We can calculate the derivative of the derivative of the objective function, that is the rate of change of the rate of change in the objective function. This is called the second derivative.

▷ **Second-Order Derivative**: Rate at which the derivative of the objective function changes.

For a function that takes multiple input variables, this is a matrix and is referred to as the Hessian matrix.

▷ **Hessian matrix**: Second derivative of a function with two or more input variables.

Simple differentiable functions can be optimized analytically using calculus. Typically, the objective functions that we are interested in cannot be solved analytically. Optimization is significantly easier if the gradient of the objective function can be calculated, and as such, there has been a lot more research into optimization algorithms that use the derivative than those that do not. Some groups of algorithms that use gradient information include:

▷ Bracketing Algorithms

▷ Local Descent Algorithms

▷ First-Order Algorithms

▷ Second-Order Algorithms

> **Note:** this taxonomy is inspired by the 2019 book *Algorithms for Optimization*. (https://amzn.to/39KZSQn)

Let's take a closer look at each in turn.

3.3.1 Bracketing Algorithms

Bracketing optimization algorithms are intended for optimization problems with one input variable where the optima is known to exist within a specific range. Bracketing algorithms are able to efficiently navigate the known range and locate the optima, although they assume only a single optima is present (referred to as unimodal objective functions). Some bracketing algorithms may be able to be used without derivative information if it is not available.

Examples of bracketing algorithms include:

▷ Fibonacci Search

▷ Golden Section Search

▷ Bisection Method

3.3.2 Local Descent Algorithms

Local descent optimization algorithms are intended for optimization problems with more than one input variable and a single global optima (e.g. unimodal objective function). Perhaps the most common example of a local descent algorithm is the line search algorithm.

▷ Line Search

There are many variations of the line search (e.g. the Brent-Dekker algorithm), but the procedure generally involves choosing a direction to move in the search space, then performing a bracketing

type search in a line or hyperplane in the chosen direction. This process is repeated until no further improvements can be made. The limitation is that it is computationally expensive to optimize each directional move in the search space.

3.3.3 First-Order Algorithms

First-order optimization algorithms explicitly involve using the first derivative (gradient) to choose the direction to move in the search space. The procedures involve first calculating the gradient of the function, then following the gradient in the opposite direction (e.g. downhill to the minimum for minimization problems) using a step size (also called the learning rate). The step size is a hyperparameter that controls how far to move in the search space, unlike "local descent algorithms" that perform a full line search for each directional move. A step size that is too small results in a search that takes a long time and can get stuck, whereas a step size that is too large will result in zig-zagging or bouncing around the search space, missing the optima completely. First-order algorithms are generally referred to as gradient descent, with more specific names referring to minor extensions to the procedure, e.g.:

▷ Gradient Descent

▷ Momentum

▷ Adagrad

▷ RMSProp

▷ Adam

The gradient descent algorithm also provides the template for the popular stochastic version of the algorithm, named Stochastic Gradient Descent (SGD) that is used to train artificial neural networks (deep learning) models. The important difference is that the gradient is appropriated rather than calculated directly, using prediction error on training data, such as one sample (stochastic), all examples (batch), or a small subset of training data (minibatch). The extensions designed to accelerate the gradient descent algorithm (momentum, etc.) can be and are commonly used with SGD.

▷ Stochastic Gradient Descent

▷ Batch Gradient Descent

▷ Minibatch Gradient Descent

3.3.4 Second-Order Algorithms

Second-order optimization algorithms explicitly involve using the second derivative (Hessian) to choose the direction to move in the search space. These algorithms are only appropriate for those objective functions where the Hessian matrix can be calculated or approximated. Examples of second-order optimization algorithms for univariate objective functions include:

▷ Newton's Method

▷ Secant Method

Second-order methods for multivariate objective functions are referred to as Quasi-Newton Methods.

▷ Quasi-Newton Method

There are many Quasi-Newton Methods, and they are typically named for the developers of the algorithm, such as:

▷ Davidson-Fletcher-Powell

▷ Broyden-Fletcher-Goldfarb-Shanno (BFGS)

▷ Limited-memory BFGS (L-BFGS)

Now that we are familiar with the so-called classical optimization algorithms, let's look at algorithms used when the objective function is not differentiable.

3.4 Non-Differential Objective Function

Optimization algorithms that make use of the derivative of the objective function are fast and efficient. Nevertheless, there are objective functions where the derivative cannot be calculated, typically because the function is complex for a variety of real-world reasons. Or the derivative can be calculated in some regions of the domain, but not all, or is not a good guide. Some difficulties on objective functions for the classical algorithms described in the previous section include:

▷ No analytical description of the function (e.g. simulation).

▷ Multiple global optima (e.g. multimodal).

▷ Stochastic function evaluation (e.g. noisy).

▷ Discontinuous objective function (e.g. regions with invalid solutions).

As such, there are optimization algorithms that do not expect first- or second-order derivatives to be available. These algorithms are sometimes referred to as black-box optimization algorithms as they assume little or nothing (relative to the classical methods) about the objective function. A grouping of these algorithms include:

▷ Direct Algorithms

▷ Stochastic Algorithms

▷ Population Algorithms

Let's take a closer look at each in turn.

3.4.1 Direct Algorithms

Direct optimization algorithms are for objective functions for which derivatives cannot be calculated. The algorithms are deterministic procedures and often assume the objective function has a single global optima, e.g. unimodal. Direct search methods are also typically referred to as a "*pattern search*" as they may navigate the search space using geometric shapes or decisions, e.g. patterns.

Gradient information is approximated directly (hence the name) from the result of the objective function comparing the relative difference between scores for points in the search space.

These direct estimates are then used to choose a direction to move in the search space and triangulate the region of the optima.

Examples of direct search algorithms include:

▷ Cyclic Coordinate Search

▷ Powell's Method

▷ Hooke-Jeeves Method

▷ Nelder-Mead Simplex Search

3.4.2 Stochastic Algorithms

Stochastic optimization algorithms are algorithms that make use of randomness in the search procedure for objective functions for which derivatives cannot be calculated. Unlike the deterministic direct search methods, stochastic algorithms typically involve a lot more sampling of the objective function, but are able to handle problems with deceptive local optima.

Stochastic optimization algorithms include:

▷ Simulated Annealing

▷ Evolution Strategy

▷ Cross-Entropy Method

3.4.3 Population Algorithms

Population optimization algorithms are stochastic optimization algorithms that maintain a pool (a population) of candidate solutions that together are used to sample, explore, and hone in on an optima. Algorithms of this type are intended for more challenging objective problems that may have noisy function evaluations and many global optima (multimodal), and finding a good or good enough solution is challenging or infeasible using other methods. The pool of candidate solutions adds robustness to the search, increasing the likelihood of overcoming local optima.

Examples of population optimization algorithms include:

▷ Genetic Algorithm

▷ Differential Evolution

▷ Particle Swarm Optimization

3.5   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

Sean Luke. *Essentials of Metaheuristics*. lulu.com, 2011.
    https://amzn.to/3lHryZr
Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. 2nd ed. Wiley, 2007.
    https://amzn.to/3ob61KA
James C. Spall. *Introduction to Stochastic Search and Optimization*. Wiley-Interscience, 2003.
    https://amzn.to/34JYN7m

Articles

Mathematical optimization. Wikipedia.
    https://en.wikipedia.org/wiki/Mathematical_optimization

3.6   Summary

In this tutorial, you discovered a guided tour of different optimization algorithms.

Specifically, you learned:

▷ Optimization algorithms may be grouped into those that use derivatives and those that do not.

▷ Classical algorithms use the first and sometimes second derivative of the objective function.

▷ Direct search and stochastic algorithms are designed for objective functions where function derivatives are unavailable.

Next, you will learn about some optimization concepts, starts with the no free lunch theorem.

Background II

No Free Lunch Theorem for Machine Learning

4

No Free Lunch Theorem is often thrown around in the field of optimization and machine learning, often with little understanding of what it means or implies. The theorem states that all optimization algorithms perform equally well when their performance is averaged across all possible problems. It implies that there is no single best optimization algorithm. Because of the close relationship between optimization, search, and machine learning, it also implies that there is no single best machine learning algorithm for predictive modeling problems such as classification and regression.

In this tutorial, you will discover the no free lunch theorem for optimization and search. After completing this tutorial, you will know:

▷ The no free lunch theorem suggests the performance of all optimization algorithms are identical, under some specific constraints.

▷ There is provably no single best optimization algorithm or machine learning algorithm.

▷ The practical implications of the theorem may be limited given we are interested in a small subset of all possible objective functions.

Let's get started.

4.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Is the No Free Lunch Theorem?
2. Implications for Optimization
3. Implications for Machine Learning

4.2   What Is the No Free Lunch Theorem?

The No Free Lunch Theorem, often abbreviated as NFL or NFLT, is a theoretical finding that suggests all optimization algorithms perform equally well when their performance is averaged over all possible objective functions.

> The NFL stated that within certain constraints, over the space of all possible problems, every optimization technique will perform as well as every other one on average (including Random Search)

— Page 203, *Essentials of Metaheuristics*, 2011.

The theorem applies to optimization generally and to search problems, as optimization can be described or framed as a search problem. The implication is that the performance of your favorite algorithm is identical to a completely naive algorithm, such as random search.

> Roughly speaking we show that for both static and time dependent optimization problems the average performance of any pair of algorithms across all possible problems is exactly identical.

— "No Free Lunch Theorems For Optimization", 1997.

An easy way to think about this finding is to consider a large table like you might have in Excel. Across the top of the table, each column represents a different optimization algorithm. Down the side of the table, each row represents a different objective function. Each cell of the table is the performance of the algorithm on the objective function, using whatever performance measure you like, as long as it is consistent across the whole table.

	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4	Algorithm 5	Algorithm 6	Algorithm 7	Algorithm 8	...
Problem 1	78.90158096	38.18696053	83.9788141	3.128185533	93.71767489	3.612131384	38.02555482	46.02033283	...
Problem 2	63.63661246	51.21726878	6.915100117	92.46504485	20.63056606	90.15194724	6.628150576	88.92628997	...
Problem 3	5.467817525	78.82129795	19.01963224	16.18471759	59.57316925	26.61430506	41.45446652	62.38540108	...
Problem 4	40.96337067	55.59045049	25.47959077	77.75563723	90.98183523	42.23275523	92.4381591	80.17316672	...
Problem 5	17.32640301	80.17604054	48.01380213	9.378352179	13.25844413	66.24497877	17.39991202	46.86218446	...
Problem 6	2.90117365	14.18732284	88.12091607	28.32526953	88.17950692	43.16349405	78.48956349	76.09121009	...
Problem 7	74.22339559	71.35440724	46.26625983	69.9710712	66.9510279	68.97533166	14.29350951	56.8139594	...
Problem 8	69.06790479	89.53420767	17.7105817	71.3419208	48.8622438	3.348772613	70.81053152	3.855765825	...
Problem 9	19.94675498	3.137513385	10.68373549	4.011603637	49.49135388	37.92530089	99.49914362	54.10622766	...
Problem 10	7.510870987	58.55534993	57.60647147	80.17271882	80.41639739	25.77488384	55.59960103	94.67596268	...
Problem 11	98.30840803	40.16271408	15.063453	80.71102508	67.38435353	2.092705478	54.93369837	34.34560747	...
Problem 12	56.35291015	99.47783881	73.23060569	79.11112105	58.89165367	51.21548188	72.3854659	54.63516655	...
Problem 13	42.95441914	5.055088383	20.45995021	60.02150262	2.129162205	0.03549031414	90.26590811	1.821852475	...
Problem 14	44.26664262	55.68963431	33.72502344	56.30721179	88.24480947	42.89040502	29.76489645	6.234549423	...
Problem 15	91.00330356	24.51201295	90.63002494	53.41813975	93.87696033	28.00711639	23.69333881	40.15298867	...
...	...	...	...	...	...	...	...	...	...
Average	100	100	100	100	100	100	100	100	

Figure 4.1: Depiction on the No Free Lunch Theorem as a Table of Algorithms and Problems

You can imagine that this table will be infinitely large. Nevertheless, in this table, we can calculate the average performance of any algorithm from all the values in its column and it will be identical to the average performance of any other algorithm column.

> If one algorithm performs better than another algorithm on one class of problems, then it will perform worse on another class of problems

— Page 6, *Algorithms for Optimization*, 2019.

Now that we are familiar with the no free lunch theorem in general, let's look at the specific implications for optimization.

4.3 Implications for Optimization

So-called black-box optimization algorithms are general optimization algorithms that can be applied to many different optimization problems and assume very little about the objective function. Examples of black-box algorithms include the genetic algorithm, simulated annealing, and particle swarm optimization.

The no free lunch theorem was proposed in an environment of the late 1990s where claims of one black-box optimization algorithm being better than another optimization algorithm were being made routinely. The theorem pushes back on this, indicating that there is no best optimization algorithm, that it is provably impossible. The theorem does state that no optimization algorithm is any better than any other optimization algorithm, on average.

> . . . known as the "no free lunch" theorem, sets a limit on how good a learner can be. The limit is pretty low: no learner can be better than random guessing!
>
> — Page 63, *The Master Algorithm*, 2018.

The catch is that the application of algorithms does not assume anything about the problem. In fact, algorithms are applied to objective functions with no prior knowledge, even such as whether the objective function is minimizing or maximizing. And this is a hard constraint of the theorem.

We often have "*some*" knowledge about the objective function being optimized. In fact, if in practice we truly knew nothing about the objective function, we could not choose an optimization algorithm.

> As elaborated by the no free lunch theorems of Wolpert and Macready, there is no reason to prefer one algorithm over another unless we make assumptions about the probability distribution over the space of possible objective functions.
>
> — Page 6, *Algorithms for Optimization*, 2019.

The beginner practitioner in the field of optimization is counseled to learn and use as much about the problem as possible in the optimization algorithm. The more we know and harness in the algorithms about the problem, the better tailored the technique is to the problem and the more likely the algorithm is expected to perform well on the problem. The no free lunch theorem supports this advice.

> We don't care about all possible worlds, only the one we live in. If we know something about the world and incorporate it into our learner, it now has an advantage over random guessing.
>
> — Page 63, *The Master Algorithm*, 2018.

Additionally, the performance is averaged over all possible objective functions and all possible optimization algorithms. Whereas in practice, we are interested in a small subset of objective functions that may have a specific structure or form and algorithms tailored to those functions.

> . . . we cannot emphasize enough that no claims whatsoever are being made in this paper concerning how well various search algorithms work in practice The focus

of this paper is on what can be said a priori without any assumptions and from mathematical principles alone concerning the utility of a search algorithm.

— "No Free Lunch Theorems For Optimization", 1997.

These implications lead some practitioners to note the limited practical value of the theorem.

This is of considerable theoretical interest but, I think, of limited practical value, because the space of all possible problems likely includes many extremely unusual and pathological problems which are rarely if ever seen in practice.

— Page 203, *Essentials of Metaheuristics*, 2011.

Now that we have reviewed the implications of the no free lunch theorem for optimization, let's review the implications for machine learning.

4.4 Implications for Machine Learning

Learning can be described or framed as an optimization problem, and most machine learning algorithms solve an optimization problem at their core. The no free lunch theorem for optimization and search is applied to machine learning, specifically supervised learning, which underlies classification and regression predictive modeling tasks. This means that all machine learning algorithms are equally effective across all possible prediction problems, e.g. random forest is as good as random predictions.

So all learning algorithms are the same in that: (1) by several definitions of "average", all algorithms have the same average off-training-set misclassification risk, (2) therefore no learning algorithm can have lower risk than another one for all f . . .

— "The Supervised Learning No-Free-Lunch Theorems", 2002.

This also has implications for the way in which algorithms are evaluated or chosen, such as choosing a learning algorithm via a k-fold cross-validation test harness or not.

. . . an algorithm that uses cross validation to choose amongst a prefixed set of learning algorithms does no better on average than one that does not.

— "The Supervised Learning No-Free-Lunch Theorems", 2002.

It also has implications for common heuristics for what constitutes a "*good*" machine learning model, such as avoiding overfitting or choosing the simplest possible model that performs well.

Another set of examples is provided by all the heuristics that people have come up with for supervised learning avoid overfitting prefer simpler to more complex models etc. [no free lunch] says that all such heuristics fail as often as they succeed.

— "The Supervised Learning No-Free-Lunch Theorems", 2002.

Given that there is no best single machine learning algorithm across all possible prediction problems, it motivates the need to continue to develop new learning algorithms and to better understand algorithms that have already been developed.

As a consequence of the no free lunch theorem, we need to develop many different types of models, to cover the wide variety of data that occurs in the real world. And

for each model, there may be many different algorithms we can use to train the model, which make different speed-accuracy-complexity tradeoffs. ""

— Pages 24–25, *Machine Learning: A Probabilistic Perspective*, 2012.

It also supports the argument of testing a suite of different machine learning algorithms for a given predictive modeling problem.

" The "*No Free Lunch*" Theorem argues that, without having substantive information about the modeling problem, there is no single model that will always do better than any other model. Because of this, a strong case can be made to try a wide variety of techniques, then determine which model to focus on. ""

— Pages 25–26, *Machine Learning: A Probabilistic Perspective*, 2012.

Nevertheless, as with optimization, the implications of the theorem are based on the choice of learning algorithms having zero knowledge of the problem that is being solved. In practice, this is not the case, and a beginner machine learning practitioner is encouraged to review the available data in order to learn something about the problem that can be incorporated into the learning algorithm. We may even want to take this one step further and say that learning is not possible without some prior knowledge and that data alone is not enough.

" In the meantime, the practical consequence of the "no free lunch" theorem is that there's no such thing as learning without knowledge. Data alone is not enough. ""

— Page 64, *The Master Algorithm*, 2018.

4.5   Further Reading

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

Papers

D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems For Optimization". *IEEE Transactions on Evolutionary Computation*, 1(1), 1997, pp. 67–82.
https://ieeexplore.ieee.org/abstract/document/585893

David H. Wolpert. "The Supervised Learning No-Free-Lunch Theorems". In: *Soft Computing and Industry*. Ed. by R. Roy et al. Springer, 2002.
https://link.springer.com/chapter/10.1007/978-1-4471-0123-9_3

David H. Wolpert. "The Lack of A Priori Distinctions Between Learning Algorithms". *Neural Computation*, 8(7), 1996, pp. 1341–1390.
https://www.mitpressjournals.org/doi/abs/10.1162/neco.1996.8.7.1341

David H. Wolpert and William G. Macready. *No Free Lunch Theorems for Search*. Tech. rep. SFI-TR-95-02-010. The Santa Fe Institute, 1995.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.7505&rep=rep1&type=pdf

Books

Sean Luke. *Essentials of Metaheuristics.* lulu.com, 2011.
    https://amzn.to/3lHryZr
Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization.* MIT Press, 2019.
    https://amzn.to/3je8O1J
Pedro Domingos. *The Master Algorithm.* Basic Books, 2018.
    https://amzn.to/3lKKGFX
Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* MIT Press, 2012.
    https://amzn.to/3nJJe8s
Max Kuhn and Kjell Johnson. *Applied Predictive Modeling.* Springer, 2013.
    https://amzn.to/3ly7nwK

Articles

No free lunch in search and optimization. Wikipedia.
    https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization
No free lunch theorem. Wikipedia.
    https://en.wikipedia.org/wiki/No_free_lunch_theorem
No Free Lunch Theorems.
    http://www.no-free-lunch.org/

4.6   Summary

In this tutorial, you discovered the no free lunch theorem for optimization and search. Specifically, you learned:

▷ The no free lunch theorem suggests the performance of all optimization algorithms are identical, under some specific constraints.

▷ There is provably no single best optimization algorithm or machine learning algorithm.

▷ The practical implications of the theorem may be limited given we are interested in a small subset of all possible objective functions.

Next, you will learn the distinction between a local and a global optimization.

Local Optimization vs. Global Optimization

5

Optimization refers to finding the set of inputs to an objective function that results in the maximum or minimum output from the objective function. It is common to describe optimization problems in terms of *local vs. global optimization*. Similarly, it is also common to describe optimization algorithms or search algorithms in terms of local vs. global search.

In this tutorial, you will discover the practical differences between local and global optimization. After completing this tutorial, you will know:

▷ Local optimization involves finding the optimal solution for a specific region of the search space, or the global optima for problems with no local optima.

▷ Global optimization involves finding the optimal solution on problems that contain local optima.

▷ How and when to use local and global search algorithms and how to use both methods in concert.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Local Optimization
2. Global Optimization
3. Local vs. Global Optimization

5.2 Local Optimization

A local optima is the extrema (minimum or maximum) of the objective function for a given region of the input space, e.g. a basin in a minimization problem.

> " . . . we seek a point that is only locally optimal, which means that it minimizes the objective function among feasible points that are near it . . . "

— Page 9, *Convex Optimization*, 2004.

An objective function may have many local optima, or it may have a single local optima, in which case the local optima is also the global optima.

 ▷ **Local Optimization**: Locate the optima for an objective function from a starting point believed to contain the optima (e.g. a basin).

Local optimization[1] or local search refers to searching for the local optima. A local optimization algorithm, also called a local search algorithm, is an algorithm intended to locate a local optima. It is suited to traversing a given region of the search space and getting close to (or finding exactly) the extrema of the function in that region.

> " . . . local optimization methods are widely used in applications where there is value in finding a good point, if not the very best. "

> — Page 9, *Convex Optimization*, 2004.

Local search algorithms typically operate on a single candidate solution and involve iteratively making small changes to the candidate solution and evaluating the change to see if it leads to an improvement and is taken as the new candidate solution.

A local optimization algorithm will locate the global optimum:

 ▷ If the local optima is the global optima, or

 ▷ If the region being searched contains the global optima.

These define the ideal use cases for using a local search algorithm.

There may be debate over what exactly constitutes a local search algorithm; nevertheless, three examples of local search algorithms using our definitions include:

 ▷ Nelder-Mead Algorithm

 ▷ BFGS Algorithm

 ▷ Hill-Climbing Algorithm

Now that we are familiar with local optimization, let's take a look at global optimization.

5.3 Global Optimization

A global optimum is the extrema (minimum or maximum) of the objective function for the entire input search space.

> " Global optimization, where the algorithm searches for the global optimum by employing mechanisms to search larger parts of the search space. "

> — Page 37, *Computational Intelligence: An Introduction*, 2007.

An objective function may have one or more than one global optima, and if more than one, it is referred to as a multimodal optimization[2] problem and each optimum will have a different input and the same objective function evaluation.

 ▷ **Global Optimization**: Locate the optima for an objective function that may contain local optima.

---

[1]https://en.wikipedia.org/wiki/Local_search_(optimization)
[2]https://en.wikipedia.org/wiki/Multimodal_distribution

An objective function always has a global optima (otherwise we would not be interested in optimizing it), although it may also have local optima that have an objective function evaluation that is not as good as the global optima. The global optima may be the same as the local optima, in which case it would be more appropriate to refer to the optimization problem as a local optimization, instead of global optimization.

The presence of the local optima is a major component of what defines the difficulty of a global optimization problem as it may be relatively easy to locate a local optima and relatively difficult to locate the global optima.

Global optimization[3] or global search refers to searching for the global optima. A global optimization algorithm, also called a global search algorithm, is intended to locate a global optima. It is suited to traversing the entire input search space and getting close to (or finding exactly) the extrema of the function.

> Global optimization is used for problems with a small number of variables, where computing time is not critical, and the value of finding the true global solution is very high.
>
> — Page 9, *Convex Optimization*, 2004.

Global search algorithms may involve managing a single or a population of candidate solutions from which new candidate solutions are iteratively generated and evaluated to see if they result in an improvement and taken as the new working state. There may be debate over what exactly constitutes a global search algorithm; nevertheless, three examples of global search algorithms using our definitions include:

▷ Genetic Algorithm

▷ Simulated Annealing

▷ Particle Swarm Optimization

Now that we are familiar with global and local optimization, let's compare and contrast the two.

5.4   Local vs. Global Optimization

Local and global search optimization algorithms solve different problems or answer different questions. A local optimization algorithm should be used when you know that you are in the region of the global optima or that your objective function contains a single optima, e.g. unimodal. A global optimization algorithm should be used when you know very little about the structure of the objective function response surface, or when you know that the function contains local optima.

> Local optimization, where the algorithm may get stuck in a local optimum without finding a global optimum.
>
> — Page 37, *Computational Intelligence: An Introduction*, 2007.

Applying a local search algorithm to a problem that requires a global search algorithm will deliver poor results as the local search will get caught (deceived) by local optima.

---

[3] https://en.wikipedia.org/wiki/Global_optimization

▷ **Local search**: When you are in the region of the global optima.

▷ **Global search**: When you know that there are local optima.

Local search algorithms often give computational complexity guarantees related to locating the global optima, as long as the assumptions made by the algorithm hold. Global search algorithms often give very few if any guarantees about locating the global optima. As such, global search is often used on problems that are sufficiently difficult that "*good*" or "*good enough*" solutions are preferred over no solutions at all. This might mean relatively good local optima instead of the true global optima if locating the global optima is intractable.

It is often appropriate to re-run or re-start the algorithm multiple times and record the optima found by each run to give some confidence that relatively good solutions have been located.

▷ **Local search**: For narrow problems where the global solution is required.

▷ **Global search**: For broad problems where the global optima might be intractable.

We often know very little about the response surface for an objective function, e.g. whether a local or global search algorithm is most appropriate. Therefore, it may be desirable to establish a baseline in performance with a local search algorithm and then explore a global search algorithm to see if it can perform better. If it cannot, it may suggest that the problem is indeed unimodal or appropriate for a local search algorithm.

▷ **Best Practice**: Establish a baseline with a local search then explore a global search on objective functions where little is known.

Local optimization is a simpler problem to solve than global optimization. As such, the vast majority of the research on mathematical optimization has been focused on local search techniques.

> A large fraction of the research on general nonlinear programming has focused on methods for local optimization, which as a consequence are well developed.
>
> — Page 9, *Convex Optimization*, 2004.

Global search algorithms are often coarse in their navigation of the search space.

> Many population methods perform well in global search, being able to avoid local minima and finding the best regions of the design space. Unfortunately, these methods do not perform as well in local search in comparison to descent methods.
>
> — Page 162, *Algorithms for Optimization*, 2019.

As such, they may locate the basin for a good local optima or the global optima, but may not be able to locate the best solution within the basin.

> Local and global optimization techniques can be combined to form hybrid training algorithms.
>
> — Page 37, *Computational Intelligence: An Introduction*, 2007.

Therefore, it is a good practice to apply a local search to the optima candidate solutions found by a global search algorithm.

▷ **Best Practice**: Apply a local search to the solutions found by a global search.

5.5   Further Reading

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

Books

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge, 2004.
    https://amzn.to/34mvCr1
Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. 2nd ed. Wiley, 2007.
    https://amzn.to/3ob61KA
Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
    https://amzn.to/3je8O1J

Articles

Local search (optimization). Wikipedia.
    https://en.wikipedia.org/wiki/Local_search_(optimization)
Global optimization. Wikipedia.
    https://en.wikipedia.org/wiki/Global_optimization

5.6   Summary

In this tutorial, you discovered the practical differences between local and global optimization. Specifically, you learned:

- ▷ Local optimization involves finding the optimal solution for a specific region of the search space, or the global optima for problems with no local optima.

- ▷ Global optimization involves finding the optimal solution on problems that contain local optima.

- ▷ How and when to use local and global search algorithms and how to use both methods in concert.

Next, you will learn about convergence and what happens if an algorithm converge prematurely.

Premature Convergence

6

Convergence refers to the limit of a process and can be a useful analytical tool when evaluating the expected performance of an optimization algorithm. It can also be a useful empirical tool when exploring the learning dynamics of an optimization algorithm, and machine learning algorithms trained using an optimization algorithm, such as deep learning neural networks. This motivates the investigation of learning curves and techniques, such as early stopping. If optimization is a process that generates candidate solutions, then convergence represents a stable point at the end of the process when no further changes or improvements are expected. *Premature convergence* refers to a failure mode for an optimization algorithm where the process stops at a stable point that does not represent a globally optimal solution.

In this tutorial, you will discover a gentle introduction to premature convergence in machine learning. After completing this tutorial, you will know:

▷ Convergence refers to the stable point found at the end of a sequence of solutions via an iterative optimization algorithm.

▷ Premature convergence refers to a stable point found too soon, perhaps close to the starting point of the search, and with a worse evaluation than expected.

▷ Greediness of an optimization algorithm provides a control over the rate of convergence of an algorithm.

Let's get started.

6.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Convergence in Machine Learning
2. Premature Convergence
3. Addressing Premature Convergence

6.2 Convergence in Machine Learning

Convergence[1] generally refers to the values of a process that have a tendency in behavior over time. It is a useful idea when working with optimization algorithms.

Optimization refers to a type of problem that requires finding a set of inputs that result in the maximum or minimum value from an objective function. Optimization is an iterative process that produces a sequence of candidate solutions until ultimately arriving upon a final solution at the end of the process. This behavior or dynamics of the optimization algorithm arriving on a stable-point final solution is referred to as convergence, e.g. the convergence of the optimization algorithms. In this way, convergence defines the termination of the optimization algorithm.

> Local descent involves iteratively choosing a descent direction and then taking a step in that direction and repeating that process until convergence or some termination condition is met.
>
> — Page 13, *Algorithms for Optimization*, 2019.

▷ **Convergence**: Stop condition for an optimization algorithm where a stable point is located and further iterations of the algorithm are unlikely to result in further improvement.

We might measure and explore the convergence of an optimization algorithm empirically, such as using learning curves. Additionally, we might also explore the convergence of an optimization algorithm analytically, such as a convergence proof or average case computational complexity.

> Strong selection pressure results in rapid, but possibly premature, convergence. Weakening the selection pressure slows down the search process . . .
>
> — Page 78, *Evolutionary Computation: A Unified Approach*, 2002.

Optimization, and the convergence of optimization algorithms, is an important concept in machine learning for those algorithms that fit (learn) on a training dataset via an iterative optimization algorithm, such as logistic regression and artificial neural networks. As such, we may choose optimization algorithms that result in better convergence behavior than other algorithms, or spend a lot of time tuning the convergence dynamics (learning dynamics) of an optimization algorithm via the hyperparameters of the optimization (e.g. learning rate). Convergence behavior can be compared, often in terms of the number of iterations of an algorithm required until convergence, to the objective function evaluation of the stable point found at convergence, and combinations of these concerns.

6.3 Premature Convergence

Premature convergence refers to the convergence of a process that has occurred too soon. In optimization, it refers to the algorithm converging upon a stable point that has worse performance than expected. Premature convergence typically afflicts complex optimization tasks where the objective function is non-convex, meaning that the response surface contains many different good solutions (stable points), perhaps with one (or a few) best solutions. If

---

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

we consider the response surface of an objective function under optimization as a geometrical landscape and we are seeking a minimum of the function, then premature optimization refers to finding a valley close to the starting point of the search that has less depth than the deepest valley in the problem domain.

> For problems that exhibit highly multi-modal (rugged) fitness landscapes or landscapes that change over time, too much exploitation generally results in premature convergence to suboptimal peaks in the space.
>
> — Page 60, *Evolutionary Computation: A Unified Approach*, 2002.

In this way, premature convergence is described as finding a locally optimal solution instead of the globally optimal solution for an optimization algorithm. It is a specific failure case for an optimization algorithm.

▷ **Premature Convergence**: Convergence of an optimization algorithm to a worse than optimal stable point that is likely close to the starting point.

Put another way, convergence signifies the end of the search process, e.g. a stable point was located and further iterations of the algorithm are not likely to improve upon the solution. Premature convergence refers to reaching this stop condition of an optimization algorithm at a less than desirable stationary point.

6.4   Addressing Premature Convergence

Premature convergence may be a relevant concern on any reasonably challenging optimization task. For example, a majority of research into the field of evolutionary computation and genetic algorithms involves identifying and overcoming the premature convergence of the algorithm on an optimization task.

> If selection focuses on the most-fit individuals, the selection pressure may cause premature convergence due to reduced diversity of the new populations.
>
> — Page 139, *Computational Intelligence: An Introduction*, 2007.

Population-based optimization algorithms, like evolutionary algorithms and swarm intelligence, often describe their dynamics in terms of the interplay between selective pressures and convergence. For example, strong selective pressures result in faster convergence and likely premature convergence. Weaker selective pressures may result in a slower convergence (greater computational cost) although perhaps locate a better or even global optima.

> An operator with a high selective pressure decreases diversity in the population more rapidly than operators with a low selective pressure, which may lead to premature convergence to suboptimal solutions. A high selective pressure limits the exploration abilities of the population.
>
> — Page 135, *Computational Intelligence: An Introduction*, 2007.

This idea of selective pressure is helpful more generally in understanding the learning dynamics of optimization algorithms. For example, an optimization that is configured to be too greedy (e.g. via hyperparameters such as the step size or learning rate) may fail due to

premature convergence, whereas the same algorithm that is configured to be less greedy may overcome premature convergence and discover a better or globally optimal solution.

Premature convergence may be encountered when using stochastic gradient descent to train a neural network model, signified by a learning curve that drops exponentially quickly then stops improving.

> The number of updates required to reach convergence usually increases with training set size. However, as m approaches infinity, the model will eventually converge to its best possible test error before SGD has sampled every example in the training set.
>
> — Page 153, *Deep Learning*, 2016.

The fact that fitting neural networks are subject to premature convergence motivates the use of methods such as learning curves to monitor and diagnose issues with the convergence of a model on a training dataset, and the use of regularization, such as early stopping, that halts the optimization algorithm prior to finding a stable point comes at the expense of worse performance on a holdout dataset. As such, much research into deep learning neural networks is ultimately directed at overcoming premature convergence.

> Empirically, it is often found that 'tanh' activation functions give rise to faster convergence of training algorithms than logistic functions.
>
> — Page 127, *Neural Networks for Pattern Recognition*, 1995.

This includes techniques such as work on weight initialization, which is critical because the initial weights of a neural network define the starting point of the optimization process, and poor initialization can lead to premature convergence.

> The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.
>
> — Page 301, *Deep Learning*, 2016.

This also includes the vast number of variations and extensions of the stochastic gradient descent optimization algorithm, such as the addition of momentum so that the algorithm does not overshoot the optima (stable point), and Adam that adds an automatically adapted step size hyperparameter (learning rate) for each parameter that is being optimized, dramatically speeding up convergence.

6.5 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
Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach*. Bradford Book, 2002.
https://amzn.to/2LjWceK

Andries P. Engelbrecht. *Computational Intelligence: An Introduction.* 2nd ed. Wiley, 2007.
    https://amzn.to/3ob61KA
Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.
    https://amzn.to/3qSk3C2
Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995.
    https://amzn.to/3nFrjyF
Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 3rd ed. Pearson, 2009.
    https://amzn.to/2HYk1Xj
Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1998.
    https://amzn.to/3kK8Osd
Kevin Patrick Murphy. *Probabilistic Machine Learning: An Introduction.* MIT Press, 2022.
    https://probml.github.io/pml-book/book1.html

Articles

Limit of a sequence. Wikipedia.
    https://en.wikipedia.org/wiki/Limit_of_a_sequence
Convergence of random variables. Wikipedia.
    https://en.wikipedia.org/wiki/Convergence_of_random_variables
Premature convergence. Wikipedia.
    https://en.wikipedia.org/wiki/Premature_convergence

6.6   Summary

In this tutorial, you discovered a gentle introduction to premature convergence in machine learning. Specifically, you learned:

▷ Convergence refers to the stable point found at the end of a sequence of solutions via an iterative optimization algorithm.

▷ Premature convergence refers to a stable point found too soon, perhaps close to the starting point of the search, and with a worse evaluation than expected.

▷ Greediness of an optimization algorithm provides a control over the rate of convergence of an algorithm.

Next, you will learn how to plot a function so you can see the progress of optimization visually.

Creating Visualization for Function Optimization

7

Function optimization involves finding the input that results in the optimal value from an objective function. Optimization algorithms navigate the search space of input variables in order to locate the optima, and both the shape of the objective function and behavior of the algorithm in the search space are opaque on real-world problems. As such, it is common to study optimization algorithms using simple low-dimensional functions that can be easily visualized directly. Additionally, the samples in the input space of these simple functions made by an optimization algorithm can be visualized with their appropriate context. Visualization of lower-dimensional functions and algorithm behavior on those functions can help to develop the intuitions that can carry over to more complex higher-dimensional function optimization problems later.

In this tutorial, you will discover how to create visualizations for function optimization in Python. After completing this tutorial, you will know:

▷ Visualization is an important tool when studying function optimization algorithms.

▷ How to visualize one-dimensional functions and samples using line plots.

▷ How to visualize two-dimensional functions and samples using contour and surface plots.

Let's get started.

7.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Visualization for Function Optimization

2. Visualize 1D Function Optimization

    (a) Test Function

    (b) Sample Test Function

    (c) Line Plot of Test Function

    (d) Scatter Plot of Test Function

    (e) Line Plot with Marked Optima

    (f) Line Plot with Samples

3. Visualize 2D Function Optimization

    (a) Test Function

    (b) Sample Test Function

    (c) Contour Plot of Test Function

    (d) Filled Contour Plot of Test Function

    (e) Filled Contour Plot of Test Function with Samples

    (f) Surface Plot of Test Function

7.2 Visualization for Function Optimization

Function optimization is a field of mathematics concerned with finding the inputs to a function that result in the optimal output for the function, typically a minimum or maximum value.

Optimization may be straightforward for simple differential functions where the solution can be calculated analytically. However, most functions we're interested in solving in applied machine learning may or may not be well behaved and may be complex, nonlinear, multivariate, and non-differentiable. As such, it is important to have an understanding of a wide range of different algorithms that can be used to address function optimization problems.

An important aspect of studying function optimization is understanding the objective function that is being optimized and understanding the behavior of an optimization algorithm over time. Visualization plays an important role when getting started with function optimization. We can select simple and well-understood test functions to study optimization algorithms. These simple functions can be plotted to understand the relationship between the input to the objective function and the output of the objective function and highlighting hills, valleys, and optima.

In addition, the samples selected from the search space by an optimization algorithm can also be plotted on top of plots of the objective function. These plots of algorithm behavior can provide insight and intuition into how specific optimization algorithms work and navigate a search space that can generalize to new problems in the future. Typically, one-dimensional or two-dimensional functions are chosen to study optimization algorithms as they are easy to visualize using standard plots, like line plots and surface plots. We will explore both in this tutorial.

First, let's explore how we might visualize a one-dimensional function optimization.

7.3 Visualize 1D Function Optimization

A one-dimensional function takes a single input variable and outputs the evaluation of that input variable. Input variables are typically continuous, represented by a real-valued floating-point value. Often, the input domain is unconstrained, although for test problems we impose a domain of interest.

7.3.1 Test Function

In this case we will explore function visualization with a simple x^2 objective function:

$$f(x) = x^2$$

This has an optimal value with an input of $x = 0.0$, which equals 0.0. The example below implements this objective function and evaluates a single input.

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

# evaluate inputs to the objective function
x = 4.0
result = objective(x)
print('f(%.3f) = %.3f' % (x, result))
```
Program 7.1: Example of a 1D objective function

Running the example evaluates the value 4.0 with the objective function, which equals 16.0.

```
f(4.000) = 16.000
```
Output 7.1: Result of Program 7.1

7.3.2 Sample the Test Function

The first thing we might want to do with a new function is define an input range of interest and sample the domain of interest using a uniform grid. This sample will provide the basis for generating a plot later. In this case, we will define a domain of interest around the optima of $x = 0.0$ from $x = -5.0$ to $x = 5.0$ and sample a grid of values in this range with 0.1 increments, such as -5.0, -4.9, -4.8, etc.

```python
...
# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# summarize some of the input domain
print(inputs[:5])
```
Program 7.2: Set up uniform input range

We can then evaluate each of the x values in our sample.

```python
...
# compute targets
results = objective(inputs)
# summarize some of the results
print(results[:5])
```
Program 7.3: Compute target from each input

Finally, we can check some of the input and their corresponding outputs.

```
...
for i in range(5):
    print('f(%.3f) = %.3f' % (inputs[i], results[i]))
```
Program 7.4: Creating a mapping of some inputs to some results

Tying this together, the complete example of sampling the input space and evaluating all points in the sample is listed below.

```
from numpy import arange

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

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# summarize some of the input domain
print(inputs[:5])
# compute targets
results = objective(inputs)
# summarize some of the results
print(results[:5])
# create a mapping of some inputs to some results
for i in range(5):
    print('f(%.3f) = %.3f' % (inputs[i], results[i]))
```
Program 7.5: Complete code to sample 1D objective function

Running the example first generates a uniform sample of input points as we expected. The input points are then evaluated using the objective function and finally, we can see a simple mapping of inputs to outputs of the objective function.

```
[-5.  -4.9 -4.8 -4.7 -4.6]
[25.   24.01 23.04 22.09 21.16]
f(-5.000) = 25.000
f(-4.900) = 24.010
f(-4.800) = 23.040
f(-4.700) = 22.090
f(-4.600) = 21.160
```
Output 7.2: Result from Program 7.5

Now that we have some confidence in generating a sample of inputs and evaluating them with the objective function, we can look at generating plots of the function.

7.3.3 Line Plot of Test Function

We could sample the input space randomly, but the benefit of a uniform line or grid of points is that it can be used to generate a smooth plot. It is smooth because the points in the input

space are ordered from smallest to largest. This ordering is important as we expect (hope) that the output of the objective function has a similar smooth relationship between values, i.e. small changes in input result in locally consistent (smooth) changes in the output of the function.

In this case, we can use the samples to generate a line plot of the objective function with the input points (x) on the x-axis of the plot and the objective function output (results) on the y-axis of the plot.

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

Program 7.6: Creating line plot of the function

Tying this together, the complete example is listed below.

```
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 = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 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 7.7: Complete code to create line plot of input vs result for a 1D objective function

Running the example creates a line plot of the objective function. We can see that the function has a large U-shape, called a parabola[1]. This is a common shape when studying curves, i.e. the study of calculus[2].

---

[1]https://en.wikipedia.org/wiki/Parabola
[2]https://en.wikipedia.org/wiki/Calculus
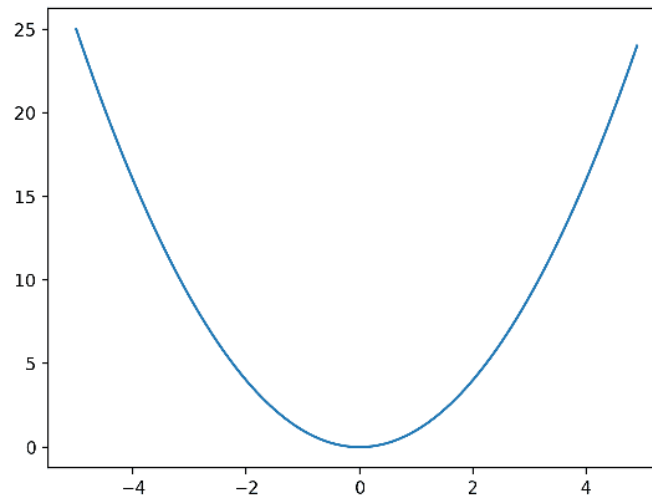
Figure 7.1: Line Plot of a One-Dimensional Function

7.3.4 Scatter Plot of Test Function

The line is a construct. It is not really the function, just a smooth summary of the function. Always keep this in mind. Recall that we, in fact, generated a sample of points in the input space and corresponding evaluation of those points. As such, it would be more accurate to create a scatter plot of points; for example:

```python
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 = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = objective(inputs)
# create a scatter plot of input vs result
pyplot.scatter(inputs, results)
# show the plot
pyplot.show()
```

Program 7.8: Create scatter plot of input vs result for a 1D objective function

Running the example creates a scatter plot of the objective function. We can see the familiar shape of the function, but we don't gain anything from plotting the points directly. The line and the smooth interpolation between the points it provides are more useful as we can draw

other points on top of the line, such as the location of the optima or the points sampled by an optimization algorithm.
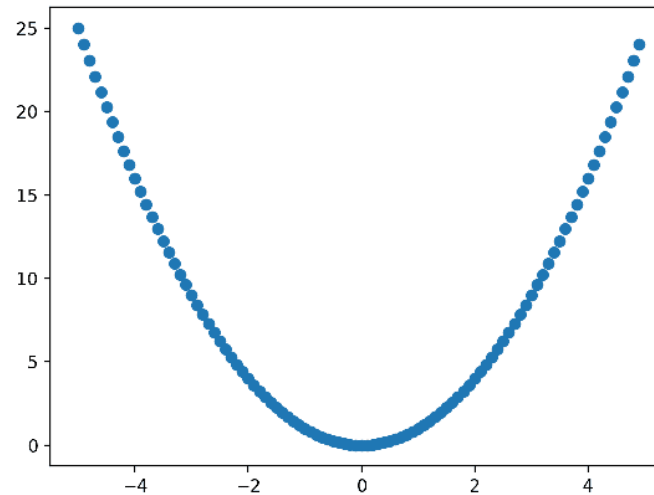
Figure 7.2: Scatter Plot of a One-Dimensional Function

7.3.5 Line Plot with Marked Optima

Next, let's draw the line plot again and this time draw a point where the known optima of the function is located. This can be helpful when studying an optimization algorithm as we might want to see how close an optimization algorithm can get to the optima.

First, we must define the input for the optima, then evaluate that point to give the x-axis and y-axis values for plotting.

```
...
optima_x = 0.0
optima_y = objective(optima_x)
```

Program 7.9: Define the known function optima

We can then plot this point with any shape or color we like, in this case, a red square.

```
...
pyplot.plot([optima_x], [optima_y], 's', color='r')
```

Program 7.10: Draw the function optima as a red square

Tying this together, the complete example of creating a line plot of the function with the optima highlighted by a point is listed below.

```python
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 = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define the known function optima
optima_x = 0.0
optima_y = objective(optima_x)
# draw the function optima as a red square
pyplot.plot([optima_x], [optima_y], 's', color='r')
# show the plot
pyplot.show()
```

Program 7.11: Create line plot of input vs result for a 1D objective function and show optima

Running the example creates the familiar line plot of the function, and this time, the optima of the function, i.e. the input that results in the minimum output of the function, is marked with a red square.
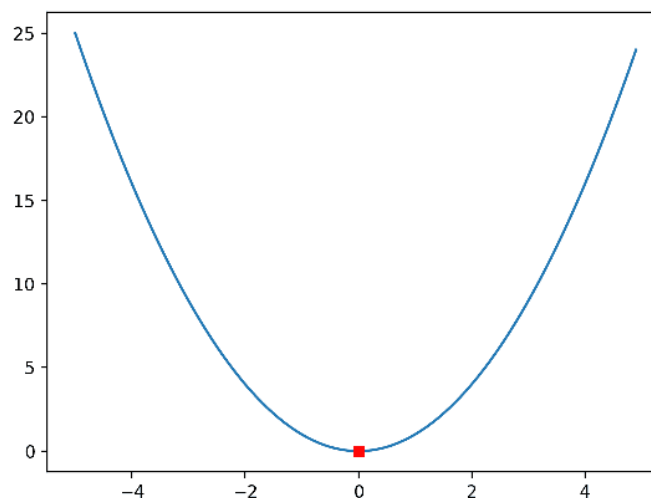
Figure 7.3: Line plot of a one-dimensional function with optima marked by a red square

This is a very simple function and the red square for the optima is easy to see. Sometimes the function might be more complex, with lots of hills and valleys, and we might want to make the optima more visible. In this case, we can draw a vertical line across the whole plot.

```
...
pyplot.axvline(x=optima_x, ls='--', color='red')
```

Program 7.12: Draw a vertical line at the optimal input

Tying this together, the complete example is listed below.

```python
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 = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define the known function optima
optima_x = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=optima_x, ls='--', color='red')
# show the plot
pyplot.show()
```

Program 7.13: Line plot of input vs result for a 1D objective function and show optima as line

Running the example creates the same plot and this time draws a red line clearly marking the point in the input space that marks the optima.
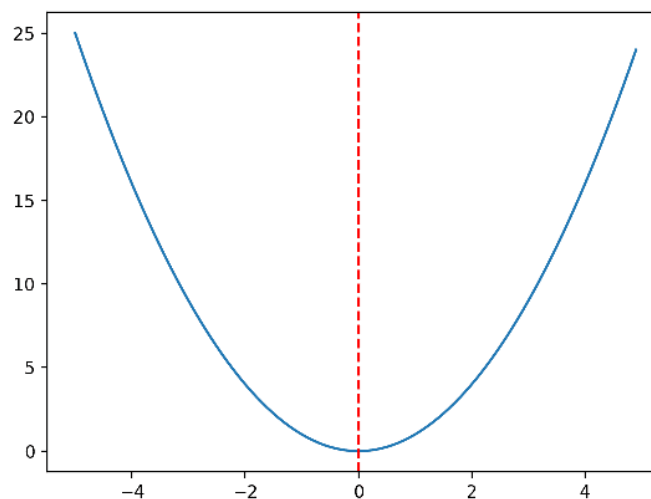
Figure 7.4: Line plot of a one-dimensional function with optima marked by a red line

7.3.6 Line Plot with Samples

Finally, we might want to draw the samples of the input space selected by an optimization algorithm. We will simulate these samples with random points drawn from the input domain.

```
...
seed(1)
sample = r_min + rand(10) * (r_max - r_min)
# evaluate the sample
sample_eval = objective(sample)
```

Program 7.14: Simulate a sample made by an optimization algorithm

We can then plot this sample, in this case using small black circles.

```
...
pyplot.plot(sample, sample_eval, 'o', color='black')
```

Program 7.15: Plot the sample as black circles

The complete example of creating a line plot of a function with the optima marked by a red line and an algorithm sample drawn with small black dots is listed below.

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

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

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = objective(inputs)
# simulate a sample made by an optimization algorithm
seed(1)
sample = r_min + rand(10) * (r_max - r_min)
# evaluate the sample
sample_eval = objective(sample)
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define the known function optima
optima_x = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=optima_x, ls='--', color='red')
# plot the sample as black circles
pyplot.plot(sample, sample_eval, 'o', color='black')
# show the plot
pyplot.show()
```

Program 7.16: Line plot of domain for a 1D function with optima and algorithm sample

Running the example creates the line plot of the domain and marks the optima with a red line as before. This time, the sample from the domain selected by an algorithm (really a random sample of points) is drawn with black dots. We can imagine that a real optimization algorithm will show points narrowing in on the domain as it searches down-hill from a starting point.
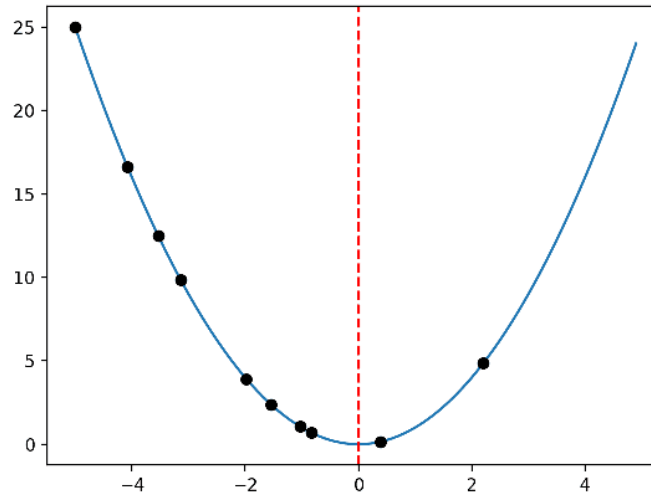
Figure 7.5: Line Plot of a One-Dimensional Function With Optima Marked by a Red Line and Samples Shown with Black Dots

Next, let's look at how we might perform similar visualizations for the optimization of a two-dimensional function.

7.4 Visualize 2D Function Optimization

A two-dimensional function is a function that takes two input variables, e.g. x and y.

7.4.1 Test Function

We can use the same x^2 function and scale it up to be a two-dimensional function; for example:

$$f(x, y) = x^2 + y^2$$

This has an optimal value with an input of $[x = 0.0, y = 0.0]$, which equals 0.0.

The example below implements this objective function and evaluates a single input.

```python
# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# evaluate inputs to the objective function
x = 4.0
```

```
y = 4.0
result = objective(x, y)
print('f(%.3f, %.3f) = %.3f' % (x, y, result))
```
Program 7.17: Example of a 2D objective function

Running the example evaluates the point $[x = 4, y = 4]$, which equals 32.

```
f(4.000, 4.000) = 32.000
```
Output 7.3: Result from Program 7.17

Next, we need a way to sample the domain so that we can, in turn, sample the objective function.

7.4.2 Sample Test Function

A common way for sampling a two-dimensional function is to first generate a uniform sample along each variable, x and y, then use these two uniform samples to create a grid of samples, called a mesh grid[3]. This is not a two-dimensional array across the input space; instead, it is two two-dimensional arrays that, when used together, define a grid across the two input variables. This is achieved by duplicating the entire x sample array for each y sample point and similarly duplicating the entire y sample array for each x sample point.

This can be achieved using the `meshgrid()` NumPy function[4]; for example:

```
...
# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# summarize some of the input domain
print(x[:5, :5])
```
Program 7.18: Using `meshgrid()` to create sample points

We can then evaluate each pair of points using our objective function.

```
...
# compute targets
results = objective(x, y)
# summarize some of the results
print(results[:5, :5])
```
Program 7.19: Evaluate targets from each sample point

Finally, we can review the mapping of some of the inputs to their corresponding output values.

---

[3]https://www.mathworks.com/help/matlab/ref/meshgrid.html
[4]https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html

```
...
for i in range(5):
    print('f(%.3f, %.3f) = %.3f' % (x[i,0], y[i,0], results[i,0]))
```

Program 7.20: Create a mapping of some inputs to some results

The example below demonstrates how we can create a uniform sample grid across the two-dimensional input space and objective function.

```
from numpy import arange
from numpy import meshgrid

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# summarize some of the input domain
print(x[:5, :5])
# compute targets
results = objective(x, y)
# summarize some of the results
print(results[:5, :5])
# create a mapping of some inputs to some results
for i in range(5):
    print('f(%.3f, %.3f) = %.3f' % (x[i,0], y[i,0], results[i,0]))
```

Program 7.21: Complete code of sampling a 2D objective function

Running the example first summarizes some points in the mesh grid, then the objective function evaluation for some points. Finally, we enumerate coordinates in the two-dimensional input space and their corresponding function evaluation.

```
[[-5.  -4.9 -4.8 -4.7 -4.6]
 [-5.  -4.9 -4.8 -4.7 -4.6]
 [-5.  -4.9 -4.8 -4.7 -4.6]
 [-5.  -4.9 -4.8 -4.7 -4.6]
 [-5.  -4.9 -4.8 -4.7 -4.6]]
[[50.   49.01 48.04 47.09 46.16]
 [49.01 48.02 47.05 46.1  45.17]
 [48.04 47.05 46.08 45.13 44.2 ]
 [47.09 46.1  45.13 44.18 43.25]
 [46.16 45.17 44.2  43.25 42.32]]
f(-5.000, -5.000) = 50.000
f(-5.000, -4.900) = 49.010
f(-5.000, -4.800) = 48.040
f(-5.000, -4.700) = 47.090
f(-5.000, -4.600) = 46.160
```

Output 7.4: Result from Program 7.21

Now that we are familiar with how to sample the input space and evaluate points, let's look at how we might plot the function.

7.4.3 Contour Plot of Test Function

A popular plot for two-dimensional functions is a contour plot[5]. This plot creates a flat representation of the objective function outputs for each x and y coordinate where the color and contour lines indicate the relative value or height of the output of the objective function. This is just like a contour map of a landscape where mountains can be distinguished from valleys.

This can be achieved using the `contour()` Matplotlib[6] function that takes the mesh grid and the evaluation of the mesh grid as input directly. We can then specify the number of levels to draw on the contour and the color scheme to use. In this case, we will use 50 levels and a popular "*jet*" color scheme where low-levels use a cold color scheme (blue) and high-levels use a hot color scheme (red).

```
...
# create a contour plot with 50 levels and jet color scheme
pyplot.contour(x, y, results, 50, alpha=1.0, cmap='jet')
# show the plot
pyplot.show()
```

Program 7.22: Creating a contour plot

Tying this together, the complete example of creating a contour plot of the two-dimensional objective function is listed below.

```
# contour plot for 2d objective function
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a contour plot with 50 levels and jet color scheme
pyplot.contour(x, y, results, 50, alpha=1.0, cmap='jet')
# show the plot
pyplot.show()
```

Program 7.23: Complete code of creating a contour plot of a 2D objective function

---

[5]https://en.wikipedia.org/wiki/Contour_line
[6]https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.contour.html

Running the example creates the contour plot. We can see that the more curved parts of the surface around the edges have more contours to show the detail, and the less curved parts of the surface in the middle have fewer contours. We can see that the lowest part of the domain is the middle, as expected.
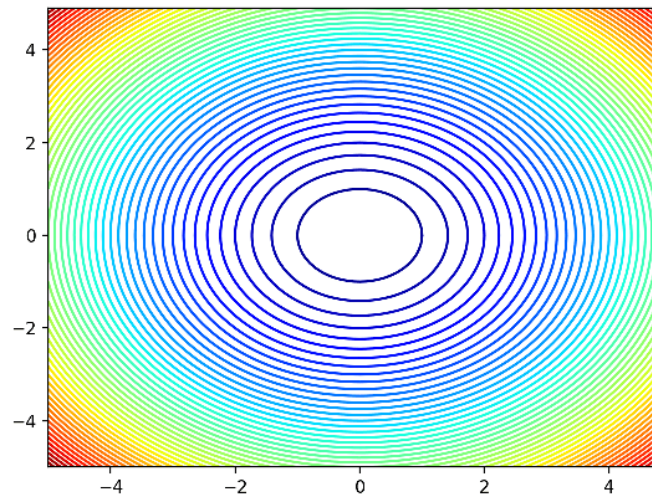
Figure 7.6: Contour Plot of a Two-Dimensional Objective Function

7.4.4 Filled Contour Plot of Test Function

It is also helpful to color the plot between the contours to show a more complete surface. Again, the colors are just a simple linear interpolation, not the true function evaluation. This must be kept in mind on more complex functions where fine detail will not be shown.

We can fill the contour plot using the `contourf()` version of the function[7] that takes the same arguments.

```
...
pyplot.contourf(x, y, results, levels=50, cmap='jet')
```
Program 7.24: Create a filled contour plot with 50 levels and jet color scheme

We can also show the optima on the plot, in this case as a white star that will stand out against the blue background color of the lowest part of the plot.

```
...
# define the known function optima
optima_x = [0.0, 0.0]
# draw the function optima as a white star
pyplot.plot([optima_x[0]], [optima_x[1]], '*', color='white')
```
Program 7.25: Draw the function optima as a white star

---

[7]https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.contourf.html

Tying this together, the complete example of a filled contour plot with the optima marked is listed below.

```python
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# define the known function optima
optima_x = [0.0, 0.0]
# draw the function optima as a white star
pyplot.plot([optima_x[0]], [optima_x[1]], '*', color='white')
# show the plot
pyplot.show()
```

Program 7.26: Filled contour plot for 2D objective function and show the optima

Running the example creates the filled contour plot that gives a better idea of the shape of the objective function. The optima at $[x = 0, y = 0]$ is then marked clearly with a white star.
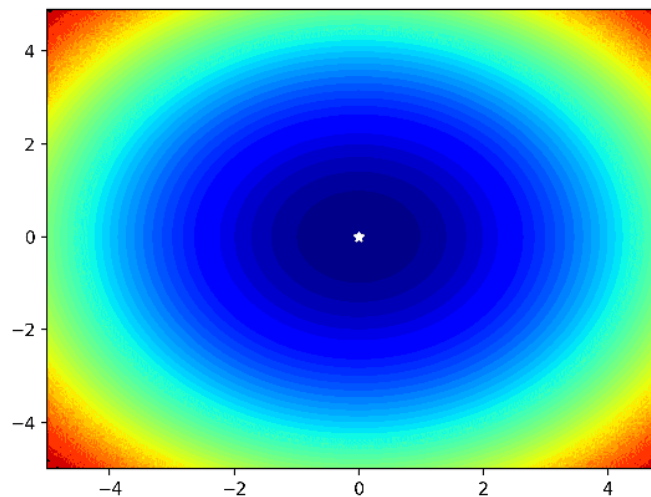
Figure 7.7: Filled Contour Plot of a Two-Dimensional Objective Function With Optima Marked by a White Star

7.4.5 Filled Contour Plot of Test Function with Samples

We may want to show the progress of an optimization algorithm to get an idea of its behavior in the context of the shape of the objective function. In this case, we can simulate the points chosen by an optimization algorithm with random coordinates in the input space.

```
...
seed(1)
sample_x = r_min + rand(10) * (r_max - r_min)
sample_y = r_min + rand(10) * (r_max - r_min)
```

Program 7.27: Simulate a sample made by an optimization algorithm

These points can then be plotted directly as black circles and their context color can give an idea of their relative quality.

```
...
# plot the sample as black circles
pyplot.plot(sample_x, sample_y, 'o', color='black')
```

Program 7.28: Plot the sample as black circles

Tying this together, the complete example of a filled contour plot with optimal and input sample plotted is listed below.

```
from numpy import arange
from numpy import meshgrid
from numpy.random import seed
from numpy.random import rand
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# simulate a sample made by an optimization algorithm
seed(1)
sample_x = r_min + rand(10) * (r_max - r_min)
sample_y = r_min + rand(10) * (r_max - r_min)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# define the known function optima
optima_x = [0.0, 0.0]
# draw the function optima as a white star
```

```
pyplot.plot([optima_x[0]], [optima_x[1]], '*', color='white')
# plot the sample as black circles
pyplot.plot(sample_x, sample_y, 'o', color='black')
# show the plot
pyplot.show()
```
Program 7.29: Filled contour plot for 2D objective function and show the optima and sample

Running the example, we can see the filled contour plot as before with the optima marked. We can now see the sample drawn as black dots and their surrounding color and relative distance to the optima gives an idea of how close the algorithm (random points in this case) got to solving the problem.
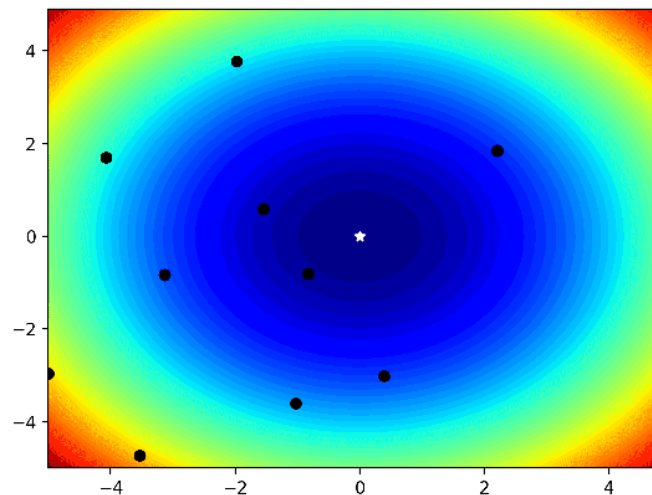
Figure 7.8: Filled Contour Plot of a Two-Dimensional Objective Function With Optima and Input Sample Marked

7.4.6 Surface Plot of Test Function

Finally, we may want to create a three-dimensional plot of the objective function to get a fuller idea of the curvature of the function.

This can be achieved using the `plot_surface()` Matplotlib function[8], that, like the contour plot, takes the mesh grid and function evaluation directly.

```
...
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
```
Program 7.30: Create a surface plot with the jet color scheme

---

[8]https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#mpl_toolkits.mplot3d.Axes3D.plot_surface

The complete example of creating a surface plot is listed below.

```python
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 7.31: Surface plot for 2D objective function

Running the example creates a three-dimensional surface plot of the objective function.
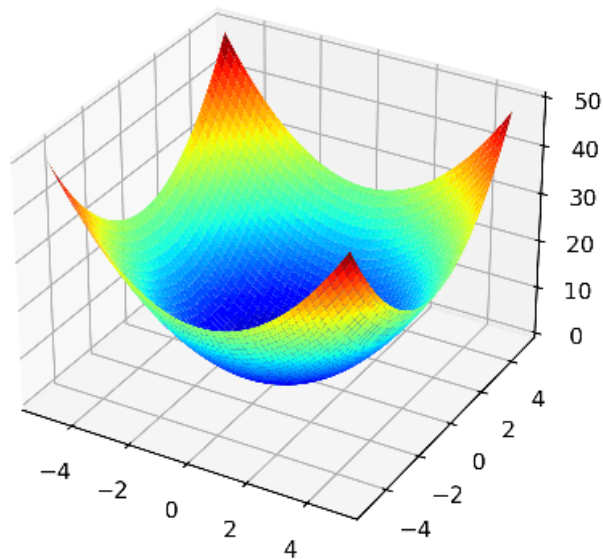
Figure 7.9: Surface Plot of a Two-Dimensional Objective Function

Additionally, the plot is interactive, meaning that you can use the mouse to drag the perspective on the surface around and view it from different angles.
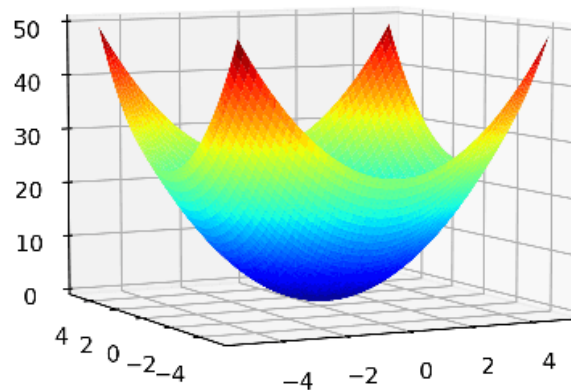
Figure 7.10: Surface Plot From a Different Angle of a Two-Dimensional Objective Function

7.5 Further Reading

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

APIs

Optimization and root finding (`scipy.optimize`*).*
    https://docs.scipy.org/doc/scipy/reference/optimize.html
Optimization (`scipy.optimize`*).*
    https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html
`numpy.meshgrid` *API.*
    https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html
`matplotlib.pyplot.contour` *API.*
    https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.contour.html
`matplotlib.pyplot.contourf` *API.*
    https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.contourf.html
`mpl_toolkits.mplot3d.Axes3D.plot_surface` *API.*
    https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html%5C#mpl_toolkits.
    mplot3d.Axes3D.plot_surface

Articles

Mathematical optimization. Wikipedia.
    https://en.wikipedia.org/wiki/Mathematical_optimization

Parabola. Wikipedia.
https://en.wikipedia.org/wiki/Parabola

7.6 Summary

In this tutorial, you discovered how to create visualizations for function optimization in Python. Specifically, you learned:

▷ Visualization is an important tool when studying function optimization algorithms.

▷ How to visualize one-dimensional functions and samples using line plots.

▷ How to visualize two-dimensional functions and samples using contour and surface plots.

Next, you will learn about the use of randomness in optimization.

Stochastic Optimization Algorithms

<div style="text-align: right; font-size: 2em;">8</div>

Stochastic optimization refers to the use of randomness in the objective function or in the optimization algorithm. Challenging optimization algorithms, such as high-dimensional nonlinear objective problems, may contain multiple local optima in which deterministic optimization algorithms may get stuck. Stochastic optimization algorithms provide an alternative approach that permits less optimal local decisions to be made within the search procedure that may increase the probability of the procedure locating the global optima of the objective function.

In this tutorial, you will discover a gentle introduction to stochastic optimization. After completing this tutorial, you will know:

▷ Stochastic optimization algorithms make use of randomness as part of the search procedure.

▷ Examples of stochastic optimization algorithms like simulated annealing and genetic algorithms.

▷ Practical considerations when using stochastic optimization algorithms such as repeated evaluations.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Is Stochastic Optimization?
2. Stochastic Optimization Algorithms
3. Practical Considerations for Stochastic Optimization

8.2 What Is Stochastic Optimization?

Optimization refers to optimization algorithms that seek the inputs to a function that result in the minimum or maximum of an objective function. Stochastic optimization or stochastic search refers to an optimization task that involves randomness in some way, such as either from the objective function or in the optimization algorithm.

> Stochastic search and optimization pertains to problems where there is randomness noise in the measurements provided to the algorithm and/or there is injected (Monte Carlo) randomness in the algorithm itself.
>
> — Page xiii, *Introduction to Stochastic Search and Optimization*, 2003.

Randomness in the objective function means that the evaluation of candidate solutions involves some uncertainty or noise and algorithms must be chosen that can make progress in the search in the presence of this noise. Randomness in the algorithm is used as a strategy, e.g. stochastic or probabilistic decisions. It is used as an alternative to deterministic decisions in an effort to improve the likelihood of locating the global optima or a better local optima.

> Standard stochastic optimization methods are brittle, sensitive to stepsize choice and other algorithmic parameters, and they exhibit instability outside of well-behaved families of objectives.
>
> — "The importance of better models in stochastic optimization", 2019.

It is more common to refer to an algorithm that uses randomness than an objective function that contains noisy evaluations when discussing stochastic optimization. This is because randomness in the objective function can be addressed by using randomness in the optimization algorithm. As such, stochastic optimization may be referred to as "*robust optimization.*" A deterministic algorithm may be misled (e.g. "*deceived*" or "*confused*") by the noisy evaluation of candidate solutions or noisy function gradients, causing the algorithm to bounce around or get stuck (i.e. fail to converge).

> Methods for stochastic optimization provide a means of coping with inherent system noise and coping with models or systems that are highly nonlinear, high dimensional, or otherwise inappropriate for classical deterministic methods of optimization.
>
> — "Stochastic Optimization", 2011.

Using randomness in an optimization algorithm allows the search procedure to perform well on challenging optimization problems that may have a nonlinear response surface. This is achieved by the algorithm taking locally suboptimal steps or moves in the search space that allow it to escape local optima.

> Randomness can help escape local optima and increase the chances of finding a global optimum.
>
> — Page 8, *Algorithms for Optimization*, 2019.

The randomness used in a stochastic optimization algorithm does not have to be true randomness; instead, pseudorandom is sufficient. A pseudorandom number generator is almost universally used in stochastic optimization. Use of randomness in a stochastic optimization algorithm does not mean that the algorithm is random. Instead, it means that some decisions made during the search procedure involve some portion of randomness. For example, we can conceptualize this as the move from the current to the next point in the search space made by the algorithm may be made according to a probability distribution relative to the optimal move.

Now that we have an idea of what stochastic optimization is, let's look at some examples of stochastic optimization algorithms.

8.3 Stochastic Optimization Algorithms

The use of randomness in the algorithms often means that the techniques are referred to as "heuristic search[1]" as they use a rough rule-of-thumb procedure that may or may not work to find the optima instead of a precise procedure. Many stochastic algorithms are inspired by a biological or natural process and may be referred to as "metaheuristics[2]" as a higher-order procedure providing the conditions for a specific search of the objective function. They are also referred to as "*black box*" optimization algorithms.

> " Metaheuristics is a rather unfortunate term often used to describe a major subfield, indeed the primary subfield, of stochastic optimization. "
>
> — Page 7, *Essentials of Metaheuristics*, 2011.

There are many stochastic optimization algorithms. Some examples of stochastic optimization algorithms include:

▷ Iterated Local Search

▷ Stochastic Hill Climbing

▷ Stochastic Gradient Descent

▷ Tabu Search

▷ Greedy Randomized Adaptive Search Procedure

Some examples of stochastic optimization algorithms that are inspired by biological or physical processes include:

▷ Simulated Annealing

▷ Evolution Strategies

▷ Genetic Algorithm

▷ Differential Evolution

▷ Particle Swarm Optimization

Now that we are familiar with some examples of stochastic optimization algorithms, let's look at some practical considerations when using them.

8.4 Practical Considerations for Stochastic Optimization

There are important considerations when using stochastic optimization algorithms. The stochastic nature of the procedure means that any single run of an algorithm will be different, given a different source of randomness used by the algorithm and, in turn, different starting points for the search and decisions made during the search. The pseudorandom number generator used as the source of randomness can be seeded to ensure the same sequence of random numbers is provided each run of the algorithm. This is good for small demonstrations and tutorials, although it is fragile as it is working against the inherent randomness of the algorithm. Instead, a given algorithm can be executed many times to control for the randomness of the procedure.

---

[1] https://en.wikipedia.org/wiki/Heuristic_(computer_science)
[2] https://en.wikipedia.org/wiki/Metaheuristic

This idea of multiple runs of the algorithm can be used in two key situations:

▷ Comparing Algorithms

▷ Evaluating Final Result

Algorithms may be compared based on the relative quality of the result found, the number of function evaluations performed, or some combination or derivation of these considerations. The result of any one run will depend upon the randomness used by the algorithm and alone cannot meaningfully represent the capability of the algorithm. Instead, a strategy of repeated evaluation should be used. Any comparison between stochastic optimization algorithms will require the repeated evaluation of each algorithm with a different source of randomness and the summarization of the probability distribution of best results found, such as the mean and standard deviation of objective values. The mean result from each algorithm can then be compared.

> In cases where multiple local minima are likely to exist, it can be beneficial to incorporate random restarts after our terminiation conditions are met where we restart our local descent method from randomly selected initial points.
>
> — Page 66, *Algorithms for Optimization*, 2019.

Similarly, any single run of a chosen optimization algorithm alone does not meaningfully represent the global optima of the objective function. Instead, a strategy of repeated evaluation should be used to develop a distribution of optimal solutions. The maximum or minimum of the distribution can be taken as the final solution, and the distribution itself will provide a point of reference and confidence that the solution found is "*relatively good*" or "*good enough*" given the resources expended.

▷ **Multi-Restart**: An approach for improving the likelihood of locating the global optima via the repeated application of a stochastic optimization algorithm to an optimization problem.

The repeated application of a stochastic optimization algorithm on an objective function is sometimes referred to as a multi-restart strategy and may be built in to the optimization algorithm itself or prescribed more generally as a procedure around the chosen stochastic optimization algorithm.

> Each time you do a random restart, the hill-climber then winds up in some (possibly new) local optimum.
>
> — Page 26, *Essentials of Metaheuristics*, 2011.

8.5 Further Reading

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

Papers

Hilal Asi and John C. Duchi. "The importance of better models in stochastic optimization". In: *Proceedings of the National Academy of Sciences*. Vol. 116. 46. 2019, pp. 22924–22930. https://www.pnas.org/content/116/46/22924

James C. Spall. "Stochastic Optimization". In: Springer, 2011.
https://link.springer.com/chapter/10.1007%2F978-3-642-21551-3_7

Books

James C. Spall. *Introduction to Stochastic Search and Optimization*. Wiley-Interscience, 2003.
https://amzn.to/34JYN7m
Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J
Sean Luke. *Essentials of Metaheuristics*. lulu.com, 2011.
https://amzn.to/3lHryZr

Articles

Stochastic optimization. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_optimization
Heuristic (computer science). Wikipedia.
https://en.wikipedia.org/wiki/Heuristic_(computer_science)
Metaheuristic. Wikipedia.
https://en.wikipedia.org/wiki/Metaheuristic

8.6 Summary

In this tutorial, you discovered a gentle introduction to stochastic optimization. Specifically, you learned:

▷ Stochastic optimization algorithms make use of randomness as part of the search procedure.

▷ Examples of stochastic optimization algorithms like simulated annealing and genetic algorithms.

▷ Practical considerations when using stochastic optimization algorithms such as repeated evaluations.

Next, you will learn about two naive optimization algorithms, the random search and grid search.

Random Search and Grid Search

9

Function optimization requires the selection of an algorithm to efficiently sample the search space and locate a good or best solution. There are many algorithms to choose from, although it is important to establish a baseline for what types of solutions are feasible or possible for a problem. This can be achieved using a naive optimization algorithm, such as a *random search* or a *grid search.*

The results achieved by a naive optimization algorithm are computationally efficient to generate and provide a point of comparison for more sophisticated optimization algorithms. Sometimes, naive algorithms are found to achieve the best performance, particularly on those problems that are noisy or non-smooth and those problems where domain expertise typically biases the choice of optimization algorithm.

In this tutorial, you will discover naive algorithms for function optimization. After completing this tutorial, you will know:

▷ The role of naive algorithms in function optimization projects.

▷ How to generate and evaluate a random search for function optimization.

▷ How to generate and evaluate a grid search for function optimization.

Let's get started.

9.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Naive Function Optimization Algorithms
2. Random Search for Function Optimization
3. Grid Search for Function Optimization

9.2   Naive Function Optimization Algorithms

There are many different algorithms you can use for optimization, but how do you know whether the results you get are any good? One approach to solving this problem is to establish a baseline

in performance using a naive optimization algorithm. A naive optimization algorithm is an algorithm that assumes nothing about the objective function that is being optimized. It can be applied with very little effort and the best result achieved by the algorithm can be used as a point of reference to compare more sophisticated algorithms. If a more sophisticated algorithm cannot achieve a better result than a naive algorithm on average, then it does not have skill on your problem and should be abandoned.

There are two naive algorithms that can be used for function optimization; they are:

▷ Random Search

▷ Grid Search

These algorithms are referred to as "*search*" algorithms because, at base, optimization can be framed as a search problem, i.e. find the inputs that minimize or maximize the output of the objective function.

There is another algorithm that can be used called "exhaustive search"[1] that enumerates all possible inputs. This is rarely used in practice as enumerating all possible inputs is not feasible, e.g. would require too much time to run. Nevertheless, if you find yourself working on an optimization problem for which all inputs can be enumerated and evaluated in reasonable time, this should be the default strategy you should use.

Let's take a closer look at each in turn.

9.3   Random Search for Function Optimization

Random search[2] is also referred to as random optimization or random sampling. Random search involves generating and evaluating random inputs to the objective function. It's effective because it does not assume anything about the structure of the objective function. This can be beneficial for problems where there is a lot of domain expertise that may influence or bias the optimization strategy, allowing non-intuitive solutions to be discovered.

> ... random sampling, which simply draws m random samples over the design space using a pseudorandom number generator. To generate a random sample x, we can sample each variable independently from a distribution.
>
> — Page 236, *Algorithms for Optimization*, 2019.

Random search may also be the best strategy for highly complex problems with noisy or non-smooth (discontinuous) areas of the search space, which can cause problems for algorithms that depend on reliable gradients. We can generate a random sample from a domain using a pseudorandom number generator. Each variable requires a well-defined bound or range and a uniformly random value can be sampled from the range, then evaluated.

Generating random samples is computationally trivial and does not take up much memory, therefore, it may be efficient to generate a large sample of inputs, then evaluate them. Each sample is independent, so samples can be evaluated in parallel if needed to accelerate the process. The example below gives an example of a simple one-dimensional minimization objective

---

[1]https://en.wikipedia.org/wiki/Brute-force_search
[2]https://en.wikipedia.org/wiki/Random_search

function and generates then evaluates a random sample of 100 inputs. The input with the best performance is then reported.

```python
from numpy.random import rand

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

# define range for input
r_min, r_max = -5.0, 5.0
# generate a random sample from the domain
sample = r_min + rand(100) * (r_max - r_min)
# evaluate the sample
sample_eval = objective(sample)
# locate the best solution
best_ix = 0
for i in range(len(sample)):
    if sample_eval[i] < sample_eval[best_ix]:
        best_ix = i
# summarize best solution
print('Best: f(%.5f) = %.5f' % (sample[best_ix], sample_eval[best_ix]))
```

Program 9.1: Example of random search for function optimization

Running the example generates a random sample of input values, which are then evaluated. The best performing point is then identified and reported.

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 result is very close to the optimal input of 0.0.

```
Best: f(-0.01762) = 0.00031
```

Output 9.1: Result from Program 9.1

We can update the example to plot the objective function and show the sample and best result. The complete example is listed below.

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

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

# define range for input
r_min, r_max = -5.0, 5.0
```

```python
# generate a random sample from the domain
sample = r_min + rand(100) * (r_max - r_min)
# evaluate the sample
sample_eval = objective(sample)
# locate the best solution
best_ix = 0
for i in range(len(sample)):
    if sample_eval[i] < sample_eval[best_ix]:
        best_ix = i
# summarize best solution
print('Best: f(%.5f) = %.5f' % (sample[best_ix], sample_eval[best_ix]))
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
# plot the sample
pyplot.scatter(sample, sample_eval)
# draw a vertical line at the best input
pyplot.axvline(x=sample[best_ix], ls='--', color='red')
# show the plot
pyplot.show()
```

Program 9.2: Example of random search for function optimization with plot

Running the example again generates the random sample and reports the best result.

```
Best: f(0.01934) = 0.00037
```

Output 9.2: Result from Program 9.2

A line plot is then created showing the shape of the objective function, the random sample, and a red line for the best result located from the sample.
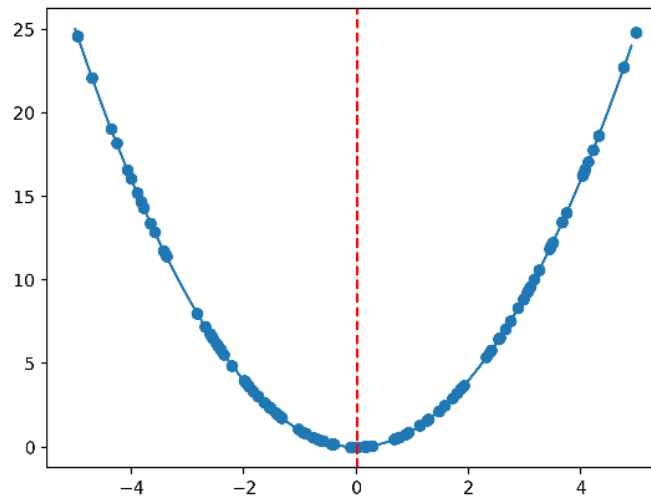
Figure 9.1: Line Plot of One-Dimensional Objective Function With Random Sample

9.4 Grid Search for Function Optimization

Grid search is also referred to as a grid sampling or full factorial sampling. Grid search involves generating uniform grid inputs for an objective function. In one-dimension, this would be inputs evenly spaced along a line. In two-dimensions, this would be a lattice of evenly spaced points across the surface, and so on for higher dimensions.

> The full factorial sampling plan places a grid of evenly spaced points over the search space. This approach is easy to implement, does not rely on randomness, and covers the space, but it uses a large number of points.
>
> — Page 235, *Algorithms for Optimization*, 2019.

Like random search, a grid search can be particularly effective on problems where domain expertise is typically used to influence the selection of specific optimization algorithms. The grid can help to quickly identify areas of a search space that may deserve more attention.

The grid of samples is typically uniform, although this does not have to be the case. For example, a log-10 scale could be used with a uniform spacing, allowing sampling to be performed across orders of magnitude. The downside is that the coarseness of the grid may step over whole regions of the search space where good solutions reside, a problem that gets worse as the number of inputs (dimensions of the search space) to the problem increases.

A grid of samples can be generated by choosing the uniform separation of points, then enumerating each variable in turn and incrementing each variable by the chosen separation. The example below gives an example of a simple two-dimensional minimization objective function and generates then evaluates a grid sample with a spacing of 0.1 for both input variables. The input with the best performance is then reported.

```python
from numpy import arange
from numpy.random import rand

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# generate a grid sample from the domain
sample = list()
step = 0.1
for x in arange(r_min, r_max+step, step):
    for y in arange(r_min, r_max+step, step):
        sample.append([x,y])
# evaluate the sample
sample_eval = [objective(x,y) for x,y in sample]
# locate the best solution
best_ix = 0
for i in range(len(sample)):
    if sample_eval[i] < sample_eval[best_ix]:
        best_ix = i
# summarize best solution
print('Best: f(%.5f,%.5f) = %.5f' % (sample[best_ix][0], sample[best_ix][1],
    sample_eval[best_ix]))
```

Program 9.3: Example of grid search for function optimization

Running the example generates a grid of input values, which are then evaluated. The best performing point is then identified and reported.

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 result finds the optima exactly.

```
Best: f(-0.00000,-0.00000) = 0.00000
```

Output 9.3: Result from Program 9.3

We can update the example to plot the objective function and show the sample and best result. The complete example is listed below.

```python
from numpy import arange
from numpy import meshgrid
from numpy.random import rand
from matplotlib import pyplot

# objective function
def objective(x, y):
```

```
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# generate a grid sample from the domain
sample = list()
step = 0.5
for x in arange(r_min, r_max+step, step):
    for y in arange(r_min, r_max+step, step):
        sample.append([x,y])
# evaluate the sample
sample_eval = [objective(x,y) for x,y in sample]
# locate the best solution
best_ix = 0
for i in range(len(sample)):
    if sample_eval[i] < sample_eval[best_ix]:
        best_ix = i
# summarize best solution
print('Best: f(%.5f,%.5f) = %.5f' % (sample[best_ix][0], sample[best_ix][1],
    sample_eval[best_ix]))
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# plot the sample as black circles
pyplot.plot([x for x,_ in sample], [y for _,y in sample], '.', color='black')
# draw the best result as a white star
pyplot.plot(sample[best_ix][0], sample[best_ix][1], '*', color='white')
# show the plot
pyplot.show()
```

Program 9.4: Example of grid search for function optimization with plot

Running the example again generates the grid sample and reports the best result.

```
Best: f(0.00000,0.00000) = 0.00000
```

Output 9.4: Result from Program 9.4

A contour plot is then created showing the shape of the objective function, the grid sample as black dots, and a white star for the best result located from the sample. Note that some of the black dots for the edge of the domain appear to be off the plot; this is just an artifact for how we are choosing to draw the dots (e.g. not centered on the sample).
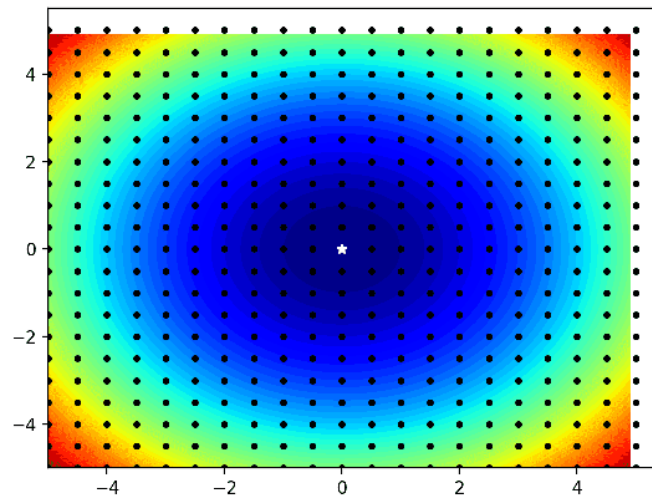
Figure 9.2: Contour Plot of One-Dimensional Objective Function With Grid Sample

9.5 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

Articles

Random search. Wikipedia.
    https://en.wikipedia.org/wiki/Random_search
Hyperparameter optimization. Wikipedia.
    https://en.wikipedia.org/wiki/Hyperparameter_optimization
Brute-force search. Wikipedia.
    https://en.wikipedia.org/wiki/Brute-force_search

9.6 Summary

In this tutorial, you discovered naive algorithms for function optimization. Specifically, you learned:

▷ The role of naive algorithms in function optimization projects.

▷ How to generate and evaluate a random search for function optimization.

▷ How to generate and evaluate a grid search for function optimization.

Next, you will see a few local optimization algorithms.

Local Optimization III

What is a Gradient in Machine Learning?

<div style="text-align: right; font-size: large">**10**</div>

Gradient is a commonly used term in optimization and machine learning. For example, deep learning neural networks are fit using stochastic gradient descent, and many standard optimization algorithms used to fit machine learning algorithms use gradient information. In order to understand what a gradient is, you need to understand what a derivative is from the field of calculus. This includes how to calculate a derivative and interpret the value. An understanding of the derivative is directly applicable to understanding how to calculate and interpret gradients as used in optimization and machine learning.

In this tutorial, you will discover a gentle introduction to the derivative and the gradient in machine learning. After completing this tutorial, you will know:

▷ The derivative of a function is the change of the function for a given input.

▷ The gradient is simply a derivative vector for a multivariate function.

▷ How to calculate and interpret derivatives of a simple function.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What is a Derivative?
2. What is a Gradient?
3. Worked Example of Calculating Derivatives
4. How to Interpret the Derivative
5. How to Calculate the Derivative of a Function

10.2 What is a Derivative?

In calculus, a derivative[1] is the rate of change at a given point in a real-valued function. For example, the derivative $f'(x)$ of function $f()$ for variable x is the rate that the function $f()$

---

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

changes at the point x. It might change a lot, i.e. be very curved, or might change a little, i.e. slight curve, or it might not change at all, i.e. flat or stationary.

A function is differentiable[2] if we can calculate the derivative at all points of input for the function variables. Not all functions are differentiable. Once we calculate the derivative, we can use it in a number of ways. For example, given an input value x and the derivative at that point $f'(x)$, we can estimate the value of the function $f(x)$ at a nearby point Δx (change in x) using the derivative, as follows:

$$f(x + \Delta x) = f(x) + f'(x) \times \Delta x$$

Here, we can see that $f'(x)$ is a line and we are estimating the value of the function at a nearby point by moving along the line by Δx. We can use derivatives in optimization problems as they tell us how to change inputs to the target function in a way that increases or decreases the output of the function, so we can get closer to the minimum or maximum of the function.

> " Derivatives are useful in optimization because they provide information about how to change a given point in order to improve the objective function. "
>
> — Page 32, *Algorithms for Optimization*, 2019.

Finding the line that can be used to approximate nearby values was the main reason for the initial development of differentiation. This line is referred to as the tangent line or the slope of the function at a given point.

> " The problem of finding the tangent line to a curve [...] involve finding the same type of limit [...] This special type of limit is called a derivative and we will see that it can be interpreted as a rate of change in any of the sciences or engineering. "
>
> — Page 104, *Calculus*, 2013.

An example of the tangent line of a point for a function is provided below, taken from page 19 of "*Algorithms for Optimization*."
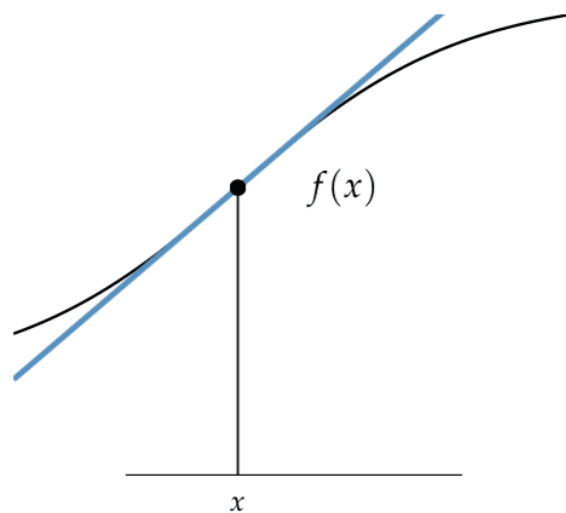
Figure 10.1: *Tangent Line of a Function at a Given Point. Taken from Algorithms for Optimization.*

---

[2]https://en.wikipedia.org/wiki/Differentiable_function

Technically, the derivative described so far is called the first derivative or first-order derivative. The second derivative[3] (or second-order derivative) is the derivative of the derivative function. That is, the rate of change of the rate of change or how much the change in the function changes.

▷ **First Derivative**: Rate of change of the target function.

▷ **Second Derivative**: Rate of change of the first derivative function.

A natural use of the second derivative is to approximate the first derivative at a nearby point, just as we can use the first derivative to estimate the value of the target function at a nearby point.

Now that we know what a derivative is, let's take a look at a gradient.

10.3   What is a Gradient?

A gradient[4] is a derivative of a function that has more than one input variable. It is a term used to refer to the derivative of a function from the perspective of the field of linear algebra. Specifically when linear algebra meets calculus, called vector calculus.

> " The gradient is the generalization of the derivative to multivariate functions. It captures the local slope of the function, allowing us to predict the effect of taking a small step from a point in any direction. "
>
> — Page 21, *Algorithms for Optimization*, 2019.

Multiple input variables together define a vector of values, i.e. a point in the input space that can be provided to the target function. The derivative of a target function with a vector of input variables similarly is a vector. This vector of derivatives for each input variable is the gradient.

▷ **Gradient (vector calculus)**: A vector of derivatives for a function that takes a vector of input variables.

You might recall from high school algebra or pre-calculus, the gradient also refers generally to the slope of a line on a two-dimensional plot. It is calculated as the rise (change on the y-axis) of the function divided by the run (change in x-axis) of the function, simplified to the rule, "*rise over run*":

▷ **Gradient (algebra):** Slope of a line, calculated as rise over run.

We can see that this is a simple and rough approximation of the derivative for a function with one variable. The derivative function from calculus is more precise as it uses limits to find the exact slope of the function at a point. This idea of gradient from algebra is related, but not directly useful to the idea of a gradient as used in optimization and machine learning. A function that takes multiple input variables, i.e. a vector of input variables, may be referred to as a multivariate function.

---

[3]https://en.wikipedia.org/wiki/Second_derivative
[4]https://en.wikipedia.org/wiki/Gradient

> The partial derivative of a function with respect to a variable is the derivative assuming all other input variables are held constant.
>
> — Page 21, *Algorithms for Optimization*, 2019.

Each component in the gradient (vector of derivatives) is called a partial derivative of the target function. A partial derivative assumes all other variables of the function are held constant.

▷ **Partial Derivative**: A derivative for one of the variables for a multivariate function.

It is useful to work with square matrices in linear algebra, and the square matrix of the second-order derivatives is referred to as the Hessian matrix[5].

> The Hessian of a multivariate function is a matrix containing all of the second derivatives with respect to the input
>
> — Page 21, *Algorithms for Optimization*, 2019.

We can use gradient and derivative interchangeably, although in the fields of optimization and machine learning, we typically use "*gradient*" as we are typically concerned with multivariate functions. Intuitions for the derivative translate directly to the gradient, only with more dimensions.

Now that we are familiar with the idea of a derivative and a gradient, let's look at a worked example of calculating derivatives.

10.4   Worked Example of Calculating Derivatives

Let's make the derivative concrete with a worked example. First, let's define a simple one-dimensional function that squares the input and defines the range of valid inputs from -1.0 to 1.0.

$$f(x) = x^2$$

The example below samples inputs from this function in 0.1 increments, calculates the function value for each input, and plots the result.

```python
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)
```

---

[5]https://en.wikipedia.org/wiki/Hessian_matrix

```
# show the plot
pyplot.show()
```

Program 10.1: 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.
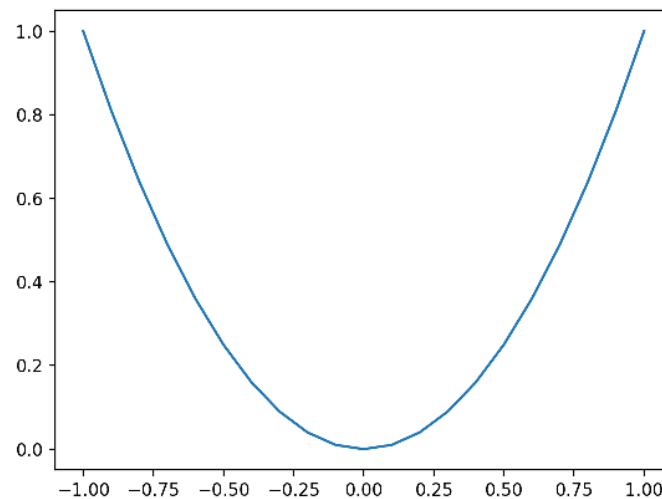
Figure 10.2: Line Plot of Simple One Dimensional Function

We can see a large change or steep curve on the sides of the shape where we would expect a large derivative and a flat area in the middle of the function where we would expect a small derivative.

Let's confirm these expectations by calculating the derivative at -0.5 and 0.5 (steep) and 0.0 (flat). The derivative for the function is calculated as follows:

$$f'(x) = 2 \times x$$

The example below calculates the derivatives for the specific input points for our objective function.

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

# calculate derivatives
d1 = derivative(-0.5)
print("f'(-0.5) = %.3f" % d1)
d2 = derivative(0.5)
print("f'(0.5) = %.3f" % d2)
d3 = derivative(0.0)
print("f'(0.0) = %.3f" % d3)
```

Program 10.2: Calculate the derivative of the objective function

Running the example prints the derivative values for specific input values. We can see that the derivative at the steep points of the function is −1 and 1 and the derivative for the flat part of the function is 0.0.

```
f'(-0.5) = -1.000
f'(0.5) = 1.000
f'(0.0) = 0.000
```

Output 10.1: Result from Program 10.2

Now that we know how to calculate derivatives of a function, let's look at how we might interpret the derivative values.

10.5   How to Interpret the Derivative

The value of the derivative can be interpreted as the rate of change (magnitude) and the direction (sign).

▷ **Magnitude of Derivative**: How much change.

▷ **Sign of Derivative**: Direction of change.

A derivative of 0.0 indicates no change in the target function, referred to as a stationary point. A function may have one or more stationary points and a local or global minimum (bottom of a valley) or maximum (peak of a mountain) of the function are examples of stationary points.

    " The gradient points in the direction of steepest ascent of the tangent hyperplane . . . "

               — Page 21, *Algorithms for Optimization*, 2019.

The sign of the derivative tells you if the target function is increasing or decreasing at that point.

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

▷ **Negative Derivative**: Function is decreasing at that point

This might be confusing because, looking at the plot from the previous section, the values of the function $f(x)$ are increasing on the y-axis for −0.5 and 0.5. The trick here is to always read the plot of the function from left to right, i.e. follow the values on the y-axis from left to right for input x-values. Indeed the values around $x = -0.5$ are decreasing if read from left to right, hence the negative derivative, and the values around $x = 0.5$ are increasing, hence the positive derivative.

We can imagine that if we wanted to find the minima of the function in the previous section using only the gradient information, we would increase the x input value if the gradient was negative to go downhill, or decrease the value of x input if the gradient was positive to go downhill. This is the basis for the gradient descent (and gradient ascent) class of optimization algorithms that have access to function gradient information.

Now that we know how to interpret derivative values, let's look at how we might find the derivative of a function.

10.6   How to Calculate the Derivative of a Function

Finding the derivative function $f'()$ that outputs the rate of change of a target function $f()$ is called differentiation. There are many approaches (algorithms) for calculating the derivative of a function. In some cases, we can calculate the derivative of a function using the tools of calculus, either manually or using an automatic solver.

General classes of techniques for calculating the derivative of a function include:

▷ Finite difference method[6]

▷ Symbolic differentiation[7]

▷ Automatic differentiation[8]

The SymPy Python library[9] can be used for symbolic differentiation. Computational libraries such as *Theano* and *TensorFlow* can be used for automatic differentiation. There are also online services you can use if your function is easy to specify in plain text. One example is the Wolfram Alpha[10] website that will calculate the derivative of the function for you; for example, entering "`Calculate the Derivative of x^2`":
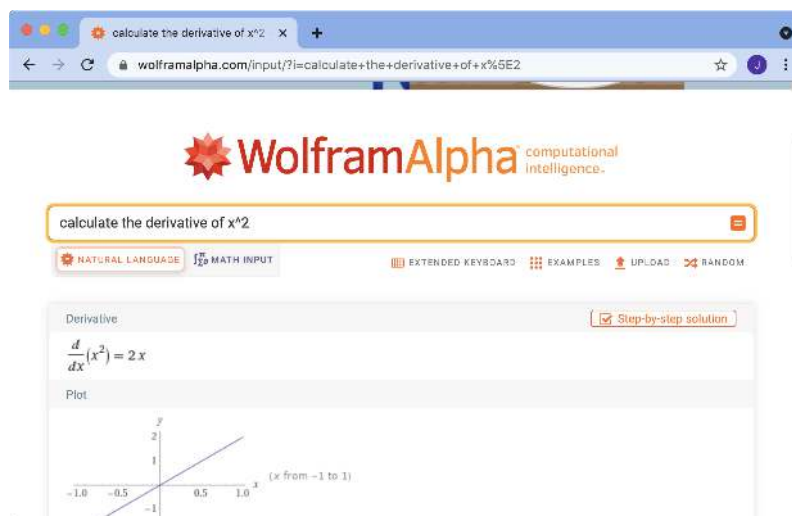
Figure 10.3: Calculate the derivative of x^2 from Wolfram Alpha

Not all functions are differentiable, and some functions that are differentiable may make it difficult to find the derivative with some methods. Calculating the derivative of a function is beyond the scope of this tutorial. Consult a good calculus textbook, such as those in the further reading section.

10.7   Further Reading

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

---

[6] https://en.wikipedia.org/wiki/Finite_differences
[7] https://en.wikipedia.org/wiki/Computer_algebra
[8] https://en.wikipedia.org/wiki/Automatic_differentiation
[9] https://www.sympy.org/
[10] https://www.wolframalpha.com

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization.* MIT Press, 2019.
    https://amzn.to/3je8O1J
James Stewart. *Calculus.* 8th ed. Cengage Learning, 2013.
    https://amzn.to/3kS9I52
Gilbert Strang. *Calculus.* 3rd ed. Wellesley-Cambridge Press, 2017.
    https://amzn.to/3fqNSEB
    (The 1991 edition is available online: https://ocw.mit.edu/resources/res-18-001-calculus-online-textbook-spring-2005/textbook/).

Articles

Derivative. Wikipedia.
    https://en.wikipedia.org/wiki/Derivative
Second derivative. Wikipedia.
    https://en.wikipedia.org/wiki/Second_derivative
Partial derivative. Wikipedia.
    https://en.wikipedia.org/wiki/Partial_derivative
Gradient. Wikipedia.
    https://en.wikipedia.org/wiki/Gradient
Differentiable function. Wikipedia.
    https://en.wikipedia.org/wiki/Differentiable_function
Jacobian matrix and determinant. Wikipedia.
    https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant
Hessian matrix. Wikipedia.
    https://en.wikipedia.org/wiki/Hessian_matrix

10.8   Summary

In this tutorial, you discovered a gentle introduction to the derivative and the gradient in machine learning. Specifically, you learned:

▷ The derivative of a function is the change of the function for a given input.

▷ The gradient is simply a derivative vector for a multivariate function.

▷ How to calculate and interpret derivatives of a simple function.

Next, we will start with the simplest example of optimizing a function with one variable.

Univariate Function Optimization

<div style="text-align: right; font-size: 3em; font-weight: bold; color: gray;">11</div>

How to Optimize a Function with One Variable?

Univariate function optimization involves finding the input to a function that results in the optimal output from an objective function. This is a common procedure in machine learning when fitting a model with one parameter or tuning a model that has a single hyperparameter. An efficient algorithm is required to solve optimization problems of this type that will find the best solution with the minimum number of evaluations of the objective function, given that each evaluation of the objective function could be computationally expensive, such as fitting and evaluating a model on a dataset. This excludes expensive grid search and random search algorithms and in favor of efficient algorithms like Brent's method.

In this tutorial, you will discover how to perform univariate function optimization in Python. After completing this tutorial, you will know:

▷ Univariate function optimization involves finding an optimal input for an objective function that takes a single continuous argument.

▷ How to perform univariate function optimization for an unconstrained convex function.

▷ How to perform univariate function optimization for an unconstrained non-convex function.

Let's get started.

11.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Univariate Function Optimization
2. Convex Univariate Function Optimization
3. Non-Convex Univariate Function Optimization

11.2   Univariate Function Optimization

We may need to find an optimal value of a function that takes a single parameter. In machine learning, this may occur in many situations, such as:

▷ Finding the coefficient of a model to fit to a training dataset.

▷ Finding the value of a single hyperparameter that results in the best model performance.

This is called univariate function optimization. We may be interested in the minimum outcome or maximum outcome of the function, although this can be simplified to minimization as a maximizing function can be made minimizing by adding a negative sign to all outcomes of the function. There may or may not be limits on the inputs to the function, so-called unconstrained or constrained optimization, and we assume that small changes in input correspond to small changes in the output of the function, i.e. it is smooth.

The function may or may not have a single optima, although we prefer that it does have a single optima and that shape of the function looks like a large basin. If this is the case, we know we can sample the function at one point and find the path down to the minima of the function. Technically, this is referred to as a convex function[1] for minimization (concave for maximization), and functions that don't have this basin shape are referred to as non-convex.

▷ **Convex Target Function**: There is a single optima and the shape of the target function leads to this optima.

Nevertheless, the target function is sufficiently complex that we don't know the derivative, meaning we cannot just use calculus to analytically compute the minimum or maximum of the function where the gradient is zero. This is referred to as a function that is non-differentiable. Although we might be able to sample the function with candidate values, we don't know the input that will result in the best outcome. This may be because of the many reasons it is expensive to evaluate candidate solutions. Therefore, we require an algorithm that efficiently samples input values to the function.

One approach to solving univariate function optimization problems is to use Brent's method[2]. Brent's method is an optimization algorithm that combines a bisecting algorithm (Dekker's method) and inverse quadratic interpolation[3]. It can be used for constrained and unconstrained univariate function optimization.

> The Brent-Dekker method is an extension of the bisection method. It is a root-finding algorithm that combines elements of the secant method and inverse quadratic interpolation. It has reliable and fast convergence properties, and it is the univariate optimization algorithm of choice in many popular numerical optimization packages.
>
> — Pages 49–51, *Algorithms for Optimization*, 2019.

Bisecting algorithms use a bracket (lower and upper) of input values and split up the input domain, bisecting it in order to locate where in the domain the optima is located, much like a binary search. Dekker's method is one way this is achieved efficiently for a continuous domain. Dekker's method gets stuck on non-convex problems. Brent's method modifies Dekker's method

---

[1]https://en.wikipedia.org/wiki/Convex_function
[2]https://en.wikipedia.org/wiki/Brent%27s_method
[3]https://en.wikipedia.org/wiki/Inverse_quadratic_interpolation

to avoid getting stuck and also approximates the second derivative of the objective function (called the Secant Method[4]) in an effort to accelerate the search. As such, Brent's method for univariate function optimization is generally preferred over most other univariate function optimization algorithms given its efficiency.

Brent's method is available in Python via the `minimize_scalar()` SciPy function[5] that takes the name of the function to be minimized. If your target function is constrained to a range, it can be specified via the "`bounds`" argument. It returns an `OptimizeResult` object[6] that is a dictionary containing the solution. Importantly, the 'x' key summarizes the input for the optima, the 'fun' key summarizes the function output for the optima, and the 'nfev' summarizes the number of evaluations of the target function that were performed.

```
...
result = minimize_scalar(objective, method='brent')
```

Program 11.1: Minimize the objective function

Now that we know how to perform univariate function optimization in Python, let's look at some examples.

11.3   Convex Univariate Function Optimization

In this section, we will explore how to solve a convex univariate function optimization problem. First, we can define a function that implements our function. In this case, we will use a simple offset version of the x^2 function, e.g. a simple parabola[7] (u-shape) function. It is a minimization objective function with an optima at -5.0.

```
def objective(x):
    return (5.0 + x)**2.0
```

Program 11.2: Objective function

We can plot a coarse grid of this function with input values from -10 to 10 to get an idea of the shape of the target function. The complete example is listed below.

```
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(x):
    return (5.0 + x)**2.0

# define range
r_min, r_max = -10.0, 10.0
# prepare inputs
```

---

[4]https://en.wikipedia.org/wiki/Secant_method
[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html
[6]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html
[7]https://en.wikipedia.org/wiki/Parabola

```
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]
# plot inputs vs target
pyplot.plot(inputs, targets, '--')
pyplot.show()
```

Program 11.3: Plot a convex target function

Running the example evaluates input values in our specified range using our target function and creates a plot of the function inputs to function outputs. We can see the U-shape of the function and that the objective is at -5.0.
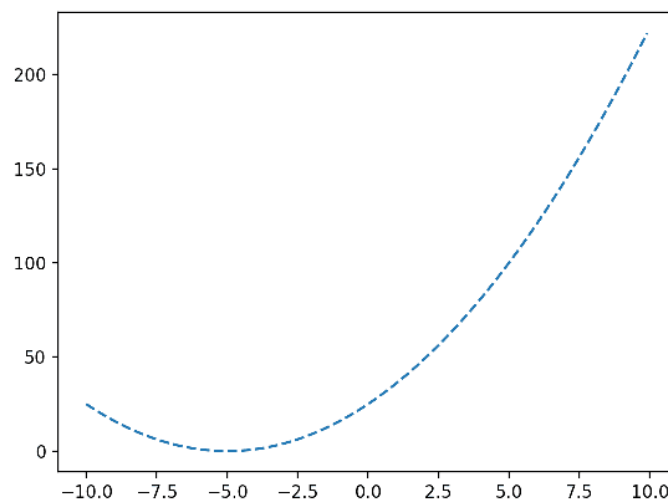
Figure 11.1: Line Plot of a Convex Objective Function

> **Note:** in a real optimization problem, we would not be able to perform so many evaluations of the objective function so easily. This simple function is used for demonstration purposes so we can learn how to use the optimization algorithm.

Next, we can use the optimization algorithm to find the optima.

```
...
result = minimize_scalar(objective, method='brent')
```

Program 11.4: Minimize the function

Once optimized, we can summarize the result, including the input and evaluation of the optima and the number of function evaluations required to locate the optima.

```
...
opt_x, opt_y = result['x'], result['fun']
print('Optimal Input x: %.6f' % opt_x)
print('Optimal Output f(x): %.6f' % opt_y)
print('Total Evaluations n: %d' % result['nfev'])
```

Program 11.5: Summarize the result

Finally, we can plot the function again and mark the optima to confirm it was located in the place we expected for this function.

```
...
# define the range
r_min, r_max = -10.0, 10.0
# prepare inputs
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]
# plot inputs vs target
pyplot.plot(inputs, targets, '--')
# plot the optima
pyplot.plot([opt_x], [opt_y], 's', color='r')
# show the plot
pyplot.show()
```

Program 11.6: Plot the function and mark the optima

The complete example of optimizing an unconstrained convex univariate function is listed below.

```
from numpy import arange
from scipy.optimize import minimize_scalar
from matplotlib import pyplot

# objective function
def objective(x):
    return (5.0 + x)**2.0

# minimize the function
result = minimize_scalar(objective, method='brent')
# summarize the result
opt_x, opt_y = result['x'], result['fun']
print('Optimal Input x: %.6f' % opt_x)
print('Optimal Output f(x): %.6f' % opt_y)
print('Total Evaluations n: %d' % result['nfev'])
# define the range
r_min, r_max = -10.0, 10.0
# prepare inputs
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]
# plot inputs vs target
pyplot.plot(inputs, targets, '--')
```

```
# plot the optima
pyplot.plot([opt_x], [opt_y], 's', color='r')
# show the plot
pyplot.show()
```

Program 11.7: Complete code to optimize a convex objective function

Running the example first solves the optimization problem and reports the result.

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 optima was located after 10 evaluations of the objective function with an input of -5.0, achieving an objective function value of 0.0.

```
Optimal Input x: -5.000000
Optimal Output f(x): 0.000000
Total Evaluations n: 10
```

Output 11.1: Result from Program 11.7

A plot of the function is created again and this time, the optima is marked as a red square.
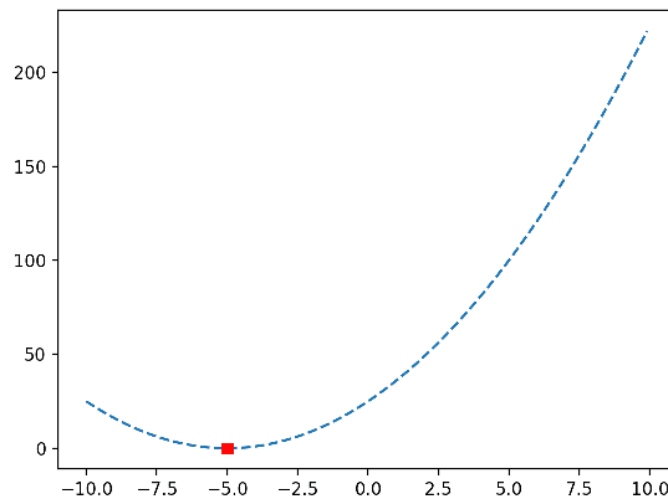
Figure 11.2: Line Plot of a Convex Objective Function with Optima Marked

11.4   Non-Convex Univariate Function Optimization

A non-convex function is one that does not resemble a basin, meaning that it may have more than one hill or valley. This can make it more challenging to locate the global optima as the

multiple hills and valleys can cause the search to get stuck and report a false or local optima instead. We can define a non-convex univariate function as follows.

```python
def objective(x):
    return (x - 2.0) * x * (x + 2.0)**2.0
```

Program 11.8: Objective function

We can sample this function and create a line plot of input values to objective values. The complete example is listed below.

```python
from numpy import arange
from matplotlib import pyplot

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

# define range
r_min, r_max = -3.0, 2.5
# prepare inputs
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]
# plot inputs vs target
pyplot.plot(inputs, targets, '--')
pyplot.show()
```

Program 11.9: Plot a non-convex univariate function

Running the example evaluates input values in our specified range using our target function and creates a plot of the function inputs to function outputs. We can see a function with one false optima around -2.0 and a global optima around 1.2.
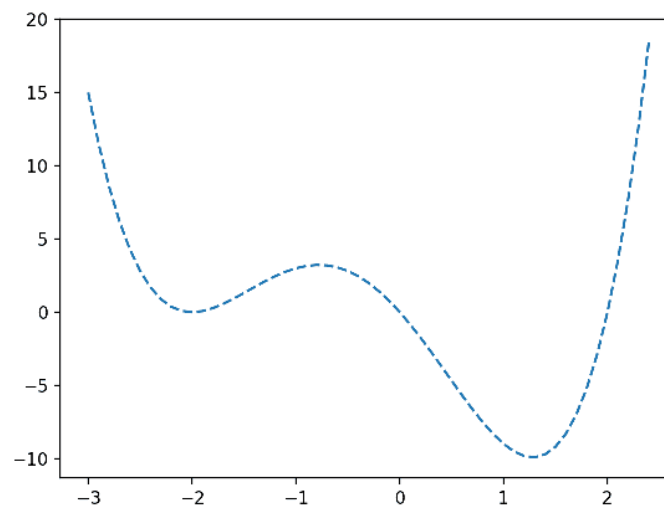
Figure 11.3: Line Plot of a Non-Convex Objective Function

Note: in a real optimization problem, we would not be able to perform so many evaluations of the objective function so easily. This simple function is used for demonstration purposes so we can learn how to use the optimization algorithm.

Next, we can use the optimization algorithm to find the optima. As before, we can call the `minimize_scalar()` function[8] to optimize the function, then summarize the result and plot the optima on a line plot.

The complete example of optimization of an unconstrained non-convex univariate function is listed below.

```python
from numpy import arange
from scipy.optimize import minimize_scalar
from matplotlib import pyplot

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

# minimize the function
result = minimize_scalar(objective, method='brent')
# summarize the result
opt_x, opt_y = result['x'], result['fun']
print('Optimal Input x: %.6f' % opt_x)
print('Optimal Output f(x): %.6f' % opt_y)
print('Total Evaluations n: %d' % result['nfev'])
# define the range
r_min, r_max = -3.0, 2.5
# prepare inputs
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]
# plot inputs vs target
pyplot.plot(inputs, targets, '--')
# plot the optima
pyplot.plot([opt_x], [opt_y], 's', color='r')
# show the plot
pyplot.show()
```

Program 11.10: Optimize non-convex objective function

Running the example first solves the optimization problem and reports the result. In this case, we can see that the optima was located after 15 evaluations of the objective function with an input of about 1.28, achieving an objective function value of about -9.91.

```
Optimal Input x: 1.280776
Optimal Output f(x): -9.914950
Total Evaluations n: 15
```

Output 11.2: Result from Program 11.10

---

[8]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html

A plot of the function is created again, and this time, the optima is marked as a red square. We can see that the optimization was not deceived by the false optima and successfully located the global optima.
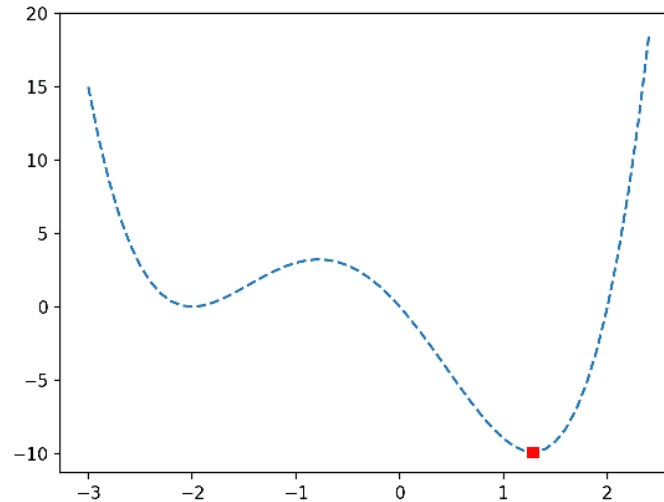
Figure 11.4: Line Plot of a Non-Convex Objective Function with Optima Marked

11.5 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

APIs

Optimization (`scipy.optimize`*).*
https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html
Optimization and root finding (`scipy.optimize`*).*
https://docs.scipy.org/doc/scipy/reference/optimize.html
`scipy.optimize.minimize_scalar` *API.*
https : / / docs . scipy . org / doc / scipy / reference / generated / scipy . optimize .
minimize_scalar.html

Articles

Brent's method. Wikipedia.
https://en.wikipedia.org/wiki/Brent%5C%27s_method

Secant method. Wikipedia.
https://en.wikipedia.org/wiki/Secant_method

11.6 Summary

In this tutorial, you discovered how to perform univariate function optimization in Python. Specifically, you learned:

▷ Univariate function optimization involves finding an optimal input for an objective function that takes a single continuous argument.

▷ How to perform univariate function optimization for an unconstrained convex function.

▷ How to perform univariate function optimization for an unconstrained non-convex function.

Next, you will be introduced to an optimization algorithm that do not need to use a gradient.

Pattern Search: The Nelder-Mead Optimization Algorithm

12

The *Nelder-Mead optimization* algorithm is a widely used approach for non-differentiable objective functions. As such, it is generally referred to as a pattern search algorithm and is used as a local or global search procedure, challenging nonlinear and potentially noisy and multimodal function optimization problems.

In this tutorial, you will discover the Nelder-Mead optimization algorithm. After completing this tutorial, you will know:

▷ The Nelder-Mead optimization algorithm is a type of pattern search that does not use function gradients.

▷ How to apply the Nelder-Mead algorithm for function optimization in Python.

▷ How to interpret the results of the Nelder-Mead algorithm on noisy and multimodal objective functions.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Nelder-Mead Algorithm
2. Nelder-Mead Example in Python
3. Nelder-Mead on Challenging Functions
   (a) Noisy Optimization Problem
   (b) Multimodal Optimization Problem

12.2   Nelder-Mead Algorithm

Nelder-Mead[1] is an optimization algorithm named after the developers of the technique, John Nelder[2] and Roger Mead[3]. The algorithm was described in their 1965 paper titled "A Simplex Method for Function Minimization"[4] and has become a standard and widely used technique for function optimization. It is appropriate for one-dimensional or multidimensional functions with numerical inputs.

Nelder-Mead is a pattern search optimization algorithm[5], which means it does not require or use function gradient information and is appropriate for optimization problems where the gradient of the function is unknown or cannot be reasonably computed. It is often used for multidimensional nonlinear function optimization problems, although it can get stuck in local optima.

> Practical performance of the Nelder-Mead algorithm is often reasonable, though stagnation has been observed to occur at nonoptimal points. Restarting can be used when stagnation is detected.
>
> — Page 239, *Numerical Optimization*, 2006.

A starting point must be provided to the algorithm, which may be the endpoint of another global optimization algorithm or a random point drawn from the domain. Given that the algorithm may get stuck, it may benefit from multiple restarts with different starting points.

> The Nelder-Mead simplex method uses a simplex to traverse the space in search of a minimum.
>
> — Page 105, *Algorithms for Optimization*, 2019).

The algorithm works by using a shape structure (called a simplex) composed of $n + 1$ points (vertices), where n is the number of input dimensions to the function. For example, on a two-dimensional problem that may be plotted as a surface, the shape structure would be composed of three points represented as a triangle.

> The Nelder-Mead method uses a series of rules that dictate how the simplex is updated based on evaluations of the objective function at its vertices.
>
> — Page 106, *Algorithms for Optimization*, 2019.

The points of the shape structure are evaluated and simple rules are used to decide how to move the points of the shape based on their relative evaluation. This includes operations such as "*reflection*," "*expansion*," "*contraction*," and "*shrinkage*" of the simplex shape on the surface of the objective function.

> In a single iteration of the Nelder-Mead algorithm, we seek to remove the vertex with the worst function value and replace it with another point with a better value. The new point is obtained by reflecting, expanding, or contracting the simplex along the line joining the worst vertex with the centroid of the remaining vertices. If we

---

[1]https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method
[2]https://en.wikipedia.org/wiki/John_Nelder
[3]https://en.wikipedia.org/wiki/Roger_Mead
[4]https://academic.oup.com/comjnl/article-abstract/7/4/308/354237
[5]https://en.wikipedia.org/wiki/Pattern_search_(optimization)

> cannot find a better point in this manner, we retain only the vertex with the best function value, and we shrink the simplex by moving all other vertices toward this value.

> — Page 238, *Numerical Optimization*, 2006.

The search stops when the points converge on an optimum, when a minimum difference between evaluations is observed, or when a maximum number of function evaluations are performed. Now that we have a high-level idea of how the algorithm works, let's look at how we might use it in practice.

12.3   Nelder-Mead Example in Python

The Nelder-Mead optimization algorithm can be used in Python via the `minimize()` function[6]. This function requires that the "`method`" argument be set to `"nelder-mead"` to use the Nelder-Mead algorithm. It takes the objective function to be minimized and an initial point for the search.

```
...
result = minimize(objective, pt, method='nelder-mead')
```

Program 12.1: Perform the search

The result is an `OptimizeResult`[7] object that contains information about the result of the optimization accessible via keys. For example, the "`success`" boolean indicates whether the search was completed successfully or not, the "`message`" provides a human-readable message about the success or failure of the search, and the "`nfev`" key indicates the number of function evaluations that were performed. Importantly, the "`x`" key specifies the input values that indicate the optima found by the search, if successful.

```
...
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
print('Solution: %s' % result['x'])
```

Program 12.2: Summarize the result

We can demonstrate the Nelder-Mead optimization algorithm on a well-behaved function to show that it can quickly and efficiently find the optima without using any derivative information from the function. In this case, we will use the x^2 function in two-dimensions, defined in the range -5.0 to 5.0 with the known optima at $[0.0, 0.0]$. We can define the `objective()` function below.

```
def objective(x):
    return x[0]**2.0 + x[1]**2.0
```

Program 12.3: Objective function

---

[6]https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html
[7]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

We will use a random point in the defined domain as a starting point for the search.

```
...
r_min, r_max = -5.0, 5.0
# define the starting point as a random sample from the domain
pt = r_min + rand(2) * (r_max - r_min)
```

Program 12.4: Define range for input

The search can then be performed. We use the default maximum number of function evaluations set via the "`maxiter`" and set to `N*200`, where `N` is the number of input variables, which is two in this case, i.e. 400 evaluations.

```
...
result = minimize(objective, pt, method='nelder-mead')
```

Program 12.5: Perform the search

After the search is finished, we will report the total function evaluations used to find the optima and the success message of the search, which we expect to be positive in this case.

```
...
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
```

Program 12.6: Summarize the result

Finally, we will retrieve the input values for located optima, evaluate it using the objective function, and report both in a human-readable manner.

```
...
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 12.7: Evaluate solution

Tying this together, the complete example of using the Nelder-Mead optimization algorithm on a simple convex objective function is listed below.

```
from scipy.optimize import minimize
from numpy.random import rand

# objective function
def objective(x):
    return x[0]**2.0 + x[1]**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# define the starting point as a random sample from the domain
pt = r_min + rand(2) * (r_max - r_min)
# perform the search
result = minimize(objective, pt, method='nelder-mead')
```

```
# summarize the result
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
# evaluate solution
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 12.8: Nelder-Mead optimization of a convex function

Running the example executes the optimization, then reports 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.

In this case, we can see that the search was successful, as we expected, and was completed after 88 function evaluations. We can see that the optima was located with inputs very close to $[0, 0]$, which evaluates to the minimum objective value of 0.0.

```
Status: Optimization terminated successfully.
Total Evaluations: 88
Solution: f([ 2.25680716e-05 -3.87021351e-05]) = 0.00000
```

Output 12.1: Result from Program 12.8

Now that we have seen how to use the Nelder-Mead optimization algorithm successfully, let's look at some examples where it does not perform so well.

12.4 Nelder-Mead on Challenging Functions

The Nelder-Mead optimization algorithm works well for a range of challenging nonlinear and non-differentiable objective functions. Nevertheless, it can get stuck on multimodal optimization problems and noisy problems. To make this concrete, let's look at an example of each.

12.4.1 Noisy Optimization Problem

A noisy objective function is a function that gives different answers each time the same input is evaluated. We can make an objective function artificially noisy by adding small Gaussian random numbers to the inputs prior to their evaluation. For example, we can define a one-dimensional version of the x^2 function and use the randn() function[8] to add small Gaussian random numbers to the input with a mean of 0.0 and a standard deviation of 0.3.

```
def objective(x):
    return (x + randn(len(x))*0.3)**2.0
```

Program 12.9: Objective function

---

[8] https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html

The noise will make the function challenging to optimize for the algorithm and it will very likely not locate the optima at $x = 0.0$. The complete example of using Nelder-Mead to optimize the noisy objective function is listed below.

```python
from scipy.optimize import minimize
from numpy.random import rand
from numpy.random import randn

# objective function
def objective(x):
    return (x + randn(len(x))*0.3)**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# define the starting point as a random sample from the domain
pt = r_min + rand(1) * (r_max - r_min)
# perform the search
result = minimize(objective, pt, method='nelder-mead')
# summarize the result
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
# evaluate solution
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 12.10: Nelder-Mead optimization of noisy one-dimensional convex function

Running the example executes the optimization, then reports 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.

In this case, the algorithm does not converge and instead uses the maximum number of function evaluations, which is 200.

```
Status: Maximum number of function evaluations has been exceeded.
Total Evaluations: 200
Solution: f([-0.6918238]) = 0.79431
```

Output 12.2: Result from Program 12.10

The algorithm may converge on some runs of the code but will arrive on a point away from the optima.

12.4.2 Multimodal Optimization Problem

Many nonlinear objective functions may have multiple optima, referred to as multimodal problems. The problem may be structured such that it has multiple global optima that have

an equivalent function evaluation, or a single global optima and multiple local optima where algorithms like the Nelder-Mead can get stuck in search of the local optima.

The Ackley function[9] is an example of the latter. It is a two-dimensional objective function that has a global optima at $[0, 0]$ but has many local optima. The example below implements the Ackley and creates a three-dimensional plot showing the global optima and multiple local optima.

```python
from numpy import arange
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi
        * x) + cos(2 * pi * y))) + e + 20

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 12.11: Ackley multimodal function

Running the example creates the surface plot of the Ackley function showing the vast number of local optima.

---

[9]https://en.wikipedia.org/wiki/Ackley_function
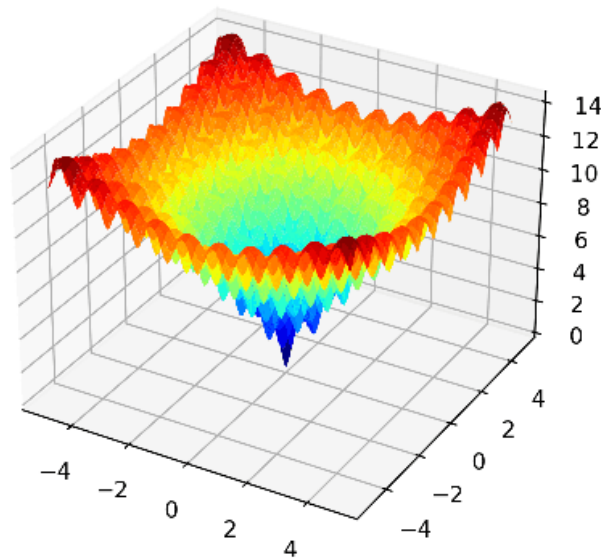
Figure 12.1: 3D Surface Plot of the Ackley Multimodal Function

We would expect the Nelder-Mead function to get stuck in one of the local optima while in search of the global optima. Initially, when the simplex is large, the algorithm may jump over many local optima, but as it contracts, it will get stuck. We can explore this with the example below that demonstrates the Nelder-Mead algorithm on the Ackley function.

```python
from scipy.optimize import minimize
from numpy.random import rand
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi

# objective function
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi ⤸
        ↳ * x) + cos(2 * pi * y))) + e + 20

# define range for input
r_min, r_max = -5.0, 5.0
# define the starting point as a random sample from the domain
pt = r_min + rand(2) * (r_max - r_min)
# perform the search
result = minimize(objective, pt, method='nelder-mead')
# summarize the result
```

```python
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
# evaluate solution
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 12.12: Nelder-Mead for multimodal function optimization

Running the example executes the optimization, then reports 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.

In this case, we can see that the search completed successfully but did not locate the global optima. It got stuck and found a local optima. Each time we run the example, we will find a different local optima given the different random starting point for the search.

```
Status: Optimization terminated successfully.
Total Evaluations: 62
Solution: f([-4.9831427 -3.98656015]) = 11.90126
```

Output 12.3: Result from Program 12.12

12.5   Further Reading

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

Papers

J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization". *The Computer Journal*, 7(4), 1965, pp. 308–313.
https://academic.oup.com/comjnl/article-abstract/7/4/308/354237

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J
Jorge Nocedal and Stephen Wright. *Numerical Optimization*. 2nd ed. Springer, 2006.
https://amzn.to/3sbjF2t

APIs

*Nelder-Mead Simplex algorithm (*method='Nelder-Mead'*).*
https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html%5C#nelder-mead-simplex-algorithm-method-nelder-mead

`scipy.optimize.minimize` *API.*
    https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html
`scipy.optimize.OptimizeResult` *API.*
    https : / / docs . scipy . org / doc / scipy / reference / generated / scipy . optimize .
    OptimizeResult.html
`numpy.random.randn` *API.*
    https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.
    html

Articles

Nelder-Mead method. Wikipedia.
    https://en.wikipedia.org/wiki/Nelder%5C%E2%5C%80%5C%93Mead_method
Nelder-Mead algorithm. Scholarpedia.
    http://www.scholarpedia.org/article/Nelder-Mead_algorithm

12.6   Summary

In this tutorial, you discovered the Nelder-Mead optimization algorithm. Specifically, you learned:

▷ The Nelder-Mead optimization algorithm is a type of pattern search that does not use function gradients.

▷ How to apply the Nelder-Mead algorithm for function optimization in Python.

▷ How to interpret the results of the Nelder-Mead algorithm on noisy and multimodal objective functions.

Next, you will learn about an algorithm that uses second-order derivatives.

Second Order: The BFGS and L-BFGS-B Optimization Algorithms

<div style="text-align: right; font-size: 3em;">**13**</div>

The Broyden, Fletcher, Goldfarb, and Shanno, or *BFGS Algorithm*, is a local search optimization algorithm. It is a type of second-order optimization algorithm, meaning that it makes use of the second-order derivative of an objective function and belongs to a class of algorithms referred to as Quasi-Newton methods that approximate the second derivative (called the Hessian) for optimization problems where the second derivative cannot be calculated. The BFGS algorithm is perhaps one of the most widely used second-order algorithms for numerical optimization and is commonly used to fit machine learning algorithms such as the logistic regression algorithm.

In this tutorial, you will discover the BFGS second-order optimization algorithm. After completing this tutorial, you will know:

▷ Second-order optimization algorithms are algorithms that make use of the second-order derivative, called the Hessian matrix for multivariate objective functions.

▷ The BFGS algorithm is perhaps the most popular second-order algorithm for numerical optimization and belongs to a group called Quasi-Newton methods.

▷ How to minimize objective functions using the BFGS and L-BFGS-B algorithms in Python.

Let's get started.

13.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Second-Order Optimization Algorithms
2. BFGS Optimization Algorithm
3. Worked Example of BFGS

13.2   Second-Order Optimization Algorithms

Optimization involves finding values for input parameters that maximize or minimize an objective function. Newton-method optimization algorithms are those algorithms that make use of the

second derivative of the objective function. You may recall from calculus that the first derivative[1] of a function is the rate of change or curvature of the function at a specific point. The derivative can be followed downhill (or uphill) by an optimization algorithm toward the minima of the function (the input values that result in the smallest output of the objective function).

Algorithms that make use of the first derivative are called first-order optimization algorithms. An example of a first-order algorithm is the gradient descent optimization algorithm.

▷ **First-Order Methods**: Optimization algorithms that make use of the first-order derivative to find the optima of an objective function.

The second-order derivative[2] is the derivative of the derivative, or the rate of change of the rate of change. The second derivative can be followed to more efficiently locate the optima of the objective function. This makes sense more generally, as the more information we have about the objective function, the easier it may be to optimize it. The second-order derivative allows us to know both which direction to move (like the first-order) but also estimate how far to move in that direction, called the step size.

> Second-order information, on the other hand, allows us to make a quadratic approximation of the objective function and approximate the right step size to reach a local minimum . . .
>
> — Page 87, *Algorithms for Optimization*, 2019.

Algorithms that make use of the second-order derivative are referred to as second-order optimization algorithms.

▷ **Second-Order Methods**: Optimization algorithms that make use of the second-order derivative to find the optima of an objective function.

An example of a second-order optimization algorithm is Newton's method. When an objective function has more than one input variable, the input variables together may be thought of as a vector, which may be familiar from linear algebra.

> The gradient is the generalization of the derivative to multivariate functions. It captures the local slope of the function, allowing us to predict the effect of taking a small step from a point in any direction.
>
> — Page 21, *Algorithms for Optimization*, 2019.

Similarly, the first derivative of multiple input variables may also be a vector, where each element is called a partial derivative. This vector of partial derivatives is referred to as the gradient.

▷ **Gradient**: Vector of partial first derivatives for multiple input variables of an objective function.

This idea generalizes to the second-order derivatives of the multivariate inputs, which is a matrix containing the second derivatives called the Hessian matrix.

▷ **Hessian**: Matrix of partial second-order derivatives for multiple input variables of an objective function.

---

[1] https://en.wikipedia.org/wiki/Derivative
[2] https://en.wikipedia.org/wiki/Second_derivative

The Hessian matrix is square and symmetric if the second derivatives are all continuous at the point where we are calculating the derivatives. This is often the case when solving real-valued optimization problems and an expectation when using many second-order methods.

> The Hessian of a multivariate function is a matrix containing all of the second derivatives with respect to the input. The second derivatives capture information about the local curvature of the function.

> — Page 21, *Algorithms for Optimization*, 2019.

As such, it is common to describe second-order optimization algorithms making use of or following the Hessian to the optima of the objective function. Now that we have a high-level understanding of second-order optimization algorithms, let's take a closer look at the BFGS algorithm.

13.3   BFGS Optimization Algorithm

BFGS is a second-order optimization algorithm. It is an acronym, named for the four co-discovers of the algorithm: Broyden, Fletcher, Goldfarb, and Shanno. It is a local search algorithm, intended for convex optimization problems with a single optima. The BFGS algorithm is perhaps best understood as belonging to a group of algorithms that are an extension to Newton's Method optimization algorithm, referred to as Quasi-Newton Methods.

Newton's method is a second-order optimization algorithm that makes use of the Hessian matrix. A limitation of Newton's method is that it requires the calculation of the inverse of the Hessian matrix. This is a computationally expensive operation and may not be stable depending on the properties of the objective function. Quasi-Newton methods are second-order optimization algorithms that approximate the inverse of the Hessian matrix using the gradient, meaning that the Hessian and its inverse do not need to be available or calculated precisely for each step of the algorithm.

> Quasi-Newton methods are among the most widely used methods for nonlinear optimization. They are incorporated in many software libraries, and they are effective in solving a wide variety of small to midsize problems, in particular when the Hessian is hard to compute.

> — Page 411, *Linear and Nonlinear Optimization*, 2009.

The main difference between different Quasi-Newton optimization algorithms is the specific way in which the approximation of the inverse Hessian is calculated. The BFGS algorithm is one specific way for updating the calculation of the inverse Hessian, instead of recalculating it every iteration. It, or its extensions, may be one of the most popular Quasi-Newton or even second-order optimization algorithms used for numerical optimization.

> The most popular quasi-Newton algorithm is the BFGS method, named for its discoverers Broyden, Fletcher, Goldfarb, and Shanno.

> — Page 136, *Numerical Optimization*, 2006.

A benefit of using the Hessian, when available, is that it can be used to determine both the direction and the step size to move in order to change the input parameters to minimize (or maximize) the objective function. Quasi-Newton methods like BFGS approximate the inverse

Hessian, which can then be used to determine the direction to move, but we no longer have the step size.

The BFGS algorithm addresses this by using a line search in the chosen direction to determine how far to move in that direction. For the derivation and calculations used by the BFGS algorithm, I recommend the resources in the further reading section at the end of this tutorial. The size of the Hessian and its inverse is proportional to the number of input parameters to the objective function. As such, the size of the matrix can become very large for hundreds, thousand, or millions of parameters.

> " . . . the BFGS algorithm must store the inverse Hessian matrix, M, that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters. "

— Page 317, *Deep Learning*, 2016.

Limited Memory BFGS[3] (or L-BFGS) is an extension to the BFGS algorithm that addresses the cost of having a large number of parameters. It does this by not requiring that the entire approximation of the inverse matrix be stored, by assuming a simplification of the inverse Hessian in the previous iteration of the algorithm (used in the approximation).

Now that we are familiar with the BFGS algorithm from a high-level, let's look at how we might make use of it.

13.4   Worked Example of BFGS

In this section, we will look at some examples of using the BFGS optimization algorithm. We can implement the BFGS algorithm for optimizing arbitrary functions in Python using the `minimize()` SciPy function[4]. The function takes a number of arguments, but most importantly, we can specify the name of the objective function as the first argument, the starting point for the search as the second argument, and specify the "`method`" argument as `"BFGS"`. The name of the function used to calculate the derivative of the objective function can be specified via the "`jac`" argument.

```
...
result = minimize(objective, pt, method='BFGS', jac=derivative)
```
Program 13.1: Perform the BFGS algorithm search

Let's look at an example.

First, we can define a simple two-dimensional objective function, a bowl function, e.g. x^2. It is simple the sum of the squared input variables with an optima at $f(0,0) = 0.0$.

```
def objective(x):
    return x[0]**2.0 + x[1]**2.0
```
Program 13.2: Objective function

---

[3]https://en.wikipedia.org/wiki/Limited-memory_BFGS
[4]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

Next, let's define a function for the derivative of the function, which is $[2x, 2y]$.

```python
def derivative(x):
    return [x[0] * 2, x[1] * 2]
```

Program 13.3: Derivative of the objective function

We will define the bounds of the function as a box with the range -5 and 5 in each dimension.

```python
...
r_min, r_max = -5.0, 5.0
```

Program 13.4: Define range for input

The starting point of the search will be a randomly generated position in the search domain.

```python
...
pt = r_min + rand(2) * (r_max - r_min)
```

Program 13.5: Define the starting point as a random sample from the domain

We can then apply the BFGS algorithm to find the minima of the objective function by specifying the name of the objective function, the initial point, the method we want to use (BFGS), and the name of the derivative function.

```python
...
result = minimize(objective, pt, method='BFGS', jac=derivative)
```

Program 13.6: Perform the BFGS algorithm search

We can then review the result reporting a message as to whether the algorithm finished successfully or not and the total number of evaluations of the objective function that were performed.

```python
...
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
```

Program 13.7: Summarize the result

Finally, we can report the input variables that were found and their evaluation against the objective function.

```python
...
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 13.8: Evaluate solution

Tying this together, the complete example is listed below.

```python
from scipy.optimize import minimize
from numpy.random import rand

# objective function
def objective(x):
    return x[0]**2.0 + x[1]**2.0

# derivative of the objective function
def derivative(x):
    return [x[0] * 2, x[1] * 2]

# define range for input
r_min, r_max = -5.0, 5.0
# define the starting point as a random sample from the domain
pt = r_min + rand(2) * (r_max - r_min)
# perform the bfgs algorithm search
result = minimize(objective, pt, method='BFGS', jac=derivative)
# summarize the result
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
# evaluate solution
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 13.9: BFGS algorithm local optimization of a convex function

Running the example applies the BFGS algorithm to our objective function and reports 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.

In this case, we can see that four iterations of the algorithm were performed and a solution very close to the optima $f(0.0, 0.0) = 0.0$ was discovered, at least to a useful level of precision.

```
Status: Optimization terminated successfully.
Total Evaluations: 4
Solution: f([0.00000000e+00 1.11022302e-16]) = 0.00000
```

Output 13.1: Result from Program 13.9

The `minimize()` function also supports the L-BFGS algorithm that has lower memory requirements than BFGS. Specifically, the L-BFGS-B version of the algorithm where the -B suffix indicates a "*boxed*" version of the algorithm, where the bounds of the domain can be specified. This can be achieved by specifying the "`method`" argument as `"L-BFGS-B"`.

```python
...
result = minimize(objective, pt, method='L-BFGS-B', jac=derivative)
```

Program 13.10: Perform the L-BFGS-B algorithm search

The complete example with this update is listed below.

```python
from scipy.optimize import minimize
from numpy.random import rand

# objective function
def objective(x):
    return x[0]**2.0 + x[1]**2.0

# derivative of the objective function
def derivative(x):
    return [x[0] * 2, x[1] * 2]

# define range for input
r_min, r_max = -5.0, 5.0
# define the starting point as a random sample from the domain
pt = r_min + rand(2) * (r_max - r_min)
# perform the l-bfgs-b algorithm search
result = minimize(objective, pt, method='L-BFGS-B', jac=derivative)
# summarize the result
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
# evaluate solution
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```

Program 13.11: L-BFGS-B algorithm local optimization of a convex function

Running the example application applies the L-BFGS-B algorithm to our objective function and reports 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.

Again, we can see that the minima to the function is found in very few evaluations.

```
Status : b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
Total Evaluations: 3
Solution: f([-1.33226763e-15 1.33226763e-15]) = 0.00000
```

Output 13.2: Result from Program 13.11

It might be a fun exercise to increase the dimensions of the test problem to millions of parameters and compare the memory usage and run time of the two algorithms.

13.5   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

Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization*. 2nd ed. SIAM, 2009.
https://amzn.to/39fWKtS

Jorge Nocedal and Stephen Wright. *Numerical Optimization*. 2nd ed. Springer, 2006.
https://amzn.to/3sbjF2t

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://amzn.to/3qSk3C2

APIs

scipy.optimize.minimize *API*.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

Articles

Broyden-Fletcher-Goldfarb-Shanno algorithm. Wikipedia.
https://en.wikipedia.org/wiki/Broyden%5C%E2%5C%80%5C%93Fletcher%5C%E2%5C%80%5C%93Goldfarb%5C%E2%5C%80%5C%93Shanno_algorithm

Limited-memory BFGS. Wikipedia.
https://en.wikipedia.org/wiki/Limited-memory_BFGS

13.6 Summary

In this tutorial, you discovered the BFGS second-order optimization algorithm. Specifically, you learned:

▷ Second-order optimization algorithms are algorithms that make use of the second-order derivative, called the Hessian matrix for multivariate objective functions.

▷ The BFGS algorithm is perhaps the most popular second-order algorithm for numerical optimization and belongs to a group called Quasi-Newton methods.

▷ How to minimize objective functions using the BFGS and L-BFGS-B algorithms in Python.

Next, you will learn about curve fitting.

Least Square: Curve Fitting with SciPy

<div align="right" style="font-size:3em">14</div>

Curve fitting is a type of optimization that finds an optimal set of parameters for a defined function that best fits a given set of observations. Unlike supervised learning, curve fitting requires that you define the function that maps examples of inputs to outputs. The mapping function, also called the basis function can have any form you like, including a straight line (linear regression), a curved line (polynomial regression), and much more. This provides the flexibility and control to define the form of the curve, where an optimization process is used to find the specific optimal parameters of the function.

In this tutorial, you will discover how to perform curve fitting in Python. After completing this tutorial, you will know:

▷ Curve fitting involves finding the optimal parameters to a function that maps examples of inputs to outputs.

▷ The SciPy Python library provides an API to fit a curve to a dataset.

▷ How to use curve fitting in SciPy to fit a range of different curves to a set of observations.

Let's get started.

14.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Curve Fitting
2. Curve Fitting Python API
3. Curve Fitting Worked Example

14.2 Curve Fitting

Curve fitting[1] is an optimization problem that finds a line that best fits a collection of observations. It is easiest to think about curve fitting in two dimensions, such as a graph.

---

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

Consider that we have collected examples of data from the problem domain with inputs and outputs. The x-axis is the independent variable or the input to the function. The y-axis is the dependent variable or the output of the function. We don't know the form of the function that maps examples of inputs to outputs, but we suspect that we can approximate the function with a standard function form.

Curve fitting involves first defining the functional form of the mapping function (also called the basis function[2] or objective function), then searching for the parameters to the function that result in the minimum error. Error is calculated by using the observations from the domain and passing the inputs to our candidate mapping function and calculating the output, then comparing the calculated output to the observed output. Once fit, we can use the mapping function to interpolate or extrapolate new points in the domain. It is common to run a sequence of input values through the mapping function to calculate a sequence of outputs, then create a line plot of the result to show how output varies with input and how well the line fits the observed points.

The key to curve fitting is the form of the mapping function. A straight line between inputs and outputs can be defined as follows:

$$y = a \times x + b$$

Where y is the calculated output, x is the input, and a and b are parameters of the mapping function found using an optimization algorithm. This is called a linear equation because it is a weighted sum of the inputs. In a linear regression model, these parameters are referred to as coefficients; in a neural network, they are referred to as weights.

This equation can be generalized to any number of inputs, meaning that the notion of curve fitting is not limited to two-dimensions (one input and one output), but could have many input variables. For example, a line mapping function for two input variables may look as follows:

$$y = a_1 \times x_1 + a_2 \times x_2 + b$$

The equation does not have to be a straight line. We can add curves in the mapping function by adding exponents. For example, we can add a squared version of the input weighted by another parameter:

$$y = a \times x + b \times x^2 + c$$

This is called polynomial regression[3], and the squared term means it is a second-degree polynomial. So far, linear equations of this type can be fit by minimizing least squares and can be calculated analytically. This means we can find the optimal values of the parameters using a little linear algebra. We might also want to add other mathematical functions to the equation, such as sine, cosine, and more. Each term is weighted with a parameter and added to the whole to give the output; for example:

$$y = a \times \sin(b \times x) + c$$

Adding arbitrary mathematical functions to our mapping function generally means we cannot calculate the parameters analytically, and instead, we will need to use an iterative optimization

---

[2]https://en.wikipedia.org/wiki/Basis_function
[3]https://en.wikipedia.org/wiki/Polynomial_regression

algorithm. This is called nonlinear least squares[4], as the objective function is no longer convex (it's nonlinear) and not as easy to solve.

Now that we are familiar with curve fitting, let's look at how we might perform curve fitting in Python.

14.3   Curve Fitting Python API

We can perform curve fitting for our dataset in Python. The SciPy open source library provides the `curve_fit()` function[5] for curve fitting via nonlinear least squares. The function takes the same input and output data as arguments, as well as the name of the mapping function to use. The mapping function must take examples of input data and some number of arguments. These remaining arguments will be the coefficients or weight constants that will be optimized by a nonlinear least squares optimization process. For example, we may have some observations from our domain loaded as input variables x and output variables y.

```
...
x_values = ...
y_values = ...
```
Program 14.1: Load input variables from a file

Next, we need to design a mapping function to fit a line to the data and implement it as a Python function that takes inputs and the arguments. It may be a straight line, in which case it would look as follows:

```
def objective(x, a, b, c):
    return a * x + b
```
Program 14.2: Objective function

We can then call the `curve_fit()` function[6] to fit a straight line to the dataset using our defined function. The function `curve_fit()` returns the optimal values for the mapping function, e.g, the coefficient values. It also returns a covariance matrix for the estimated parameters, but we can ignore that for now.

```
...
popt, _ = curve_fit(objective, x_values, y_values)
```
Program 14.3: Fit a curve

Once fit, we can use the optimal parameters and our mapping function `objective()` to calculate the output for any arbitrary input. This might include the output for the examples we have already collected from the domain, it might include new values that interpolate observed values, or it might include extrapolated values outside of the limits of what was observed.

---

[4]https://en.wikipedia.org/wiki/Non-linear_least_squares
[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html
[6]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

```
...
# define new input values
x_new = ...
# unpack optima parameters for the objective function
a, b, c = popt
# use optimal parameters to calculate new values
y_new = objective(x_new, a, b, c)
```

Program 14.4: Use result of `curve_fit()` *to extrapolate values*

Now that we are familiar with using the curve fitting API, let's look at a worked example.

14.4   Curve Fitting Worked Example

We will develop a curve to fit some real world observations of economic data. In this example, we will use the so-called "*Longley's Economic Regression*" dataset; you can learn more about it here:

▷ Longley's Economic Regression (longley.csv)
   https://raw.githubusercontent.com/jbrownlee/Datasets/master/longley.csv

▷ Longley's Economic Regression Description (longley.names)
   https://github.com/jbrownlee/Datasets/blob/master/longley.names

We will download the dataset automatically as part of the worked example. There are seven input variables and 16 rows of data, where each row defines a summary of economic details for a year between 1947 to 1962. In this example, we will explore fitting a line between population size and the number of people employed for each year. The example below loads the dataset from the URL, selects the input variable as "*population,*" and the output variable as "*employed*" and creates a scatter plot.

```
from pandas import read_csv
from matplotlib import pyplot
# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/longley.csv'
dataframe = read_csv(url, header=None)
data = dataframe.values
# choose the input and output variables
x, y = data[:, 4], data[:, -1]
# plot input vs output
pyplot.scatter(x, y)
pyplot.show()
```

Program 14.5: Plot "Population" vs "Employed"

Running the example loads the dataset, selects the variables, and creates a scatter plot. We can see that there is a relationship between the two variables. Specifically, that as the population increases, the total number of employees increases. It is not unreasonable to think we can fit a line to this data.
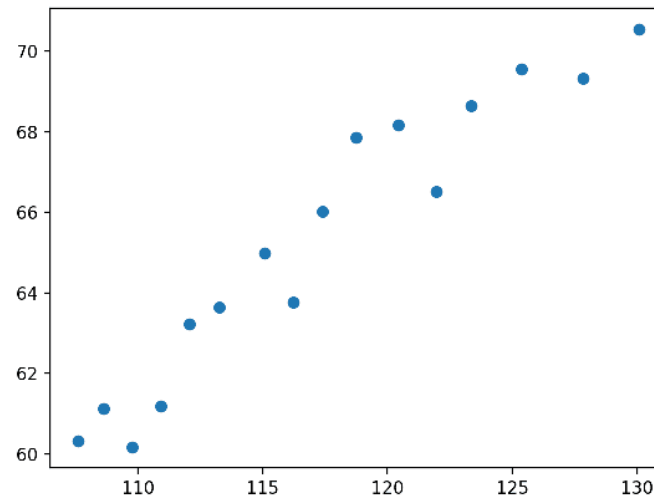
Figure 14.1: Scatter Plot of Population vs. Total Employed

First, we will try fitting a straight line to this data, as follows:

```
def objective(x, a, b):
    return a * x + b
```

Program 14.6: Define the true objective function

We can use curve fitting to find the optimal values of "a" and "b" and summarize the values that were found:

```
...
# curve fit
popt, _ = curve_fit(objective, x, y)
# summarize the parameter values
a, b = popt
print('y = %.5f * x + %.5f' % (a, b))
```

Program 14.7: Fit a curve and summarize the values found

We can then create a scatter plot as before.

```
...
pyplot.scatter(x, y)
```

Program 14.8: Plot input vs output

On top of the scatter plot, we can draw a line for the function with the optimized parameter values. This involves first defining a sequence of input values between the minimum and maximum values observed in the dataset (e.g. between about 120 and about 130).

```
...
x_line = arange(min(x), max(x), 1)
```

Program 14.9: Define a sequence of inputs between the smallest and largest known inputs

We can then calculate the output value for each input value.

```
...
y_line = objective(x_line, a, b)
```

Program 14.10: Calculate the output for the range

Then create a line plot of the inputs vs. the outputs to see a line:

```
...
pyplot.plot(x_line, y_line, '--', color='red')
```

Program 14.11: Create a line plot for the mapping function

Tying this together, the example below uses curve fitting to find the parameters of a straight line for our economic data.

```python
from numpy import arange
from pandas import read_csv
from scipy.optimize import curve_fit
from matplotlib import pyplot

# define the true objective function
def objective(x, a, b):
    return a * x + b

# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/longley.csv'
dataframe = read_csv(url, header=None)
data = dataframe.values
# choose the input and output variables
x, y = data[:, 4], data[:, -1]
# curve fit
popt, _ = curve_fit(objective, x, y)
# summarize the parameter values
a, b = popt
print('y = %.5f * x + %.5f' % (a, b))
# plot input vs output
pyplot.scatter(x, y)
# define a sequence of inputs between the smallest and largest known inputs
x_line = arange(min(x), max(x), 1)
# calculate the output for the range
y_line = objective(x_line, a, b)
# create a line plot for the mapping function
pyplot.plot(x_line, y_line, '--', color='red')
pyplot.show()
```

Program 14.12: Fit a straight line to the economic data

Running the example performs curve fitting and finds the optimal parameters to our objective function. First, the values of the parameters are reported.

```
y = 0.48488 * x + 8.38067
```

Output 14.1: Result from Program 14.12

Next, a plot is created showing the original data and the line that was fit to the data. We can see that it is a reasonably good fit.
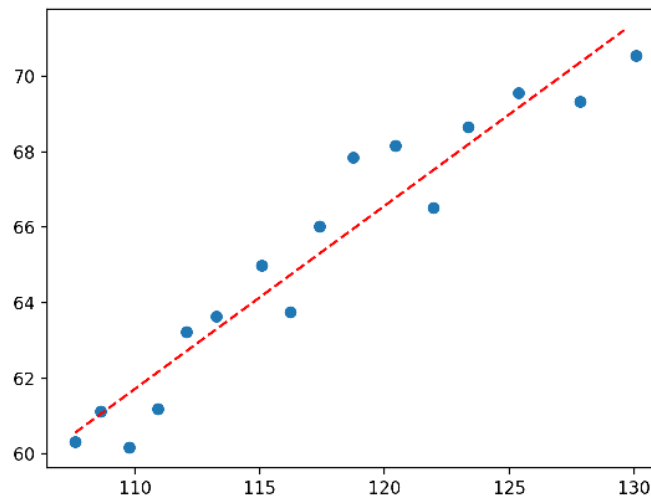
Figure 14.2: Plot of Straight Line Fit to Economic Dataset

So far, this is not very exciting as we could achieve the same effect by fitting a linear regression model on the dataset. Let's try a polynomial regression model by adding squared terms to the objective function.

```python
def objective(x, a, b, c):
    return a * x + b * x**2 + c
```

Program 14.13: Define the true objective function

Tying this together, the complete example is listed below.

```python
from numpy import arange
from pandas import read_csv
from scipy.optimize import curve_fit
from matplotlib import pyplot

# define the true objective function
def objective(x, a, b, c):
    return a * x + b * x**2 + c

# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/longley.csv'
```

```
dataframe = read_csv(url, header=None)
data = dataframe.values
# choose the input and output variables
x, y = data[:, 4], data[:, -1]
# curve fit
popt, _ = curve_fit(objective, x, y)
# summarize the parameter values
a, b, c = popt
print('y = %.5f * x + %.5f * x^2 + %.5f' % (a, b, c))
# plot input vs output
pyplot.scatter(x, y)
# define a sequence of inputs between the smallest and largest known inputs
x_line = arange(min(x), max(x), 1)
# calculate the output for the range
y_line = objective(x_line, a, b, c)
# create a line plot for the mapping function
pyplot.plot(x_line, y_line, '--', color='red')
pyplot.show()
```

Program 14.14: Fit a second degree polynomial to the economic data

First the optimal parameters are reported.

```
y = 3.25443 * x + -0.01170 * x^2 + -155.02783
```

Output 14.2: Result from Program 14.14

Next, a plot is created showing the line in the context of the observed values from the domain. We can see that the second-degree polynomial equation that we defined is visually a better fit for the data than the straight line that we tested first.
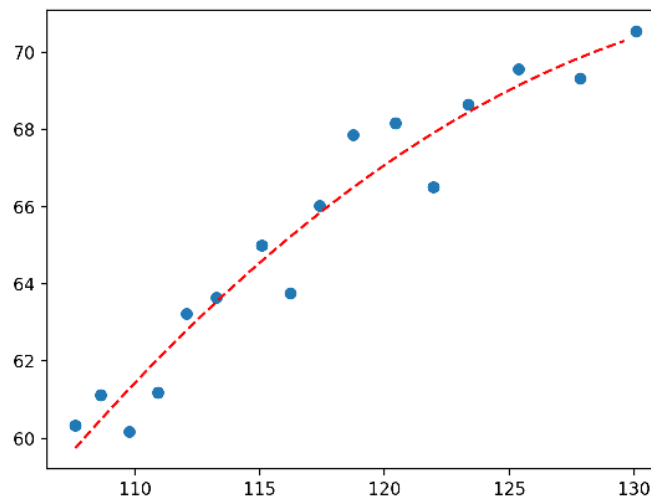
Figure 14.3: Plot of Second Degree Polynomial Fit to Economic Dataset

We could keep going and add more polynomial terms to the equation to better fit the curve. For example, below is an example of a fifth-degree polynomial fit to the data.

```python
from numpy import arange
from pandas import read_csv
from scipy.optimize import curve_fit
from matplotlib import pyplot

# define the true objective function
def objective(x, a, b, c, d, e, f):
    return (a * x) + (b * x**2) + (c * x**3) + (d * x**4) + (e * x**5) + f

# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/longley.csv'
dataframe = read_csv(url, header=None)
data = dataframe.values
# choose the input and output variables
x, y = data[:, 4], data[:, -1]
# curve fit
popt, _ = curve_fit(objective, x, y)
# summarize the parameter values
a, b, c, d, e, f = popt
# plot input vs output
pyplot.scatter(x, y)
# define a sequence of inputs between the smallest and largest known inputs
x_line = arange(min(x), max(x), 1)
# calculate the output for the range
y_line = objective(x_line, a, b, c, d, e, f)
# create a line plot for the mapping function
pyplot.plot(x_line, y_line, '--', color='red')
pyplot.show()
```

Program 14.15: Fit a fifth degree polynomial to the economic data

Running the example fits the curve and plots the result, again capturing slightly more nuance in how the relationship in the data changes over time.
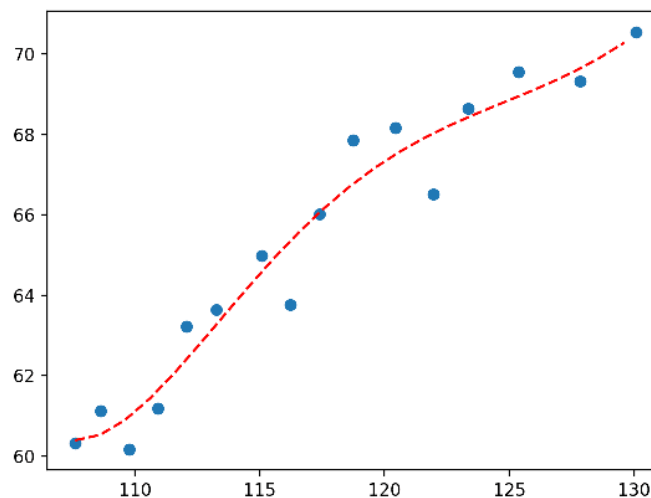
Figure 14.4: Plot of Fifth Degree Polynomial Fit to Economic Dataset

Importantly, we are not limited to linear regression or polynomial regression. We can use any arbitrary basis function. For example, perhaps we want a line that has wiggles to capture the short-term movement in observation. We could add a sine curve to the equation and find the parameters that best integrate this element in the equation. For example, an arbitrary function that uses a sine wave and a second degree polynomial is listed below:

```python
def objective(x, a, b, c, d):
    return a * sin(b - x) + c * x**2 + d
```

Program 14.16: Define the true objective function

The complete example of fitting a curve using this basis function is listed below.

```python
from numpy import sin
from numpy import sqrt
from numpy import arange
from pandas import read_csv
from scipy.optimize import curve_fit
from matplotlib import pyplot

# define the true objective function
def objective(x, a, b, c, d):
    return a * sin(b - x) + c * x**2 + d

# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/longley.csv'
dataframe = read_csv(url, header=None)
data = dataframe.values
# choose the input and output variables
x, y = data[:, 4], data[:, -1]
# curve fit
popt, _ = curve_fit(objective, x, y)
# summarize the parameter values
a, b, c, d = popt
print(popt)
# plot input vs output
pyplot.scatter(x, y)
# define a sequence of inputs between the smallest and largest known inputs
x_line = arange(min(x), max(x), 1)
# calculate the output for the range
y_line = objective(x_line, a, b, c, d)
# create a line plot for the mapping function
pyplot.plot(x_line, y_line, '--', color='red')
pyplot.show()
```

Program 14.17: Fit a line to the economic data

Running the example fits a curve and plots the result. We can see that adding a sine wave has the desired effect showing a periodic wiggle with an upward trend that provides another way of capturing the relationships in the data.
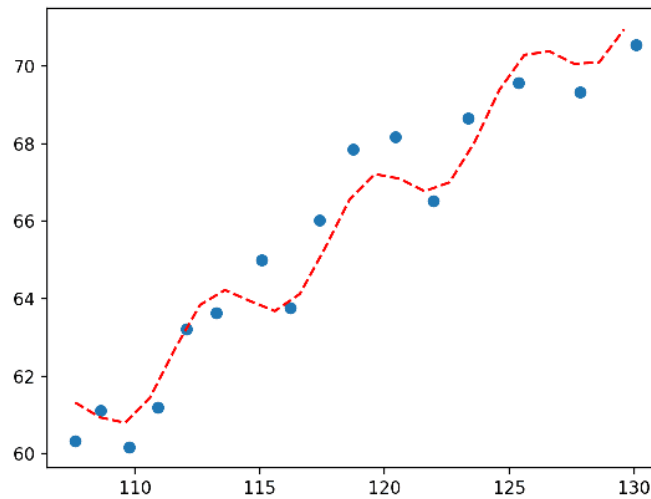
Figure 14.5: Plot of Sine Wave Fit to Economic Dataset

How do you choose the best fit?

If you want the best fit, you would model the problem as a regression supervised learning problem and test a suite of algorithms in order to discover which is best at minimizing the error. In this case, curve fitting is appropriate when you want to define the function explicitly, then discover the parameters of your function that best fit a line to the data.

14.5   Further Reading

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

Books

Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.
https://amzn.to/36yvG9w

APIs

`scipy.optimize.curve_fit` *API*.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html
`numpy.random.randn` *API*.
https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html

Articles

Curve fitting. Wikipedia.
https://en.wikipedia.org/wiki/Curve_fitting

Non-linear least squares. Wikipedia.
    https://en.wikipedia.org/wiki/Non-linear_least_squares
Polynomial regression. Wikipedia.
    https://en.wikipedia.org/wiki/Polynomial_regression

14.6 Summary

In this tutorial, you discovered how to perform curve fitting in Python. Specifically, you learned:

- ▷ Curve fitting involves finding the optimal parameters to a function that maps examples of inputs to outputs.

- ▷ Unlike supervised learning, curve fitting requires that you define the function that maps examples of inputs to outputs.

- ▷ How to use curve fitting in SciPy to fit a range of different curves to a set of observations.

    Next, you will learn about hill climbing algorithm.

Stochastic Hill Climbing

<div style="text-align: right; font-size: 3em;">**15**</div>

Stochastic Hill climbing is an optimization algorithm. It makes use of randomness as part of the search process. This makes the algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well. It is also a local search algorithm, meaning that it modifies a single solution and searches the relatively local area of the search space until the local optima is located. This means that it is appropriate on unimodal optimization problems or for use after the application of a global optimization algorithm.

In this tutorial, you will discover the hill climbing optimization algorithm for function optimization After completing this tutorial, you will know:

▷ Hill climbing is a stochastic local search algorithm for function optimization.

▷ How to implement the hill climbing algorithm from scratch in Python.

▷ How to apply the hill climbing algorithm and inspect the results of the algorithm.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Hill Climbing Algorithm
2. Hill Climbing Algorithm Implementation
3. Example of Applying the Hill Climbing Algorithm

15.2 Hill Climbing Algorithm

The stochastic hill climbing algorithm is a stochastic local search optimization algorithm. It takes an initial point as input and a step size, where the step size is a distance within the search space. The algorithm takes the initial point as the current best candidate solution and generates a new point within the step size distance of the provided point. The generated point is evaluated, and if it is equal or better than the current point, it is taken as the current point. The generation of the new point uses randomness, often referred to as Stochastic Hill Climbing.

This means that the algorithm can skip over bumpy, noisy, discontinuous, or deceptive regions of the response surface as part of the search.

> " Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. "
>
> — Page 124, *Artificial Intelligence: A Modern Approach*, 2009.

It is important that different points with equal evaluation are accepted as it allows the algorithm to continue to explore the search space, such as across flat regions of the response surface. It may also be helpful to put a limit on these so-called "*sideways*" moves to avoid an infinite loop.

> " If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves "
>
> — Page 123, *Artificial Intelligence: A Modern Approach*, 2009.

This process continues until a stop condition is met, such as a maximum number of function evaluations or no improvement within a given number of function evaluations. The algorithm takes its name from the fact that it will (stochastically) climb the hill of the response surface to the local optima. This does not mean it can only be used for maximizing objective functions; it is just a name. In fact, typically, we minimize functions instead of maximize them.

> " The hill-climbing search algorithm (steepest-ascent version) [...] is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. "
>
> — Page 122, *Artificial Intelligence: A Modern Approach*, 2009.

As a local search algorithm, it can get stuck in local optima. Nevertheless, multiple restarts may allow the algorithm to locate the global optimum.

> " Random-restart hill climbing [...] conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. "
>
> — Page 124, *Artificial Intelligence: A Modern Approach*, 2009.

The step size must be large enough to allow better nearby points in the search space to be located, but not so large that the search jumps over out of the region that contains the local optima.

15.3 Hill Climbing Algorithm Implementation

At the time of writing, the SciPy library does not provide an implementation of stochastic hill climbing. Nevertheless, we can implement it ourselves. First, we must define our objective function and the bounds on each input variable to the objective function. The objective function is just a Python function we will name `objective()`. The bounds will be a 2D array with one dimension for each input variable that defines the minimum and maximum for the variable. For example, a one-dimensional objective function and bounds would be defined as follows:

```python
def objective(x):
    # placeholder: to return a scalar value based on x
    return 0

# define range for input
bounds = asarray([[-5.0, 5.0]])
```

Program 15.1: Objective function

Next, we can generate our initial solution as a random point within the bounds of the problem, then evaluate it using the objective function.

```python
...
# generate an initial point
solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
# evaluate the initial point
solution_eval = objective(solution)
```

Program 15.2: Evaluate a random initial point

Now we can loop over a predefined number of iterations of the algorithm defined as "`n_iterations`", such as 100 or 1,000.

```python
...
for i in range(n_iterations):
    ...
```

Program 15.3: Run the hill climb

The first step of the algorithm iteration is to take a step. This requires a predefined "`step_size`" parameter, which is relative to the bounds of the search space. We will take a random step with a Gaussian distribution where the mean is our current point and the standard deviation is defined by the "`step_size`". That means that about 99 percent of the steps taken will be within (3 * `step_size`) of the current point.

```python
...
candidate = solution + randn(len(bounds)) * step_size
```

Program 15.4: Take one random step (normal distribution)

We don't have to take steps in this way. You may wish to use a uniform distribution between 0 and the step size. For example:

```python
...
candidate = solution + rand(len(bounds)) * step_size
```

Program 15.5: Take one random step (uniform distribution)

Next we need to evaluate the new candidate solution with the objective function.

```python
...
candidte_eval = objective(candidate)
```

Program 15.6: Evaluate candidate opint

We then need to check if the evaluation of this new point is as good as or better than the current best point, and if it is, replace our current best point with this new point.

```
...
if candidte_eval <= solution_eval:
    # store the new point
    solution, solution_eval = candidate, candidte_eval
    # report progress
    print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
```

Program 15.7: Check if we should keep the new point

And that's it.

We can implement this hill climbing algorithm as a reusable function that takes the name of the objective function, the bounds of each input variable, the total iterations and steps as arguments, and returns the best solution found and its evaluation.

```
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]
```

Program 15.8: Hill climbing local search algorithm

Now that we know how to implement the hill climbing algorithm in Python, let's look at how we might use it to optimize an objective function.

15.4   Example of Applying the Hill Climbing Algorithm

In this section, we will apply the hill climbing optimization algorithm to an objective function.

First, let's define our objective function. We will use a simple one-dimensional x^2 objective function with the bounds $[-5, 5]$. The example below defines the function, then creates a line plot of the response surface of the function for a grid of input values and marks the optima at $f(0.0) = 0.0$ with a red line.

```python
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(x):
    return x[0]**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = [objective([x]) for x in inputs]
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define optimal input value
x_optima = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=x_optima, ls='--', color='red')
# show the plot
pyplot.show()
```

Program 15.9: Convex unimodal optimization function

Running the example creates a line plot of the objective function and clearly marks the function optima.
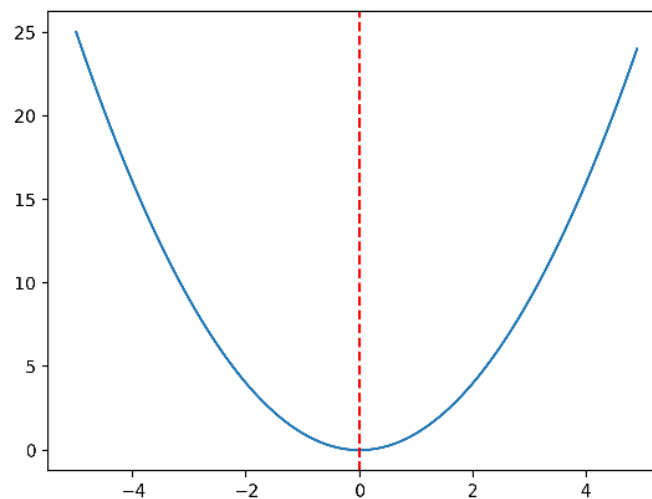
Figure 15.1: Line Plot of Objective Function With Optima Marked with a Dashed Red Line

Next, we can apply the hill climbing algorithm to the objective function.

First, we will seed the pseudorandom number generator. This is not required in general, but in this case, I want to ensure we get the same results (same sequence of random numbers) each time we run the algorithm so we can plot the results later.

```
...
seed(5)
```

Program 15.10: Seed the pseudorandom number generator

Next, we can define the configuration of the search. In this case, we will search for 1,000 iterations of the algorithm and use a step size of 0.1. Given that we are using a Gaussian function for generating the step, this means that about 99 percent of all steps taken will be within a distance of 0.1×3 of a given point, i.e. three standard deviations.

```
...
n_iterations = 1000
# define the maximum step size
step_size = 0.1
```

Program 15.11: Define the number if iterations and step size

Next, we can perform the search and report the results.

```
...
best, score = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 15.12: Perform the hill climbing search

Tying this all together, the complete example is listed below.

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

# objective function
def objective(x):
    return x[0]**2.0

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
```

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

    # seed the pseudorandom number generator
    seed(5)
    # define range for input
    bounds = asarray([[-5.0, 5.0]])
    # define the total iterations
    n_iterations = 1000
    # define the maximum step size
    step_size = 0.1
    # perform the hill climbing search
    best, score = hillclimbing(objective, bounds, n_iterations, step_size)
    print('Done!')
    print('f(%s) = %f' % (best, score))
```

Program 15.13: Hill climbing search of a one-dimensional objective function

Running the example reports the progress of the search, including the iteration number, the input to the function, and the response from the objective function each time an improvement was detected. At the end of the search, the best solution is found and its evaluation is reported. In this case we can see about 36 improvements over the 1,000 iterations of the algorithm and a solution that is very close to the optimal input of 0.0 that evaluates to $f(0.0) = 0.0$.

```
>1 f([-2.74290923]) = 7.52355
>3 f([-2.65873147]) = 7.06885
>4 f([-2.52197291]) = 6.36035
>5 f([-2.46450214]) = 6.07377
>7 f([-2.44740961]) = 5.98981
>9 f([-2.28364676]) = 5.21504
>12 f([-2.19245939]) = 4.80688
>14 f([-2.01001538]) = 4.04016
>15 f([-1.86425287]) = 3.47544
>22 f([-1.79913002]) = 3.23687
>24 f([-1.57525573]) = 2.48143
>25 f([-1.55047719]) = 2.40398
>26 f([-1.51783757]) = 2.30383
>27 f([-1.49118756]) = 2.22364
>28 f([-1.45344116]) = 2.11249
>30 f([-1.33055275]) = 1.77037
>32 f([-1.17805016]) = 1.38780
>33 f([-1.15189314]) = 1.32686
>36 f([-1.03852644]) = 1.07854
>37 f([-0.99135322]) = 0.98278
>38 f([-0.79448984]) = 0.63121
>39 f([-0.69837955]) = 0.48773
>42 f([-0.69317313]) = 0.48049
>46 f([-0.61801423]) = 0.38194
>48 f([-0.48799625]) = 0.23814
>50 f([-0.22149135]) = 0.04906
>54 f([-0.20017144]) = 0.04007
>57 f([-0.15994446]) = 0.02558
```

```
>60 f([-0.15492485]) = 0.02400
>61 f([-0.03572481]) = 0.00128
>64 f([-0.03051261]) = 0.00093
>66 f([-0.0074283]) = 0.00006
>78 f([-0.00202357]) = 0.00000
>119 f([0.00128373]) = 0.00000
>120 f([-0.00040911]) = 0.00000
>314 f([-0.00017051]) = 0.00000
Done!
f([-0.00017051]) = 0.000000
```

Output 15.1: Result from Program 15.13

It can be interesting to review the progress of the search as a line plot that shows the change in the evaluation of the best solution each time there is an improvement. We can update the `hillclimbing()` to keep track of the objective function evaluations each time there is an improvement and return this list of scores.

```python
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    scores = list()
    scores.append(solution_eval)
    for i in range(n_iterations):
        # take a step
        candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # keep track of scores
            scores.append(solution_eval)
            # report progress
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval, scores]
```

Program 15.14: Hill climbing local search algorithm

We can then create a line plot of these scores to see the relative change in objective function for each improvement found during the search.

```python
...
pyplot.plot(scores, '.-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Program 15.15: Line plot of best scores

Tying this together, the complete example of performing the search and plotting the objective function scores of the improved solutions during the search is listed below.

```python
from numpy import asarray
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed
from matplotlib import pyplot

# objective function
def objective(x):
    return x[0]**2.0

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    scores = list()
    scores.append(solution_eval)
    for i in range(n_iterations):
        # take a step
        candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # keep track of scores
            scores.append(solution_eval)
            # report progress
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval, scores]

# seed the pseudorandom number generator
seed(5)
# define range for input
bounds = asarray([[-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# perform the hill climbing search
best, score, scores = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
# line plot of best scores
pyplot.plot(scores, '.-')
pyplot.xlabel('Improvement Number')
```

```
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Program 15.16: Hill climbing search of a one-dimensional objective function

Running the example performs the search and reports the results as before. A line plot is created showing the objective function evaluation for each improvement during the hill climbing search. We can see about 36 changes to the objective function evaluation during the search, with large changes initially and very small to imperceptible changes towards the end of the search as the algorithm converged on the optima.
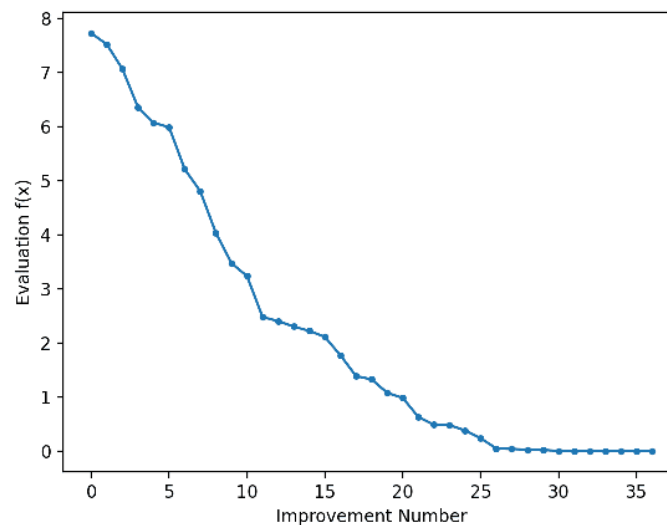
Figure 15.2: Line Plot of Objective Function Evaluation for Each Improvement During the Hill Climbing Search

Given that the objective function is one-dimensional, it is straightforward to plot the response surface as we did above. It can be interesting to review the progress of the search by plotting the best candidate solutions found during the search as points in the response surface. We would expect a sequence of points running down the response surface to the optima. This can be achieved by first updating the `hillclimbing()` function to keep track of each best candidate solution as it is located during the search, then return a list of best solutions.

```
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    solutions = list()
    solutions.append(solution)
    for i in range(n_iterations):
        # take a step
        candidate = solution + randn(len(bounds)) * step_size
```

```
            # evaluate candidate point
            candidte_eval = objective(candidate)
            # check if we should keep the new point
            if candidte_eval <= solution_eval:
                # store the new point
                solution, solution_eval = candidate, candidte_eval
                # keep track of solutions
                solutions.append(solution)
                # report progress
                print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval, solutions]
```

Program 15.17: Hill climbing local search algorithm

We can then create a plot of the response surface of the objective function and mark the optima as before.

```
...
# sample input range uniformly at 0.1 increments
inputs = arange(bounds[0,0], bounds[0,1], 0.1)
# create a line plot of input vs result
pyplot.plot(inputs, [objective([x]) for x in inputs], '--')
# draw a vertical line at the optimal input
pyplot.axvline(x=[0.0], ls='--', color='red')
```

Program 15.18: Create plot of response surface

Finally, we can plot the sequence of candidate solutions found by the search as black dots.

```
...
pyplot.plot(solutions, [objective(x) for x in solutions], 'o', color='black')
```

Program 15.19: Plot the sample as black circles

Tying this together, the complete example of plotting the sequence of improved solutions on the response surface of the objective function is listed below.

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

# objective function
def objective(x):
    return x[0]**2.0

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
```

```python
        # run the hill climb
        solutions = list()
        solutions.append(solution)
        for i in range(n_iterations):
            # take a step
            candidate = solution + randn(len(bounds)) * step_size
            # evaluate candidate point
            candidte_eval = objective(candidate)
            # check if we should keep the new point
            if candidte_eval <= solution_eval:
                # store the new point
                solution, solution_eval = candidate, candidte_eval
                # keep track of solutions
                solutions.append(solution)
                # report progress
                print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
        return [solution, solution_eval, solutions]

# seed the pseudorandom number generator
seed(5)
# define range for input
bounds = asarray([[-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# perform the hill climbing search
best, score, solutions = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
# sample input range uniformly at 0.1 increments
inputs = arange(bounds[0,0], bounds[0,1], 0.1)
# create a line plot of input vs result
pyplot.plot(inputs, [objective([x]) for x in inputs], '--')
# draw a vertical line at the optimal input
pyplot.axvline(x=[0.0], ls='--', color='red')
# plot the sample as black circles
pyplot.plot(solutions, [objective(x) for x in solutions], 'o', color='black')
pyplot.show()
```

Program 15.20: Hill climbing search of a one-dimensional objective function

Running the example performs the hill climbing search and reports the results as before. A plot of the response surface is created as before showing the familiar bowl shape of the function with a vertical red line marking the optima of the function. The sequence of best solutions found during the search is shown as black dots running down the bowl shape to the optima.
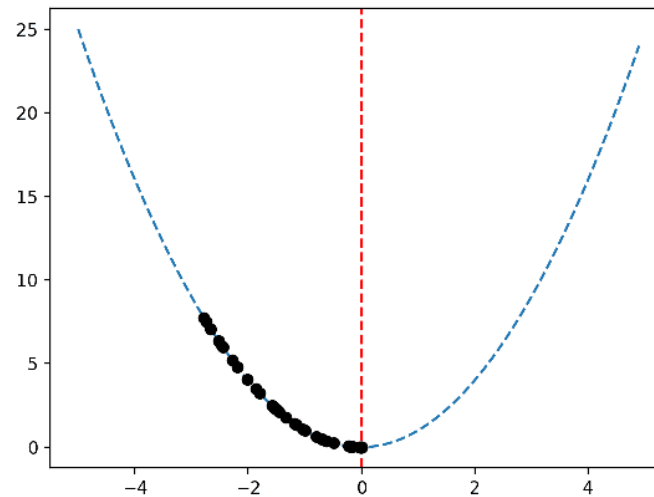
Figure 15.3: Response Surface of Objective Function With Sequence of Best Solutions Plotted as Black Dots

15.5 Further Reading

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

Books

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 3rd ed. Pearson, 2009.
https://amzn.to/2HYk1Xj

APIs

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

Articles

Hill climbing. Wikipedia.
https://en.wikipedia.org/wiki/Hill_climbing

Stochastic hill climbing. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_hill_climbing

15.6 Summary

In this tutorial, you discovered the hill climbing optimization algorithm for function optimization. Specifically, you learned:

 ▷ Hill climbing is a stochastic local search algorithm for function optimization.

 ▷ How to implement the hill climbing algorithm from scratch in Python.

 ▷ How to apply the hill climbing algorithm and inspect the results of the algorithm.

Next, you will learn about an algorithm that repeats hill climbing with different starting points.

Iterated Local Search

<div style="text-align: right; font-size: 2em;">**16**</div>

Iterated Local Search is a stochastic global optimization algorithm. It involves the repeated application of a local search algorithm to modified versions of a good solution found previously. In this way, it is like a clever version of the stochastic hill climbing with random restarts algorithm. The intuition behind the algorithm is that random restarts can help to locate many local optima in a problem and that better local optima are often close to other local optima. Therefore modest perturbations to existing local optima may locate better or even best solutions to an optimization problem.

In this tutorial, you will discover how to implement the iterated local search algorithm from scratch. After completing this tutorial, you will know:

▷ Iterated local search is a stochastic global search optimization algorithm that is a smarter version of stochastic hill climbing with random restarts.

▷ How to implement stochastic hill climbing with random restarts from scratch.

▷ How to implement and apply the iterated local search algorithm to a nonlinear objective function.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is Iterated Local Search
2. Ackley Objective Function
3. Stochastic Hill Climbing Algorithm
4. Stochastic Hill Climbing With Random Restarts
5. Iterated Local Search Algorithm

16.2   What Is Iterated Local Search

Iterated Local Search[1], or ILS for short, is a stochastic global search optimization algorithm. It is related to or an extension of stochastic hill climbing and stochastic hill climbing with random starts.

> It's essentially a more clever version of Hill-Climbing with Random Restarts.
>
> — Page 26, *Essentials of Metaheuristics*, 2011.

Stochastic hill climbing[2] is a local search algorithm that involves making random modifications to an existing solution and accepting the modification only if it results in better results than the current working solution. Local search algorithms in general can get stuck in local optima. One approach to address this problem is to restart the search from a new randomly selected starting point. The restart procedure can be performed many times and may be triggered after a fixed number of function evaluations or if no further improvement is seen for a given number of algorithm iterations. This algorithm is called stochastic hill climbing with random restarts.

> The simplest possibility to improve upon a cost found by LocalSearch is to repeat the search from another starting point.
>
> — Page 132, *Handbook of Metaheuristics*, 2019.

Iterated local search is similar to stochastic hill climbing with random restarts, except rather than selecting a random starting point for each restart, a point is selected based on a modified version of the best point found so far during the broader search. The perturbation of the best solution so far is like a large jump in the search space to a new region, whereas the perturbations made by the stochastic hill climbing algorithm are much smaller, confined to a specific region of the search space.

> The heuristic here is that you can often find better local optima near to the one you're presently in, and walking from local optimum to local optimum in this way often outperforms just trying new locations entirely at random.
>
> — Page 26, *Essentials of Metaheuristics*, 2011.

This allows the search to be performed at two levels. The hill climbing algorithm is the local search for getting the most out of a specific candidate solution or region of the search space, and the restart approach allows different regions of the search space to be explored. In this way, the algorithm Iterated Local Search explores multiple local optima in the search space, increasing the likelihood of locating the global optima. The Iterated Local Search was proposed for combinatorial optimization problems, such as the traveling salesman problem (TSP), although it can be applied to continuous function optimization by using different step sizes in the search space: smaller steps for the hill climbing and larger steps for the random restart.

Now that we are familiar with the Iterated Local Search algorithm, let's explore how to implement the algorithm from scratch.

---

[1]https://en.wikipedia.org/wiki/Iterated_local_search
[2]https://en.wikipedia.org/wiki/Hill_climbing

16.3   Ackley Objective Function

First, let's define a channeling optimization problem as the basis for implementing the Iterated Local Search algorithm. The Ackley function[3] is an example of a multimodal objective function that has a single global optima and multiple local optima in which a local search might get stuck. As such, a global optimization technique is required. It is a two-dimensional objective function that has a global optima at [0,0], which evaluates to 0.0.

The example below implements the Ackley and creates a three-dimensional surface plot showing the global optima and multiple local optima.

```python
from numpy import arange
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi
        * x) + cos(2 * pi * y))) + e + 20

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 16.1: Ackley multimodal function

Running the example creates the surface plot of the Ackley function showing the vast number of local optima.

---

[3]https://en.wikipedia.org/wiki/Ackley_function
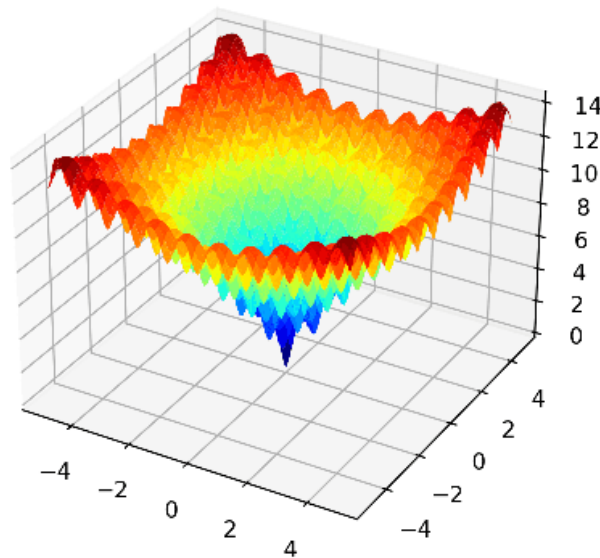
Figure 16.1: 3D Surface Plot of the Ackley Multimodal Function

We will use this as the basis for implementing and comparing a simple stochastic hill climbing algorithm, stochastic hill climbing with random restarts, and finally iterated local search. We would expect a stochastic hill climbing algorithm to get stuck easily in local minima. We would expect stochastic hill climbing with restarts to find many local minima, and we would expect iterated local search to perform better than either method on this problem if configured appropriately.

16.4   Stochastic Hill Climbing Algorithm

Core to the Iterated Local Search algorithm is a local search, and in this tutorial, we will use the Stochastic Hill Climbing algorithm for this purpose. The Stochastic Hill Climbing algorithm involves first generating a random starting point and current working solution, then generating perturbed versions of the current working solution and accepting them if they are better than the current working solution. Given that we are working on a continuous optimization problem, a solution is a vector of values to be evaluated by the objective function, in this case, a point in a two-dimensional space bounded by -5 and 5. We can generate a random point by sampling the search space with a uniform probability distribution. For example:

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

Program 16.2: Generate a random point in the search space

We can generate perturbed versions of a currently working solution using a Gaussian probability distribution with the mean of the current values in the solution and a standard deviation controlled by a hyperparameter that controls how far the search is allowed to explore from the current working solution. We will refer to this hyperparameter as "`step_size`", for example:

```
...
candidate = solution + randn(len(bounds)) * step_size
```

Program 16.3: Generate a perturbed version of a current working solution

Importantly, we must check that generated solutions are within the search space. This can be achieved with a custom function named `in_bounds()` that takes a candidate solution and the bounds of the search space and returns `True` if the point is in the search space, `False` otherwise.

```
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True
```

Program 16.4: Check if a point is within the bounds of the search

This function can then be called during the hill climb to confirm that new points are in the bounds of the search space, and if not, new points can be generated.

Tying this together, the function `hillclimbing()` below implements the stochastic hill climbing local search algorithm. It takes the name of the objective function, bounds of the problem, number of iterations, and steps size as arguments and returns the best solution and its evaluation.

```
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = None
    while solution is None or not in_bounds(solution, bounds):
        solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
```

```
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]
```

Program 16.5: Hill climbing local search algorithm

We can test this algorithm on the Ackley function. We will fix the seed for the pseudorandom number generator to ensure we get the same results each time the code is run. The algorithm will be run for 1,000 iterations and a step size of 0.05 units will be used; both hyperparameters were chosen after a little trial and error. At the end of the run, we will report the best solution found.

```
...
# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.05
# perform the hill climbing search
best, score = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 16.6: Hill climb and report the best solution found

Tying this together, the complete example of applying the stochastic hill climbing algorithm to the Ackley objective function is listed below.

```
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi
        * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
```

```python
        return True

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
    # generate an initial point
    solution = None
    while solution is None or not in_bounds(solution, bounds):
        solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.05
# perform the hill climbing search
best, score = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 16.7: Hill climbing search of the Ackley objective function

Running the example performs the stochastic hill climbing search on the objective function. Each improvement found during the search is reported and the best solution is then reported at the end of the search.

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 about 13 improvements during the search and a final solution of about $f(-0.981, 1.965)$, resulting in an evaluation of about 5.381, which is far from $f(0.0, 0.0) = 0$.

```
>0 f([-0.85618854 2.1495965 ]) = 6.46986
>1 f([-0.81291816 2.03451957]) = 6.07149
>5 f([-0.82903902 2.01531685]) = 5.93526
>7 f([-0.83766043 1.97142393]) = 5.82047
>9 f([-0.89269139 2.02866012]) = 5.68283
>12 f([-0.8988359 1.98187164]) = 5.55899
>13 f([-0.9122303 2.00838942]) = 5.55566
>14 f([-0.94681334 1.98855174]) = 5.43024
>15 f([-0.98117198 1.94629146]) = 5.39010
>23 f([-0.97516403 1.97715161]) = 5.38735
>39 f([-0.98628044 1.96711371]) = 5.38241
>362 f([-0.9808789 1.96858459]) = 5.38233
>629 f([-0.98102417 1.96555308]) = 5.38194
Done!
f([-0.98102417 1.96555308]) = 5.381939
```

Output 16.1: Result from Program 16.7

Next, we will modify the algorithm to perform random restarts and see if we can achieve better results.

16.5   Stochastic Hill Climbing With Random Restarts

The Stochastic Hill Climbing With Random Restarts algorithm involves the repeated running of the Stochastic Hill Climbing algorithm and keeping track of the best solution found. First, let's modify the `hillclimbing()` function to take the starting point of the search rather than generating it randomly. This will help later when we implement the Iterated Local Search algorithm later.

```python
def hillclimbing(objective, bounds, n_iterations, step_size, start_pt):
    # store the initial point
    solution = start_pt
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
    return [solution, solution_eval]
```

Program 16.8: Hill climbing local search algorithm

Next, we can implement the random restart algorithm by repeatedly calling the `hillclimbing()` function a fixed number of times. Each call, we will generate a new randomly selected starting point for the hill climbing search.

```
...
# generate a random initial point for the search
start_pt = None
while start_pt is None or not in_bounds(start_pt, bounds):
    start_pt = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
# perform a stochastic hill climbing search
solution, solution_eval = hillclimbing(objective, bounds, n_iter, step_size,  ↪
  ↳   start_pt)
```
Program 16.9: Stochastic hill climbing with random initial point

We can then inspect the result and keep it if it is better than any result of the search we have seen so far.

```
...
if solution_eval < best_eval:
    best, best_eval = solution, solution_eval
print('Restart %d, best: f(%s) = %.5f' % (n, best, best_eval))
```
Program 16.10: Check for new best

Tying this together, the `random_restarts()` function implemented the stochastic hill climbing algorithm with random restarts.

```
def random_restarts(objective, bounds, n_iter, step_size, n_restarts):
    best, best_eval = None, 1e+10
    # enumerate restarts
    for n in range(n_restarts):
        # generate a random initial point for the search
        start_pt = None
        while start_pt is None or not in_bounds(start_pt, bounds):
            start_pt = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -  ↪
              ↳   bounds[:, 0])
        # perform a stochastic hill climbing search
        solution, solution_eval = hillclimbing(objective, bounds, n_iter,  ↪
          ↳   step_size, start_pt)
        # check for new best
        if solution_eval < best_eval:
            best, best_eval = solution, solution_eval
            print('Restart %d, best: f(%s) = %.5f' % (n, best, best_eval))
    return [best, best_eval]
```
Program 16.11: Hill climbing with random restarts algorithm

We can then apply this algorithm to the Ackley objective function. In this case, we will limit the number of random restarts to 30, chosen arbitrarily.

The complete example is listed below.

```python
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi ↩
    ↪ * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size, start_pt):
    # store the initial point
    solution = start_pt
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
    return [solution, solution_eval]

# hill climbing with random restarts algorithm
def random_restarts(objective, bounds, n_iter, step_size, n_restarts):
    best, best_eval = None, 1e+10
    # enumerate restarts
    for n in range(n_restarts):
        # generate a random initial point for the search
        start_pt = None
        while start_pt is None or not in_bounds(start_pt, bounds):
```

```
            start_pt = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
               ↳ bounds[:, 0])
        # perform a stochastic hill climbing search
        solution, solution_eval = hillclimbing(objective, bounds, n_iter,
           ↳ step_size, start_pt)
        # check for new best
        if solution_eval < best_eval:
            best, best_eval = solution, solution_eval
            print('Restart %d, best: f(%s) = %.5f' % (n, best, best_eval))
    return [best, best_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iter = 1000
# define the maximum step size
step_size = 0.05
# total number of random restarts
n_restarts = 30
# perform the hill climbing search
best, score = random_restarts(objective, bounds, n_iter, step_size, n_restarts)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 16.12: Hill climbing search with random restarts of the Ackley objective function

Running the example will perform a stochastic hill climbing with random restarts search for the Ackley objective function. Each time an improved overall solution is discovered, it is reported and the final best solution found by the search is summarized.

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 three improvements during the search and that the best solution found was approximately $f(0.002, 0.002)$, which evaluated to about 0.009, which is much better than a single run of the hill climbing algorithm.

```
Restart 0, best: f([-0.98102417 1.96555308]) = 5.38194
Restart 2, best: f([1.96522236 0.98120013]) = 5.38191
Restart 4, best: f([0.00223194 0.00258853]) = 0.00998
Done!
f([0.00223194 0.00258853]) = 0.009978
```

Output 16.2: Result from Program 16.12

Next, let's look at how we can implement the iterated local search algorithm.

16.6   Iterated Local Search Algorithm

The Iterated Local Search algorithm is a modified version of the stochastic hill climbing with random restarts algorithm. The important difference is that the starting point for each application of the stochastic hill climbing algorithm is a perturbed version of the best point found so far. We can implement this algorithm by using the `random_restarts()` function as a starting point. Each restart iteration, we can generate a modified version of the best solution found so far instead of a random starting point. This can be achieved by using a step size hyperparameter, much like is used in the stochastic hill climber. In this case, a larger step size value will be used given the need for larger perturbations in the search space.

```
...
start_pt = None
while start_pt is None or not in_bounds(start_pt, bounds):
    start_pt = best + randn(len(bounds)) * p_size
```

Program 16.13: Generate an initial point as a perturbed version of the last best

Tying this together, the `iterated_local_search()` function is defined below.

```
def iterated_local_search(objective, bounds, n_iter, step_size, n_restarts, p_size):
    # define starting point
    best = None
    while best is None or not in_bounds(best, bounds):
        best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate current best point
    best_eval = objective(best)
    # enumerate restarts
    for n in range(n_restarts):
        # generate an initial point as a perturbed version of the last best
        start_pt = None
        while start_pt is None or not in_bounds(start_pt, bounds):
            start_pt = best + randn(len(bounds)) * p_size
        # perform a stochastic hill climbing search
        solution, solution_eval = hillclimbing(objective, bounds, n_iter,
            step_size, start_pt)
        # check for new best
        if solution_eval < best_eval:
            best, best_eval = solution, solution_eval
            print('Restart %d, best: f(%s) = %.5f' % (n, best, best_eval))
    return [best, best_eval]
```

Program 16.14: Iterated local search algorithm

We can then apply the algorithm to the Ackley objective function. In this case, we will use a larger step size value of 1.0 for the random restarts, chosen after a little trial and error.

The complete example is listed below.

```python
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi \
     * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size, start_pt):
    # store the initial point
    solution = start_pt
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
    return [solution, solution_eval]

# iterated local search algorithm
def iterated_local_search(objective, bounds, n_iter, step_size, n_restarts, p_size):
    # define starting point
    best = None
    while best is None or not in_bounds(best, bounds):
        best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate current best point
    best_eval = objective(best)
    # enumerate restarts
```

```
        for n in range(n_restarts):
            # generate an initial point as a perturbed version of the last best
            start_pt = None
            while start_pt is None or not in_bounds(start_pt, bounds):
                start_pt = best + randn(len(bounds)) * p_size
            # perform a stochastic hill climbing search
            solution, solution_eval = hillclimbing(objective, bounds, n_iter,
            ↳    step_size, start_pt)
            # check for new best
            if solution_eval < best_eval:
                best, best_eval = solution, solution_eval
                print('Restart %d, best: f(%s) = %.5f' % (n, best, best_eval))
        return [best, best_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iter = 1000
# define the maximum step size
s_size = 0.05
# total number of random restarts
n_restarts = 30
# perturbation step size
p_size = 1.0
# perform the hill climbing search
best, score = iterated_local_search(objective, bounds, n_iter, s_size,
↳    n_restarts, p_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 16.15: Iterated local search of the Ackley objective function

Running the example will perform an Iterated Local Search of the Ackley objective function. Each time an improved overall solution is discovered, it is reported and the final best solution found by the search is summarized at the end of the run.

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 four improvements during the search and that the best solution found was two very small inputs that are close to zero, which evaluated to about 0.0003, which is better than either a single run of the hill climber or the hill climber with restarts.

```
Restart 0, best: f([-0.96775653 0.96853129]) = 3.57447
Restart 3, best: f([-4.50618519e-04 9.51020713e-01]) = 2.57996
Restart 5, best: f([ 0.00137423 -0.00047059]) = 0.00416
Restart 22, best: f([ 1.16431936e-04 -3.31358206e-06]) = 0.00033
```

```
Done!
f([ 1.16431936e-04 -3.31358206e-06]) = 0.000330
```

Output 16.3: Result from Program 16.15

16.7   Further Reading

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

Books

Sean Luke. *Essentials of Metaheuristics.* lulu.com, 2011.
    https://amzn.to/3lHryZr
Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics.* 3rd ed. Springer, 2019.
    https://amzn.to/2IIq0Qt

Articles

Hill climbing. Wikipedia.
    https://en.wikipedia.org/wiki/Hill_climbing
Iterated local search. Wikipedia.
    https://en.wikipedia.org/wiki/Iterated_local_search

16.8   Summary

In this tutorial, you discovered how to implement the iterated local search algorithm from scratch. Specifically, you learned:

▷ Iterated local search is a stochastic global search optimization algorithm that is a smarter version of stochastic hill climbing with random restarts.

▷ How to implement stochastic hill climbing with random restarts from scratch.

▷ How to implement and apply the iterated local search algorithm to a nonlinear objective function.

Next, you will depart from local optimization and learn about global optimization, start with genetic algorithm.

IV Global Optimization

Simple Genetic Algorithm from Scratch

<div style="text-align: right">**17**</div>

The *genetic algorithm* is a stochastic global optimization algorithm. It may be one of the most popular and widely known biologically inspired algorithms, along with artificial neural networks. The algorithm is a type of evolutionary algorithm and performs an optimization procedure inspired by the biological theory of evolution by means of natural selection with a binary representation and simple operators based on genetic recombination and genetic mutations.

In this tutorial, you will discover the genetic algorithm optimization algorithm. After completing this tutorial, you will know:

▷ Genetic algorithm is a stochastic optimization algorithm inspired by evolution.

▷ How to implement the genetic algorithm from scratch in Python.

▷ How to apply the genetic algorithm to a continuous objective function.

Let's get started.

17.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Genetic Algorithm
2. Genetic Algorithm From Scratch
3. Genetic Algorithm for OneMax
4. Genetic Algorithm for Continuous Function Optimization

17.2 Genetic Algorithm

The Genetic Algorithm[1] is a stochastic global search optimization algorithm. It is inspired by the biological theory of evolution by means of natural selection. Specifically, the new synthesis that combines an understanding of genetics with the theory.

---

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

> ❝ Genetic algorithms (algorithm 9.4) borrow inspiration from biological evolution, where fitter individuals are more likely to pass on their genes to the next generation. ❞

— Page 148, *Algorithms for Optimization*, 2019.

The algorithm uses analogs of a genetic representation (bitstrings), fitness (function evaluations), genetic recombination (crossover of bitstrings), and mutation (flipping bits). The algorithm works by first creating a population of a fixed size of random bitstrings. The main loop of the algorithm is repeated for a fixed number of iterations or until no further improvement is seen in the best solution over a given number of iterations.

One iteration of the algorithm is like an evolutionary generation. First, the population of bitstrings (candidate solutions) are evaluated using the objective function. The objective function evaluation for each candidate solution is taken as the fitness of the solution, which may be minimized or maximized. Then, parents are selected based on their fitness. A given candidate solution may be used as parent zero or more times. A simple and effective approach to selection involves drawing k candidates from the population randomly and selecting the member from the group with the best fitness. This is called tournament selection where k is a hyperparameter and set to a value such as 3. This simple approach simulates a more costly fitness-proportionate selection scheme.

> ❝ In tournament selection, each parent is the fittest out of k randomly chosen chromosomes of the population ❞

— Page 151, *Algorithms for Optimization*, 2019.

Parents are used as the basis for generating the next generation of candidate points and one parent for each position in the population is required. Parents are then taken in pairs and used to create two children. Recombination is performed using a crossover operator. This involves selecting a random split point on the bit string, then creating a child with the bits up to the split point from the first parent and from the split point to the end of the string from the second parent. This process is then inverted for the second child.

For example the two parents:

  ▷ parent1 = 00000

  ▷ parent2 = 11111

May result in two cross-over children:

  ▷ child1 = 00011

  ▷ child2 = 11100

This is called one point crossover, and there are many other variations of the operator. Crossover is applied probabilistically for each pair of parents, meaning that in some cases, copies of the parents are taken as the children instead of the recombination operator. Crossover is controlled by a hyperparameter set to a large value, such as 80 percent or 90 percent.

> ❝ Crossover is the Genetic Algorithm's distinguishing feature. It involves mixing and matching parts of two parents to form children. How you do that mixing and matching depends on the representation of the individuals. ❞

— Page 36, *Essentials of Metaheuristics*, 2011.

Mutation involves flipping bits in created children candidate solutions. Typically, the mutation rate is set to $1/L$, where L is the length of the bitstring.

> Each bit in a binary-valued chromosome typically has a small probability of being flipped. For a chromosome with m bits, this mutation rate is typically set to $1/m$, yielding an average of one mutation per child chromosome.

<div align="right">— Page 155, Algorithms for Optimization, 2019.</div>

For example, if a problem used a bitstring with 20 bits, then a good default mutation rate would be $(1/20) = 0.05$ or a probability of 5 percent. This defines the simple genetic algorithm procedure. It is a large field of study, and there are many extensions to the algorithm.

Now that we are familiar with the simple genetic algorithm procedure, let's look at how we might implement it from scratch.

17.3 Genetic Algorithm From Scratch

In this section, we will develop an implementation of the genetic algorithm. The first step is to create a population of random bitstrings. We could use boolean values `True` and `False`, string values `"0"` and `"1"`, or integer values `0` and `1`. In this case, we will use integer values.

We can generate an array of integer values in a range using the `randint()` function[2], and we can specify the range as values starting at 0 and less than 2, e.g. 0 or 1. We will also represent a candidate solution as a list instead of a NumPy array to keep things simple. An initial population of random bitstring can be created as follows, where "`n_pop`" is a hyperparameter that controls the population size and "`n_bits`" is a hyperparameter that defines the number of bits in a single candidate solution:

```
...
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
```

<div align="center">Program 17.1: Initial population of random bitstring</div>

Next, we can enumerate over a fixed number of algorithm iterations, in this case, controlled by a hyperparameter named "`n_iter`".

```
...
    for gen in range(n_iter):
        ...
```

<div align="center">Program 17.2: Enumerate generations</div>

The first step in the algorithm iteration is to evaluate all candidate solutions. We will use a function named `objective()` as a generic objective function and call it to get a fitness score, which we will minimize.

```
...
scores = [objective(c) for c in pop]
```

<div align="center">Program 17.3: Evaluate all candidates in the population</div>

---

[2]https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

We can then select parents that will be used to create children. The tournament selection procedure can be implemented as a function that takes the population and returns one selected parent. The k value is fixed at 3 with a default argument, but you can experiment with different values if you like.

```python
def selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]
```

Program 17.4: Tournament selection

We can then call this function one time for each position in the population to create a list of parents.

```python
...
selected = [selection(pop, scores) for _ in range(n_pop)]
```

Program 17.5: Select parents

We can then create the next generation. This first requires a function to perform crossover. This function will take two parents and the crossover rate. The crossover rate is a hyperparameter that determines whether crossover is performed or not, and if not, the parents are copied into the next generation. It is a probability and typically has a large value close to 1.0.

The `crossover()` function below implements crossover using a draw of a random number in the range [0,1] to determine if crossover is performed, then selecting a valid split point if crossover is to be performed.

```python
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string
        pt = randint(1, len(p1)-2)
        # perform crossover
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]
```

Program 17.6: Crossover two parents to create two children

We also need a function to perform mutation. This procedure simply flips bits with a low probability controlled by the "`r_mut`" hyperparameter.

```
def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        # check for a mutation
        if rand() < r_mut:
            # flip the bit
            bitstring[i] = 1 - bitstring[i]
```

Program 17.7: Mutation operator

We can then loop over the list of parents and create a list of children to be used as the next generation, calling the crossover and mutation functions as needed.

```
...
children = list()
for i in range(0, n_pop, 2):
    # get selected parents in pairs
    p1, p2 = selected[i], selected[i+1]
    # crossover and mutation
    for c in crossover(p1, p2, r_cross):
        # mutation
        mutation(c, r_mut)
        # store for next generation
        children.append(c)
```

Program 17.8: Create the next generation

We can tie all of this together into a function named `genetic_algorithm()` that takes the name of the objective function and the hyperparameters of the search, and returns the best solution found during the search.

```
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(pop[0])
    # enumerate generations
    for gen in range(n_iter):
        # evaluate all candidates in the population
        scores = [objective(c) for c in pop]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
        # select parents
        selected = [selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
```

```
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]
```
Program 17.9: Genetic algorithm

Now that we have developed an implementation of the genetic algorithm, let's explore how we might apply it to an objective function.

17.4   Genetic Algorithm for OneMax

In this section, we will apply the genetic algorithm to a binary string-based optimization problem. The problem is called OneMax and evaluates a binary string based on the number of 1s in the string. For example, a bitstring with a length of 20 bits will have a score of 20 for a string of all 1s. Given we have implemented the genetic algorithm to minimize the objective function, we can add a negative sign to this evaluation so that large positive values become large negative values. The `onemax()` function below implements this and takes a bitstring of integer values as input and returns the negative sum of the values.

```
def onemax(x):
    return -sum(x)
```
Program 17.10: Objective function

Next, we can configure the search. The search will run for 100 iterations and we will use 20 bits in our candidate solutions, meaning the optimal fitness will be -20.0. The population size will be 100, and we will use a crossover rate of 90 percent and a mutation rate of 5 percent. This configuration was chosen after a little trial and error.

```
...
# define the total iterations
n_iter = 100
# bits
n_bits = 20
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / float(n_bits)
```
Program 17.11: Define the hyperparameters

The search can then be called and the best result reported.

```
...
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 17.12: Perform the genetic algorithm search

Tying this together, the complete example of applying the genetic algorithm to the OneMax objective function is listed below.

```python
from numpy.random import randint
from numpy.random import rand

# objective function
def onemax(x):
    return -sum(x)

# tournament selection
def selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string
        pt = randint(1, len(p1)-2)
        # perform crossover
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        # check for a mutation
        if rand() < r_mut:
            # flip the bit
            bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(pop[0])
```

```
    # enumerate generations
    for gen in range(n_iter):
        # evaluate all candidates in the population
        scores = [objective(c) for c in pop]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
        # select parents
        selected = [selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]

# define the total iterations
n_iter = 100
# bits
n_bits = 20
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / float(n_bits)
# perform the genetic algorithm search
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 17.13: Genetic algorithm search of the one max optimization problem

Running the example will report the best result as it is found along the way, then the final best solution at the end of the search, which we would expect to be the optimal solution.

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 found the optimal solution after about eight generations.

```
>0, new best f([1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]) = -14.000
>0, new best f([1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0]) = -15.000
>1, new best f([1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1]) = -16.000
>2, new best f([0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]) = -17.000
>2, new best f([1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -19.000
>8, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Done!
f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000
```

Output 17.1: Result from Program 17.13

17.5 Genetic Algorithm for Continuous Function Optimization

Optimizing the OneMax function is not very interesting; we are more likely to want to optimize a continuous function. For example, we can define the x^2 minimization function that takes input variables and has an optima at Âă$f(0,0) = 0.0$.

```
def objective(x):
    return x[0]**2.0 + x[1]**2.0
```

Program 17.14: Objective function

We can minimize this function with a genetic algorithm. First, we must define the bounds of each input variable.

```
...
bounds = [[-5.0, 5.0], [-5.0, 5.0]]
```

Program 17.15: Define range for input

We will take the "`n_bits`" hyperparameter as a number of bits per input variable to the objective function and set it to 16 bits.

```
...
n_bits = 16
```

Program 17.16: Define bits per variable

This means our actual bit string will have $(16 \times 2) = 32$ bits, given the two input variables. We must update our mutation rate accordingly.

```
...
r_mut = 1.0 / (float(n_bits) * len(bounds))
```

Program 17.17: Define mutation rate

Next, we need to ensure that the initial population creates random bitstrings that are large enough.

```
...
pop = [randint(0, 2, n_bits*len(bounds)).tolist() for _ in range(n_pop)]
```

Program 17.18: Initial population of random bitstring

Finally, we need to decode the bitstrings to numbers prior to evaluating each with the objective function. We can achieve this by first decoding each substring to an integer, then scaling the integer to the desired range. This will give a vector of values in the range that can then be provided to the objective function for evaluation. The `decode()` function below implements this, taking the bounds of the function, the number of bits per variable, and a bitstring as input and returns a list of decoded real values.

```python
def decode(bounds, n_bits, bitstring):
    decoded = list()
    largest = 2**n_bits
    for i in range(len(bounds)):
        # extract the substring
        start, end = i * n_bits, (i * n_bits)+n_bits
        substring = bitstring[start:end]
        # convert bitstring to a string of chars
        chars = ''.join([str(s) for s in substring])
        # convert string to integer
        integer = int(chars, 2)
        # scale integer to desired range
        value = bounds[i][0] + (integer/largest) * (bounds[i][1] - bounds[i][0])
        # store
        decoded.append(value)
    return decoded
```

Program 17.19: Decode bitstring to numbers

We can then call this at the beginning of the algorithm loop to decode the population, then evaluate the decoded version of the population.

```python
...
# decode population
decoded = [decode(bounds, n_bits, p) for p in pop]
# evaluate all candidates in the population
scores = [objective(d) for d in decoded]
```

Program 17.20: Evaluate candidates in the population

Tying this together, the complete example of the genetic algorithm for continuous function optimization is listed below.

```python
from numpy.random import randint
from numpy.random import rand

# objective function
def objective(x):
    return x[0]**2.0 + x[1]**2.0

# decode bitstring to numbers
```

```python
def decode(bounds, n_bits, bitstring):
    decoded = list()
    largest = 2**n_bits
    for i in range(len(bounds)):
        # extract the substring
        start, end = i * n_bits, (i * n_bits)+n_bits
        substring = bitstring[start:end]
        # convert bitstring to a string of chars
        chars = ''.join([str(s) for s in substring])
        # convert string to integer
        integer = int(chars, 2)
        # scale integer to desired range
        value = bounds[i][0] + (integer/largest) * (bounds[i][1] - bounds[i][0])
        # store
        decoded.append(value)
    return decoded

# tournament selection
def selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string
        pt = randint(1, len(p1)-2)
        # perform crossover
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        # check for a mutation
        if rand() < r_mut:
            # flip the bit
            bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits*len(bounds)).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(decode(bounds, n_bits, pop[0]))
```

```python
    # enumerate generations
    for gen in range(n_iter):
        # decode population
        decoded = [decode(bounds, n_bits, p) for p in pop]
        # evaluate all candidates in the population
        scores = [objective(d) for d in decoded]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(">%d, new best f(%s) = %f" % (gen,  decoded[i], scores[i]))
        # select parents
        selected = [selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]

# define range for input
bounds = [[-5.0, 5.0], [-5.0, 5.0]]
# define the total iterations
n_iter = 100
# bits per variable
n_bits = 16
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / (float(n_bits) * len(bounds))
# perform the genetic algorithm search
best, score = genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop,
    r_cross, r_mut)
print('Done!')
decoded = decode(bounds, n_bits, best)
print('f(%s) = %f' % (decoded, score))
```

Program 17.21: Genetic algorithm search for continuous function optimization

Running the example reports the best decoded results along the way and the best decoded solution at the end of the run.

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 discovers an input very close to $f(0.0, 0.0) = 0.0$.

```
>0, new best f([-0.785064697265625, -0.807647705078125]) = 1.268621
>0, new best f([0.385894775390625, 0.342864990234375]) = 0.266471
>1, new best f([-0.342559814453125, -0.1068115234375]) = 0.128756
>2, new best f([-0.038909912109375, 0.30242919921875]) = 0.092977
>2, new best f([0.145721435546875, 0.1849365234375]) = 0.055436
>3, new best f([0.14404296875, -0.029754638671875]) = 0.021634
>5, new best f([0.066680908203125, 0.096435546875]) = 0.013746
>5, new best f([-0.036468505859375, -0.10711669921875]) = 0.012804
>6, new best f([-0.038909912109375, -0.099639892578125]) = 0.011442
>7, new best f([-0.033111572265625, 0.09674072265625]) = 0.010455
>7, new best f([-0.036468505859375, 0.05584716796875]) = 0.004449
>10, new best f([0.058746337890625, 0.008087158203125]) = 0.003517
>10, new best f([-0.031585693359375, 0.008087158203125]) = 0.001063
>12, new best f([0.022125244140625, 0.008087158203125]) = 0.000555
>13, new best f([0.022125244140625, 0.00701904296875]) = 0.000539
>13, new best f([-0.013885498046875, 0.008087158203125]) = 0.000258
>16, new best f([-0.011444091796875, 0.00518798828125]) = 0.000158
>17, new best f([-0.0115966796875, 0.00091552734375]) = 0.000135
>17, new best f([-0.004730224609375, 0.00335693359375]) = 0.000034
>20, new best f([-0.004425048828125, 0.00274658203125]) = 0.000027
>21, new best f([-0.002288818359375, 0.00091552734375]) = 0.000006
>22, new best f([-0.001983642578125, 0.00091552734375]) = 0.000005
>22, new best f([-0.001983642578125, 0.0006103515625]) = 0.000004
>24, new best f([-0.001373291015625, 0.001068115234375]) = 0.000003
>25, new best f([-0.001373291015625, 0.00091552734375]) = 0.000003
>26, new best f([-0.001373291015625, 0.0006103515625]) = 0.000002
>27, new best f([-0.001068115234375, 0.0006103515625]) = 0.000002
>29, new best f([-0.000152587890625, 0.00091552734375]) = 0.000001
>33, new best f([-0.0006103515625, 0.0]) = 0.000000
>34, new best f([-0.000152587890625, 0.00030517578125]) = 0.000000
>43, new best f([-0.00030517578125, 0.0]) = 0.000000
>60, new best f([-0.000152587890625, 0.000152587890625]) = 0.000000
>65, new best f([-0.000152587890625, 0.0]) = 0.000000
Done!
f([-0.000152587890625, 0.0]) = 0.000000
```

Output 17.2: Result from Program 17.21

17.6   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
Sean Luke. *Essentials of Metaheuristics*. lulu.com, 2011.
https://amzn.to/3lHryZr
David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 13th ed. Addison-Wesley, 1989.
https://amzn.to/3jADHgZ
Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.
https://amzn.to/3kK8Osd
Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. 2nd ed. Wiley, 2007.
https://amzn.to/3ob61KA

APIs

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

Articles

Genetic algorithm. Wikipedia.
https://en.wikipedia.org/wiki/Genetic_algorithm
Genetic algorithms. Scholarpedia.
http://www.scholarpedia.org/article/Genetic_algorithms

17.7   Summary

In this tutorial, you discovered the genetic algorithm optimization. Specifically, you learned:

▷ Genetic algorithm is a stochastic optimization algorithm inspired by evolution.

▷ How to implement the genetic algorithm from scratch in Python.

▷ How to apply the genetic algorithm to a continuous objective function.

Next, you will learn about evolution strategies.

Evolution Strategies

<div style="text-align: right; font-size: 3em;">18</div>

Evolution strategies is a stochastic global optimization algorithm. It is an evolutionary algorithm related to others, such as the genetic algorithm, although it is designed specifically for continuous function optimization.

In this tutorial, you will discover how to implement the evolution strategies optimization algorithm. After completing this tutorial, you will know:

▷ Evolution Strategies is a stochastic global optimization algorithm inspired by the biological theory of evolution by natural selection.

▷ There is a standard terminology for Evolution Strategies and two common versions of the algorithm referred to as (μ, λ)-ES and $(\mu + \lambda)$-ES.

▷ How to implement and apply the Evolution Strategies algorithm to continuous objective functions.

Let's get started.

18.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Evolution Strategies
2. Develop a (μ, λ)-ES
3. Develop a $(\mu + \lambda)$-ES

18.2   Evolution Strategies

Evolution Strategies[1], sometimes referred to as Evolution Strategy (singular) or ES, is a stochastic global optimization algorithm. The technique was developed in the 1960s to be implemented manually by engineers for minimal drag designs in a wind tunnel.

---

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

> " The family of algorithms known as Evolution Strategies (ES) were developed by
> Ingo Rechenberg and Hans-Paul Schwefel at the Technical University of Berlin in
> the mid 1960s. "

— Page 31, *Essentials of Metaheuristics*, 2011.

Evolution Strategies is a type of evolutionary algorithm and is inspired by the biological theory of evolution by means of natural selection. Unlike other evolutionary algorithms, it does not use any form of crossover; instead, modification of candidate solutions is limited to mutation operators. In this way, Evolution Strategies may be thought of as a type of parallel stochastic hill climbing. The algorithm involves a population of candidate solutions that initially are randomly generated. Each iteration of the algorithm involves first evaluating the population of solutions, then deleting all but a subset of the best solutions, referred to as truncation selection. The remaining solutions (the parents) each are used as the basis for generating a number of new candidate solutions (mutation) that replace or compete with the parents for a position in the population for consideration in the next iteration of the algorithm (generation).

There are a number of variations of this procedure and a standard terminology to summarize the algorithm. The size of the population is referred to as λ (lambda) and the number of parents selected each iteration is referred to as μ (mu). The number of children created from each parent is calculated as (λ/μ) and parameters should be chosen so that the division has no remainder.

⊳ μ: The number of parents selected each iteration.

⊳ λ: Size of the population.

⊳ λ/μ: Number of children generated from each selected parent.

A bracket notation is used to describe the algorithm configuration, i.e. (μ, λ)-ES. For example, if $\mu = 5$ and $\lambda = 20$, then it would be summarized as $(5, 20)$-ES. A comma (,) separating the μ and λ parameters indicates that the children replace the parents directly each iteration of the algorithm.

⊳ (μ, λ)-ES: A version of evolution strategies where children replace parents.

A plus (+) separation of the μ and λ parameters indicates that the children and the parents together will define the population for the next iteration.

⊳ $(\mu + \lambda)$-ES: A version of evolution strategies where children and parents are added to the population.

A stochastic hill climbing algorithm can be implemented as an Evolution Strategy and would have the notation $(1 + 1)$-ES. This is the simile or canonical ES algorithm and there are many extensions and variants described in the literature.

Now that we are familiar with Evolution Strategies we can explore how to implement the algorithm.

18.3 Develop a (μ, λ)-ES

In this section, we will develop a (μ, λ)-ES, that is, a version of the algorithm where children replace parents. First, let's define a challenging optimization problem as the basis for implementing the algorithm.

The Ackley function[2] is an example of a multimodal objective function that has a single global optima and multiple local optima in which a local search might get stuck. As such, a global optimization technique is required. It is a two-dimensional objective function that has a global optima at [0,0], which evaluates to 0.0. The example below implements the Ackley and creates a three-dimensional surface plot showing the global optima and multiple local optima.

```python
from numpy import arange
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi ⤸
        ↳ * x) + cos(2 * pi * y))) + e + 20

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 18.1: Ackley multimodal function

Running the example creates the surface plot of the Ackley function showing the vast number of local optima.

---

[2]https://en.wikipedia.org/wiki/Ackley_function
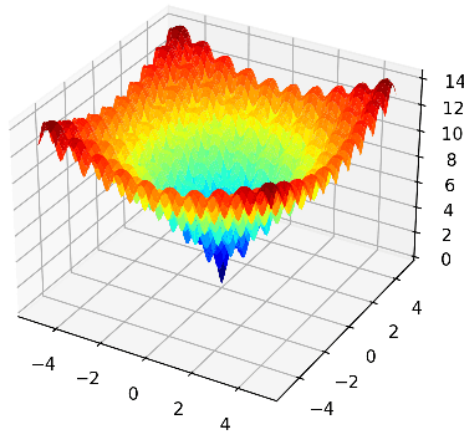
Figure 18.1: 3D Surface Plot of the Ackley Multimodal Function

We will be generating random candidate solutions as well as modified versions of existing candidate solutions. It is important that all candidate solutions are within the bounds of the search problem. To achieve this, we will develop a function to check whether a candidate solution is within the bounds of the search and then discard it and generate another solution if it is not. The `in_bounds()` function below will take a candidate solution (point) and the definition of the bounds of the search space (bounds) and return True if the solution is within the bounds of the search or False otherwise.

```
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True
```

Program 18.2: Check if a point is within the bounds of the search

We can then use this function when generating the initial population of λ (i.e., "`lam`") random candidate solutions. For example:

```
...
population = list()
for _ in range(lam):
    candidate = None
    while candidate is None or not in_bounds(candidate, bounds):
        candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    population.append(candidate)
```

Program 18.3: Initial population

Next, we can iterate over a fixed number of iterations of the algorithm. Each iteration first involves evaluating each candidate solution in the population. We will calculate the scores and store them in a separate parallel list.

```
...
scores = [objective(c) for c in population]
```
Program 18.4: Evaluate fitness for the population

Next, we need to select the μ (i.e., "mu") parents with the best scores, lowest scores in this case, as we are minimizing the objective function. We will do this in two steps. First, we will rank the candidate solutions based on their scores in ascending order so that the solution with the lowest score has a rank of 0, the next has a rank 1, and so on. We can use a double call of the `argsort()` function[3] to achieve this. We will then use the ranks and select those parents that have a rank below the value "mu." This means if mu is set to 5 to select 5 parents, only those parents with a rank between 0 and 4 will be selected.

```
...
# rank scores in ascending order
ranks = argsort(argsort(scores))
# select the indexes for the top mu ranked solutions
selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
```
Program 18.5: Select parents by ranking

We can then create children for each selected parent. First, we must calculate the total number of children to create per parent.

```
...
n_children = int(lam / mu)
```
Program 18.6: Calculate the number of children per parent

We can then iterate over each parent and create modified versions of each. We will create children using a similar technique used in stochastic hill climbing. Specifically, each variable will be sampled using a Gaussian distribution with the current value as the mean and the standard deviation provided as a "`step_size`" hyperparameter.

```
...
for _ in range(n_children):
    child = None
    while child is None or not in_bounds(child, bounds):
        child = population[i] + randn(len(bounds)) * step_size
```
Program 18.7: Create children for parent

We can also check if each selected parent is better than the best solution seen so far so that we can return the best solution at the end of the search.

---

[3]https://numpy.org/doc/stable/reference/generated/numpy.argsort.html

```
...
if scores[i] < best_eval:
    best, best_eval = population[i], scores[i]
    print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
```

Program 18.8: Check if this parent is the best solution ever seen

The created children can be added to a list and we can replace the population with the list of children at the end of the algorithm iteration.

```
...
population = children
```

Program 18.9: Replace population with children

We can tie all of this together into a function named `es_comma()` that performs the comma version of the Evolution Strategy algorithm. The function takes the name of the objective function, the bounds of the search space, the number of iterations, the step size, and the mu and lambda hyperparameters and returns the best solution found during the search and its evaluation.

```
def es_comma(objective, bounds, n_iter, step_size, mu, lam):
    best, best_eval = None, 1e+10
    # calculate the number of children per parent
    n_children = int(lam / mu)
    # initial population
    population = list()
    for _ in range(lam):
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
                bounds[:, 0])
        population.append(candidate)
    # perform the search
    for epoch in range(n_iter):
        # evaluate fitness for the population
        scores = [objective(c) for c in population]
        # rank scores in ascending order
        ranks = argsort(argsort(scores))
        # select the indexes for the top mu ranked solutions
        selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
        # create children from parents
        children = list()
        for i in selected:
            # check if this parent is the best solution ever seen
            if scores[i] < best_eval:
                best, best_eval = population[i], scores[i]
                print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
            # create children for parent
            for _ in range(n_children):
                child = None
                while child is None or not in_bounds(child, bounds):
                    child = population[i] + randn(len(bounds)) * step_size
```

```
                children.append(child)
        # replace population with children
        population = children
    return [best, best_eval]
```

Program 18.10: Evolution strategy (μ, λ) algorithm

Next, we can apply this algorithm to our Ackley objective function. We will run the algorithm for 5,000 iterations and use a step size of 0.15 in the search space. We will use a population size (λ) of 100 select 20 parents (μ). These hyperparameters were chosen after a little trial and error. At the end of the search, we will report the best candidate solution found during the search.

```
...
# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iter = 5000
# define the maximum step size
step_size = 0.15
# number of parents selected
mu = 20
# the number of children generated by parents
lam = 100
# perform the evolution strategy (mu, lambda) search
best, score = es_comma(objective, bounds, n_iter, step_size, mu, lam)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 18.11: Search and report the best candidate solution

Tying this together, the complete example of applying the comma version of the Evolution Strategies algorithm to the Ackley objective function is listed below.

```
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import argsort
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi
     * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
```

```python
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True

# evolution strategy (mu, lambda) algorithm
def es_comma(objective, bounds, n_iter, step_size, mu, lam):
    best, best_eval = None, 1e+10
    # calculate the number of children per parent
    n_children = int(lam / mu)
    # initial population
    population = list()
    for _ in range(lam):
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
                bounds[:, 0])
        population.append(candidate)
    # perform the search
    for epoch in range(n_iter):
        # evaluate fitness for the population
        scores = [objective(c) for c in population]
        # rank scores in ascending order
        ranks = argsort(argsort(scores))
        # select the indexes for the top mu ranked solutions
        selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
        # create children from parents
        children = list()
        for i in selected:
            # check if this parent is the best solution ever seen
            if scores[i] < best_eval:
                best, best_eval = population[i], scores[i]
                print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
            # create children for parent
            for _ in range(n_children):
                child = None
                while child is None or not in_bounds(child, bounds):
                    child = population[i] + randn(len(bounds)) * step_size
                children.append(child)
        # replace population with children
        population = children
    return [best, best_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iter = 5000
# define the maximum step size
step_size = 0.15
```

```
# number of parents selected
mu = 20
# the number of children generated by parents
lam = 100
# perform the evolution strategy (mu, lambda) search
best, score = es_comma(objective, bounds, n_iter, step_size, mu, lam)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 18.12: Complete code for (μ, λ)-ES search of Ackley objective function

Running the example reports the candidate solution and scores each time a better solution is found, then reports the best solution found at the end of the search.

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 about 22 improvements to performance were seen during the search and the best solution is close to the optima. No doubt, this solution can be provided as a starting point to a local search algorithm to be further refined, a common practice when using a global optimization algorithm like ES.

```
0, Best: f([-0.82977995 2.20324493]) = 6.91249
0, Best: f([-1.03232526 0.38816734]) = 4.49240
1, Best: f([-1.02971385 0.21986453]) = 3.68954
2, Best: f([-0.98361735 0.19391181]) = 3.40796
2, Best: f([-0.98189724 0.17665892]) = 3.29747
2, Best: f([-0.07254927 0.67931431]) = 3.29641
3, Best: f([-0.78716147 0.02066442]) = 2.98279
3, Best: f([-1.01026218 -0.03265665]) = 2.69516
3, Best: f([-0.08851828 0.26066485]) = 2.00325
4, Best: f([-0.23270782 0.04191618]) = 1.66518
4, Best: f([-0.01436704 0.03653578]) = 0.15161
7, Best: f([0.01247004 0.01582657]) = 0.06777
9, Best: f([0.00368129 0.00889718]) = 0.02970
25, Best: f([ 0.00666975 -0.0045051 ]) = 0.02449
33, Best: f([-0.00072633 -0.00169092]) = 0.00530
211, Best: f([2.05200123e-05 1.51343187e-03]) = 0.00434
315, Best: f([ 0.00113528 -0.00096415]) = 0.00427
418, Best: f([ 0.00113735 -0.00030554]) = 0.00337
491, Best: f([ 0.00048582 -0.00059587]) = 0.00219
704, Best: f([-6.91643854e-04 -4.51583644e-05]) = 0.00197
1504, Best: f([ 2.83063223e-05 -4.60893027e-04]) = 0.00131
3725, Best: f([ 0.00032757 -0.00023643]) = 0.00115
Done!
f([ 0.00032757 -0.00023643]) = 0.001147
```

Output 18.1: Result from Program 18.12

Now that we are familiar with how to implement the comma version of evolution strategies, let's look at how we might implement the plus version.

18.4 Develop a $(\mu + \lambda)$-ES

The plus version of the Evolution Strategies algorithm is very similar to the comma version. The main difference is that children and the parents comprise the population at the end instead of just the children. This allows the parents to compete with the children for selection in the next iteration of the algorithm. This can result in a more greedy behavior by the search algorithm and potentially premature convergence to local optima (suboptimal solutions). The benefit is that the algorithm is able to exploit good candidate solutions that were found and focus intently on candidate solutions in the region, potentially finding further improvements.

We can implement the plus version of the algorithm by modifying the function to add parents to the population when creating the children.

```
...
children.append(population[i])
```

Program 18.13: Keep the parent

The updated version of the function with this addition, and with a new name es_plus(), is listed below.

```python
def es_plus(objective, bounds, n_iter, step_size, mu, lam):
    best, best_eval = None, 1e+10
    # calculate the number of children per parent
    n_children = int(lam / mu)
    # initial population
    population = list()
    for _ in range(lam):
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
              ↪  bounds[:, 0])
        population.append(candidate)
    # perform the search
    for epoch in range(n_iter):
        # evaluate fitness for the population
        scores = [objective(c) for c in population]
        # rank scores in ascending order
        ranks = argsort(argsort(scores))
        # select the indexes for the top mu ranked solutions
        selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
        # create children from parents
        children = list()
        for i in selected:
            # check if this parent is the best solution ever seen
            if scores[i] < best_eval:
                best, best_eval = population[i], scores[i]
                print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
            # keep the parent
            children.append(population[i])
            # create children for parent
            for _ in range(n_children):
```

```
                child = None
                while child is None or not in_bounds(child, bounds):
                    child = population[i] + randn(len(bounds)) * step_size
                children.append(child)
        # replace population with children
        population = children
    return [best, best_eval]
```

Program 18.14: Evolution strategy $(\mu + \lambda)$ algorithm

We can apply this version of the algorithm to the Ackley objective function with the same hyperparameters used in the previous section.

The complete example is listed below.

```
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import argsort
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi  ⤸
     ↳  * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True

# evolution strategy (mu + lambda) algorithm
def es_plus(objective, bounds, n_iter, step_size, mu, lam):
    best, best_eval = None, 1e+10
    # calculate the number of children per parent
    n_children = int(lam / mu)
    # initial population
    population = list()
    for _ in range(lam):
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -     ⤸
             ↳  bounds[:, 0])
        population.append(candidate)
```

```
    # perform the search
    for epoch in range(n_iter):
        # evaluate fitness for the population
        scores = [objective(c) for c in population]
        # rank scores in ascending order
        ranks = argsort(argsort(scores))
        # select the indexes for the top mu ranked solutions
        selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
        # create children from parents
        children = list()
        for i in selected:
            # check if this parent is the best solution ever seen
            if scores[i] < best_eval:
                best, best_eval = population[i], scores[i]
                print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
            # keep the parent
            children.append(population[i])
            # create children for parent
            for _ in range(n_children):
                child = None
                while child is None or not in_bounds(child, bounds):
                    child = population[i] + randn(len(bounds)) * step_size
                children.append(child)
        # replace population with children
        population = children
    return [best, best_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iter = 5000
# define the maximum step size
step_size = 0.15
# number of parents selected
mu = 20
# the number of children generated by parents
lam = 100
# perform the evolution strategy (mu + lambda) search
best, score = es_plus(objective, bounds, n_iter, step_size, mu, lam)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 18.15: Complete code for $(\mu + \lambda)$*-ES search of Ackley objective function*

Running the example reports the candidate solution and scores each time a better solution is found, then reports the best solution found at the end of the search.

> ⚠️ **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 about 24 improvements to performance were seen during the search. We can also see that a better final solution was found with an evaluation of 0.000532, compared to 0.001147 found with the comma version on this objective function.

```
0, Best: f([-0.82977995 2.20324493]) = 6.91249
0, Best: f([-1.03232526 0.38816734]) = 4.49240
1, Best: f([-1.02971385 0.21986453]) = 3.68954
2, Best: f([-0.96315064 0.21176994]) = 3.48942
2, Best: f([-0.9524528 -0.19751564]) = 3.39266
2, Best: f([-1.02643442 0.14956346]) = 3.24784
2, Best: f([-0.90172166 0.15791013]) = 3.17090
2, Best: f([-0.15198636 0.42080645]) = 3.08431
3, Best: f([-0.76669476 0.03852254]) = 3.06365
3, Best: f([-0.98979547 -0.01479852]) = 2.62138
3, Best: f([-0.10194792 0.33439734]) = 2.52353
3, Best: f([0.12633886 0.27504489]) = 2.24344
4, Best: f([-0.01096566 0.22380389]) = 1.55476
4, Best: f([0.16241469 0.12513091]) = 1.44068
5, Best: f([-0.0047592 0.13164993]) = 0.77511
5, Best: f([ 0.07285478 -0.0019298 ]) = 0.34156
6, Best: f([-0.0323925 -0.06303525]) = 0.32951
6, Best: f([0.00901941 0.0031937 ]) = 0.02950
32, Best: f([ 0.00275795 -0.00201658]) = 0.00997
109, Best: f([-0.00204732 0.00059337]) = 0.00615
195, Best: f([-0.00101671 0.00112202]) = 0.00434
555, Best: f([ 0.00020392 -0.00044394]) = 0.00139
2804, Best: f([3.86555110e-04 6.42776651e-05]) = 0.00111
4357, Best: f([ 0.00013889 -0.0001261 ]) = 0.00053
Done!
f([ 0.00013889 -0.0001261 ]) = 0.000532
```

Output 18.2: Result from Program 18.15

18.5   Further Reading

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

Papers

Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution Strategies — A Comprehensive Introduction". *Natural Computing*, 1, 2002, pp. 3–52.
https://link.springer.com/article/10.1023/A:1015059928466

Books

Sean Luke. *Essentials of Metaheuristics*. lulu.com, 2011.
https://amzn.to/3lHryZr
Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J

Andries P. Engelbrecht. *Computational Intelligence: An Introduction.* 2nd ed. Wiley, 2007.
https://amzn.to/3ob61KA

Articles

Evolution strategy. Wikipedia.
https://en.wikipedia.org/wiki/Evolution_strategy
Evolution strategies. Scholarpedia.
http://www.scholarpedia.org/article/Evolution_strategies

18.6 Summary

In this tutorial, you discovered how to implement the evolution strategies optimization algorithm. Specifically, you learned:

▷ Evolution Strategies is a stochastic global optimization algorithm inspired by the biological theory of evolution by natural selection.

▷ There is a standard terminology for Evolution Strategies and two common versions of the algorithm referred to as (μ, λ)-ES and $(\mu + \lambda)$-ES.

▷ How to implement and apply the Evolution Strategies algorithm to continuous objective functions.

Next, you will learn about differential evolution.

Differential Evolution

19

Differential evolution is a heuristic approach for the global optimization of nonlinear and non-differentiable continuous space functions. The differential evolution algorithm belongs to a broader family of evolutionary computing algorithms. Similar to other popular direct search approaches, such as genetic algorithms and evolution strategies, the differential evolution algorithm starts with an initial population of candidate solutions. These candidate solutions are iteratively improved by introducing mutations into the population, and retaining the fittest candidate solutions that yield a lower objective function value. The differential evolution algorithm is advantageous over the aforementioned popular approaches because it can handle nonlinear and non-differentiable multi-dimensional objective functions, while requiring very few control parameters to steer the minimisation. These characteristics make the algorithm easier and more practical to use.

In this tutorial, you will discover the differential evolution algorithm for global optimization. After completing this tutorial, you will know:

▷ Differential evolution is a heuristic approach for the global optimization of nonlinear and non-differentiable continuous space functions.

▷ How to implement the differential evolution algorithm from scratch in Python.

▷ How to apply the differential evolution algorithm to a real-valued 2D objective function.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Differential Evolution
2. Differential Evolution Algorithm From Scratch
3. Differential Evolution Algorithm on the Sphere Function

19.2   Differential Evolution

Differential evolution is a heuristic approach for the global optimization of nonlinear and non-differentiable continuous space functions. For a minimisation algorithm to be considered practical, it is expected to fulfil five different requirements:

> (1) Ability to handle non-differentiable, nonlinear and multimodal cost functions.
> (2) Parallelizability to cope with computation intensive cost functions.
> (3) Ease of use, i.e. few control variables to steer the minimization. These variables should also be robust and easy to choose.
> (4) Good convergence properties, i.e. consistent convergence to the global minimum in consecutive independent trials.
>
> — "Differential evolution", 1997.

The strength of the differential evolution algorithm stems from the fact that it was designed to fulfil all of the above requirements.

> Differential Evolution (DE) is arguably one of the most powerful and versatile evolutionary optimizers for the continuous parameter spaces in recent times.
>
> — "Recent advances in differential evolution — An updated survey", 2016.

The algorithm begins by randomly initiating a population of real-valued decision vectors, also known as genomes or chromosomes. These represent the candidates solutions to the multi-dimensional optimization problem. At each iteration, the algorithm introduces mutations into the population to generate new candidate solutions. The mutation process adds the weighted difference between two population vectors to a third vector, to produce a mutated vector. The parameters of the mutated vector are again mixed with the parameters of another predetermined vector, the target vector, during a process known as crossover that aims to increase the diversity of the perturbed parameter vectors. The resulting vector is known as the trial vector.

> DE generates new parameter vectors by adding the weighted difference between two population vectors to a third vector. Let this operation be called mutation. In order to increase the diversity of the perturbed parameter vectors, crossover is introduced.
>
> — "Differential evolution", 1997.

These mutations are generated according to a mutation strategy, which follows a general naming convention of DE/x/y/z, where DE stands for Differential Evolution, while x denotes the vector to be mutated, y denotes the number of difference vectors considered for the mutation of x, and z is the type of crossover in use. For instance, the popular strategies:

▷ DE/rand/1/bin

▷ DE/best/2/bin

Specify that vector x can either be picked randomly (rand) from the population, or else the vector with the lowest cost (best) is selected; that the number of difference vectors under consideration is either 1 or 2; and that crossover is performed according to independent binomial (bin) experiments. The DE/best/2/bin strategy, in particular, appears to be highly beneficial in improving the diversity of the population if the population size is large enough.

> The usage of two difference vectors seems to improve the diversity of the population if the number of population vectors NP is high enough.
>
> — "Differential evolution", 1997.

A final selection operation replaces the target vector, or the parent, by the trial vector, its offspring, if the latter yields a lower objective function value. Hence, the fitter offspring now becomes a member of the newly generated population, and subsequently participates in the mutation of further population members. These iterations continue until a termination criterion is reached.

> The iterations continue till a termination criterion (such as exhaustion of maximum functional evaluations) is satisfied.
>
> — "Recent advances in differential evolution — An updated survey", 2016.

The differential evolution algorithm requires very few parameters to operate, namely the population size, NP, a real and constant scale factor, $F \in [0, 2]$, that weights the differential variation during the mutation process, and a crossover rate, $CR \in [0, 1]$, that is determined experimentally. This makes the algorithm easy and practical to use.

> In addition, the canonical DE requires very few control parameters (3 to be precise: the scale factor, the crossover rate and the population size) — a feature that makes it easy to use for the practitioners.
>
> — "Recent advances in differential evolution — An updated survey", 2016.

There have been further variants to the canonical differential evolution algorithm described above, which one may read on in *Recent advances in differential evolution - An updated survey*[1], 2016.

Now that we are familiar with the differential evolution algorithm, let's look at how to implement it from scratch.

19.3   Differential Evolution Algorithm From Scratch

In this section, we will explore how to implement the differential evolution algorithm from scratch. The differential evolution algorithm begins by generating an initial population of candidate solutions. For this purpose, we shall use the `rand()` function to generate an array of random values sampled from a uniform distribution over the range, $[0, 1]$. We will then scale these values to change the range of their distribution to (lower bound, upper bound), where the bounds are specified in the form of a 2D array with each dimension corresponding to each input variable.

```
...
pop = bounds[:, 0] + (rand(pop_size, len(bounds)) * (bounds[:, 1] - bounds[:, 0]))
```
Program 19.1: Initialise population of candidate solutions randomly within the specified bounds

---
[1] https://link.springer.com/article/10.1007/s10462-009-9137-2

It is within these same bounds that the objective function will also be evaluated. An objective function of choice and the bounds on each input variable may, therefore, be defined as follows:

```python
def obj(x):
    return 0

# define lower and upper bounds
bounds = asarray([-5.0, 5.0])
```

Program 19.2: Define objective function

We can evaluate our initial population of candidate solutions by passing it to the objective function as input argument.

```python
...
obj_all = [obj(ind) for ind in pop]
```

Program 19.3: Evaluate initial population of candidate solutions

We shall be replacing the values in `obj_all` with better ones as the population evolves and converges towards an optimal solution. We can then loop over a predefined number of iterations of the algorithm, such as 100 or 1,000, as specified by parameter, `iter`, as well as over all candidate solutions.

```python
...
for i in range(iter):
    # iterate over all candidate solutions
    for j in range(pop_size):
        ...
```

Program 19.4: Run iterations of the algorithm

The first step of the algorithm iteration performs a mutation process. For this purpose, three random candidates, a, b and c, that are not the current one, are randomly selected from the population and a mutated vector is generated by computing: $a + F \times (b - c)$. Recall that $F \in [0, 2]$ and denotes the mutation scale factor.

```python
...
candidates = [candidate for candidate in range(pop_size) if candidate != j]
a, b, c = pop[choice(candidates, 3, replace=False)]
```

Program 19.5: Choose three candidates that are not the current one

The mutation process is performed by the function, mutation, to which we pass `[a, b, c]` and F as input arguments.

```python
def mutation(x, F):
    return x[0] + F * (x[1] - x[2])
...
# perform mutation
mutated = mutation([a, b, c], F)
...
```

Program 19.6: Define mutation operation

Since we are operating within a bounded range of values, we need to check whether the newly mutated vector is also within the specified bounds, and if not, clip its values to the upper or lower limits as necessary. This check is carried out by the function, `check_bounds`.

```
def check_bounds(mutated, bounds):
    mutated_bound = [clip(mutated[i], bounds[i, 0], bounds[i, 1]) for i in
    ↳    range(len(bounds))]
    return mutated_bound
```

Program 19.7: Define boundary check operation

The next step performs crossover, where specific values of the current vector (`target`) are replaced by the corresponding values in the `mutated` vector, to create a `trial` vector. The decision of which values to replace is based on whether a uniform random value generated for each input variable falls below a crossover rate. If it does, then the corresponding values from the `mutated` vector are copied to the `target` vector. The crossover process is implemented by the `crossover()` function, which takes the `mutated` and `target` vectors as input, as well as the crossover rate, $cr \in [0, 1]$, and the number of input variables.

```
def crossover(mutated, target, dims, cr):
    # generate a uniform random value for every dimension
    p = rand(dims)
    # generate trial vector by binomial crossover
    trial = [mutated[i] if p[i] < cr else target[i] for i in range(dims)]
    return trial
...
# perform crossover
trial = crossover(mutated, pop[j], len(bounds), cr)
...
```

Program 19.8: Define crossover operation

A final selection step replaces the `target` vector by the `trial` vector if the latter yields a lower objective function value. For this purpose, we evaluate both vectors on the objective function and subsequently perform selection, storing the new objective function value in `obj_all` if the `trial` vector is found to be the fittest of the two.

```
...
# compute objective function value for target vector
obj_target = obj(pop[j])
# compute objective function value for trial vector
obj_trial = obj(trial)
# perform selection
if obj_trial < obj_target:
    # replace the target vector with the trial vector
    pop[j] = trial
    # store the new objective function value
    obj_all[j] = obj_trial
```

Program 19.9: Selection of the best vector

We can tie all steps together into a `differential_evolution()` function that takes as input arguments the population size, the bounds of each input variable, the total number of iterations,

the mutation scale factor and the crossover rate, and returns the best solution found and its evaluation.

```python
def differential_evolution(pop_size, bounds, iter, F, cr):
    # initialise population of candidate solutions randomly within the specified
    #   bounds
    pop = bounds[:, 0] + (rand(pop_size, len(bounds)) * (bounds[:, 1] - bounds[:,
      0]))
    # evaluate initial population of candidate solutions
    obj_all = [obj(ind) for ind in pop]
    # find the best performing vector of initial population
    best_vector = pop[argmin(obj_all)]
    best_obj = min(obj_all)
    prev_obj = best_obj
    # run iterations of the algorithm
    for i in range(iter):
        # iterate over all candidate solutions
        for j in range(pop_size):
            # choose three candidates, a, b and c, that are not the current one
            candidates = [candidate for candidate in range(pop_size) if candidate
              != j]
            a, b, c = pop[choice(candidates, 3, replace=False)]
            # perform mutation
            mutated = mutation([a, b, c], F)
            # check that lower and upper bounds are retained after mutation
            mutated = check_bounds(mutated, bounds)
            # perform crossover
            trial = crossover(mutated, pop[j], len(bounds), cr)
            # compute objective function value for target vector
            obj_target = obj(pop[j])
            # compute objective function value for trial vector
            obj_trial = obj(trial)
            # perform selection
            if obj_trial < obj_target:
                # replace the target vector with the trial vector
                pop[j] = trial
                # store the new objective function value
                obj_all[j] = obj_trial
        # find the best performing vector at each iteration
        best_obj = min(obj_all)
        # store the lowest objective function value
        if best_obj < prev_obj:
            best_vector = pop[argmin(obj_all)]
            prev_obj = best_obj
            # report progress at each iteration
            print('Iteration: %d f([%s]) = %.5f' % (i, around(best_vector,
              decimals=5), best_obj))
    return [best_vector, best_obj]
```

Program 19.10: Differential evolution algorithm

Now that we have implemented the differential evolution algorithm, let's investigate how to use it to optimize an objective function.

19.4 Differential Evolution Algorithm on the Sphere Function

In this section, we will apply the differential evolution algorithm to an objective function. We will use a simple two-dimensional sphere objective function specified within the bounds, $[-5, 5]$. The sphere function is continuous, convex and unimodal, and is characterised by a single global minimum at $f(0, 0) = 0.0$.

```python
def obj(x):
    return x[0]**2.0 + x[1]**2.0
```
Program 19.11: Define objective function

We will minimise this objective function with the differential evolution algorithm, based on the strategy DE/rand/1/bin. In order to do so, we must define values for the algorithm parameters, specifically for the population size, the number of iterations, the mutation scale factor and the crossover rate. We set these values empirically to, 10, 100, 0.5 and 0.7 respectively.

```python
...
# define population size
pop_size = 10
# define number of iterations
iter = 100
# define scale factor for mutation
F = 0.5
# define crossover rate for recombination
cr = 0.7
```
Program 19.12: Define hyperparameters

We also define the bounds of each input variable.

```python
...
bounds = asarray([(-5.0, 5.0), (-5.0, 5.0)])
```
Program 19.13: Define lower and upper bounds for every dimension

Next, we carry out the search and report the results.

```python
...
solution = differential_evolution(pop_size, bounds, iter, F, cr)
```
Program 19.14: Perform differential evolution

Tying this all together, the complete example is listed below.

```python
from numpy.random import rand
from numpy.random import choice
from numpy import asarray
from numpy import clip
from numpy import argmin
from numpy import min
from numpy import around
```

```python
# define objective function
def obj(x):
    return x[0]**2.0 + x[1]**2.0

# define mutation operation
def mutation(x, F):
    return x[0] + F * (x[1] - x[2])

# define boundary check operation
def check_bounds(mutated, bounds):
    mutated_bound = [clip(mutated[i], bounds[i, 0], bounds[i, 1]) for i in
      range(len(bounds))]
    return mutated_bound

# define crossover operation
def crossover(mutated, target, dims, cr):
    # generate a uniform random value for every dimension
    p = rand(dims)
    # generate trial vector by binomial crossover
    trial = [mutated[i] if p[i] < cr else target[i] for i in range(dims)]
    return trial

def differential_evolution(pop_size, bounds, iter, F, cr):
    # initialise population of candidate solutions randomly within the specified
      bounds
    pop = bounds[:, 0] + (rand(pop_size, len(bounds)) * (bounds[:, 1] - bounds[:,
      0]))
    # evaluate initial population of candidate solutions
    obj_all = [obj(ind) for ind in pop]
    # find the best performing vector of initial population
    best_vector = pop[argmin(obj_all)]
    best_obj = min(obj_all)
    prev_obj = best_obj
    # run iterations of the algorithm
    for i in range(iter):
        # iterate over all candidate solutions
        for j in range(pop_size):
            # choose three candidates, a, b and c, that are not the current one
            candidates = [candidate for candidate in range(pop_size) if candidate
              != j]
            a, b, c = pop[choice(candidates, 3, replace=False)]
            # perform mutation
            mutated = mutation([a, b, c], F)
            # check that lower and upper bounds are retained after mutation
            mutated = check_bounds(mutated, bounds)
            # perform crossover
            trial = crossover(mutated, pop[j], len(bounds), cr)
            # compute objective function value for target vector
            obj_target = obj(pop[j])
            # compute objective function value for trial vector
            obj_trial = obj(trial)
            # perform selection
            if obj_trial < obj_target:
```

```
                 # replace the target vector with the trial vector
                 pop[j] = trial
                 # store the new objective function value
                 obj_all[j] = obj_trial
         # find the best performing vector at each iteration
         best_obj = min(obj_all)
         # store the lowest objective function value
         if best_obj < prev_obj:
             best_vector = pop[argmin(obj_all)]
             prev_obj = best_obj
             # report progress at each iteration
             print('Iteration: %d f([%s]) = %.5f' % (i, around(best_vector,
                 decimals=5), best_obj))
     return [best_vector, best_obj]

# define population size
pop_size = 10
# define lower and upper bounds for every dimension
bounds = asarray([(-5.0, 5.0), (-5.0, 5.0)])
# define number of iterations
iter = 100
# define scale factor for mutation
F = 0.5
# define crossover rate for recombination
cr = 0.7

# perform differential evolution
solution = differential_evolution(pop_size, bounds, iter, F, cr)
print('\nSolution: f([%s]) = %.5f' % (around(solution[0], decimals=5), solution[1]))
```

Program 19.15: Differential evolution search of the two-dimensional sphere objective function

Running the example reports the progress of the search including the iteration number, and the response from the objective function each time an improvement is detected. At the end of the search, the best solution is found and its evaluation is reported.

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 converges very close to $f(0.0, 0.0) = 0.0$ in about 33 improvements out of 100 iterations.

```
Iteration: 1 f([[ 0.89709 -0.45082]]) = 1.00800
Iteration: 2 f([[-0.5382 0.29676]]) = 0.37773
Iteration: 3 f([[ 0.41884 -0.21613]]) = 0.22214
Iteration: 4 f([[0.34737 0.29676]]) = 0.20873
Iteration: 5 f([[ 0.20692 -0.1747 ]]) = 0.07334
Iteration: 7 f([[-0.23154 -0.00557]]) = 0.05364
Iteration: 8 f([[ 0.11956 -0.02632]]) = 0.01499
```

```
Iteration: 11 f([[ 0.01535 -0.02632]]) = 0.00093
Iteration: 15 f([[0.01918 0.01603]]) = 0.00062
Iteration: 18 f([[0.01706 0.00775]]) = 0.00035
Iteration: 20 f([[0.00467 0.01275]]) = 0.00018
Iteration: 21 f([[ 0.00288 -0.00175]]) = 0.00001
Iteration: 27 f([[ 0.00286 -0.00175]]) = 0.00001
Iteration: 30 f([[-0.00059 0.00044]]) = 0.00000
Iteration: 37 f([[-1.5e-04 8.0e-05]]) = 0.00000
Iteration: 41 f([[-1.e-04 -8.e-05]]) = 0.00000
Iteration: 43 f([[-4.e-05 6.e-05]]) = 0.00000
Iteration: 48 f([[-2.e-05 6.e-05]]) = 0.00000
Iteration: 49 f([[-6.e-05 0.e+00]]) = 0.00000
Iteration: 50 f([[-4.e-05 1.e-05]]) = 0.00000
Iteration: 51 f([[1.e-05 1.e-05]]) = 0.00000
Iteration: 55 f([[1.e-05 0.e+00]]) = 0.00000
Iteration: 64 f([[-0. -0.]]) = 0.00000
Iteration: 68 f([[ 0. -0.]]) = 0.00000
Iteration: 72 f([[-0. 0.]]) = 0.00000
Iteration: 77 f([[-0. 0.]]) = 0.00000
Iteration: 79 f([[0. 0.]]) = 0.00000
Iteration: 84 f([[ 0. -0.]]) = 0.00000
Iteration: 86 f([[-0. -0.]]) = 0.00000
Iteration: 87 f([[-0. -0.]]) = 0.00000
Iteration: 95 f([[-0. 0.]]) = 0.00000
Iteration: 98 f([[-0. 0.]]) = 0.00000
Solution: f([[-0. 0.]]) = 0.00000
```

Output 19.1: Result from Program 19.15

We can plot the objective function values returned at every improvement by modifying the `differential_evolution()` function slightly to keep track of the objective function values and return this in the list, `obj_iter`.

```python
def differential_evolution(pop_size, bounds, iter, F, cr):
    # initialise population of candidate solutions randomly within the specified ⤸
        bounds
    pop = bounds[:, 0] + (rand(pop_size, len(bounds)) * (bounds[:, 1] - bounds[:, ⤸
        0]))
    # evaluate initial population of candidate solutions
    obj_all = [obj(ind) for ind in pop]
    # find the best performing vector of initial population
    best_vector = pop[argmin(obj_all)]
    best_obj = min(obj_all)
    prev_obj = best_obj
    # initialise list to store the objective function value at each iteration
    obj_iter = list()
    # run iterations of the algorithm
    for i in range(iter):
        # iterate over all candidate solutions
        for j in range(pop_size):
            # choose three candidates, a, b and c, that are not the current one
            candidates = [candidate for candidate in range(pop_size) if candidate ⤸
                != j]
```

```
                a, b, c = pop[choice(candidates, 3, replace=False)]
                # perform mutation
                mutated = mutation([a, b, c], F)
                # check that lower and upper bounds are retained after mutation
                mutated = check_bounds(mutated, bounds)
                # perform crossover
                trial = crossover(mutated, pop[j], len(bounds), cr)
                # compute objective function value for target vector
                obj_target = obj(pop[j])
                # compute objective function value for trial vector
                obj_trial = obj(trial)
                # perform selection
                if obj_trial < obj_target:
                    # replace the target vector with the trial vector
                    pop[j] = trial
                    # store the new objective function value
                    obj_all[j] = obj_trial
            # find the best performing vector at each iteration
            best_obj = min(obj_all)
            # store the lowest objective function value
            if best_obj < prev_obj:
                best_vector = pop[argmin(obj_all)]
                prev_obj = best_obj
                obj_iter.append(best_obj)
                # report progress at each iteration
                print('Iteration: %d f([%s]) = %.5f' % (i, around(best_vector, ↲
                 ↳  decimals=5), best_obj))
        return [best_vector, best_obj, obj_iter]
```

Program 19.16: Differential evolution function with value of each iteration stored

We can then create a line plot of these objective function values to see the relative changes at every improvement during the search.

```
from matplotlib import pyplot
...
# perform differential evolution
solution = differential_evolution(pop_size, bounds, iter, F, cr)
...
# line plot of best objective function values
pyplot.plot(solution[2], '.-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Program 19.17: Perform differential evolution and plot the value of each iteration

Tying this together, the complete example is listed below.

```
from numpy.random import rand
from numpy.random import choice
from numpy import asarray
from numpy import clip
from numpy import argmin
```

```python
from numpy import min
from numpy import around
from matplotlib import pyplot

# define objective function
def obj(x):
    return x[0]**2.0 + x[1]**2.0

# define mutation operation
def mutation(x, F):
    return x[0] + F * (x[1] - x[2])

# define boundary check operation
def check_bounds(mutated, bounds):
    mutated_bound = [clip(mutated[i], bounds[i, 0], bounds[i, 1]) for i in
        range(len(bounds))]
    return mutated_bound

# define crossover operation
def crossover(mutated, target, dims, cr):
    # generate a uniform random value for every dimension
    p = rand(dims)
    # generate trial vector by binomial crossover
    trial = [mutated[i] if p[i] < cr else target[i] for i in range(dims)]
    return trial

def differential_evolution(pop_size, bounds, iter, F, cr):
    # initialise population of candidate solutions randomly within the specified
        bounds
    pop = bounds[:, 0] + (rand(pop_size, len(bounds)) * (bounds[:, 1] - bounds[:,
        0]))
    # evaluate initial population of candidate solutions
    obj_all = [obj(ind) for ind in pop]
    # find the best performing vector of initial population
    best_vector = pop[argmin(obj_all)]
    best_obj = min(obj_all)
    prev_obj = best_obj
    # initialise list to store the objective function value at each iteration
    obj_iter = list()
    # run iterations of the algorithm
    for i in range(iter):
        # iterate over all candidate solutions
        for j in range(pop_size):
            # choose three candidates, a, b and c, that are not the current one
            candidates = [candidate for candidate in range(pop_size) if candidate
                != j]
            a, b, c = pop[choice(candidates, 3, replace=False)]
            # perform mutation
            mutated = mutation([a, b, c], F)
            # check that lower and upper bounds are retained after mutation
            mutated = check_bounds(mutated, bounds)
            # perform crossover
            trial = crossover(mutated, pop[j], len(bounds), cr)
            # compute objective function value for target vector
```

```python
            obj_target = obj(pop[j])
            # compute objective function value for trial vector
            obj_trial = obj(trial)
            # perform selection
            if obj_trial < obj_target:
                # replace the target vector with the trial vector
                pop[j] = trial
                # store the new objective function value
                obj_all[j] = obj_trial
        # find the best performing vector at each iteration
        best_obj = min(obj_all)
        # store the lowest objective function value
        if best_obj < prev_obj:
            best_vector = pop[argmin(obj_all)]
            prev_obj = best_obj
            obj_iter.append(best_obj)
            # report progress at each iteration
            print('Iteration: %d f([%s]) = %.5f' % (i, around(best_vector,
            ↳  decimals=5), best_obj))
    return [best_vector, best_obj, obj_iter]

# define population size
pop_size = 10
# define lower and upper bounds for every dimension
bounds = asarray([(-5.0, 5.0), (-5.0, 5.0)])
# define number of iterations
iter = 100
# define scale factor for mutation
F = 0.5
# define crossover rate for recombination
cr = 0.7

# perform differential evolution
solution = differential_evolution(pop_size, bounds, iter, F, cr)
print('\nSolution: f([%s]) = %.5f' % (around(solution[0], decimals=5), solution[1]))

# line plot of best objective function values
pyplot.plot(solution[2], '.-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Program 19.18: Differential evolution search of the two-dimensional sphere objective function and plot

Running the example creates a line plot. The line plot shows the objective function evaluation for each improvement, with large changes initially and very small changes towards the end of the search as the algorithm converged on the optima.
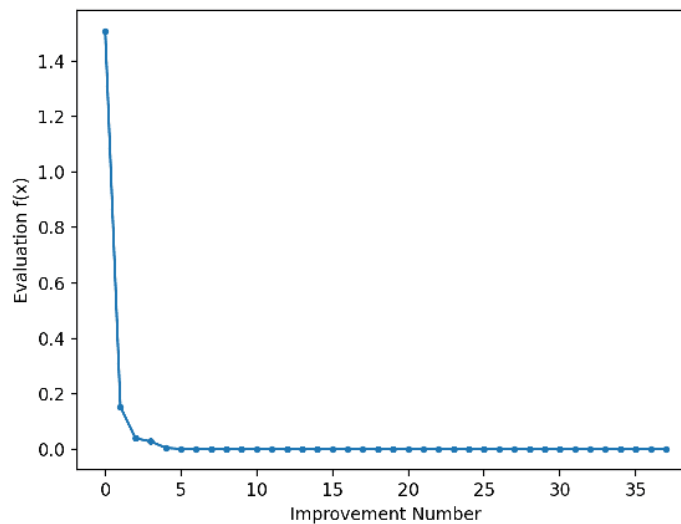
Figure 19.1: Line Plot of Objective Function Evaluation for Each Improvement During the Differential Evolution Search

19.5 Further Reading

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

Papers

Rainer Storn and Kenneth Price. "Differential evolution — A simple and efficient heuristic for global optimization over continuous spaces". *Journal of Global Optimization*, 11(4), 1997, pp. 341–359.
https://link.springer.com/article/10.1023/A:1008202821328

Swagatam Das, Sankha Subhra Mullick, and P. N. Suganthan. "Recent advances in differential evolution — An updated survey". *Swarm and Evolutionary Computation*, 27, Apr. 2016, pp. 1–30.
https://www.sciencedirect.com/science/article/abs/pii/S2210650216000146

Books

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

Articles

Differential evolution. Wikipedia.
https://en.wikipedia.org/wiki/Differential_evolution

19.6   Summary

In this tutorial, you discovered the differential evolution algorithm. Specifically, you learned:

- ▷ Differential evolution is a heuristic approach for the global optimization of nonlinear and non- differentiable continuous space functions.

- ▷ How to implement the differential evolution algorithm from scratch in Python.

- ▷ How to apply the differential evolution algorithm to a real-valued 2D objective function.

   Next, you will learn about simulated annealing, an algorithm that is not in the family of evolution algorithms.

Simulated Annealing from Scratch

<div style="text-align: right; font-size: 3em; color: gray;">20</div>

Simulated Annealing is a stochastic global search optimization algorithm. This means that it makes use of randomness as part of the search process. This makes the algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well. Like the stochastic hill climbing local search algorithm, it modifies a single solution and searches the relatively local area of the search space until the local optima is located. Unlike the hill climbing algorithm, it may accept worse solutions as the current working solution. The likelihood of accepting worse solutions starts high at the beginning of the search and decreases with the progress of the search, giving the algorithm the opportunity to first locate the region for the global optima, escaping local optima, then hill climb to the optima itself.

In this tutorial, you will discover the simulated annealing optimization algorithm for function optimization. After completing this tutorial, you will know:

▷ Simulated annealing is a stochastic global search algorithm for function optimization.

▷ How to implement the simulated annealing algorithm from scratch in Python.

▷ How to use the simulated annealing algorithm and inspect the results of the algorithm.

Let's get started.

20.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Simulated Annealing
2. Implement Simulated Annealing
3. Simulated Annealing Worked Example

20.2   Simulated Annealing

Simulated Annealing[1] is a stochastic global search optimization algorithm. The algorithm is inspired by annealing in metallurgy[2] where metal is heated to a high temperature quickly, then cooled slowly, which increases its strength and makes it easier to work with. The annealing process works by first exciting the atoms in the material at a high temperature, allowing the atoms to move around a lot, then decreasing their excitement slowly, allowing the atoms to fall into a new, more stable configuration.

> When hot, the atoms in the material are more free to move around, and, through random motion, tend to settle into better positions. A slow cooling brings the material to an ordered, crystalline state.
>
> — Page 128, *Algorithms for Optimization*, 2019.

The simulated annealing optimization algorithm can be thought of as a modified version of stochastic hill climbing. Stochastic hill climbing maintains a single candidate solution and takes steps of a random but constrained size from the candidate in the search space. If the new point is better than the current point, then the current point is replaced with the new point. This process continues for a fixed number of iterations. Simulated annealing executes the search in the same way. The main difference is that new points that are not as good as the current point (worse points) are accepted sometimes. A worse point is accepted probabilistically where the likelihood of accepting a solution worse than the current solution is a function of the temperature of the search and how much worse the solution is than the current solution.

> The algorithm varies from Hill-Climbing in its decision of when to replace S, the original candidate solution, with R, its newly tweaked child. Specifically: if R is better than S, we'll always replace S with R as usual. But if R is worse than S, we may still replace S with R with a certain probability
>
> — Page 23, *Essentials of Metaheuristics*, 2011.

The initial temperature for the search is provided as a hyperparameter and decreases with the progress of the search. A number of different schemes (annealing schedules) may be used to decrease the temperature during the search from the initial value to a very low value, although it is common to calculate temperature as a function of the iteration number. A popular example for calculating temperature is the so-called "*fast simulated annealing*," calculated as follows

$$\text{temperature} = \frac{\text{initial temperature}}{\text{iteration number} + 1}$$

We add one to the iteration number in the case that iteration numbers start at zero, to avoid a divide by zero error. The acceptance of worse solutions uses the temperature as well as the difference between the objective function evaluation of the worse solution and the current solution. A value is calculated between 0 and 1 using this information, indicating the likelihood of accepting the worse solution. This distribution is then sampled using a random number, which, if less than the value, means the worse solution is accepted.

---

[1]https://en.wikipedia.org/wiki/Simulated_annealing
[2]https://en.wikipedia.org/wiki/Annealing_(metallurgy)

> " It is this acceptance probability, known as the Metropolis criterion, that allows the algorithm to escape from local minima when the temperature is high. "

— Page 128, *Algorithms for Optimization*, 2019.

This is called the metropolis acceptance criterion and for minimization is calculated as follows:

$$\text{criterion} = \exp\left(-\frac{\text{objective(new)} - \text{objective(current)}}{\text{temperature}}\right)$$

Where exp() is e (the mathematical constant)[3] raised to a power of the provided argument, and objective(new), and objective(current) are the objective function evaluation of the new (worse) and current candidate solutions. The effect is that poor solutions have more chances of being accepted early in the search and less likely of being accepted later in the search. The intent is that the high temperature at the beginning of the search will help the search locate the basin for the global optima and the low temperature later in the search will help the algorithm hone in on the global optima.

> " The temperature starts high, allowing the process to freely move about the search space, with the hope that in this phase the process will find a good region with the best local minimum. The temperature is then slowly brought down, reducing the stochasticity and forcing the search to converge to a minimum "

— Page 128, *Algorithms for Optimization*, 2019.

Now that we are familiar with the simulated annealing algorithm, let's look at how to implement it from scratch.

20.3   Implement Simulated Annealing

In this section, we will explore how we might implement the simulated annealing optimization algorithm from scratch. First, we must define our objective function and the bounds on each input variable to the objective function. The objective function is just a Python function we will name `objective()`. The bounds will be a 2D array with one dimension for each input variable that defines the minimum and maximum for the variable. For example, a one-dimensional objective function and bounds would be defined as follows:

```python
def objective(x):
    return 0

# define range for input
bounds = asarray([[-5.0, 5.0]])
```

Program 20.1: Objective function

Next, we can generate our initial point as a random point within the bounds of the problem, then evaluate it using the objective function.

---

[3]https://en.wikipedia.org/wiki/E_(mathematical_constant)

```
...
# generate an initial point
best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
# evaluate the initial point
best_eval = objective(best)
```

Program 20.2: Evaluate objective function at a random point

We need to maintain the "current" solution that is the focus of the search and that may be replaced with better solutions.

```
...
curr, curr_eval = best, best_eval
```

Program 20.3: Save the current working solution

Now we can loop over a predefined number of iterations of the algorithm defined as "n_iterations", such as 100 or 1,000.

```
...
for i in range(n_iterations):
    ...
```

Program 20.4: Run the algorithm

The first step of the algorithm iteration is to generate a new candidate solution from the current working solution, e.g. take a step. This requires a predefined "step_size" parameter, which is relative to the bounds of the search space. We will take a random step with a Gaussian distribution where the mean is our current point and the standard deviation is defined by the "step_size". That means that about 99 percent of the steps taken will be within 3 * step_size of the current point.

```
...
candidate = solution + randn(len(bounds)) * step_size
```

Program 20.5: Take a random step (normal distribution)

We don't have to take steps in this way. You may wish to use a uniform distribution between 0 and the step size. For example:

```
...
candidate = solution + rand(len(bounds)) * step_size
```

Program 20.6: Take a random step (uniform distribution)

Next, we need to evaluate it.

```
...
candidate_eval = objective(candidate)
```

Program 20.7: Evaluate candidate point

We then need to check if the evaluation of this new point is as good as or better than the current best point, and if it is, replace our current best point with this new point. This is separate from the current working solution that is the focus of the search.

```
...
if candidate_eval < best_eval:
    # store new best point
    best, best_eval = candidate, candidate_eval
    # report progress
    print('>%d f(%s) = %.5f' % (i, best, best_eval))
```
Program 20.8: Check for new best solution

Next, we need to prepare to replace the current working solution. The first step is to calculate the difference between the objective function evaluation of the current solution and the current working solution.

```
...
diff = candidate_eval - curr_eval
```
Program 20.9: Difference between candidate and current point evaluation

Next, we need to calculate the current temperature, using the fast annealing schedule, where "temp" is the initial temperature provided as an argument.

```
...
t = temp / float(i + 1)
```
Program 20.10: Calculate temperature for current epoch

We can then calculate the likelihood of accepting a solution with worse performance than our current working solution.

```
...
metropolis = exp(-diff / t)
```
Program 20.11: Calculate metropolis acceptance criterion

Finally, we can accept the new point as the current working solution if it has a better objective function evaluation (the difference is negative) or if the objective function is worse, but we probabilistically decide to accept it.

```
...
if diff < 0 or rand() < metropolis:
    # store the new current point
    curr, curr_eval = candidate, candidate_eval
```
Program 20.12: Check if we should keep the new point

And that's it.

We can implement this simulated annealing algorithm as a reusable function that takes the name of the objective function, the bounds of each input variable, the total iterations, step size, and initial temperature as arguments, and returns the best solution found and its evaluation.

```python
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):
    # generate an initial point
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval]
```

Program 20.13: Simulated annealing algorithm

Now that we know how to implement the simulated annealing algorithm in Python, let's look at how we might use it to optimize an objective function.

20.4   Simulated Annealing Worked Example

In this section, we will apply the simulated annealing optimization algorithm to an objective function. First, let's define our objective function. We will use a simple one-dimensional x^2 objective function with the bounds $[-5, 5]$. The example below defines the function, then creates a line plot of the response surface of the function for a grid of input values, and marks the optima at $f(0.0) = 0.0$ with a red line

```python
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(x):
    return x[0]**2.0
```

```
# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = [objective([x]) for x in inputs]
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define optimal input value
x_optima = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=x_optima, ls='--', color='red')
# show the plot
pyplot.show()
```

Program 20.14: Convex unimodal optimization function

Running the example creates a line plot of the objective function and clearly marks the function optima.
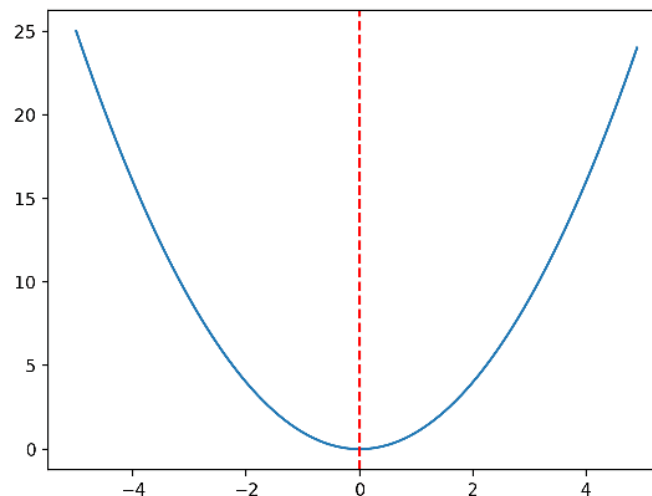
Figure 20.1: Line Plot of Objective Function With Optima Marked With a Dashed Red Line

Before we apply the optimization algorithm to the problem, let's take a moment to understand the acceptance criterion a little better. First, the fast annealing schedule is an exponential function of the number of iterations. We can make this clear by creating a plot of the temperature for each algorithm iteration. We will use an initial temperature of 10 and 100 algorithm iterations, both arbitrarily chosen. The complete example is listed below.

```
from matplotlib import pyplot
# total iterations of algorithm
iterations = 100
# initial temperature
```

```
initial_temp = 10
# array of iterations from 0 to iterations - 1
iterations = [i for i in range(iterations)]
# temperatures for each iterations
temperatures = [initial_temp/float(i + 1) for i in iterations]
# plot iterations vs temperatures
pyplot.plot(iterations, temperatures)
pyplot.xlabel('Iteration')
pyplot.ylabel('Temperature')
pyplot.show()
```

Program 20.15: Explore temperature vs algorithm iteration for simulated annealing

Running the example calculates the temperature for each algorithm iteration and creates a plot of algorithm iteration (x-axis) vs. temperature (y-axis). We can see that temperature drops rapidly, exponentially, not linearly, such that after 20 iterations it is below 1 and stays low for the remainder of the search.
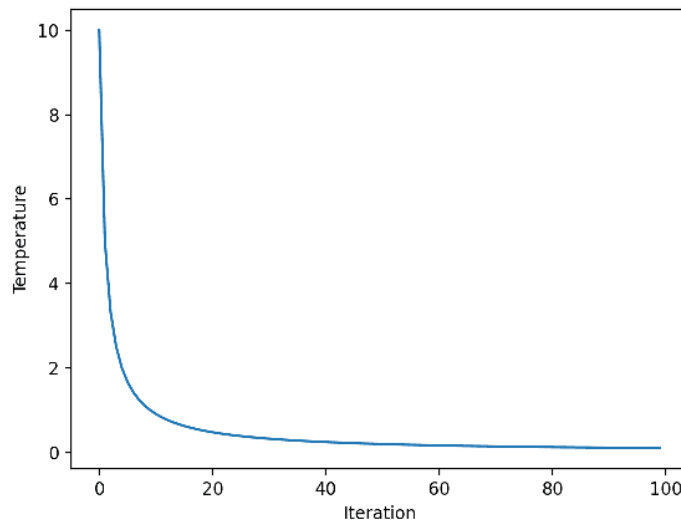
Figure 20.2: Line Plot of Temperature vs. Algorithm Iteration for Fast Annealing

Next, we can get a better idea of how the metropolis acceptance criterion changes over time with the temperature. Recall that the criterion is a function of temperature, but is also a function of how different the objective evaluation of the new point is compared to the current working solution. As such, we will plot the criterion for a few different "*differences in objective function value*" to see the effect it has on acceptance probability. The complete example is listed below.

```
from math import exp
from matplotlib import pyplot
# total iterations of algorithm
iterations = 100
# initial temperature
initial_temp = 10
```

```
# array of iterations from 0 to iterations - 1
iterations = [i for i in range(iterations)]
# temperatures for each iterations
temperatures = [initial_temp/float(i + 1) for i in iterations]
# metropolis acceptance criterion
differences = [0.01, 0.1, 1.0]
for d in differences:
    metropolis = [exp(-d/t) for t in temperatures]
    # plot iterations vs metropolis
    label = 'diff=%.2f' % d
    pyplot.plot(iterations, metropolis, label=label)
# inalize plot
pyplot.xlabel('Iteration')
pyplot.ylabel('Metropolis Criterion')
pyplot.legend()
pyplot.show()
```

Program 20.16: Explore metropolis acceptance criterion for simulated annealing

Running the example calculates the metropolis acceptance criterion for each algorithm iteration using the temperature shown for each iteration (shown in the previous section). The plot has three lines for three differences between the new worse solution and the current working solution. We can see that the worse the solution is (the larger the difference), the less likely the model is to accept the worse solution regardless of the algorithm iteration, as we might expect. We can also see that in all cases, the likelihood of accepting worse solutions decreases with algorithm iteration.
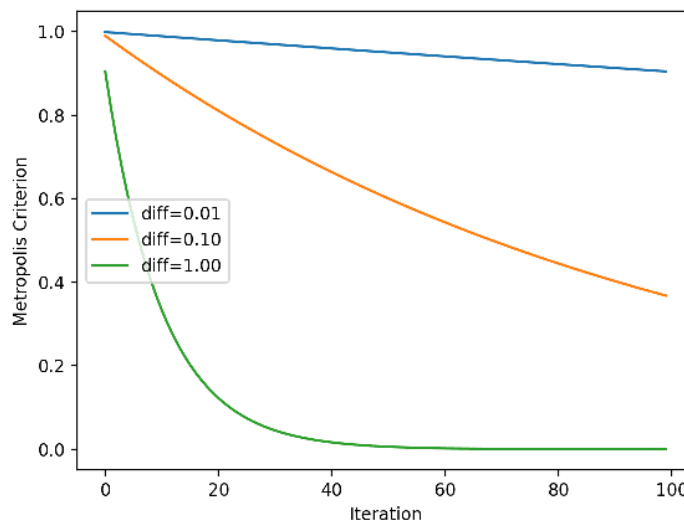
Figure 20.3: Line Plot of Metropolis Acceptance Criterion vs. Algorithm Iteration for Simulated Annealing

Now that we are more familiar with the behavior of the temperature and metropolis acceptance criterion over time, let's apply simulated annealing to our test problem. First, we will seed the pseudorandom number generator. This is not required in general, but in this case,

I want to ensure we get the same results (same sequence of random numbers) each time we run the algorithm so we can plot the results later.

```
...
seed(1)
```

Program 20.17: Seed the pseudorandom number generator

Next, we can define the configuration of the search. In this case, we will search for 1,000 iterations of the algorithm and use a step size of 0.1. Given that we are using a Gaussian function for generating the step, this means that about 99 percent of all steps taken will be within a distance of (0.1×3) of a given point, i.e. three standard deviations. We will also use an initial temperature of 10.0. The search procedure is more sensitive to the annealing schedule than the initial temperature, as such, initial temperature values are almost arbitrary.

```
...
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# initial temperature
temp = 10
```

Program 20.18: Set up hyperparameters

Next, we can perform the search and report the results.

```
...
best, score = simulated_annealing(objective, bounds, n_iterations, step_size, temp)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 20.19: Perform the simulated annealing search

Tying this all together, the complete example is listed below.

```python
from numpy import asarray
from numpy import exp
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(x):
    return x[0]**2.0

# simulated annealing algorithm
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):
    # generate an initial point
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
```

```python
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# initial temperature
temp = 10
# perform the simulated annealing search
best, score = simulated_annealing(objective, bounds, n_iterations, step_size, temp)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 20.20: Simulated annealing search of a one-dimensional objective function

Running the example reports the progress of the search including the iteration number, the input to the function, and the response from the objective function each time an improvement was detected. At the end of the search, the best solution is found and its evaluation is reported.

> **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 about 20 improvements over the 1,000 iterations of the algorithm and a solution that is very close to the optimal input of 0.0 that evaluates to $f(0.0) = 0.0$.

```
>34 f([-0.78753544]) = 0.62021
>35 f([-0.76914239]) = 0.59158
>37 f([-0.68574854]) = 0.47025
>39 f([-0.64797564]) = 0.41987
>40 f([-0.58914623]) = 0.34709
>41 f([-0.55446029]) = 0.30743
>42 f([-0.41775702]) = 0.17452
>43 f([-0.35038542]) = 0.12277
>50 f([-0.15799045]) = 0.02496
>66 f([-0.11089772]) = 0.01230
>67 f([-0.09238208]) = 0.00853
>72 f([-0.09145261]) = 0.00836
>75 f([-0.05129162]) = 0.00263
>93 f([-0.02854417]) = 0.00081
>144 f([0.00864136]) = 0.00007
>149 f([0.00753953]) = 0.00006
>167 f([-0.00640394]) = 0.00004
>225 f([-0.00044965]) = 0.00000
>503 f([-0.00036261]) = 0.00000
>512 f([0.00013605]) = 0.00000
Done!
f([0.00013605]) = 0.000000
```

Output 20.1: Result from Program 20.20

It can be interesting to review the progress of the search as a line plot that shows the change in the evaluation of the best solution each time there is an improvement. We can update the `simulated_annealing()` to keep track of the objective function evaluations each time there is an improvement and return this list of scores.

```python
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):
    # generate an initial point
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # keep track of scores
            scores.append(best_eval)
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
```

```
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval, scores]
```

Program 20.21: Simulated annealing algorithm with the scores

We can then create a line plot of these scores to see the relative change in objective function for each improvement found during the search.

```
...
pyplot.plot(scores, '.-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Program 20.22: Line plot of best scores

Tying this together, the complete example of performing the search and plotting the objective function scores of the improved solutions during the search is listed below.

```
from numpy import asarray
from numpy import exp
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed
from matplotlib import pyplot

# objective function
def objective(x):
    return x[0]**2.0

# simulated annealing algorithm
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):
    # generate an initial point
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    scores = list()
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
```

```python
            # store new best point
            best, best_eval = candidate, candidate_eval
            # keep track of scores
            scores.append(best_eval)
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval, scores]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# initial temperature
temp = 10
# perform the simulated annealing search
best, score, scores = simulated_annealing(objective, bounds, n_iterations,
    step_size, temp)
print('Done!')
print('f(%s) = %f' % (best, score))
# line plot of best scores
pyplot.plot(scores, '.-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Program 20.23: Simulated annealing search of a one-dimensional objective function

Running the example performs the search and reports the results as before. A line plot is created showing the objective function evaluation for each improvement during the hill climbing search. We can see about 20 changes to the objective function evaluation during the search with large changes initially and very small to imperceptible changes towards the end of the search as the algorithm converged on the optima.
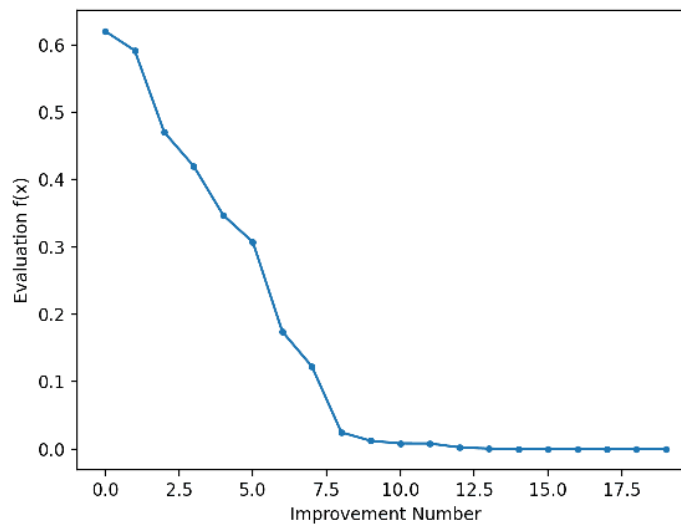
Figure 20.4: Line Plot of Objective Function Evaluation for Each Improvement During the Simulated Annealing Search

20.5 Further Reading

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

Papers

S. Kirkpatrickc, D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". *Science*, 220(4598), 1983, pp. 671–680.
https://science.sciencemag.org/content/220/4598/671.abstract

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J
Sean Luke. *Essentials of Metaheuristics*. lulu.com, 2011.
https://amzn.to/3lHryZr

Articles

Simulated annealing. Wikipedia.
https://en.wikipedia.org/wiki/Simulated_annealing
Annealing (metallurgy). Wikipedia.
https://en.wikipedia.org/wiki/Annealing_(metallurgy)

20.6   Summary

In this tutorial, you discovered the simulated annealing optimization algorithm for function optimization. Specifically, you learned:

▷ Simulated annealing is a stochastic global search algorithm for function optimization.

▷ How to implement the simulated annealing algorithm from scratch in Python.

▷ How to use the simulated annealing algorithm and inspect the results of the algorithm.

Next, we start to learn about various gradient descent algorithms.

Gradient Descent

V

Gradient Descent Optimization from Scratch

<div style="text-align: right; font-size: 3em; color: gray;">21</div>

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.
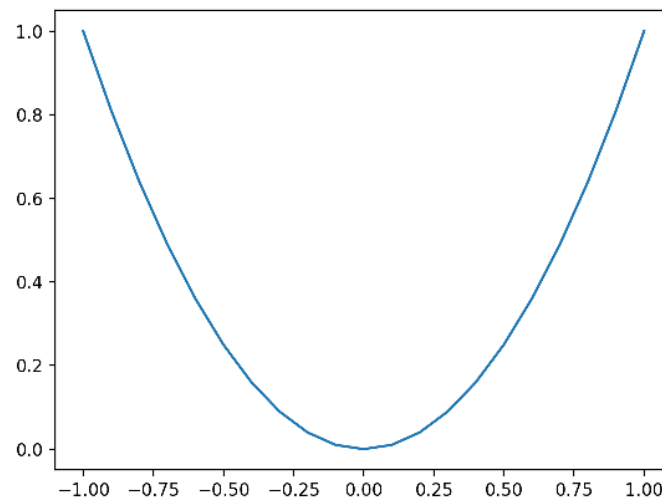
Figure 21.1: Line Plot of Simple One-Dimensional Function

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
```

```python
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
```

```python
# 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.
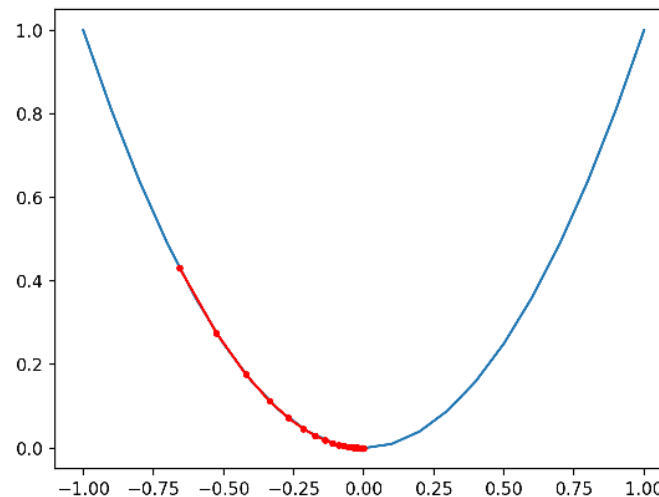
Figure 21.2: Plot of the Progress of Gradient Descent on a One Dimensional Objective Function

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.

Gradient Descent with Momentum

22

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function. A problem with gradient descent is that it can bounce around the search space on optimization problems that have large amounts of curvature or noisy gradients, and it can get stuck in flat spots in the search space that have no gradient. *Momentum* is an extension to the gradient descent optimization algorithm that allows the search to build inertia in a direction in the search space and overcome the oscillations of noisy gradients and coast across flat spots of the search space.

In this tutorial, you will discover the gradient descent with momentum algorithm. After completing this tutorial, you will know:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be accelerated by using momentum from past updates to the search position.

▷ How to implement gradient descent optimization with momentum and develop an intuition for its behavior.

Let's get started.

22.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Gradient Descent
2. Momentum
3. Gradient Descent With Momentum
   (a) One-Dimensional Test Problem
   (b) Gradient Descent Optimization
   (c) Visualization of Gradient Descent Optimization
   (d) Gradient Descent Optimization With Momentum
   (e) Visualization of Gradient Descent Optimization With Momentum

22.2   Gradient Descent

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.

> " 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[2], or simply the "*derivative*," 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 a specific input. 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. The gradient descent algorithm requires a target function that is being optimized and the derivative function for the objective 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. 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, also called the learning rate.

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.

Now that we are familiar with the gradient descent optimization algorithm, let's take a look at momentum.

---

[1]https://en.wikipedia.org/wiki/Gradient_descent
[2]https://en.wikipedia.org/wiki/Derivative
[3]https://en.wikipedia.org/wiki/Gradient

22.3 Momentum

Momentum is an extension to the gradient descent optimization algorithm, often referred to as *gradient descent with momentum*. It is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result. A problem with the gradient descent algorithm is that the progression of the search can bounce around the search space based on the gradient. For example, the search may progress downhill towards the minima, but during this progression, it may move in another direction, even uphill, depending on the gradient of specific points (sets of parameters) encountered during the search.

This can slow down the progress of the search, especially for those optimization problems where the broader trend or shape of the search space is more useful than specific gradients along the way. One approach to this problem is to add history to the parameter update equation based on the gradient encountered in the previous updates. This change is based on the metaphor of momentum from physics where acceleration in a direction can be accumulated from past updates.

> The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion.
>
> — Page 296, *Deep Learning*, 2016.

Momentum involves adding an additional hyperparameter that controls the amount of history (momentum) to include in the update equation, i.e. the step to a new point in the search space. The value for the hyperparameter is defined in the range 0.0 to 1.0 and often has a value close to 1.0, such as 0.8, 0.9, or 0.99. A momentum of 0.0 is the same as gradient descent without momentum. First, let's break the gradient descent update equation down into two parts: the calculation of the change to the position and the update of the old position to the new position. The change in the parameters is calculated as the gradient for the point scaled by the step size α.

$$\Delta x = \alpha \times f'(x)$$

The new position is calculated by simply subtracting the change from the current point

$$x_{\text{new}} = x - \Delta x$$

Momentum involves maintaining the change in the position and using it in the subsequent calculation of the change in position. If we think of updates over time, then the update at the current iteration or time (t) will add the change used at the previous time $(t-1)$ weighted by the momentum hyperparameter η, as follows:

$$\Delta x(t) = \alpha \times f'(x(t-1)) + \eta \times \Delta x(t-1)$$

The update to the position is then performed as before.

$$x(t) = x(t-1) - \Delta x(t)$$

The change in the position accumulates magnitude and direction of changes over the iterations of the search, proportional to the size of the momentum hyperparameter. For example, a large

momentum (e.g. 0.9) will mean that the update is strongly influenced by the previous update, whereas a modest momentum (0.2) will mean very little influence.

> The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
>
> — Page 296, *Deep Learning*, 2016.

Momentum has the effect of dampening down the change in the gradient and, in turn, the step size with each new point in the search space.

> Momentum can increase speed when the cost surface is highly nonspherical because it damps the size of the steps along directions of high curvature thus yielding a larger effective learning rate along the directions of low curvature.
>
> — Page 21, *Neural Networks: Tricks of the Trade*, 2012.

Momentum is most useful in optimization problems where the objective function has a large amount of curvature (i.e. changes a lot), meaning that the gradient may change a lot over relatively small regions of the search space.

> The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.
>
> — Page 296, *Deep Learning*, 2016.

It is also helpful when the gradient is estimated, such as from a simulation, and may be noisy, e.g. when the gradient has a high variance. Finally, momentum is helpful when the search space is flat or nearly flat, e.g. zero gradient. The momentum allows the search to progress in the same direction as before the flat spot and helpfully cross the flat region. Now that we are familiar with what momentum is, let's look at a worked example.

22.4 Gradient Descent with Momentum

In this section, we will first implement the gradient descent optimization algorithm, then update it to use momentum and compare results.

22.4.1 One-Dimensional Test Problem

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.

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

Program 22.1: 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 22.2: 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 22.3: Plot the objective function input and result

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

```
# plot of simple 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 22.4: 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-shape called a parabola.
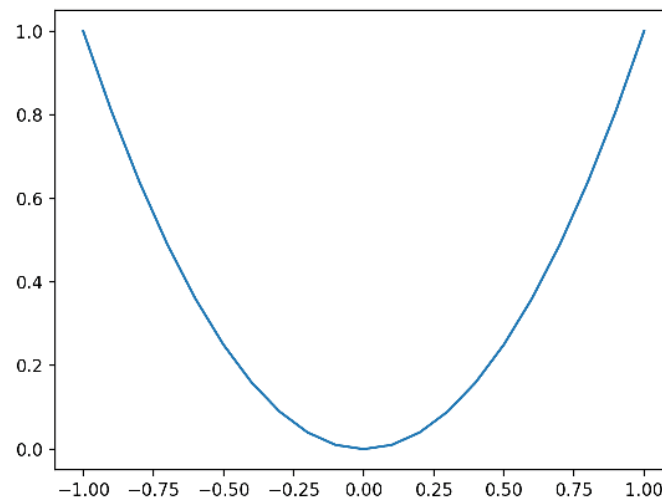
Figure 22.1: Line Plot of Simple One Dimensional Function

22.5 Gradient Descent Optimization

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

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

Program 22.5: Derivative of objective function

We can define a function that implements the gradient descent optimization algorithm. The procedure involves starting with a randomly selected point in the search space, then calculating the gradient, updating the position in the search space, evaluating the new position, and reporting the progress. This process is then repeated for a fixed number of iterations. The final point and its evaluation are then returned from the function.

The function `gradient_descent()` below implements this and 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.

```python
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 22.6: Gradient descent algorithm

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. The seed for the pseudorandom number generator is fixed so that we always get the same sequence of random numbers, and in this case, it ensures that we get the same starting point for the search each time the code is run (e.g. something interesting far from the optima).

```
...
# seed the pseudo random number generator
seed(4)
# 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 22.7: Perform gradient descent search

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

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

# 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]

 # seed the pseudo random number generator
 seed(4)
 # 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 22.8: 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 27 iterations, with a function evaluation of about 0.0. Note the optima for this function is at $f(0.0) = 0.0$. We would expect that gradient descent with momentum will accelerate the optimization procedure and find a similarly evaluated solution in fewer iterations.

```
>0 f([0.74724774]) = 0.55838
>1 f([0.59779819]) = 0.35736
>2 f([0.47823856]) = 0.22871
>3 f([0.38259084]) = 0.14638
>4 f([0.30607268]) = 0.09368
>5 f([0.24485814]) = 0.05996
>6 f([0.19588651]) = 0.03837
>7 f([0.15670921]) = 0.02456
>8 f([0.12536737]) = 0.01572
>9 f([0.10029389]) = 0.01006
>10 f([0.08023512]) = 0.00644
>11 f([0.06418809]) = 0.00412
>12 f([0.05135047]) = 0.00264
>13 f([0.04108038]) = 0.00169
>14 f([0.0328643]) = 0.00108
```

```
>15 f([0.02629144]) = 0.00069
>16 f([0.02103315]) = 0.00044
>17 f([0.01682652]) = 0.00028
>18 f([0.01346122]) = 0.00018
>19 f([0.01076897]) = 0.00012
>20 f([0.00861518]) = 0.00007
>21 f([0.00689214]) = 0.00005
>22 f([0.00551372]) = 0.00003
>23 f([0.00441097]) = 0.00002
>24 f([0.00352878]) = 0.00001
>25 f([0.00282302]) = 0.00001
>26 f([0.00225842]) = 0.00001
>27 f([0.00180673]) = 0.00000
>28 f([0.00144539]) = 0.00000
>29 f([0.00115631]) = 0.00000
Done!
f([0.00115631]) = 0.000001
```

Output 22.1: Result from Program 22.8

22.6 Visualization of Gradient Descent Optimization

Next, we can 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.

```python
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 22.9: Gradient descent algorithm with the scores stored

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

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

Program 22.10: 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 22.11: 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 22.12: 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 numpy.random import seed
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]

# seed the pseudo random number generator
seed(4)
# 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 22.13: 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 more than halfway up the right 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.
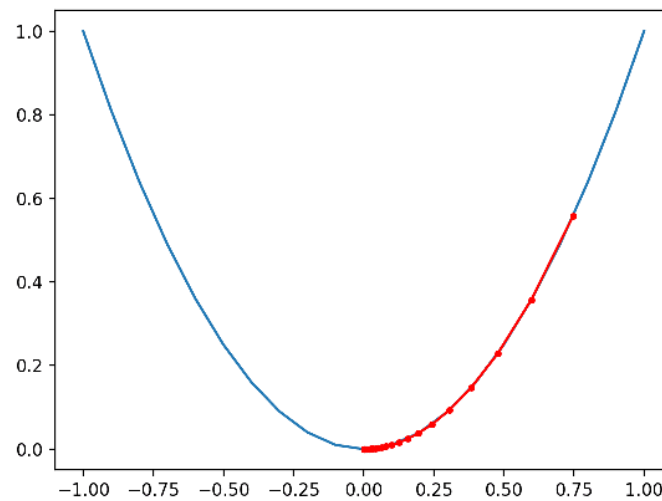
Figure 22.2: Plot of the Progress of Gradient Descent on a One Dimensional Objective Function

22.7 Gradient Descent Optimization With Momentum

Next, we can update the gradient descent optimization algorithm to use momentum. This can be achieved by updating the `gradient_descent()` function to take a "`momentum`" argument that defines the amount of momentum used during the search. The change made to the solution must be remembered from the previous iteration of the loop, with an initial value of 0.0.

```
...
change = 0.0
```

Program 22.14: Keep track of the change

We can then break the update procedure down into first calculating the gradient, then calculating the change to the solution, calculating the position of the new solution, then saving the change for the next iteration.

```
...
# calculate gradient
gradient = derivative(solution)
# calculate update
new_change = step_size * gradient + momentum * change
# take a step
solution = solution - new_change
# save the change
change = new_change
```

Program 22.15: Update procedure with the change saved for the next iteration

The updated version of the `gradient_descent()` function with these changes is listed below.

```python
def gradient_descent(objective, derivative, bounds, n_iter, step_size, momentum):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # keep track of the change
    change = 0.0
    # run the gradient descent
    for i in range(n_iter):
        # calculate gradient
        gradient = derivative(solution)
        # calculate update
        new_change = step_size * gradient + momentum * change
        # take a step
        solution = solution - new_change
        # save the change
        change = new_change
        # evaluate candidate point
        solution_eval = objective(solution)
        # report progress
        print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]
```

Program 22.16: Gradient descent algorithm with the change saved in each iteration

We can then choose a momentum value and pass it to the gradient_descent() function. After a little trial and error, a momentum value of 0.3 was found to be effective on this problem, given the fixed step size of 0.1.

```python
...
# define momentum
momentum = 0.3
# perform the gradient descent search with momentum
best, score = gradient_descent(objective, derivative, bounds, n_iter, step_size, ⤸
    momentum)
```

Program 22.17: Gradient descent search with momentum

Tying this together, the complete example of gradient descent optimization with momentum is listed below.

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

# 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, momentum):
    # generate an initial point
```

```
        solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
        # keep track of the change
        change = 0.0
        # run the gradient descent
        for i in range(n_iter):
            # calculate gradient
            gradient = derivative(solution)
            # calculate update
            new_change = step_size * gradient + momentum * change
            # take a step
            solution = solution - new_change
            # save the change
            change = new_change
            # evaluate candidate point
            solution_eval = objective(solution)
            # report progress
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
        return [solution, solution_eval]

# seed the pseudo random number generator
seed(4)
# 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
# define momentum
momentum = 0.3
# perform the gradient descent search with momentum
best, score = gradient_descent(objective, derivative, bounds, n_iter, step_size, ⤸
    momentum)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 22.18: Example of gradient descent with momentum for a one-dimensional function

Running the example starts with a random point in the search space, then applies the gradient descent algorithm with momentum, 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 13 iterations, with a function evaluation of about 0.0. As expected, this is faster (fewer iterations) than gradient descent without momentum, using the same starting point and step size that took 27 iterations.

```
>0 f([0.74724774]) = 0.55838
>1 f([0.54175461]) = 0.29350
>2 f([0.37175575]) = 0.13820
>3 f([0.24640494]) = 0.06072
>4 f([0.15951871]) = 0.02545
>5 f([0.1015491]) = 0.01031
>6 f([0.0638484]) = 0.00408
>7 f([0.03976851]) = 0.00158
>8 f([0.02459084]) = 0.00060
>9 f([0.01511937]) = 0.00023
>10 f([0.00925406]) = 0.00009
>11 f([0.00564365]) = 0.00003
>12 f([0.0034318]) = 0.00001
>13 f([0.00208188]) = 0.00000
>14 f([0.00126053]) = 0.00000
>15 f([0.00076202]) = 0.00000
>16 f([0.00046006]) = 0.00000
>17 f([0.00027746]) = 0.00000
>18 f([0.00016719]) = 0.00000
>19 f([0.00010067]) = 0.00000
>20 f([6.05804744e-05]) = 0.00000
>21 f([3.64373635e-05]) = 0.00000
>22 f([2.19069576e-05]) = 0.00000
>23 f([1.31664443e-05]) = 0.00000
>24 f([7.91100141e-06]) = 0.00000
>25 f([4.75216828e-06]) = 0.00000
>26 f([2.85408468e-06]) = 0.00000
>27 f([1.71384267e-06]) = 0.00000
>28 f([1.02900153e-06]) = 0.00000
>29 f([6.17748881e-07]) = 0.00000
Done!
f([6.17748881e-07]) = 0.000000
```

Output 22.2: Result from Program 22.18

22.8 Visualization of Gradient Descent Optimization With Momentum

Finally, we can visualize the progress of the gradient descent optimization algorithm with momentum. The complete example is listed below.

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

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

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

# gradient descent algorithm
def gradient_descent(objective, derivative, bounds, n_iter, step_size, momentum):
    # track all solutions
    solutions, scores = list(), list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # keep track of the change
    change = 0.0
    # run the gradient descent
    for i in range(n_iter):
        # calculate gradient
        gradient = derivative(solution)
        # calculate update
        new_change = step_size * gradient + momentum * change
        # take a step
        solution = solution - new_change
        # save the change
        change = new_change
        # 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]

# seed the pseudo random number generator
seed(4)
# 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
# define momentum
momentum = 0.3
# perform the gradient descent search with momentum
solutions, scores = gradient_descent(objective, derivative, bounds, n_iter,
    step_size, momentum)
# 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 22.19: Example of plotting gradient descent with momentum for a one-dimensional function

Running the example performs the gradient descent search with momentum 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, if we compare the plot to the plot created previously for the performance of gradient descent (without momentum), we can see that the search indeed reaches the optima in fewer steps, noted with fewer distinct red dots on the path to the bottom of the basin.
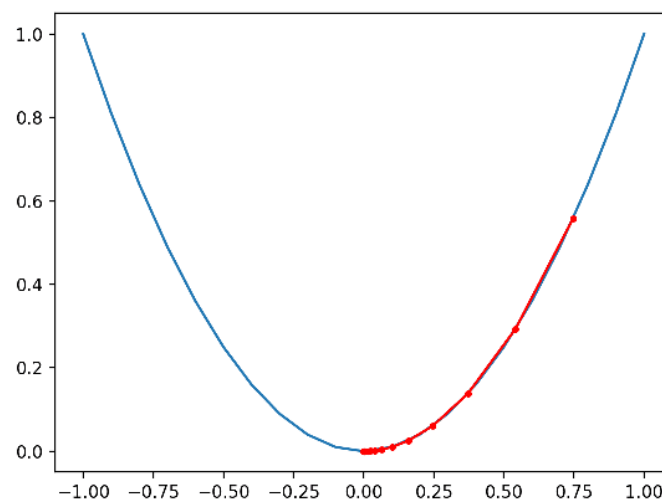
Figure 22.3: Plot of the Progress of Gradient Descent With Momentum on a One Dimensional Objective Function

As an extension, try different values for momentum, such as 0.8, and review the resulting plot.

22.9 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
Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.
https://amzn.to/3qSk3C2

Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller. *Neural Networks: Tricks of the Trade.* 2nd ed. Springer, 2012.
https://amzn.to/3ac5S4Q

Russell Reed. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks.* Bradford Books, 1999.
https://amzn.to/380Yjvd

Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995.
https://amzn.to/3nFrjyF

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

Stochastic gradient descent. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_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

22.10   Summary

In this tutorial, you discovered the gradient descent with momentum algorithm. Specifically, you learned:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be accelerated by using momentum from past updates to the search position.

▷ How to implement gradient descent optimization with momentum and develop an intuition for its behavior.

Next, we will learn about a variation of gradient descent algorithm that can automatically adjust the step size.

Gradient Descent with AdaGrad 23

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function. A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable. This can be a problem on objective functions that have different amounts of curvature in different dimensions, and in turn, may require a different sized step to a new point. *Adaptive Gradients*, or *AdaGrad* for short, is an extension of the gradient descent optimization algorithm that allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients seen for the variable (partial derivatives) seen over the course of the search.

In this tutorial, you will discover how to develop the gradient descent with adaptive gradients optimization algorithm from scratch. After completing this tutorial, you will know:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable in the objective function, called adaptive gradients or AdaGrad.

▷ How to implement the AdaGrad optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Let's get started.

23.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Gradient Descent
2. Adaptive Gradient (AdaGrad)
3. Gradient Descent With AdaGrad
   (a) Two-Dimensional Test Problem
   (b) Gradient Descent Optimization With AdaGrad
   (c) Visualization of AdaGrad

23.2   Gradient Descent

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.

> " 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[2], or simply the "*derivative,*" 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 a specific input. 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.

The gradient descent algorithm requires a target function that is being optimized and the derivative function for the objective 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. 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.

Now that we are familiar with the gradient descent optimization algorithm, let's take a look at AdaGrad.

---

[1]https://en.wikipedia.org/wiki/Gradient_descent
[2]https://en.wikipedia.org/wiki/Derivative
[3]https://en.wikipedia.org/wiki/Gradient

23.3 Adaptive Gradient (AdaGrad)

The Adaptive Gradient algorithm, or AdaGrad for short, is an extension to the gradient descent optimization algorithm. The algorithm was described by John Duchi, et al. in their 2011 paper titled "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization[4]." It is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result.

> The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.
>
> — Page 307, *Deep Learning*, 2016.

A problem with the gradient descent algorithm is that the step size (learning rate) is the same for each variable or dimension in the search space. It is possible that better performance can be achieved using a step size that is tailored to each variable, allowing larger movements in dimensions with a consistently steep gradient and smaller movements in dimensions with less steep gradients. AdaGrad is designed to specifically explore the idea of automatically tailoring the step size for each dimension in the search space.

> The adaptive subgradient method, or Adagrad, adapts a learning rate for each component of x
>
> — Page 77, *Algorithms for Optimization*, 2019.

This is achieved by first calculating a step size for a given dimension, then using the calculated step size to make a movement in that dimension using the partial derivative. This process is then repeated for each dimension in the search space.

> Adagrad dulls the influence of parameters with consistently high gradients, thereby increasing the influence of parameters with infrequent updates.
>
> — Page 77, *Algorithms for Optimization*, 2019.

AdaGrad is suited to objective functions where the curvature of the search space is different in different dimensions, allowing a more effective optimization given the customization of the step size in each dimension. The algorithm requires that you set an initial step size for all input variables as per normal, such as 0.1 or 0.001, or similar. Although, the benefit of the algorithm is that it is not as sensitive to the initial learning rate as the gradient descent algorithm.

> Adagrad is far less sensitive to the learning rate parameter alpha. The learning rate parameter is typically set to a default value of 0.01.
>
> — Page 77, *Algorithms for Optimization*, 2019.

An internal variable is then maintained for each input variable that is the sum of the squared partial derivatives for the input variable observed during the search. This sum of the squared partial derivatives is then used to calculate the step size α for the variable by dividing the initial step size value η (i.e. hyperparameter value specified at the start of the run) by the

---

[4]https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

square root of the sum of the squared partial derivatives S.

$$\alpha = \frac{\eta}{\sqrt{S}}$$

It is possible for the square root of the sum of squared partial derivatives S to result in a value of 0.0, resulting in a divide by zero error. Therefore, a tiny value can be added to the denominator to avoid this possibility, such as 10^{-8}.

$$\alpha = \frac{\eta}{10^{-8} + \sqrt{S}}$$

Where α is the calculated step size for an input variable for a given point during the search, η is the initial step size, and S is the sum of the squared partial derivatives for the input variable seen during the search so far. The custom step size α is then used to calculate the value for the variable in the next point or solution in the search.

$$x(t + 1) = x(t) - \alpha \times f'(x(t))$$

This process is then repeated for each input variable until a new point in the search space is created and can be evaluated. Importantly, the partial derivative for the current solution (iteration of the search) is included in the sum of the square root of partial derivatives. We could maintain an array of partial derivatives or squared partial derivatives for each input variable, but this is not necessary. Instead, we simply maintain the sum of the squared partial derivatives and add new values to this sum along the way. Now that we are familiar with the AdaGrad algorithm, let's explore how we might implement it and evaluate its performance.

23.4   Gradient Descent With AdaGrad

In this section, we will explore how to implement the gradient descent optimization algorithm with adaptive gradients.

23.4.1 Two-Dimensional Test Problem

First, let's define an optimization function. We will use a simple two-dimensional function that squares the input of each dimension and define the range of valid inputs from -1.0 to 1.0. The `objective()` function below implements this function.

```python
def objective(x, y):
    return x**2.0 + y**2.0
```

Program 23.1: Objective function

We can create a three-dimensional plot of the dataset to get a feeling for the curvature of the response surface. The complete example of plotting the objective function is listed below.

```python
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot
```

```
# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -1.0, 1.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 23.2: 3D plot of the test function

Running the example creates a three-dimensional surface plot of the objective function. We can see the familiar bowl shape with the global minima at $f(0, 0) = 0$.
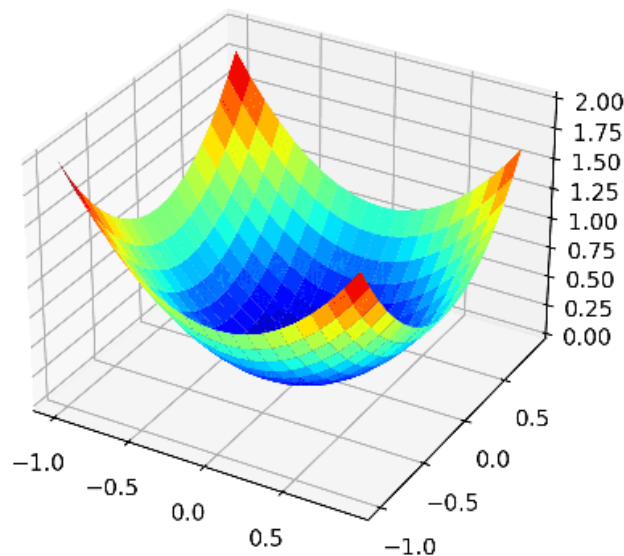
Figure 23.1: Three-Dimensional Plot of the Test Objective Function

We can also create a two-dimensional plot of the function. This will be helpful later when we want to plot the progress of the search. The example below creates a contour plot of the objective function.

```python
from numpy import asarray
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# show the plot
pyplot.show()
```

Program 23.3: Contour plot of the test function

Running the example creates a two-dimensional contour plot of the objective function. We can see the bowl shape compressed to contours shown with a color gradient. We will use this plot to plot the specific points explored during the progress of the search.
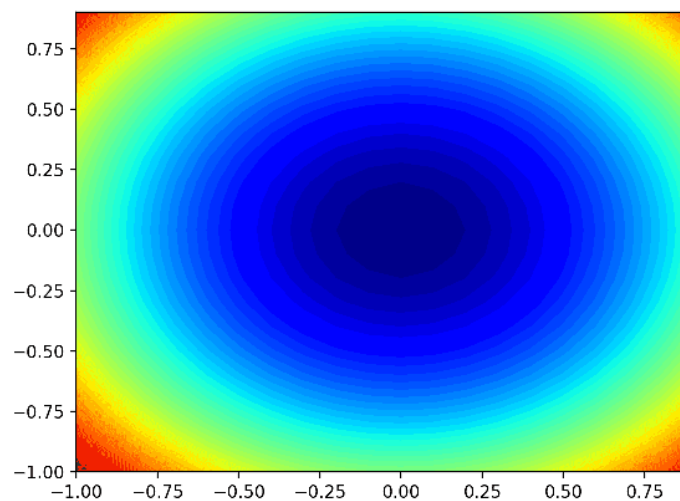
Figure 23.2: Two-Dimensional Contour Plot of the Test Objective Function

Now that we have a test objective function, let's look at how we might implement the AdaGrad optimization algorithm.

23.4.2 Gradient Descent Optimization With AdaGrad

We can apply the gradient descent with adaptive gradient algorithm to the test problem. First, we need a function that calculates the derivative for this function.

$$f(x) = x^2$$
$$f'(x) = 2 \times x$$

The derivative of x^2 is $2x$ in each dimension. The `derivative()` function implements this below.

```
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])
```

Program 23.4: Derivative of objective function

Next, we can implement gradient descent with adaptive gradients. First, we can select a random point in the bounds of the problem as a starting point for the search. This assumes we have an array that defines the bounds of the search with one row for each dimension and the first column defines the minimum and the second column defines the maximum of the dimension.

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

Program 23.5: Generate an initial point

Next, we need to initialize the sum of the squared partial derivatives for each dimension to 0.0 values.

```
...
sq_grad_sums = [0.0 for _ in range(bounds.shape[0])]
```

Program 23.6: List of the sum square gradients for each variable

We can then enumerate a fixed number of iterations of the search optimization algorithm defined by a "`n_iter`" hyperparameter.

```
...
for it in range(n_iter):
    ...
```

Program 23.7: Run the gradient descent

The first step is to calculate the gradient for the current solution using the `derivative()` function.

```
...
gradient = derivative(solution[0], solution[1])
```

Program 23.8: Calculate gradient

We then need to calculate the square of the partial derivative of each variable and add them to the running sum of these values.

```
...
for i in range(gradient.shape[0]):
    sq_grad_sums[i] += gradient[i]**2.0
```

Program 23.9: Update the sum of the squared partial derivatives

We can then use the sum squared partial derivatives and gradient to calculate the next point. We will do this one variable at a time, first calculating the step size for the variable, then the new value for the variable. These values are built up in an array until we have a completely new solution that is in the steepest descent direction from the current point using the custom step sizes.

```
...
new_solution = list()
for i in range(solution.shape[0]):
    # calculate the step size for this variable
    alpha = step_size / (1e-8 + sqrt(sq_grad_sums[i]))
    # calculate the new position in this variable
    value = solution[i] - alpha * gradient[i]
    # store this variable
    new_solution.append(value)
```

Program 23.10: Build a solution one variable at a time

This new solution can then be evaluated using the `objective()` function and the performance of the search can be reported.

```
...
# evaluate candidate point
solution = asarray(new_solution)
solution_eval = objective(solution[0], solution[1])
# report progress
print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
```

Program 23.11: Evaluate candidate point and report progress

And that's it.

We can tie all of this together into a function named `adagrad()` that takes the names of the objective function and the derivative function, an array with the bounds of the domain, and hyperparameter values for the total number of algorithm iterations and the initial learning rate, and returns the final solution and its evaluation. This complete function is listed below.

```
def adagrad(objective, derivative, bounds, n_iter, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the sum square gradients for each variable
    sq_grad_sums = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the sum of the squared partial derivatives
```

```
        for i in range(gradient.shape[0]):
            sq_grad_sums[i] += gradient[i]**2.0
        # build a solution one variable at a time
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_sums[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
            # store this variable
            new_solution.append(value)
        # evaluate candidate point
        solution = asarray(new_solution)
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return [solution, solution_eval]
```

Program 23.12: Gradient descent algorithm with AdaGrad

i | **Note:** We are using simple Python lists and imperative programming style instead of NumPy arrays or list compressions intentionally to make the code more readable for Python beginners.

We can then define our hyperparameters and call the `adagrad()` function to optimize our test objective function. In this case, we will use 50 iterations of the algorithm and an initial learning rate of 0.1, both chosen after a little trial and error.

```
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.1
# perform the gradient descent search with adagrad
best, score = adagrad(objective, derivative, bounds, n_iter, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 23.13: Perform gradient descent search with AdaGrad

Tying all of this together, the complete example of gradient descent optimization with adaptive gradients is listed below.

```python
from math import sqrt
from numpy import asarray
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with adagrad
def adagrad(objective, derivative, bounds, n_iter, step_size):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the sum square gradients for each variable
    sq_grad_sums = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the sum of the squared partial derivatives
        for i in range(gradient.shape[0]):
            sq_grad_sums[i] += gradient[i]**2.0
        # build a solution one variable at a time
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_sums[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
            # store this variable
            new_solution.append(value)
        # evaluate candidate point
        solution = asarray(new_solution)
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return [solution, solution_eval]

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.1
# perform the gradient descent search with adagrad
best, score = adagrad(objective, derivative, bounds, n_iter, step_size)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 23.14: *Gradient descent optimization with AdaGrad for a two-dimensional test function*

Running the example applies the AdaGrad optimization algorithm to our test problem and reports the performance of the search for each iteration of the algorithm.

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 a near-optimal solution was found after perhaps 35 iterations of the search, with input values near 0.0 and 0.0, evaluating to 0.0.

```
>0 f([-0.06595599  0.34064899]) = 0.12039
>1 f([-0.02902286  0.27948766]) = 0.07896
>2 f([-0.0129815   0.23463749]) = 0.05522
>3 f([-0.00582483  0.1993997 ]) = 0.03979
>4 f([-0.00261527  0.17071256]) = 0.02915
>5 f([-0.00117437  0.14686138]) = 0.02157
>6 f([-0.00052736  0.12676134]) = 0.01607
>7 f([-0.00023681  0.10966762]) = 0.01203
>8 f([-0.00010634  0.09503809]) = 0.00903
>9 f([-4.77542704e-05  8.24607972e-02]) = 0.00680
>10 f([-2.14444463e-05  7.16123835e-02]) = 0.00513
>11 f([-9.62980437e-06  6.22327049e-02]) = 0.00387
>12 f([-4.32434258e-06  5.41085063e-02]) = 0.00293
>13 f([-1.94188148e-06  4.70624414e-02]) = 0.00221
>14 f([-8.72017797e-07  4.09453989e-02]) = 0.00168
>15 f([-3.91586740e-07  3.56309531e-02]) = 0.00127
>16 f([-1.75845235e-07  3.10112252e-02]) = 0.00096
>17 f([-7.89647442e-08  2.69937139e-02]) = 0.00073
>18 f([-3.54597657e-08  2.34988084e-02]) = 0.00055
>19 f([-1.59234984e-08  2.04577993e-02]) = 0.00042
>20 f([-7.15057749e-09  1.78112581e-02]) = 0.00032
>21 f([-3.21102543e-09  1.55077005e-02]) = 0.00024
>22 f([-1.44193729e-09  1.35024688e-02]) = 0.00018
>23 f([-6.47513760e-10  1.17567908e-02]) = 0.00014
>24 f([-2.90771361e-10  1.02369798e-02]) = 0.00010
>25 f([-1.30573263e-10  8.91375193e-03]) = 0.00008
>26 f([-5.86349941e-11  7.76164047e-03]) = 0.00006
>27 f([-2.63305247e-11  6.75849105e-03]) = 0.00005
>28 f([-1.18239380e-11  5.88502652e-03]) = 0.00003
>29 f([-5.30963626e-12  5.12447017e-03]) = 0.00003
>30 f([-2.38433568e-12  4.46221948e-03]) = 0.00002
>31 f([-1.07070548e-12  3.88556303e-03]) = 0.00002
>32 f([-4.80809073e-13  3.38343471e-03]) = 0.00001
>33 f([-2.15911255e-13  2.94620023e-03]) = 0.00001
>34 f([-9.69567190e-14  2.56547145e-03]) = 0.00001
>35 f([-4.35392094e-14  2.23394494e-03]) = 0.00000
>36 f([-1.95516389e-14  1.94526160e-03]) = 0.00000
>37 f([-8.77982370e-15  1.69388439e-03]) = 0.00000
>38 f([-3.94265180e-15  1.47499203e-03]) = 0.00000
>39 f([-1.77048011e-15  1.28438640e-03]) = 0.00000
>40 f([-7.95048604e-16  1.11841198e-03]) = 0.00000
```

```
>41 f([-3.57023093e-16  9.73885702e-04]) = 0.00000
>42 f([-1.60324146e-16  8.48035867e-04]) = 0.00000
>43 f([-7.19948720e-17  7.38448972e-04]) = 0.00000
>44 f([-3.23298874e-17  6.43023418e-04]) = 0.00000
>45 f([-1.45180009e-17  5.59929193e-04]) = 0.00000
>46 f([-6.51942732e-18  4.87572776e-04]) = 0.00000
>47 f([-2.92760228e-18  4.24566574e-04]) = 0.00000
>48 f([-1.31466380e-18  3.69702307e-04]) = 0.00000
>49 f([-5.90360555e-19  3.21927835e-04]) = 0.00000
Done!
f([-5.90360555e-19  3.21927835e-04]) = 0.000000
```

Output 23.1: Result from Program 23.14

23.4.3 Visualization of AdaGrad

We can plot the progress of the search on a contour plot of the domain. This can provide an intuition for the progress of the search over the iterations of the algorithm. We must update the `adagrad()` function to maintain a list of all solutions found during the search, then return this list at the end of the search. The updated version of the function with these changes is listed below.

```python
def adagrad(objective, derivative, bounds, n_iter, step_size):
    # track all solutions
    solutions = list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the sum square gradients for each variable
    sq_grad_sums = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the sum of the squared partial derivatives
        for i in range(gradient.shape[0]):
            sq_grad_sums[i] += gradient[i]**2.0
        # build solution
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the learning rate for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_sums[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
            new_solution.append(value)
        # store the new solution
        solution = asarray(new_solution)
        solutions.append(solution)
        # evaluate candidate point
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return solutions
```

Program 23.15: Gradient descent algorithm with AdaGrad

We can then execute the search as before, and this time retrieve the list of solutions instead of the best final solution.

```
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.1
# perform the gradient descent search
solutions = adagrad(objective, derivative, bounds, n_iter, step_size)
```

Program 23.16: Perform the gradient descent search

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

```
...
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
```

Program 23.17: Create contour plot of the objective function

Finally, we can plot each solution found during the search as a white dot connected by a line.

```
...
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
```

Program 23.18: Plot the samples as black circles

Tying this all together, the complete example of performing the AdaGrad optimization on the test problem and plotting the results on a contour plot is listed below.

```
from math import sqrt
from numpy import asarray
from numpy import arange
from numpy.random import rand
from numpy.random import seed
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
```

```python
        return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with adagrad
def adagrad(objective, derivative, bounds, n_iter, step_size):
    # track all solutions
    solutions = list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the sum square gradients for each variable
    sq_grad_sums = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the sum of the squared partial derivatives
        for i in range(gradient.shape[0]):
            sq_grad_sums[i] += gradient[i]**2.0
        # build solution
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the learning rate for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_sums[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
            new_solution.append(value)
        # store the new solution
        solution = asarray(new_solution)
        solutions.append(solution)
        # evaluate candidate point
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return solutions

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.1
# perform the gradient descent search
solutions = adagrad(objective, derivative, bounds, n_iter, step_size)
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
```

```
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# plot the sample as black circles
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
# show the plot
pyplot.show()
```

Program 23.19: Example of plotting the AdaGrad search on a contour plot of the test function

Running the example performs the search as before, except in this case, a contour plot of the objective function is created and a white dot is shown for each solution found during the search, starting above the optima and progressively getting closer to the optima at the center of the plot.
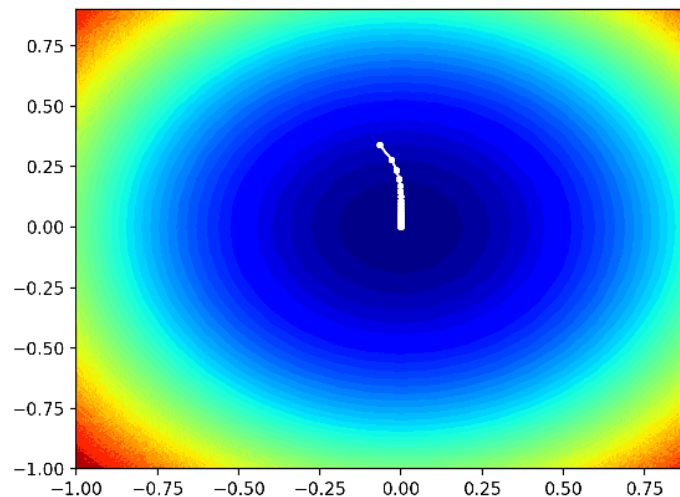
Figure 23.3: Contour Plot of the Test Objective Function with AdaGrad Search Results Shown

23.5 Further Reading

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

Papers

John Duchi, Elad Haza, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research*, 12(61), 2011, pp. 2121–2159.
https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

Books

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

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://amzn.to/3qSk3C2

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

Stochastic gradient descent. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

An overview of gradient descent optimization algorithms. 2016.
https://ruder.io/optimizing-gradient-descent/index.html

23.6   Summary

In this tutorial, you discovered how to develop the gradient descent with adaptive gradients optimization algorithm from scratch. Specifically, you learned:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable in the objective function, called adaptive gradients or AdaGrad.

▷ How to implement the AdaGrad optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Next, we will learn about another variation of gradient descent that is very similar to AdaGrad.

Gradient Descent with RMSProp

<div style="text-align: right; font-size: 3em; color: #bbb;">**24**</div>

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function. A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable. AdaGrad, for short, is an extension of the gradient descent optimization algorithm that allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients seen for the variable (partial derivatives) over the course of the search. A limitation of AdaGrad is that it can result in a very small step size for each parameter by the end of the search that can slow the progress of the search down too much and may mean not locating the optima. *Root Mean Squared Propagation*, or *RMSProp*, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter. The use of a decaying moving average allows the algorithm to forget early gradients and focus on the most recently observed partial gradients seen during the progress of the search, overcoming the limitation of AdaGrad.

In this tutorial, you will discover how to develop the gradient descent with RMSProp optimization algorithm from scratch. After completing this tutorial, you will know:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable using a decaying movingÂă average of partial derivatives, called RMSProp.

▷ How to implement the RMSProp optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Let's get started.

24.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Gradient Descent
2. Root Mean Squared Propagation (RMSProp)
3. Gradient Descent With RMSProp

    (a) Two-Dimensional Test Problem

    (b) Gradient Descent Optimization With RMSProp

    (c) Visualization of RMSProp

24.2 Gradient Descent

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.

> 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[2], or simply the "derivative," 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 a specific input. 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. The gradient descent algorithm requires a target function that is being optimized and the derivative function for the objective 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. 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.

---

[1]https://en.wikipedia.org/wiki/Gradient_descent
[2]https://en.wikipedia.org/wiki/Derivative
[3]https://en.wikipedia.org/wiki/Gradient

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.

Now that we are familiar with the gradient descent optimization algorithm, let's take a look at RMSProp.

24.3   Root Mean Squared Propagation (RMSProp)

Root Mean Squared Propagation, or RMSProp for short, is an extension to the gradient descent optimization algorithm. It is an unpublished extension, first described in Geoffrey Hinton's lecture notes for his Coursera course on neural networks, specifically Lecture 6e titled "rmsprop: Divide the gradient by a running average of its recent magnitude[4]."

RMSProp is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result. It is related to another extension to gradient descent called Adaptive Gradient, or AdaGrad. AdaGrad is designed to specifically explore the idea of automatically tailoring the step size (learning rate) for each parameter in the search space. This is achieved by first calculating a step size for a given dimension, then using the calculated step size to make a movement in that dimension using the partial derivative. This process is then repeated for each dimension in the search space. Adagrad calculates the step size for each parameter by first summing the partial derivatives for the parameter seen so far during the search, then dividing the initial step size hyperparameter by the square root of the sum of the squared partial derivatives. The calculation of the custom step size for one parameter is as follows:

$$\alpha = \frac{\eta}{\sqrt{S}}$$

Where α is the calculated step size for an input variable for a given point during the search, η is the initial step size, and S is the sum of the squared partial derivatives for the input variable seen during the search so far. This has the effect of smoothing out the oscillations in the search for optimization problems that have a lot of curvature in the search space.

> " AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure. "
>
> — Pages 307–308, *Deep Learning*, 2016.

A problem with AdaGrad is that it can slow the search down too much, resulting in very small learning rates for each parameter or dimension of the search by the end of the run. This has the effect of stopping the search too soon, before the minimal can be located.

> " RMSProp extends Adagrad to avoid the effect of a monotonically decreasing learning rate. "
>
> — Page 78, *Algorithms for Optimization*, 2019.

---

[4]http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf

RMSProp can be thought of as an extension of AdaGrad in that it uses a decaying average or moving average of the partial derivatives instead of the sum in the calculation of the learning rate for each parameter. This is achieved by adding a new hyperparameter we will call ρ (rho) that acts like momentum for the partial derivatives.

> RMSProp maintains a decaying average of squared gradients.

— Page 78, *Algorithms for Optimization*, 2019.

Using a decaying moving average of the partial derivative allows the search to forget early partial derivative values and focus on the most recently seen shape of the search space.

> RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

— Page 308, *Deep Learning*, 2016.

The calculation of the mean squared partial derivative for one parameter is as follows:

$$s(t + 1) = s(t) \times \rho + f'(x(t))^2 \times (1 - \rho)$$

Where $s(t+1)$ is the decaying moving average of the squared partial derivative for one parameter for the current iteration of the algorithm, $s(t)$ is the decaying moving average squared partial derivative for the previous iteration, $f'(x(t))^2$ is the squared partial derivative for the current parameter, and ρ is a hyperparameter, typically with the value of 0.9 like momentum. Given that we are using a decaying average of the partial derivatives and calculating the square root of this average gives the technique its name, i.e., square root of the mean squared partial derivatives or root mean square (RMS). For example, with the initial step size as η, the custom step size α for a parameter may be written as:

$$\alpha(t + 1) = \frac{\eta}{10^{-8} + \sqrt{s(t + 1)}}$$

Once we have the custom step size for the parameter, we can update the parameter using the custom step size and the partial derivative $f'(x(t))$.

$$x(t + 1) = x(t) - \alpha(t + 1) \times f'(x(t))$$

This process is then repeated for each input variable until a new point in the search space is created and can be evaluated. RMSProp is a very effective extension of gradient descent and is one of the preferred approaches generally used to fit deep learning neural networks.

> Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

— Page 308, *Deep Learning*, 2016.

Now that we are familiar with the RMSprop algorithm, let's explore how we might implement it and evaluate its performance.

24.4 Gradient Descent With RMSProp

In this section, we will explore how to implement the gradient descent optimization algorithm with adaptive gradients using the RMSProp algorithm.

24.4.1 Two-Dimensional Test Problem

First, let's define an optimization function. We will use a simple two-dimensional function that squares the input of each dimension and define the range of valid inputs from -1.0 to 1.0. The `objective()` function below implements this function

```python
def objective(x, y):
    return x**2.0 + y**2.0
```

Program 24.1: Objective function

We can create a three-dimensional plot of the dataset to get a feeling for the curvature of the response surface. The complete example of plotting the objective function is listed below.

```python
# 3d plot of the test function
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -1.0, 1.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 24.2: 3D plot of the test function

Running the example creates a three-dimensional surface plot of the objective function. We can see the familiar bowl shape with the global minima at $f(0, 0) = 0$.
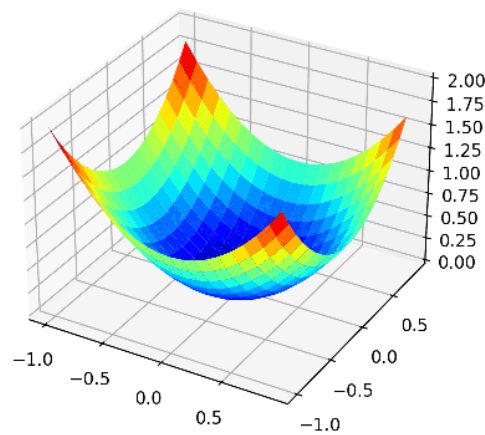
Figure 24.1: Three-Dimensional Plot of the Test Objective Function

We can also create a two-dimensional plot of the function. This will be helpful later when we want to plot the progress of the search. The example below creates a contour plot of the objective function.

```python
from numpy import asarray
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# show the plot
pyplot.show()
```

Program 24.3: Contour plot of the test function

Running the example creates a two-dimensional contour plot of the objective function. We can see the bowl shape compressed to contours shown with a color gradient. We will use this plot to plot the specific points explored during the progress of the search.
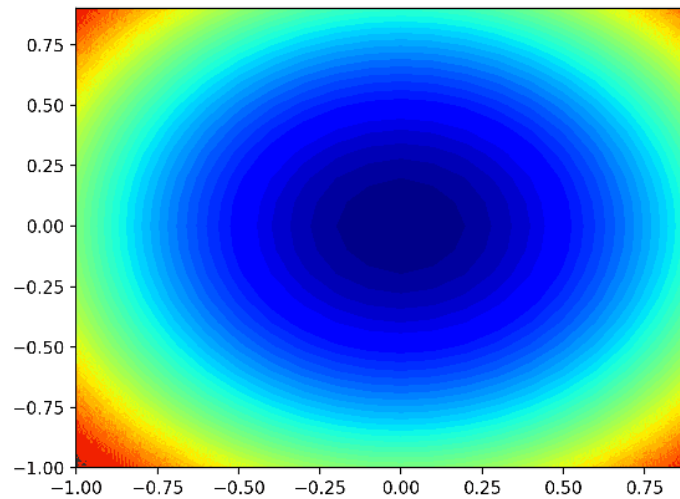
Figure 24.2: Two-Dimensional Contour Plot of the Test Objective Function

Now that we have a test objective function, let's look at how we might implement the RMSProp optimization algorithm.

24.4.2 Gradient Descent Optimization With RMSProp

We can apply the gradient descent with RMSProp to the test problem. First, we need a function that calculates the derivative for this function.

$$f(x) = x^2$$
$$f'(x) = 2 \times x$$

The derivative of x^2 is $2x$ in each dimension. The `derivative()` function implements this below.

```python
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])
```
Program 24.4: Derivative of objective function

Next, we can implement gradient descent optimization. First, we can select a random point in the bounds of the problem as a starting point for the search. This assumes we have an array that defines the bounds of the search with one row for each dimension and the first column defines the minimum and the second column defines the maximum of the dimension.

```python
...
solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
```
Program 24.5: Generate an initial point

Next, we need to initialize the decay average of the squared partial derivatives for each dimension to 0.0 values.

```
...
sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
```
Program 24.6: List of the average square gradients for each variable

We can then enumerate a fixed number of iterations of the search optimization algorithm defined by a "n_iter" hyperparameter.

```
...
for it in range(n_iter):
    ...
```
Program 24.7: Run the gradient descent

The first step is to calculate the gradient for the current solution using the `derivative()` function.

```
...
gradient = derivative(solution[0], solution[1])
```
Program 24.8: Calculate gradient

We then need to calculate the square of the partial derivative and update the decaying average of the squared partial derivatives with the "rho" hyperparameter.

```
...
for i in range(gradient.shape[0]):
    # calculate the squared gradient
    sg = gradient[i]**2.0
    # update the moving average of the squared gradient
    sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
```
Program 24.9: Update the average of the squared partial derivatives

We can then use the moving average of the squared partial derivatives and gradient to calculate the step size for the next point. We will do this one variable at a time, first calculating the step size for the variable, then the new value for the variable. These values are built up in an array until we have a completely new solution that is in the steepest descent direction from the current point using the custom step sizes.

```
...
new_solution = list()
for i in range(solution.shape[0]):
    # calculate the step size for this variable
    alpha = step_size / (1e-8 + sqrt(sq_grad_avg[i]))
    # calculate the new position in this variable
    value = solution[i] - alpha * gradient[i]
    # store this variable
    new_solution.append(value)
```
Program 24.10: Build a solution one variable at a time

This new solution can then be evaluated using the `objective()` function and the performance of the search can be reported.

```
...
solution = asarray(new_solution)
solution_eval = objective(solution[0], solution[1])
# report progress
print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
```

Program 24.11: Evaluate candidate point

And that's it.

We can tie all of this together into a function named `rmsprop()` that takes the names of the objective function and the derivative function, an array with the bounds of the domain and hyperparameter values for the total number of algorithm iterations and the initial learning rate, and returns the final solution and its evaluation. This complete function is listed below.

```
def rmsprop(objective, derivative, bounds, n_iter, step_size, rho):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build a solution one variable at a time
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_avg[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
            # store this variable
            new_solution.append(value)
        # evaluate candidate point
        solution = asarray(new_solution)
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return [solution, solution_eval]
```

Program 24.12: Gradient descent algorithm with RMSProp

> **i** **Note:** We are using simple Python lists and imperative programming style instead of NumPy arrays or list compressions intentionally to make the code more readable for Python beginners.

We can then define our hyperparameters and call the `rmsprop()` function to optimize our test objective function. In this case, we will use 50 iterations of the algorithm, an initial learning rate of 0.01, and a value of 0.99 for the rho hyperparameter, all chosen after a little trial and error.

```
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.01
# momentum for rmsprop
rho = 0.99
# perform the gradient descent search with rmsprop
best, score = rmsprop(objective, derivative, bounds, n_iter, step_size, rho)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Output 24.1: Perform gradient descent search with RMSProp

Tying all of this together, the complete example of gradient descent optimization with RMSProp is listed below.

```python
from math import sqrt
from numpy import asarray
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with rmsprop
def rmsprop(objective, derivative, bounds, n_iter, step_size, rho):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
```

```
                # update the moving average of the squared gradient
                sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
            # build a solution one variable at a time
            new_solution = list()
            for i in range(solution.shape[0]):
                # calculate the step size for this variable
                alpha = step_size / (1e-8 + sqrt(sq_grad_avg[i]))
                # calculate the new position in this variable
                value = solution[i] - alpha * gradient[i]
                # store this variable
                new_solution.append(value)
            # evaluate candidate point
            solution = asarray(new_solution)
            solution_eval = objective(solution[0], solution[1])
            # report progress
            print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
        return [solution, solution_eval]

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.01
# momentum for rmsprop
rho = 0.99
# perform the gradient descent search with rmsprop
best, score = rmsprop(objective, derivative, bounds, n_iter, step_size, rho)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 24.13: *Gradient descent optimization with RMSProp for a two-dimensional test function*

Running the example applies the RMSProp optimization algorithm to our test problem and reports the performance of the search for each iteration of the algorithm.

⚠ **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 a near optimal solution was found after perhaps 33 iterations of the search, with input values near 0.0 and 0.0, evaluating to 0.0.

```
...
>30 f([-9.61030898e-14 3.19352553e-03]) = 0.00001
>31 f([-3.42767893e-14 2.71513758e-03]) = 0.00001
>32 f([-1.21143047e-14 2.30636623e-03]) = 0.00001
>33 f([-4.24204875e-15 1.95738936e-03]) = 0.00000
>34 f([-1.47154482e-15 1.65972553e-03]) = 0.00000
```

```
>35 f([-5.05629595e-16 1.40605727e-03]) = 0.00000
>36 f([-1.72064649e-16 1.19007691e-03]) = 0.00000
>37 f([-5.79813754e-17 1.00635204e-03]) = 0.00000
>38 f([-1.93445677e-17 8.50208253e-04]) = 0.00000
>39 f([-6.38906842e-18 7.17626999e-04]) = 0.00000
>40 f([-2.08860690e-18 6.05156738e-04]) = 0.00000
>41 f([-6.75689941e-19 5.09835645e-04]) = 0.00000
>42 f([-2.16291217e-19 4.29124484e-04]) = 0.00000
>43 f([-6.84948980e-20 3.60848338e-04]) = 0.00000
>44 f([-2.14551097e-20 3.03146089e-04]) = 0.00000
>45 f([-6.64629576e-21 2.54426642e-04]) = 0.00000
>46 f([-2.03575780e-21 2.13331041e-04]) = 0.00000
>47 f([-6.16437387e-22 1.78699710e-04]) = 0.00000
>48 f([-1.84495110e-22 1.49544152e-04]) = 0.00000
>49 f([-5.45667355e-23 1.25022522e-04]) = 0.00000
Done!
f([-5.45667355e-23 1.25022522e-04]) = 0.000000
```

Output 24.2: Result from Program 24.13

24.4.3 Visualization of RMSProp

We can plot the progress of the search on a contour plot of the domain. This can provide an intuition for the progress of the search over the iterations of the algorithm. We must update the `rmsprop()` function to maintain a list of all solutions found during the search, then return this list at the end of the search. The updated version of the function with these changes is listed below.

```python
def rmsprop(objective, derivative, bounds, n_iter, step_size, rho):
    # track all solutions
    solutions = list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build solution
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the learning rate for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_avg[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
```

```
            new_solution.append(value)
        # store the new solution
        solution = asarray(new_solution)
        solutions.append(solution)
        # evaluate candidate point
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return solutions
```

Program 24.14: Gradient descent algorithm with RMSProp

We can then execute the search as before, and this time retrieve the list of solutions instead of the best final solution.

```
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.01
# momentum for rmsprop
rho = 0.99
# perform the gradient descent search with rmsprop
solutions = rmsprop(objective, derivative, bounds, n_iter, step_size, rho)
```

Program 24.15: Perform the gradient descent search with RMSProp

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

```
...
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
```

Program 24.16: Contour plot of the objective function

Finally, we can plot each solution found during the search as a white dot connected by a line.

```
...
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
```

Program 24.17: Plot the sample as black circles

Tying this all together, the complete example of performing the RMSProp optimization on the test problem and plotting the results on a contour plot is listed below.

```python
from math import sqrt
from numpy import asarray
from numpy import arange
from numpy.random import rand
from numpy.random import seed
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with rmsprop
def rmsprop(objective, derivative, bounds, n_iter, step_size, rho):
    # track all solutions
    solutions = list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build solution
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the learning rate for this variable
            alpha = step_size / (1e-8 + sqrt(sq_grad_avg[i]))
            # calculate the new position in this variable
            value = solution[i] - alpha * gradient[i]
            new_solution.append(value)
        # store the new solution
        solution = asarray(new_solution)
        solutions.append(solution)
        # evaluate candidate point
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
```

```
    return solutions

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 50
# define the step size
step_size = 0.01
# momentum for rmsprop
rho = 0.99
# perform the gradient descent search with rmsprop
solutions = rmsprop(objective, derivative, bounds, n_iter, step_size, rho)
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# plot the sample as black circles
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
# show the plot
pyplot.show()
```

Program 24.18: Example of plotting the RMSProp search on a contour plot of the test function

Running the example performs the search as before, except in this case, the contour plot of the objective function is created. In this case, we can see that a white dot is shown for each solution found during the search, starting above the optima and progressively getting closer to the optima at the center of the plot.
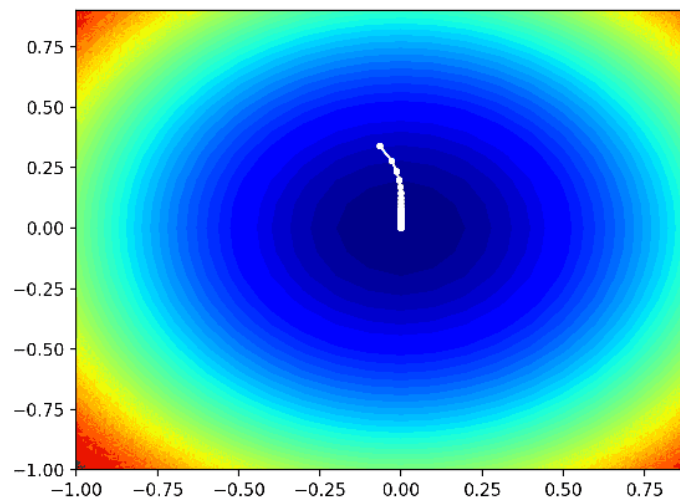
Figure 24.3: Contour plot of the test objective function with RMSProp search results shown

24.5 Further Reading

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

Papers

Geoffrey Hinton. "Lecture 6e, rmsprop: Divide the gradient by a running average of its recent magnitude". Lecture Notes, Neural Networks for Machine Learning.
http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J
Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://amzn.to/3qSk3C2

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

Stochastic gradient descent. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

An overview of gradient descent optimization algorithms. 2016.
https://ruder.io/optimizing-gradient-descent/index.html

24.6   Summary

In this tutorial, you discovered how to develop gradient descent with RMSProp optimization algorithm from scratch. Specifically, you learned:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable using a decaying average of partial derivatives, called RMSProp.

▷ How to implement the RMSProp optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Next, we will learn about an algorithm that further extends on AdaGrad and RMSProp.

Gradient Descent with Adadelta

<div style="text-align: right; font-size: 3em; color: #ccc;">25</div>

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function. A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable. AdaGrad and RMSProp are extensions to gradient descent that add a self-adaptive learning rate for each parameter for the objective function. *Adadelta* can be considered a further extension of gradient descent that builds upon AdaGrad and RMSProp and changes the calculation of the custom step size so that the units are consistent and in turn no longer requires an initial learning rate hyperparameter.

In this tutorial, you will discover how to develop the gradient descent with Adadelta optimization algorithm from scratch. After completing this tutorial, you will know:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable using a decaying average of partial derivatives, called Adadelta.

▷ How to implement the Adadelta optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Let's get started.

25.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Gradient Descent
2. Adadelta Algorithm
3. Gradient Descent With Adadelta
   (a) Two-Dimensional Test Problem
   (b) Gradient Descent Optimization With Adadelta
   (c) Visualization of Adadelta

25.2 Gradient Descent

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.

> " 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[2], or simply the "derivative," 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 a specific input. 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. The gradient descent algorithm requires a target function that is being optimized and the derivative function for the objective 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. 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.

Now that we are familiar with the gradient descent optimization algorithm, let's take a look at Adadelta.

---

[1]https://en.wikipedia.org/wiki/Gradient_descent
[2]https://en.wikipedia.org/wiki/Derivative
[3]https://en.wikipedia.org/wiki/Gradient

25.3 Adadelta Algorithm

Adadelta (or "ADADELTA") is an extension to the gradient descent optimization algorithm. The algorithm was described in the 2012 paper by Matthew Zeiler[4] titled "ADADELTA: An Adaptive Learning Rate Method[5]." Adadelta is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result. It is best understood as an extension of the AdaGrad and RMSProp algorithms.

AdaGrad is an extension of gradient descent that calculates a step size (learning rate) for each parameter for the objective function each time an update is made. The step size is calculated by first summing the partial derivatives for the parameter seen so far during the search, then dividing the initial step size hyperparameter by the square root of the sum of the squared partial derivatives. The calculation of the custom step size for one parameter with AdaGrad is as follows:

$$\alpha = \frac{\eta}{\sqrt{S}}$$

Where α is the calculated step size for an input variable for a given point during the search, η is the initial step size, and S is the sum of the squared partial derivatives for the input variable seen during the search so far (including the current iteration).

RMSProp can be thought of as an extension of AdaGrad in that it uses a decaying average or moving average of the partial derivatives instead of the sum in the calculation of the step size for each parameter. This is achieved by adding a new hyperparameter "ρ" (rho) that acts like a momentum for the partial derivatives. The calculation of the decaying moving average squared partial derivative for one parameter is as follows:

$$s(t+1) = s(t) \times \rho + f'(x(t))^2 \times (1 - \rho)$$

Where $s(t+1)$ is the mean squared partial derivative for one parameter for the current iteration of the algorithm, $s(t)$ is the decaying moving average squared partial derivative for the previous iteration, $f'(x(t))^2$ is the squared partial derivative for the current parameter, and rho is a hyperparameter, typically with the value of 0.9 like momentum.

Adadelta is a further extension of RMSProp designed to improve the convergence of the algorithm and to remove the need for a manually specified initial learning rate.

> " The idea presented in this paper was derived from ADAGRAD in order to improve upon the two main drawbacks of the method: 1) the continual decay of learning rates throughout training, and 2) the need for a manually selected global learning rate. "

— "ADADELTA: An Adaptive Learning Rate Method", 2012.

The decaying moving average of the squared partial derivative is calculated for each parameter, as with RMSProp. The key difference is in the calculation of the step size for a parameter that uses the decaying average of the delta or change in parameter. This choice of numerator was to ensure that both parts of the calculation have the same units.

---

[4]https://www.linkedin.com/in/mattzeiler/
[5]https://arxiv.org/abs/1212.5701

> After independently deriving the RMSProp update, the authors noticed that the
> units in the update equations for gradient descent, momentum and Adagrad do not
> match. To fix this, they use an exponentially decaying average of the square updates

— Pages 78–79, *Algorithms for Optimization*, 2019.

First, the custom step size is calculated as the square root of the decaying moving average of the change in the delta divided by the square root of the decaying moving average of the squared partial derivatives.

$$\alpha(t+1) = \frac{\epsilon + \sqrt{\delta(t)}}{\epsilon + \sqrt{s(t)}}$$

Where $\alpha(t+1)$ is the custom step size for a parameter for a given update, ϵ is a hyperparameter that is added to the numerator and denominator to avoid a divide by zero error, $\delta(t)$ is the decaying moving average of the squared change to the parameter (calculated in the last iteration), and $s(t)$ is the decaying moving average of the squared partial derivative (calculated in the current iteration).

The ϵ hyperparameter is set to a small value such as 10^{-3} or 10^{-8}. In addition to avoiding a divide by zero error, it also helps with the first step of the algorithm when the decaying moving average squared change and decaying moving average squared gradient are zero. Next, the change to the parameter is calculated as the custom step size multiplied by the partial derivative

$$\Delta x(t+1) = \alpha(t+1) \times f'(x(t))$$

Next, the decaying average of the squared change to the parameter is updated.

$$\delta(t+1) = \delta(t) \times \rho + \Delta x(t+1)^2 \times (1 - \rho)$$

Where $\delta(t+1)$ is the decaying average of the change to the variable to be used in the next iteration, $\Delta x(t+1)$ was calculated in the step before and ρ is a hyperparameter that acts like momentum and has a value like 0.9. Finally, the new value for the variable is calculated using the change.

$$x(t+1) = x(t) - \Delta x(t+1)$$

This process is then repeated for each variable for the objective function, then the entire process is repeated to navigate the search space for a fixed number of algorithm iterations. Now that we are familiar with the Adadelta algorithm, let's explore how we might implement it and evaluate its performance.

25.4 Gradient Descent With Adadelta

In this section, we will explore how to implement the gradient descent optimization algorithm with Adadelta.

25.4.1 Two-Dimensional Test Problem

First, let's define an optimization function. We will use a simple two-dimensional function that squares the input of each dimension and define the range of valid inputs from -1.0 to 1.0. The `objective()` function below implements this function

```python
def objective(x, y):
    return x**2.0 + y**2.0
```

Program 25.1: Objective function

We can create a three-dimensional plot of the dataset to get a feeling for the curvature of the response surface. The complete example of plotting the objective function is listed below.

```python
# 3d plot of the test function
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -1.0, 1.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 25.2: 3D plot of the test function

Running the example creates a three dimensional surface plot of the objective function. We can see the familiar bowl shape with the global minima at $f(0,0) = 0$.
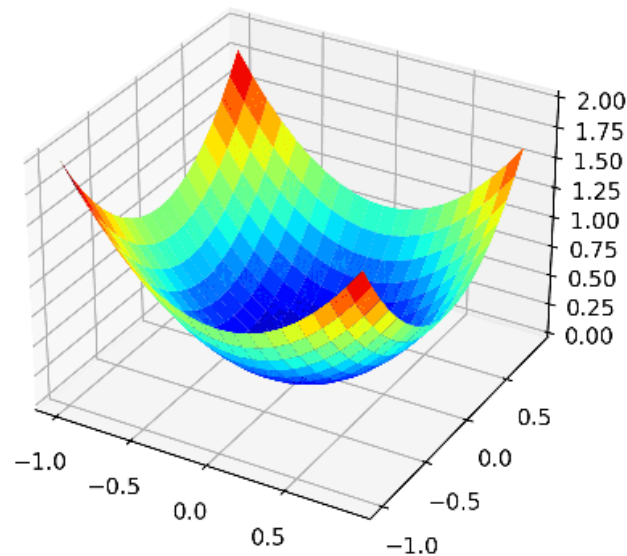
Figure 25.1: Three-Dimensional Plot of the Test Objective Function

We can also create a two-dimensional plot of the function. This will be helpful later when we want to plot the progress of the search. The example below creates a contour plot of the objective function.

```python
from numpy import asarray
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# show the plot
pyplot.show()
```

Program 25.3: Contour plot of the test function

Running the example creates a two-dimensional contour plot of the objective function. We can see the bowl shape compressed to contours shown with a color gradient. We will use this plot to plot the specific points explored during the progress of the search.
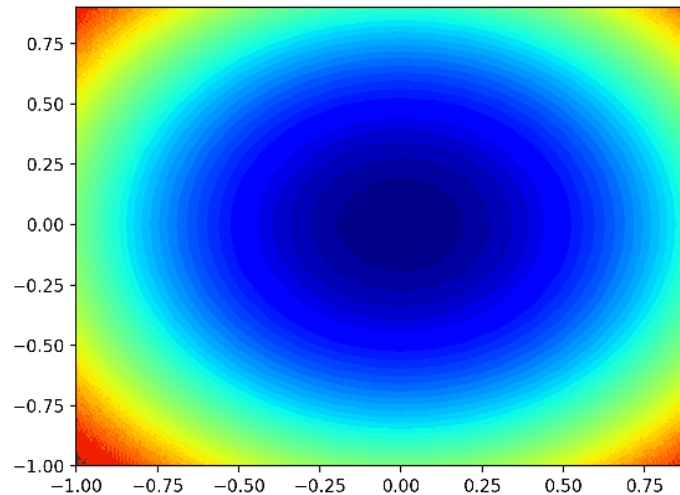
Figure 25.2: *Two-Dimensional Contour Plot of the Test Objective Function*

Now that we have a test objective function, let's look at how we might implement the Adadelta optimization algorithm.

25.4.2 Gradient Descent Optimization With Adadelta

We can apply the gradient descent with Adadelta to the test problem. First, we need a function that calculates the derivative for this function.

$$f(x) = x^2$$
$$f'(x) = 2 \times x$$

The derivative of x^2 is $2x$ in each dimension. The `derivative()` function implements this below.

```
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])
```

Program 25.4: *Derivative of objective function*

Next, we can implement gradient descent optimization. First, we can select a random point in the bounds of the problem as a starting point for the search. This assumes we have an array that defines the bounds of the search with one row for each dimension and the first column defines the minimum and the second column defines the maximum of the dimension.

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

Program 25.5: *Generate an initial point*

Next, we need to initialize the decaying average of the squared partial derivatives and squared change for each dimension to 0.0 values.

```
...
# list of the average square gradients for each variable
sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
# list of the average parameter updates
sq_para_avg = [0.0 for _ in range(bounds.shape[0])]
```

Program 25.6: List of average square gradient and average parameter updates

We can then enumerate a fixed number of iterations of the search optimization algorithm defined by a "`n_iter`" hyperparameter.

```
...
for it in range(n_iter):
    ...
```

Program 25.7: Run the gradient descent

The first step is to calculate the gradient for the current solution using the `derivative()` function.

```
...
gradient = derivative(solution[0], solution[1])
```

Program 25.8: Calculate gradient

We then need to calculate the square of the partial derivative and update the decaying moving average of the squared partial derivatives with the "`rho`" hyperparameter.

```
...
for i in range(gradient.shape[0]):
    # calculate the squared gradient
    sg = gradient[i]**2.0
    # update the moving average of the squared gradient
    sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
```

Program 25.9: Update the average of the squared partial derivatives

We can then use the decaying moving average of the squared partial derivatives and gradient to calculate the step size for the next point. We will do this one variable at a time.

```
...
new_solution = list()
for i in range(solution.shape[0]):
    ...
```

Program 25.10: Build solution

First, we will calculate the custom step size for this variable on this iteration using the decaying moving average of the squared changes and squared partial derivatives, as well as the "`ep`" hyperparameter.

```
...
alpha = (ep + sqrt(sq_para_avg[i])) / (ep + sqrt(sq_grad_avg[i]))
```
Program 25.11: Calculate the step size for this variable

Next, we can use the custom step size and partial derivative to calculate the change to the variable.

```
...
change = alpha * gradient[i]
```
Program 25.12: Calculate the change

We can then use the change to update the decaying moving average of the squared change using the "rho" hyperparameter.

```
...
sq_para_avg[i] = (sq_para_avg[i] * rho) + (change**2.0 * (1.0-rho))
```
Program 25.13: Update the moving average of squared parameter changes

Finally, we can change the variable and store the result before moving on to the next variable.

```
...
# calculate the new position in this variable
value = solution[i] - change
# store this variable
new_solution.append(value)
```
Program 25.14: Calculate the new position

This new solution can then be evaluated using the `objective()` function and the performance of the search can be reported.

```
...
# evaluate candidate point
solution = asarray(new_solution)
solution_eval = objective(solution[0], solution[1])
# report progress
print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
```
Program 25.15: Evaluate candidate point and report progress

And that's it.

We can tie all of this together into a function named `adadelta()` that takes the names of the objective function and the derivative function, an array with the bounds of the domain and hyperparameter values for the total number of algorithm iterations and `rho`, and returns the final solution and its evaluation. The `ep` hyperparameter can also be taken as an argument, although has a sensible default value of `1e-3`. This complete function is listed below.

```python
def adadelta(objective, derivative, bounds, n_iter, rho, ep=1e-3):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # list of the average parameter updates
    sq_para_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build a solution one variable at a time
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = (ep + sqrt(sq_para_avg[i])) / (ep + sqrt(sq_grad_avg[i]))
            # calculate the change
            change = alpha * gradient[i]
            # update the moving average of squared parameter changes
            sq_para_avg[i] = (sq_para_avg[i] * rho) + (change**2.0 * (1.0-rho))
            # calculate the new position in this variable
            value = solution[i] - change
            # store this variable
            new_solution.append(value)
        # evaluate candidate point
        solution = asarray(new_solution)
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return [solution, solution_eval]
```

Program 25.16: Gradient descent algorithm with Adadelta

> **Note:** We are using simple Python lists and imperative programming style instead of NumPy arrays or list compressions intentionally to make the code more readable for Python beginners.

We can then define our hyperparameters and call the `adadelta()` function to optimize our test objective function. In this case, we will use 120 iterations of the algorithm and a value of 0.99 for the `rho` hyperparameter, chosen after a little trial and error.

```python
...
# seed the pseudo random number generator
seed(1)
# define range for input
```

```
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 120
# momentum for adadelta
rho = 0.99
# perform the gradient descent search with adadelta
best, score = adadelta(objective, derivative, bounds, n_iter, rho)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 25.17: Perform gradient descent search with Adadelta

Tying all of this together, the complete example of gradient descent optimization with Adadelta is listed below.

```
from math import sqrt
from numpy import asarray
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with adadelta
def adadelta(objective, derivative, bounds, n_iter, rho, ep=1e-3):
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # list of the average parameter updates
    sq_para_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build a solution one variable at a time
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = (ep + sqrt(sq_para_avg[i])) / (ep + sqrt(sq_grad_avg[i]))
            # calculate the change
            change = alpha * gradient[i]
            # update the moving average of squared parameter changes
```

```
            sq_para_avg[i] = (sq_para_avg[i] * rho) + (change**2.0 * (1.0-rho))
            # calculate the new position in this variable
            value = solution[i] - change
            # store this variable
            new_solution.append(value)
        # evaluate candidate point
        solution = asarray(new_solution)
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return [solution, solution_eval]

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 120
# momentum for adadelta
rho = 0.99
# perform the gradient descent search with adadelta
best, score = adadelta(objective, derivative, bounds, n_iter, rho)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 25.18: Gradient descent optimization with Adadelta for a two-dimensional test function

Running the example applies the Adadelta optimization algorithm to our test problem and reports performance of the search for each iteration of the algorithm.

⚠️ **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 a near optimal solution was found after perhaps 105 iterations of the search, with input values near 0.0 and 0.0, evaluating to 0.0.

```
...
>100 f([-1.45142626e-07 2.71163181e-03]) = 0.00001
>101 f([-1.24898699e-07 2.56875692e-03]) = 0.00001
>102 f([-1.07454197e-07 2.43328237e-03]) = 0.00001
>103 f([-9.24253035e-08 2.30483111e-03]) = 0.00001
>104 f([-7.94803792e-08 2.18304501e-03]) = 0.00000
>105 f([-6.83329263e-08 2.06758392e-03]) = 0.00000
>106 f([-5.87354975e-08 1.95812477e-03]) = 0.00000
>107 f([-5.04744185e-08 1.85436071e-03]) = 0.00000
>108 f([-4.33652179e-08 1.75600036e-03]) = 0.00000
>109 f([-3.72486699e-08 1.66276699e-03]) = 0.00000
>110 f([-3.19873691e-08 1.57439783e-03]) = 0.00000
>111 f([-2.74627662e-08 1.49064334e-03]) = 0.00000
>112 f([-2.3572602e-08 1.4112666e-03]) = 0.00000
```

```
>113 f([-2.02286891e-08 1.33604264e-03]) = 0.00000
>114 f([-1.73549914e-08 1.26475787e-03]) = 0.00000
>115 f([-1.48859650e-08 1.19720951e-03]) = 0.00000
>116 f([-1.27651224e-08 1.13320504e-03]) = 0.00000
>117 f([-1.09437923e-08 1.07256172e-03]) = 0.00000
>118 f([-9.38004754e-09 1.01510604e-03]) = 0.00000
>119 f([-8.03777865e-09 9.60673346e-04]) = 0.00000
Done!
f([-8.03777865e-09 9.60673346e-04]) = 0.000001
```

Output 25.1: Result from Program 25.18

25.4.3 Visualization of Adadelta

We can plot the progress of the Adadelta search on a contour plot of the domain. This can provide an intuition for the progress of the search over the iterations of the algorithm. We must update the `adadelta()` function to maintain a list of all solutions found during the search, then return this list at the end of the search. The updated version of the function with these changes is listed below.

```python
def adadelta(objective, derivative, bounds, n_iter, rho, ep=1e-3):
    # track all solutions
    solutions = list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # list of the average parameter updates
    sq_para_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build solution
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = (ep + sqrt(sq_para_avg[i])) / (ep + sqrt(sq_grad_avg[i]))
            # calculate the change
            change = alpha * gradient[i]
            # update the moving average of squared parameter changes
            sq_para_avg[i] = (sq_para_avg[i] * rho) + (change**2.0 * (1.0-rho))
            # calculate the new position in this variable
            value = solution[i] - change
            # store this variable
            new_solution.append(value)
```

```
        # store the new solution
        solution = asarray(new_solution)
        solutions.append(solution)
        # evaluate candidate point
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return solutions
```
Program 25.19: Gradient descent algorithm with Adadelta

We can then execute the search as before, and this time retrieve the list of solutions instead of the best final solution.

```
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 120
# rho for adadelta
rho = 0.99
# perform the gradient descent search with adadelta
solutions = adadelta(objective, derivative, bounds, n_iter, rho)
```
Program 25.20: Perform gradient descent search with Adadelta

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

```
...
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
```
Program 25.21: Create contour plot of the objective function

Finally, we can plot each solution found during the search as a white dot connected by a line.

```
...
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
```
Program 25.22: Plot the sample as black circles

Tying this all together, the complete example of performing the Adadelta optimization on the test problem and plotting the results on a contour plot is listed below.

```python
from math import sqrt
from numpy import asarray
from numpy import arange
from numpy.random import rand
from numpy.random import seed
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with adadelta
def adadelta(objective, derivative, bounds, n_iter, rho, ep=1e-3):
    # track all solutions
    solutions = list()
    # generate an initial point
    solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # list of the average square gradients for each variable
    sq_grad_avg = [0.0 for _ in range(bounds.shape[0])]
    # list of the average parameter updates
    sq_para_avg = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent
    for it in range(n_iter):
        # calculate gradient
        gradient = derivative(solution[0], solution[1])
        # update the average of the squared partial derivatives
        for i in range(gradient.shape[0]):
            # calculate the squared gradient
            sg = gradient[i]**2.0
            # update the moving average of the squared gradient
            sq_grad_avg[i] = (sq_grad_avg[i] * rho) + (sg * (1.0-rho))
        # build solution
        new_solution = list()
        for i in range(solution.shape[0]):
            # calculate the step size for this variable
            alpha = (ep + sqrt(sq_para_avg[i])) / (ep + sqrt(sq_grad_avg[i]))
            # calculate the change
            change = alpha * gradient[i]
            # update the moving average of squared parameter changes
            sq_para_avg[i] = (sq_para_avg[i] * rho) + (change**2.0 * (1.0-rho))
            # calculate the new position in this variable
            value = solution[i] - change
            # store this variable
            new_solution.append(value)
        # store the new solution
        solution = asarray(new_solution)
        solutions.append(solution)
        # evaluate candidate point
```

```
        solution_eval = objective(solution[0], solution[1])
        # report progress
        print('>%d f(%s) = %.5f' % (it, solution, solution_eval))
    return solutions

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 120
# rho for adadelta
rho = 0.99
# perform the gradient descent search with adadelta
solutions = adadelta(objective, derivative, bounds, n_iter, rho)
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# plot the sample as black circles
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
# show the plot
pyplot.show()
```

Program 25.23: Example of plotting the Adadelta search on a contour plot of the test function

Running the example performs the search as before, except in this case, the contour plot of the objective function is created. In this case, we can see that a white dot is shown for each solution found during the search, starting above the optima and progressively getting closer to the optima at the center of the plot.
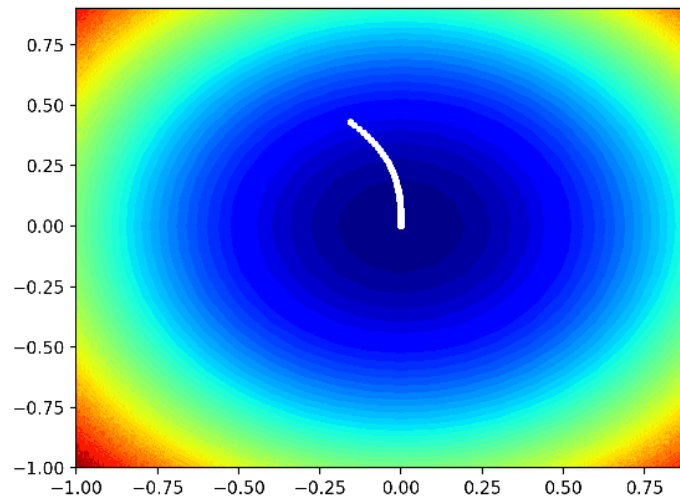
Figure 25.3: Contour Plot of the Test Objective Function With Adadelta Search Results Shown

25.5 Further Reading

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

Papers

Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". arXiv 1212.5701. 2012.
https://arxiv.org/abs/1212.5701

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J
Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://amzn.to/3qSk3C2

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
Stochastic gradient descent. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent
An overview of gradient descent optimization algorithms. 2016.
https://ruder.io/optimizing-gradient-descent/index.html

25.6 Summary

In this tutorial, you discovered how to develop the gradient descent with Adadelta optimization algorithm from scratch. Specifically, you learned:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable using a decaying average of partial derivatives, called Adadelta.

▷ How to implement the Adadelta optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Next, you will learn about Adam, another variant of gradient descent.

Adam Optimization Algorithm

26

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function. A limitation of gradient descent is that a single step size (learning rate) is used for all input variables. Extensions to gradient descent like AdaGrad and RMSProp update the algorithm to use a separate step size for each input variable but may result in a step size that rapidly decreases to very small values. The *Adaptive Movement Estimation* algorithm, or *Adam* for short, is an extension to gradient descent and a natural successor to techniques like AdaGrad and RMSProp that automatically adapts a learning rate for each input variable for the objective function and further smooths the search process by using an exponentially decreasing moving average of the gradient to make updates to variables.

In this tutorial, you will discover how to develop gradient descent with Adam optimization algorithm from scratch. After completing this tutorial, you will know:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable using a decaying average of partial derivatives, called Adam.

▷ How to implement the Adam optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Let's get started.

26.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Gradient Descent

2. Adam Optimization Algorithm

3. Gradient Descent With Adam

    (a) Two-Dimensional Test Problem

    (b) Gradient Descent Optimization With Adam

(c) Visualization of Adam

26.2 Gradient Descent

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.

> 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[2], or simply the "*derivative*," 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 a specific input. 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.

The gradient descent algorithm requires a target function that is being optimized and the derivative function for the objective 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. 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.

---

[1]https://en.wikipedia.org/wiki/Gradient_descent
[2]https://en.wikipedia.org/wiki/Derivative
[3]https://en.wikipedia.org/wiki/Gradient

Now that we are familiar with the gradient descent optimization algorithm, let's take a look at the Adam algorithm.

26.3   Adam Optimization Algorithm

Adaptive Movement Estimation algorithm, or Adam for short, is an extension to the gradient descent optimization algorithm. The algorithm was described in the 2014 paper by Diederik Kingma and Jimmy Lei Ba titled "Adam: A Method for Stochastic Optimization[4]." Adam is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result. This is achieved by calculating a step size for each input parameter that is being optimized. Importantly, each step size is automatically adapted throughput the search process based on the gradients (partial derivatives) encountered for each variable.

> We propose Adam, a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation
>
> — "Adam: A Method for Stochastic Optimization", 2015.

This involves maintaining a first and second moment of the gradient, e.g. an exponentially decaying mean gradient (first moment) and variance (second moment) for each input variable.

> The moving averages themselves are estimates of the 1st moment (the mean) and the 2nd raw moment (the uncentered variance) of the gradient.
>
> — "Adam: A Method for Stochastic Optimization", 2015.

Let's step through each element of the algorithm. First, we must maintain a moment vector and exponentially weighted infinity norm for each parameter being optimized as part of the search, referred to as m and ν (really the Greek letter nu) respectively. They are initialized to 0.0 at the start of the search.

$$m = 0$$
$$\nu = 0$$

The algorithm is executed iteratively over time t starting at $t = 1$, and each iteration involves calculating a new set of parameter values x, i.e. going from $x(t-1)$ to $x(t)$. It is perhaps easy to understand the algorithm if we focus on updating one parameter, which generalizes to updating all parameters via vector operations. First, the gradient (partial derivatives) are calculated for the current time step.

$$g(t) = f'(x(t-1))$$

Next, the first moment is updated using the gradient and a hyperparameter β_1.

$$m(t) = \beta_1 \times m(t-1) + (1 - \beta_1) \times g(t)$$

---
[4]https://arxiv.org/abs/1412.6980

Then the second moment is updated using the squared gradient and a hyperparameter β_2.

$$\nu(t) = \beta_2 \times \nu(t-1) + (1 - \beta_2) \times g(t)^2$$

The first and second moments are biased because they are initialized with zero values.

> " ... these moving averages are initialized as (vectors of) 0's, leading to moment estimates that are biased towards zero, especially during the initial timesteps, and especially when the decay rates are small (i.e. the betas are close to 1). The good news is that this initialization bias can be easily counteracted, resulting in bias-corrected estimates ...
>
> — "Adam: A Method for Stochastic Optimization", 2015.

Next the first and second moments are bias-corrected, starting with the first moment:

$$\hat{m}(t) = \frac{m(t)}{1 - \beta_1(t)}$$

And then the second moment:

$$\hat{\nu}(t) = \frac{\nu(t)}{1 - \beta_2(t)}$$

Note, $\beta_1(t)$ and $\beta_2(t)$ refer to the β_1 and β_2 hyperparameters that are decayed on a schedule over the iterations of the algorithm. A static decay schedule can be used, although the paper recommend the following:

$$\beta_1(t) = \beta_1^t$$
$$\beta_2(t) = \beta_2^t$$

Finally, we can calculate the value for the parameter for this iteration.

$$x(t) = x(t-1) - \frac{\alpha \times \hat{m}(t)}{\sqrt{\hat{\nu}(t)} + \epsilon}$$

Where α is the step size hyperparameter, ϵ is a small value such as 10^{-8} that ensures we do not encounter a divide by zero error. Note, a more efficient reordering of the update rule listed in the paper can be used:

$$\alpha(t) = \alpha \frac{\sqrt{1 - \beta_2(t)}}{1 - \beta_1(t)}$$
$$x(t) = x(t-1) - \frac{\alpha(t) \times m(t)}{\sqrt{\nu(t)} + \epsilon}$$

To review, there are three hyperparameters for the algorithm, they are:

  ▷ α: Initial step size (learning rate), a typical value is 0.001.

  ▷ β_1: Decay factor for first momentum, a typical value is 0.9.

  ▷ β_2: Decay factor for infinity norm, a typical value is 0.999.

And that's it. For full derivation of the Adam algorithm in the context of the Adam algorithm, I recommend reading the paper.

▷ Diederik P. Kingma and Jimmy Lei Ba, "Adam: A Method for Stochastic Optimization," in *Proc. 3rd Int. Conf. Learning Representations (ICLR)*, 2015.[5]

Next, let's look at how we might implement the algorithm from scratch in Python.

26.4   Gradient Descent With Adam

In this section, we will explore how to implement the gradient descent optimization algorithm with Adam.

26.4.1 Two-Dimensional Test Problem

First, let's define an optimization function. We will use a simple two-dimensional function that squares the input of each dimension and define the range of valid inputs from −1.0 to 1.0. The `objective()` function below implements this function

```python
def objective(x, y):
    return x**2.0 + y**2.0
```

Program 26.1: Objective function

We can create a three-dimensional plot of the dataset to get a feeling for the curvature of the response surface. The complete example of plotting the objective function is listed below.

```python
# 3d plot of the test function
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
r_min, r_max = -1.0, 1.0
# sample input range uniformly at 0.1 increments
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a surface plot with the jet color scheme
fig, ax = pyplot.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, results, cmap='jet')
# show the plot
pyplot.show()
```

Program 26.2: 3D plot of the test function

---

[5]https://arxiv.org/abs/1412.6980

Running the example creates a three-dimensional surface plot of the objective function. We can see the familiar bowl shape with the global minima at $f(0,0) = 0$.
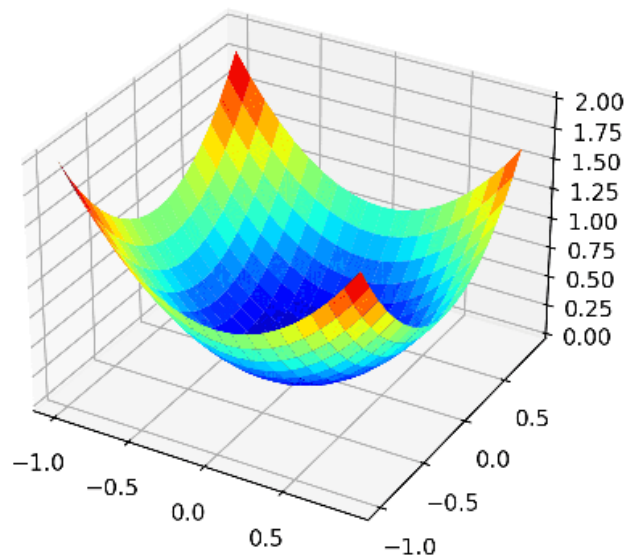
Figure 26.1: Three-Dimensional Plot of the Test Objective Function

We can also create a two-dimensional plot of the function. This will be helpful later when we want to plot the progress of the search. The example below creates a contour plot of the objective function.

```python
from numpy import asarray
from numpy import arange
from numpy import meshgrid
from matplotlib import pyplot

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
```

```
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# show the plot
pyplot.show()
```

Program 26.3: Contour plot of the test function

Running the example creates a two-dimensional contour plot of the objective function. We can see the bowl shape compressed to contours shown with a color gradient. We will use this plot to plot the specific points explored during the progress of the search.
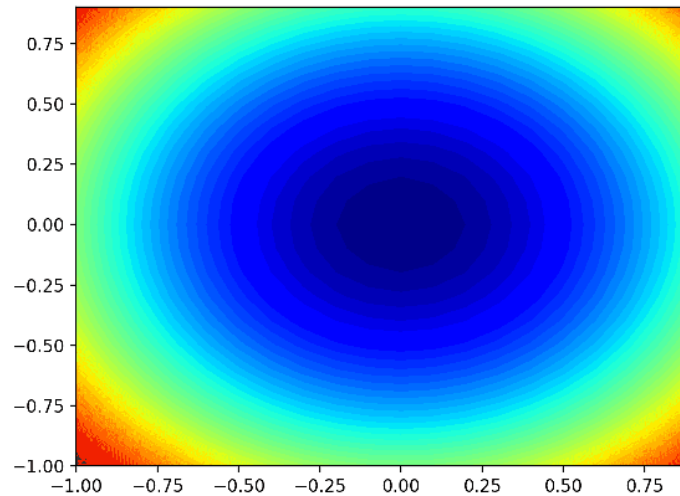
Figure 26.2: Two-Dimensional Contour Plot of the Test Objective Function

Now that we have a test objective function, let's look at how we might implement the Adam optimization algorithm.

26.4.2 Gradient Descent Optimization With Adam

We can apply the gradient descent with Adam to the test problem. First, we need a function that calculates the derivative for this function.

$$f(x) = x^2$$
$$f'(x) = 2 \times x$$

The derivative of x^2 is $2x$ in each dimension. The `derivative()` function implements this below.

```
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])
```

Program 26.4: Derivative of objective function

Next, we can implement gradient descent optimization. First, we can select a random point in the bounds of the problem as a starting point for the search. This assumes we have an array that defines the bounds of the search with one row for each dimension and the first column defines the minimum and the second column defines the maximum of the dimension.

```
...
# generate an initial point
x = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
score = objective(x[0], x[1])
```
Program 26.5: Evaluate the objective function at a random point

Next, we need to initialize the first and second moments to zero.

```
...
m = [0.0 for _ in range(bounds.shape[0])]
v = [0.0 for _ in range(bounds.shape[0])]
```
Program 26.6: Initialize first and second moments

We then run a fixed number of iterations of the algorithm defined by the "n_iter" hyperparameter.

```
...
for t in range(n_iter):
    ...
```
Program 26.7: Run iterations of gradient descent

The first step is to calculate the gradient for the current solution using the `derivative()` function.

```
...
gradient = derivative(solution[0], solution[1])
```
Program 26.8: Calculate gradient

The first step is to calculate the derivative for the current set of parameters.

```
...
g = derivative(x[0], x[1])
```
Program 26.9: Calculate gradient

Next, we need to perform the Adam update calculations. We will perform these calculations one variable at a time using an imperative programming style for readability. In practice, I recommend using NumPy vector operations for efficiency.

```
...
for i in range(x.shape[0]):
    ...
```
Program 26.10: Build a solution one variable at a time

First, we need to calculate the first moment.

```
...
# m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
```
Program 26.11: Calculate the first moment

Then the second moment.

```
...
# v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2
```
Program 26.12: Calculate the second moment

Then the bias correction for the first and second moments.

```
...
# mhat(t) = m(t) / (1 - beta1(t))
mhat = m[i] / (1.0 - beta1**(t+1))
# vhat(t) = v(t) / (1 - beta2(t))
vhat = v[i] / (1.0 - beta2**(t+1))
```
Program 26.13: Calculate the bias correction for the first and second moments

Then finally the updated variable value.

```
...
# x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + eps)
x[i] = x[i] - alpha * mhat / (sqrt(vhat) + eps)
```
Program 26.14: Update the variable

This is then repeated for each parameter that is being optimized. At the end of the iteration we can evaluate the new parameter values and report the performance of the search.

```
...
# evaluate candidate point
score = objective(x[0], x[1])
# report progress
print('>%d f(%s) = %.5f' % (t, x, score))
```
Program 26.15: Evaluate candidate point and report progress

We can tie all of this together into a function named `adam()` that takes the names of the objective and derivative functions as well as the algorithm hyperparameters, and returns the best solution found at the end of the search and its evaluation. This complete function is listed below.

```
def adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2, eps=1e-8):
    # generate an initial point
    x = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    score = objective(x[0], x[1])
    # initialize first and second moments
    m = [0.0 for _ in range(bounds.shape[0])]
    v = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent updates
    for t in range(n_iter):
        # calculate gradient g(t)
        g = derivative(x[0], x[1])
        # build a solution one variable at a time
        for i in range(x.shape[0]):
```

```
            # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
            m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
            # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
            v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2
            # mhat(t) = m(t) / (1 - beta1(t))
            mhat = m[i] / (1.0 - beta1**(t+1))
            # vhat(t) = v(t) / (1 - beta2(t))
            vhat = v[i] / (1.0 - beta2**(t+1))
            # x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + eps)
            x[i] = x[i] - alpha * mhat / (sqrt(vhat) + eps)
        # evaluate candidate point
        score = objective(x[0], x[1])
        # report progress
        print('>%d f(%s) = %.5f' % (t, x, score))
    return [x, score]
```
Program 26.16: Gradient descent algorithm with Adam

ℹ **Note:** We are using simple Python lists and imperative programming style instead of NumPy arrays or list compressions intentionally to make the code more readable for Python beginners.

We can then define our hyperparameters and call the `adam()` function to optimize our test objective function. In this case, we will use 60 iterations of the algorithm with an initial steps size of 0.02 and `beta1` and `beta2` values of 0.8 and 0.999 respectively. These hyperparameter values were found after a little trial and error.

```
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 60
# steps size
alpha = 0.02
# factor for average gradient
beta1 = 0.8
# factor for average squared gradient
beta2 = 0.999
# perform the gradient descent search with adam
best, score = adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2)
print('Done!')
print('f(%s) = %f' % (best, score))
```
Program 26.17: Perform gradient descent search with Adam

Tying all of this together, the complete example of gradient descent optimization with Adam is listed below.

```python
from math import sqrt
from numpy import asarray
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with adam
def adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2, eps=1e-8):
    # generate an initial point
    x = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    score = objective(x[0], x[1])
    # initialize first and second moments
    m = [0.0 for _ in range(bounds.shape[0])]
    v = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent updates
    for t in range(n_iter):
        # calculate gradient g(t)
        g = derivative(x[0], x[1])
        # build a solution one variable at a time
        for i in range(x.shape[0]):
            # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
            m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
            # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
            v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2
            # mhat(t) = m(t) / (1 - beta1(t))
            mhat = m[i] / (1.0 - beta1**(t+1))
            # vhat(t) = v(t) / (1 - beta2(t))
            vhat = v[i] / (1.0 - beta2**(t+1))
            # x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + eps)
            x[i] = x[i] - alpha * mhat / (sqrt(vhat) + eps)
        # evaluate candidate point
        score = objective(x[0], x[1])
        # report progress
        print('>%d f(%s) = %.5f' % (t, x, score))
    return [x, score]

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 60
# steps size
alpha = 0.02
# factor for average gradient
beta1 = 0.8
# factor for average squared gradient
```

```
beta2 = 0.999
# perform the gradient descent search with adam
best, score = adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Program 26.18: Gradient descent optimization with Adam for a two-dimensional test function

Running the example applies the Adam optimization algorithm to our test problem and reports the performance of the search for each iteration of the algorithm.

⚠ **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 a near-optimal solution was found after perhaps 53 iterations of the search, with input values near 0.0 and 0.0, evaluating to 0.0.

```
...
>50 f([-0.00056912 -0.00321961]) = 0.00001
>51 f([-0.00052452 -0.00286514]) = 0.00001
>52 f([-0.00043908 -0.00251304]) = 0.00001
>53 f([-0.0003283  -0.00217044]) = 0.00000
>54 f([-0.00020731 -0.00184302]) = 0.00000
>55 f([-8.95352320e-05 -1.53514076e-03]) = 0.00000
>56 f([ 1.43050285e-05 -1.25002847e-03]) = 0.00000
>57 f([ 9.67123406e-05 -9.89850279e-04]) = 0.00000
>58 f([ 0.00015359 -0.00075587]) = 0.00000
>59 f([ 0.00018407 -0.00054858]) = 0.00000
Done!
f([ 0.00018407 -0.00054858]) = 0.000000
```

Output 26.1: Result from Program 26.18

26.4.3 Visualization of Adam

We can plot the progress of the Adam search on a contour plot of the domain. This can provide an intuition for the progress of the search over the iterations of the algorithm. We must update the `adam()` function to maintain a list of all solutions found during the search, then return this list at the end of the search. The updated version of the function with these changes is listed below.

```
def adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2, eps=1e-8):
    solutions = list()
    # generate an initial point
    x = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    score = objective(x[0], x[1])
    # initialize first and second moments
    m = [0.0 for _ in range(bounds.shape[0])]
```

```python
        v = [0.0 for _ in range(bounds.shape[0])]
        # run the gradient descent updates
        for t in range(n_iter):
            # calculate gradient g(t)
            g = derivative(x[0], x[1])
            # build a solution one variable at a time
            for i in range(bounds.shape[0]):
                # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
                m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
                # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
                v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2
                # mhat(t) = m(t) / (1 - beta1(t))
                mhat = m[i] / (1.0 - beta1**(t+1))
                # vhat(t) = v(t) / (1 - beta2(t))
                vhat = v[i] / (1.0 - beta2**(t+1))
                # x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + ep)
                x[i] = x[i] - alpha * mhat / (sqrt(vhat) + eps)
            # evaluate candidate point
            score = objective(x[0], x[1])
            # keep track of solutions
            solutions.append(x.copy())
            # report progress
            print('>%d f(%s) = %.5f' % (t, x, score))
    return solutions
```

Program 26.19: Gradient descent algorithm with Adam

We can then execute the search as before, and this time retrieve the list of solutions instead of the best final solution.

```python
...
# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 60
# steps size
alpha = 0.02
# factor for average gradient
beta1 = 0.8
# factor for average squared gradient
beta2 = 0.999
# perform the gradient descent search with adam
solutions = adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2)
```

Program 26.20: Perform the gradient descent search with Adam

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

```python
...
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
```

```
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
```

Program 26.21: Create contour plot of the objective function

Finally, we can plot each solution found during the search as a white dot connected by a line.

```
...
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
```

Program 26.22: Plot the sample as black circles

Tying this all together, the complete example of performing the Adam optimization on the test problem and plotting the results on a contour plot is listed below.

```
from math import sqrt
from numpy import asarray
from numpy import arange
from numpy.random import rand
from numpy.random import seed
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# objective function
def objective(x, y):
    return x**2.0 + y**2.0

# derivative of objective function
def derivative(x, y):
    return asarray([x * 2.0, y * 2.0])

# gradient descent algorithm with adam
def adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2, eps=1e-8):
    solutions = list()
    # generate an initial point
    x = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    score = objective(x[0], x[1])
    # initialize first and second moments
    m = [0.0 for _ in range(bounds.shape[0])]
    v = [0.0 for _ in range(bounds.shape[0])]
    # run the gradient descent updates
    for t in range(n_iter):
        # calculate gradient g(t)
        g = derivative(x[0], x[1])
        # build a solution one variable at a time
        for i in range(bounds.shape[0]):
            # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
            m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
```

```
                # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
                v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2
                # mhat(t) = m(t) / (1 - beta1(t))
                mhat = m[i] / (1.0 - beta1**(t+1))
                # vhat(t) = v(t) / (1 - beta2(t))
                vhat = v[i] / (1.0 - beta2**(t+1))
                # x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + ep)
                x[i] = x[i] - alpha * mhat / (sqrt(vhat) + eps)
            # evaluate candidate point
            score = objective(x[0], x[1])
            # keep track of solutions
            solutions.append(x.copy())
            # report progress
            print('>%d f(%s) = %.5f' % (t, x, score))
        return solutions

# seed the pseudo random number generator
seed(1)
# define range for input
bounds = asarray([[-1.0, 1.0], [-1.0, 1.0]])
# define the total iterations
n_iter = 60
# steps size
alpha = 0.02
# factor for average gradient
beta1 = 0.8
# factor for average squared gradient
beta2 = 0.999
# perform the gradient descent search with adam
solutions = adam(objective, derivative, bounds, n_iter, alpha, beta1, beta2)
# sample input range uniformly at 0.1 increments
xaxis = arange(bounds[0,0], bounds[0,1], 0.1)
yaxis = arange(bounds[1,0], bounds[1,1], 0.1)
# create a mesh from the axis
x, y = meshgrid(xaxis, yaxis)
# compute targets
results = objective(x, y)
# create a filled contour plot with 50 levels and jet color scheme
pyplot.contourf(x, y, results, levels=50, cmap='jet')
# plot the sample as black circles
solutions = asarray(solutions)
pyplot.plot(solutions[:, 0], solutions[:, 1], '.-', color='w')
# show the plot
pyplot.show()
```

Program 26.23: Example of plotting the Adam search on a contour plot of the test function

Running the example performs the search as before, except in this case, a contour plot of the objective function is created. In this case, we can see that a white dot is shown for each solution found during the search, starting above the optima and progressively getting closer to the optima at the center of the plot.
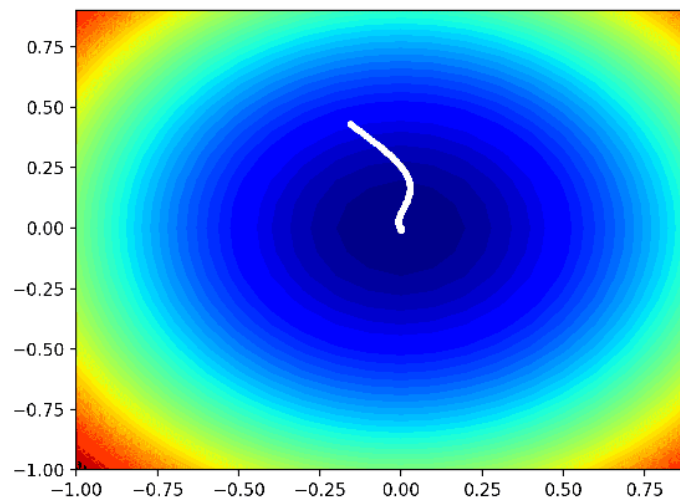
Figure 26.3: Contour Plot of the Test Objective Function With Adam Search Results Shown

26.5 Further Reading

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

Papers

Diederik P. Kingma and Jimmy Lei Ba. "Adam: A Method for Stochastic Optimization". In: *Proc. 3rd Int. Conf. Learning Representations (ICLR)*. 2015.
https://arxiv.org/abs/1412.6980

Books

Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.
https://amzn.to/3je8O1J
Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://amzn.to/3qSk3C2

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
Stochastic gradient descent. Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent
An overview of gradient descent optimization algorithms. 2016.
https://ruder.io/optimizing-gradient-descent/index.html

26.6   Summary

In this tutorial, you discovered how to develop gradient descent with Adam optimization algorithm from scratch. Specifically, you learned:

▷ Gradient descent is an optimization algorithm that uses the gradient of the objective function to navigate the search space.

▷ Gradient descent can be updated to use an automatically adaptive step size for each input variable using a decaying average of partial derivatives, called Adam.

▷ How to implement the Adam optimization algorithm from scratch and apply it to an objective function and evaluate the results.

Here we finish with all gradient descent algorithms. Next we will see how the different optimization algorithms can be used.

VI Projects

Use Optimization Algorithms to Manually Fit Regression Models

27

Regression models are fit on training data using linear regression and local search optimization algorithms. Models like linear regression and logistic regression are trained by least squares optimization, and this is the most efficient approach to finding coefficients that minimize error for these models. Nevertheless, it is possible to use alternate *optimization algorithms to fit a regression model* to a training dataset. This can be a useful exercise to learn more about how regression functions and the central nature of optimization in applied machine learning. It may also be required for regression with data that does not meet the requirements of a least squares optimization procedure.

In this tutorial, you will discover how to manually optimize the coefficients of regression models. After completing this tutorial, you will know:

▷ How to develop the inference models for regression from scratch.

▷ How to optimize the coefficients of a linear regression model for predicting numeric values.

▷ How to optimize the coefficients of a logistic regression model using stochastic hill climbing.

Let's get started.

27.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Optimize Regression Models
2. Optimize a Linear Regression Model
3. Optimize a Logistic Regression Model

27.2   Optimize Regression Models

Regression models, like linear regression and logistic regression, are well-understood algorithms from the field of statistics. Both algorithms are linear, meaning the output of the model is a

weighted sum of the inputs. Linear regression[1] is designed for "*regression*" problems that require a number to be predicted, and logistic regression[2] is designed for "*classification*" problems that require a class label to be predicted. These regression models involve the use of an optimization algorithm to find a set of coefficients for each input to the model that minimizes the prediction error. Because the models are linear and well understood, efficient optimization algorithms can be used.

In the case of linear regression, the coefficients can be found by least squares optimization, which can be solved using linear algebra. In the case of logistic regression, a local search optimization algorithm is commonly used. It is possible to use any arbitrary optimization algorithm to train linear and logistic regression models. That is, we can define a regression model and use a given optimization algorithm to find a set of coefficients for the model that result in a minimum of prediction error or a maximum of classification accuracy.

Using alternate optimization algorithms is expected to be less efficient on average than using the recommended optimization. Nevertheless, it may be more efficient in some specific cases, such as if the input data does not meet the expectations of the model like a Gaussian distribution and is uncorrelated with outer inputs. It can also be an interesting exercise to demonstrate the central nature of optimization in training machine learning algorithms, and specifically regression models.

Next, let's explore how to train a linear regression model using stochastic hill climbing.

27.3   Optimize a Linear Regression Model

The linear regression model might be the simplest predictive model that learns from data. The model has one coefficient for each input and the predicted output is simply the weights of some inputs and coefficients. In this section, we will optimize the coefficients of a linear regression model. First, let's define a synthetic regression problem that we can use as the focus of optimizing the model. We can use the `make_regression()` function[3] to define a regression problem with 1,000 rows and 10 input variables. The example below creates the dataset and summarizes the shape of the data.

```python
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=2, noise=0.2, ⤸
    random_state=1)
# summarize the shape of the dataset
print(X.shape, y.shape)
```

Program 27.1: Define a regression dataset

Running the example prints the shape of the created dataset, confirming our expectations.

```
(1000, 10) (1000,)
```

Output 27.1: Result from Program 27.1

---

[1]https://en.wikipedia.org/wiki/Linear_regression
[2]https://en.wikipedia.org/wiki/Logistic_regression
[3]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html

Next, we need to define a linear regression model. Before we optimize the model coefficients, we must develop the model and our confidence in how it works. Let's start by developing a function that calculates the activation of the model for a given input row of data from the dataset. This function will take the row of data and the coefficients for the model and calculate the weighted sum of the input with the addition of an extra y-intercept (also called the offset or bias) coefficient. The `predict_row()` function below implements this. We are using simple Python lists and imperative programming style instead of NumPy arrays or list comprehension intentionally to make the code more readable for Python beginners. Feel free to optimize it.

```python
def predict_row(row, coefficients):
    # add the bias, the last coefficient
    result = coefficients[-1]
    # add the weighted input
    for i in range(len(row)):
        result += coefficients[i] * row[i]
    return result
```

Program 27.2: Linear regression

Next, we can call the `predict_row()` function for each row in a given dataset. The `predict_dataset()` function below implements this. Again, we are intentionally using a simple imperative coding style for readability instead of list comprehension.

```python
def predict_dataset(X, coefficients):
    yhats = list()
    for row in X:
        # make a prediction
        yhat = predict_row(row, coefficients)
        # store the prediction
        yhats.append(yhat)
    return yhats
```

Program 27.3: Use model coefficients to generate predictions for a dataset of rows

Finally, we can use the model to make predictions on our synthetic dataset to confirm it is all working correctly. We can generate a random set of model coefficients using the `rand()` function[4]. Recall that we need one coefficient for each input (ten inputs in this dataset) plus an extra weight for the y-intercept coefficient.

```python
...
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=2, noise=0.2, ⤸
    random_state=1)
# determine the number of coefficients
n_coeff = X.shape[1] + 1
# generate random coefficients
coefficients = rand(n_coeff)
```

Program 27.4: Define dataset and generate random model coefficients

We can then use these coefficients with the dataset to make predictions.

---

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

```
...
yhat = predict_dataset(X, coefficients)
```

Program 27.5: Generate predictions for dataset

We can evaluate the mean squared error of these predictions.

```
...
score = mean_squared_error(y, yhat)
print('MSE: %f' % score)
```

Program 27.6: Calculate model prediction error

That's it. We can tie all of this together and demonstrate our linear regression model for regression predictive modeling. The complete example is listed below.

```
from numpy.random import rand
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error

# linear regression
def predict_row(row, coefficients):
    # add the bias, the last coefficient
    result = coefficients[-1]
    # add the weighted input
    for i in range(len(row)):
        result += coefficients[i] * row[i]
    return result

# use model coefficients to generate predictions for a dataset of rows
def predict_dataset(X, coefficients):
    yhats = list()
    for row in X:
        # make a prediction
        yhat = predict_row(row, coefficients)
        # store the prediction
        yhats.append(yhat)
    return yhats

# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=2, noise=0.2,
    random_state=1)
# determine the number of coefficients
n_coeff = X.shape[1] + 1
# generate random coefficients
coefficients = rand(n_coeff)
# generate predictions for dataset
yhat = predict_dataset(X, coefficients)
# calculate model prediction error
score = mean_squared_error(y, yhat)
print('MSE: %f' % score)
```

Program 27.7: Linear regression model

Running the example generates a prediction for each example in the training dataset, then prints the mean squared error for the predictions.

> ⚠️ **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 would expect a large error given a set of random weights, and that is what we see in this case, with an error value of about 7,307 units.

```
MSE: 7307.756740
```
Output 27.2: Result from Program 27.7

We can now optimize the coefficients of the dataset to achieve low error on this dataset. First, we need to split the dataset into train and test sets. It is important to hold back some data not used in optimizing the model so that we can prepare a reasonable estimate of the performance of the model when used to make predictions on new data. We will use 67 percent of the data for training and the remaining 33 percent as a test set for evaluating the performance of the model.

```
...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
```
Program 27.8: Split data into train test sets

Next, we can develop a stochastic hill climbing algorithm. The optimization algorithm requires an objective function to optimize. It must take a set of coefficients and return a score that is to be minimized or maximized corresponding to a better model. In this case, we will evaluate the mean squared error of the model with a given set of coefficients and return the error score, which must be minimized. The `objective()` function below implements this, given the dataset and a set of coefficients, and returns the error of the model.

```
def objective(X, y, coefficients):
    # generate predictions for dataset
    yhat = predict_dataset(X, coefficients)
    # calculate accuracy
    score = mean_squared_error(y, yhat)
    return score
```
Program 27.9: Objective function

Next, we can define the stochastic hill climbing algorithm. The algorithm will require an initial solution (e.g. random coefficients) and will iteratively keep making small changes to the solution and checking if it results in a better performing model. The amount of change made to the current solution is controlled by a `step_size` hyperparameter. This process will continue for a fixed number of iterations, also provided as a hyperparameter. The `hillclimbing()` function below implements this, taking the dataset, objective function, initial solution, and hyperparameters as arguments and returns the best set of coefficients found and the estimated performance.

```
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = solution + randn(len(solution)) * step_size
        # evaluate candidate point
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %.5f' % (i, solution_eval))
    return [solution, solution_eval]
```

Program 27.10: Hill climbing local search algorithm

We can then call this function, passing in an initial set of coefficients as the initial solution and the training dataset as the dataset to optimize the model against.

```
...
# define the total iterations
n_iter = 2000
# define the maximum step size
step_size = 0.15
# determine the number of coefficients
n_coef = X.shape[1] + 1
# define the initial solution
solution = rand(n_coef)
# perform the hill climbing search
coefficients, score = hillclimbing(X_train, y_train, objective, solution, n_iter, ⤸
    step_size)
print('Done!')
print('Coefficients: %s' % coefficients)
print('Train MSE: %f' % (score))
```

Program 27.11: Perform hill climbing search

Finally, we can evaluate the best model on the test dataset and report the performance.

```
...
# generate predictions for the test dataset
yhat = predict_dataset(X_test, coefficients)
# calculate accuracy
score = mean_squared_error(y_test, yhat)
print('Test MSE: %f' % (score))
```

Program 27.12: Evaluate the best model on test dataset

Tying this together, the complete example of optimizing the coefficients of a linear regression model on the synthetic regression dataset is listed below.

```python
from numpy.random import randn
from numpy.random import rand
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# linear regression
def predict_row(row, coefficients):
    # add the bias, the last coefficient
    result = coefficients[-1]
    # add the weighted input
    for i in range(len(row)):
        result += coefficients[i] * row[i]
    return result

# use model coefficients to generate predictions for a dataset of rows
def predict_dataset(X, coefficients):
    yhats = list()
    for row in X:
        # make a prediction
        yhat = predict_row(row, coefficients)
        # store the prediction
        yhats.append(yhat)
    return yhats

# objective function
def objective(X, y, coefficients):
    # generate predictions for dataset
    yhat = predict_dataset(X, coefficients)
    # calculate accuracy
    score = mean_squared_error(y, yhat)
    return score

# hill climbing local search algorithm
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = solution + randn(len(solution)) * step_size
        # evaluate candidate point
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %.5f' % (i, solution_eval))
    return [solution, solution_eval]

# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=2, noise=0.2,
    random_state=1)
```

```
# split into train test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
# define the total iterations
n_iter = 2000
# define the maximum step size
step_size = 0.15
# determine the number of coefficients
n_coef = X.shape[1] + 1
# define the initial solution
solution = rand(n_coef)
# perform the hill climbing search
coefficients, score = hillclimbing(X_train, y_train, objective, solution, n_iter, ⤸
    step_size)
print('Done!')
print('Coefficients: %s' % coefficients)
print('Train MSE: %f' % (score))
# generate predictions for the test dataset
yhat = predict_dataset(X_test, coefficients)
# calculate accuracy
score = mean_squared_error(y_test, yhat)
print('Test MSE: %f' % (score))
```

Program 27.13: Optimize linear regression coefficients for regression dataset

Running the example will report the iteration number and mean squared error each time there is an improvement made to the model. At the end of the search, the performance of the best set of coefficients on the training dataset is reported and the performance of the same model on the test dataset is calculated and reported.

⚠ **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 optimization algorithm found a set of coefficients that achieved an error of about 0.08 on both the train and test datasets. The fact that the algorithm found a model with very similar performance on train and test datasets is a good sign, showing that the model did not overfit (over-optimize) to the training dataset. This means the model generalizes well to new data.

```
...
>1546 0.35426
>1567 0.32863
>1572 0.32322
>1619 0.24890
>1665 0.24800
>1691 0.24162
>1715 0.15893
>1809 0.15337
>1892 0.14656
>1956 0.08042
Done!
```

```
Coefficients: [ 1.30559829e-02 -2.58299382e-04  3.33118191e+00  3.20418534e-02
  1.36497902e-01  8.65445367e+01  2.78356715e-02 -8.50901499e-02
  8.90078243e-02  6.15779867e-02 -3.85657793e-02]
Train MSE: 0.080415
Test MSE: 0.080779
```

Output 27.3: Result from Program 27.13

Now that we are familiar with how to manually optimize the coefficients of a linear regression model, let's look at how we can extend the example to optimize the coefficients of a logistic regression model for classification.

27.4   Optimize a Logistic Regression Model

A Logistic Regression model is an extension of linear regression for classification predictive modeling. Logistic regression is for binary classification tasks, meaning datasets that have two class labels, class=0 and class=1. The output first involves calculating the weighted sum of the inputs, then passing this weighted sum through a logistic function, also called a sigmoid function. The result is a Binomial probability between 0 and 1 for the example belonging to class=1.

In this section, we will build on what we learned in the previous section to optimize the coefficients of regression models for classification. We will develop the model and test it with random coefficients, then use stochastic hill climbing to optimize the model coefficients. First, let's define a synthetic binary classification problem that we can use as the focus of optimizing the model. We can use the `make_classification()` function[5] to define a binary classification problem with 1,000 rows and five input variables. The example below creates the dataset and summarizes the shape of the data.

```
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# summarize the shape of the dataset
print(X.shape, y.shape)
```

Program 27.14: Define a binary classification dataset

Running the example prints the shape of the created dataset, confirming our expectations.

```
(1000, 5) (1000,)
```

Output 27.4: Result from Program 27.14

Next, we need to define a logistic regression model. Let's start by updating the `predict_row()` function to pass the weighted sum of the input and coefficients through a logistic function. The logistic function is defined as:

$$\text{logistic} = \frac{1}{1 + \exp(-\text{result})}$$

---

[5]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

Where result is the weighted sum of the inputs and the coefficients and exp()Âăis e (Euler's number)[6] raised to the power of the provided value, implemented via the `exp()` function[7]. The updated `predict_row()` function is listed below.

```
def predict_row(row, coefficients):
    # add the bias, the last coefficient
    result = coefficients[-1]
    # add the weighted input
    for i in range(len(row)):
        result += coefficients[i] * row[i]
    # logistic function
    logistic = 1.0 / (1.0 + exp(-result))
    return logistic
```

Program 27.15: Logistic regression

That's about it in terms of changes for linear regression to logistic regression. As with linear regression, we can test the model with a set of random model coefficients.

```
...
# determine the number of coefficients
n_coeff = X.shape[1] + 1
# generate random coefficients
coefficients = rand(n_coeff)
# generate predictions for dataset
yhat = predict_dataset(X, coefficients)
```

Program 27.16: Test the model with random coefficients

The predictions made by the model are probabilities for an example belonging to class=1. We can round the prediction to be integer values 0 and 1 for the expected class labels.

```
...
yhat = [round(y) for y in yhat]
```

Program 27.17: Round predictions to labels

We can evaluate the classification accuracy of these predictions.

```
...
score = accuracy_score(y, yhat)
print('Accuracy: %f' % score)
```

Program 27.18: Calculate accuracy

That's it. We can tie all of this together and demonstrate our simple logistic regression model for binary classification. The complete example is listed below.

---

[6]https://en.wikipedia.org/wiki/E_(mathematical_constant)
[7]https://docs.python.org/3/library/math.html#math.exp

```python
from math import exp
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

# logistic regression
def predict_row(row, coefficients):
    # add the bias, the last coefficient
    result = coefficients[-1]
    # add the weighted input
    for i in range(len(row)):
        result += coefficients[i] * row[i]
    # logistic function
    logistic = 1.0 / (1.0 + exp(-result))
    return logistic

# use model coefficients to generate predictions for a dataset of rows
def predict_dataset(X, coefficients):
    yhats = list()
    for row in X:
        # make a prediction
        yhat = predict_row(row, coefficients)
        # store the prediction
        yhats.append(yhat)
    return yhats

# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# determine the number of coefficients
n_coeff = X.shape[1] + 1
# generate random coefficients
coefficients = rand(n_coeff)
# generate predictions for dataset
yhat = predict_dataset(X, coefficients)
# round predictions to labels
yhat = [round(y) for y in yhat]
# calculate accuracy
score = accuracy_score(y, yhat)
print('Accuracy: %f' % score)
```

Program 27.19: Logistic regression function for binary classification

Running the example generates a prediction for each example in the training dataset then prints the classification accuracy for the predictions.

⚠️ **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 would expect about 50 percent accuracy given a set of random weights and a dataset with an equal number of examples in each class, and that is approximately what we see in this case.

```
Accuracy: 0.540000
```

Output 27.5: Result from Program 27.19

We can now optimize the weights of the dataset to achieve good accuracy on this dataset. The stochastic hill climbing algorithm used for linear regression can be used again for logistic regression. The important difference is an update to the `objective()` function to round the predictions and evaluate the model using classification accuracy instead of mean squared error.

```python
def objective(X, y, coefficients):
    # generate predictions for dataset
    yhat = predict_dataset(X, coefficients)
    # round predictions to labels
    yhat = [round(y) for y in yhat]
    # calculate accuracy
    score = accuracy_score(y, yhat)
    return score
```

Program 27.20: Objective function

The `hillclimbing()` function also must be updated to maximize the score of solutions instead of minimizing in the case of linear regression.

```python
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = solution + randn(len(solution)) * step_size
        # evaluate candidate point
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %.5f' % (i, solution_eval))
    return [solution, solution_eval]
```

Program 27.21: Hill climbing local search algorithm

Finally, the coefficients found by the search can be evaluated using classification accuracy at the end of the run.

```python
...
# generate predictions for the test dataset
yhat = predict_dataset(X_test, coefficients)
# round predictions to labels
yhat = [round(y) for y in yhat]
# calculate accuracy
score = accuracy_score(y_test, yhat)
print('Test Accuracy: %f' % (score))
```

Program 27.22: Evaluate the model with the coefficients found by the search

Tying this all together, the complete example of using stochastic hill climbing to maximize classification accuracy of a logistic regression model is listed below.

```python
from math import exp
from numpy.random import randn
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# logistic regression
def predict_row(row, coefficients):
    # add the bias, the last coefficient
    result = coefficients[-1]
    # add the weighted input
    for i in range(len(row)):
        result += coefficients[i] * row[i]
    # logistic function
    logistic = 1.0 / (1.0 + exp(-result))
    return logistic

# use model coefficients to generate predictions for a dataset of rows
def predict_dataset(X, coefficients):
    yhats = list()
    for row in X:
        # make a prediction
        yhat = predict_row(row, coefficients)
        # store the prediction
        yhats.append(yhat)
    return yhats

# objective function
def objective(X, y, coefficients):
    # generate predictions for dataset
    yhat = predict_dataset(X, coefficients)
    # round predictions to labels
    yhat = [round(y) for y in yhat]
    # calculate accuracy
    score = accuracy_score(y, yhat)
    return score

# hill climbing local search algorithm
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = solution + randn(len(solution)) * step_size
        # evaluate candidate point
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval >= solution_eval:
```

```
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %.5f' % (i, solution_eval))
    return [solution, solution_eval]

# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# split into train test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
# define the total iterations
n_iter = 2000
# define the maximum step size
step_size = 0.1
# determine the number of coefficients
n_coef = X.shape[1] + 1
# define the initial solution
solution = rand(n_coef)
# perform the hill climbing search
coefficients, score = hillclimbing(X_train, y_train, objective, solution, n_iter,
    step_size)
print('Done!')
print('Coefficients: %s' % coefficients)
print('Train Accuracy: %f' % (score))
# generate predictions for the test dataset
yhat = predict_dataset(X_test, coefficients)
# round predictions to labels
yhat = [round(y) for y in yhat]
# calculate accuracy
score = accuracy_score(y_test, yhat)
print('Test Accuracy: %f' % (score))
```

Program 27.23: Optimize logistic regression model with a stochastic hill climber

Running the example will report the iteration number and classification accuracy each time there is an improvement made to the model. At the end of the search, the performance of the best set of coefficients on the training dataset is reported and the performance of the same model on the test dataset is calculated and reported.

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 optimization algorithm found a set of weights that achieved about 87.3 percent accuracy on the training dataset and about 83.9 percent accuracy on the test dataset.

```
...
>200 0.85672
>225 0.85672
```

```
>230 0.85672
>245 0.86418
>281 0.86418
>285 0.86716
>294 0.86716
>306 0.86716
>316 0.86716
>317 0.86716
>320 0.86866
>348 0.86866
>362 0.87313
>784 0.87313
>1649 0.87313
Done!
Coefficients: [-0.04652756  0.23243427  2.58587637 -0.45528253 -0.4954355
  -0.42658053]
Train Accuracy: 0.873134
Test Accuracy: 0.839394
```

Output 27.6: Result from Program 27.23

27.5   Further Reading

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

APIs

sklearn.datasets.make_regression *API*.
   https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_
   regression.html
sklearn.datasets.make_classification *API*.
   https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_
   classification.html
sklearn.metrics.mean_squared_error *API*.
   https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_
   squared_error.html
numpy.random.rand *API*.
   https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.
   html

Articles

Linear regression. Wikipedia.
   https://en.wikipedia.org/wiki/Linear_regression
Logistic regression. Wikipedia.
   https://en.wikipedia.org/wiki/Logistic_regression

27.6 Summary

In this tutorial, you discovered how to manually optimize the coefficients of regression models. Specifically, you learned:

$\triangleright$ How to develop the inference models for regression from scratch.

$\triangleright$ How to optimize the coefficients of a linear regression model for predicting numeric values.

$\triangleright$ How to optimize the coefficients of a logistic regression model using stochastic hill climbing.

Next, we will see another example, using stochastic hill climbing to optimize a neural network.

Optimize Neural Network Models

<div align="right">

28

</div>

Deep learning neural network models are fit on training data using the stochastic gradient descent optimization algorithm. Updates to the weights of the model are made, using the backpropagation of error algorithm. The combination of the optimization and weight update algorithm was carefully chosen and is the most efficient approach known to fit neural networks. Nevertheless, it is possible to use alternate optimization algorithms to fit a neural network model to a training dataset. This can be a useful exercise to learn more about how neural networks function and the central nature of optimization in applied machine learning. It may also be required for neural networks with unconventional model architectures and non-differentiable transfer functions.

In this tutorial, you will discover how to manually optimize the weights of neural network models. After completing this tutorial, you will know:

▷ How to develop the forward inference pass for neural network models from scratch.

▷ How to optimize the weights of a Perceptron model for binary classification.

▷ How to optimize the weights of a Multilayer Perceptron model using stochastic hill climbing.

Let's get started.

28.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Optimize Neural Networks
2. Optimize a Perceptron Model
3. Optimize a Multilayer Perceptron

28.2 Optimize Neural Networks

Deep learning or neural networks are a flexible type of machine learning. They are models composed of nodes and layers inspired by the structure and function of the brain. A neural network model works by propagating a given input vector through one or more layers to produce

a numeric output that can be interpreted for classification or regression predictive modeling. Models are trained by repeatedly exposing the model to examples of input and output and adjusting the weights to minimize the error of the model's output compared to the expected output. This is called the stochastic gradient descent optimization algorithm. The weights of the model are adjusted using a specific rule from calculus that assigns error proportionally to each weight in the network. This is called the backpropagation algorithm.

The stochastic gradient descent optimization algorithm with weight updates made using backpropagation is the best way to train neural network models. However, it is not the only way to train a neural network. It is possible to use any arbitrary optimization algorithm to train a neural network model. That is, we can define a neural network model architecture and use a given optimization algorithm to find a set of weights for the model that results in a minimum of prediction error or a maximum of classification accuracy. Using alternate optimization algorithms is expected to be less efficient on average than using stochastic gradient descent with backpropagation. Nevertheless, it may be more efficient in some specific cases, such as non-standard network architectures or non-differential transfer functions.

It can also be an interesting exercise to demonstrate the central nature of optimization in training machine learning algorithms, and specifically neural networks. Next, let's explore how to train a simple one-node neural network called a Perceptron model using stochastic hill climbing.

28.3   Optimize a Perceptron Model

The Perceptron algorithm is the simplest type of artificial neural network. It is a model of a single neuron that can be used for two-class classification problems and provides the foundation for later developing much larger networks. In this section, we will optimize the weights of a Perceptron neural network model. First, let's define a synthetic binary classification problem that we can use as the focus of optimizing the model. We can use the `make_classification()` function[1] to define a binary classification problem with 1,000 rows and five input variables. The example below creates the dataset and summarizes the shape of the data.

```python
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# summarize the shape of the dataset
print(X.shape, y.shape)
```

Program 28.1: Define a binary classification dataset

Running the example prints the shape of the created dataset, confirming our expectations.

```
(1000, 5) (1000,)
```

Output 28.1: Result from Program 28.1

---

[1]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

Next, we need to define a Perceptron model. The Perceptron model has a single node that has one input weight for each column in the dataset. Each input is multiplied by its corresponding weight to give a weighted sum and a bias weight is then added, like an intercept coefficient in a regression model. This weighted sum is called the activation. Finally, the activation is interpreted and used to predict the class label, 1 for a positive activation and 0 for a negative activation. Before we optimize the model weights, we must develop the model and our confidence in how it works.

Let's start by defining a function for interpreting the activation of the model. This is called the activation function, or the transfer function; the latter name is more traditional and is my preference. The `transfer()` function below takes the activation of the model and returns a class label, class=1 for a positive or zero activation and class=0 for a negative activation. This is called a step transfer function.

```python
def transfer(activation):
    if activation >= 0.0:
        return 1
    return 0
```

Program 28.2: Transfer function

Next, we can develop a function that calculates the activation of the model for a given input row of data from the dataset. This function will take the row of data and the weights for the model and calculate the weighted sum of the input with the addition of the bias weight. The `activate()` function below implements this.

> **Note:** We are using simple Python lists and imperative programming style instead of NumPy arrays or list compressions intentionally to make the code more readable for Python beginners.

```python
def activate(row, weights):
    # add the bias, the last weight
    activation = weights[-1]
    # add the weighted input
    for i in range(len(row)):
        activation += weights[i] * row[i]
    return activation
```

Program 28.3: Activation function

Next, we can use the `activate()` and `transfer()` functions together to generate a prediction for a given row of data. The `predict_row()` function below implements this.

```python
def predict_row(row, weights):
    # activate for input
    activation = activate(row, weights)
    # transfer for activation
    return transfer(activation)
```

Program 28.4: Use model weights to predict 0 or 1 for a given row of data

Next, we can call the `predict_row()` function for each row in a given dataset. The `predict_dataset()` function below implements this. Again, we are intentionally using simple imperative coding style for readability instead of list comprehension.

```python
def predict_dataset(X, weights):
    yhats = list()
    for row in X:
        yhat = predict_row(row, weights)
        yhats.append(yhat)
    return yhats
```
Program 28.5: Use model weights to generate predictions for a dataset of rows

Finally, we can use the model to make predictions on our synthetic dataset to confirm it is all working correctly. We can generate a random set of model weights using the `rand()` function[2]. Recall that we need one weight for each input (five inputs in this dataset) plus an extra weight for the bias weight.

```python
...
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# determine the number of weights
n_weights = X.shape[1] + 1
# generate random weights
weights = rand(n_weights)
```
Program 28.6: Define dataset and generate random weights

We can then use these weights with the dataset to make predictions.

```python
...
yhat = predict_dataset(X, weights)
```
Program 28.7: Generate predictions for dataset

We can evaluate the classification accuracy of these predictions.

```python
...
score = accuracy_score(y, yhat)
print(score)
```
Program 28.8: Calculate accuracy

That's it. We can tie all of this together and demonstrate our simple Perceptron model for classification. The complete example is listed below.

```python
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
```

---

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

```
# transfer function
def transfer(activation):
    if activation >= 0.0:
        return 1
    return 0

# activation function
def activate(row, weights):
    # add the bias, the last weight
    activation = weights[-1]
    # add the weighted input
    for i in range(len(row)):
        activation += weights[i] * row[i]
    return activation

# use model weights to predict 0 or 1 for a given row of data
def predict_row(row, weights):
    # activate for input
    activation = activate(row, weights)
    # transfer for activation
    return transfer(activation)

# use model weights to generate predictions for a dataset of rows
def predict_dataset(X, weights):
    yhats = list()
    for row in X:
        yhat = predict_row(row, weights)
        yhats.append(yhat)
    return yhats

# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# determine the number of weights
n_weights = X.shape[1] + 1
# generate random weights
weights = rand(n_weights)
# generate predictions for dataset
yhat = predict_dataset(X, weights)
# calculate accuracy
score = accuracy_score(y, yhat)
print(score)
```

Program 28.9: Simple perceptron model for binary classification

Running the example generates a prediction for each example in the training dataset then prints the classification accuracy for the predictions.

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 would expect about 50 percent accuracy given a set of random weights and a dataset with an equal number of examples in each class, and that is approximately what we see in this case.

```
0.548
```

Output 28.2: Result from Program 28.9

We can now optimize the weights of the dataset to achieve good accuracy on this dataset. First, we need to split the dataset into train and test sets. It is important to hold back some data not used in optimizing the model so that we can prepare a reasonable estimate of the performance of the model when used to make predictions on new data. We will use 67 percent of the data for training and the remaining 33 percent as a test set for evaluating the performance of the model.

```
...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
```

Program 28.10: Split data into train test sets

Next, we can develop a stochastic hill climbing algorithm. The optimization algorithm requires an objective function to optimize. It must take a set of weights and return a score that is to be minimized or maximized corresponding to a better model. In this case, we will evaluate the accuracy of the model with a given set of weights and return the classification accuracy, which must be maximized. The `objective()` function below implements this, given the dataset and a set of weights, and returns the accuracy of the model

```
def objective(X, y, weights):
    # generate predictions for dataset
    yhat = predict_dataset(X, weights)
    # calculate accuracy
    score = accuracy_score(y, yhat)
    return score
```

Program 28.11: Objective function

Next, we can define the stochastic hill climbing algorithm. The algorithm will require an initial solution (e.g. random weights) and will iteratively keep making small changes to the solution and checking if it results in a better performing model. The amount of change made to the current solution is controlled by a `step_size` hyperparameter. This process will continue for a fixed number of iterations, also provided as a hyperparameter. The `hillclimbing()` function below implements this, taking the dataset, objective function, initial solution, and hyperparameters as arguments and returns the best set of weights found and the estimated performance.

```
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = solution + randn(len(solution)) * step_size
        # evaluate candidate point
```

```
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %.5f' % (i, solution_eval))
    return [solution, solution_eval]
```

Program 28.12: Hill climbing local search algorithm

We can then call this function, passing in a set of weights as the initial solution and the training dataset as the dataset to optimize the model against.

```
...
# define the total iterations
n_iter = 1000
# define the maximum step size
step_size = 0.05
# determine the number of weights
n_weights = X.shape[1] + 1
# define the initial solution
solution = rand(n_weights)
# perform the hill climbing search
weights, score = hillclimbing(X_train, y_train, objective, solution, n_iter,
    step_size)
print('Done!')
print('f(%s) = %f' % (weights, score))
```

Program 28.13: Perform hill climbing search

Finally, we can evaluate the best model on the test dataset and report the performance.

```
...
# generate predictions for the test dataset
yhat = predict_dataset(X_test, weights)
# calculate accuracy
score = accuracy_score(y_test, yhat)
print('Test Accuracy: %.5f' % (score * 100))
```

Program 28.14: Evaluate the best model on the test dataset

Tying this together, the complete example of optimizing the weights of a Perceptron model on the synthetic binary optimization dataset is listed below.

```
from numpy import asarray
from numpy.random import randn
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# transfer function
def transfer(activation):
```

```python
    if activation >= 0.0:
        return 1
    return 0

# activation function
def activate(row, weights):
    # add the bias, the last weight
    activation = weights[-1]
    # add the weighted input
    for i in range(len(row)):
        activation += weights[i] * row[i]
    return activation

# # use model weights to predict 0 or 1 for a given row of data
def predict_row(row, weights):
    # activate for input
    activation = activate(row, weights)
    # transfer for activation
    return transfer(activation)

# use model weights to generate predictions for a dataset of rows
def predict_dataset(X, weights):
    yhats = list()
    for row in X:
        yhat = predict_row(row, weights)
        yhats.append(yhat)
    return yhats

# objective function
def objective(X, y, weights):
    # generate predictions for dataset
    yhat = predict_dataset(X, weights)
    # calculate accuracy
    score = accuracy_score(y, yhat)
    return score

# hill climbing local search algorithm
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = solution + randn(len(solution)) * step_size
        # evaluate candidate point
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %.5f' % (i, solution_eval))
    return [solution, solution_eval]
```

```
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# split into train test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
# define the total iterations
n_iter = 1000
# define the maximum step size
step_size = 0.05
# determine the number of weights
n_weights = X.shape[1] + 1
# define the initial solution
solution = rand(n_weights)
# perform the hill climbing search
weights, score = hillclimbing(X_train, y_train, objective, solution, n_iter,
    step_size)
print('Done!')
print('f(%s) = %f' % (weights, score))
# generate predictions for the test dataset
yhat = predict_dataset(X_test, weights)
# calculate accuracy
score = accuracy_score(y_test, yhat)
print('Test Accuracy: %.5f' % (score * 100))
```

Program 28.15: Hill climbing to optimize weights of a perceptron model for classification

Running the example will report the iteration number and classification accuracy each time there is an improvement made to the model. At the end of the search, the performance of the best set of weights on the training dataset is reported and the performance of the same model on the test dataset is calculated and reported.

> **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 optimization algorithm found a set of weights that achieved about 88.5 percent accuracy on the training dataset and about 81.8 percent accuracy on the test dataset.

```
...
>111 0.88060
>119 0.88060
>126 0.88209
>134 0.88209
>205 0.88209
>262 0.88209
>280 0.88209
>293 0.88209
>297 0.88209
>336 0.88209
```

```
>373 0.88209
>437 0.88358
>463 0.88507
>630 0.88507
>701 0.88507
Done!
f([ 0.0097317 0.13818088 1.17634326 -0.04296336 0.00485813 -0.14767616]) = 0.885075
Test Accuracy: 81.81818
```

Output 28.3: Result from Program 28.15

Now that we are familiar with how to manually optimize the weights of a Perceptron model, let's look at how we can extend the example to optimize the weights of a Multilayer Perceptron (MLP) model.

28.4   Optimize a Multilayer Perceptron

A Multilayer Perceptron (MLP) model is a neural network with one or more layers, where each layer has one or more nodes. It is an extension of a Perceptron model and is perhaps the most widely used neural network (deep learning) model. In this section, we will build on what we learned in the previous section to optimize the weights of MLP models with an arbitrary number of layers and nodes per layer. First, we will develop the model and test it with random weights, then use stochastic hill climbing to optimize the model weights. When using MLPs for binary classification, it is common to use a sigmoid transfer function (also called the logistic function) instead of the step transfer function used in the Perceptron. This function outputs a real-value between 0 and 1 that represents a binomial probability distribution, i.e. the probability that an example belongs to class=1. The `transfer()` function below implements this.

```python
def transfer(activation):
    # sigmoid transfer function
    return 1.0 / (1.0 + exp(-activation))
```

Program 28.16: Transfer function

We can use the same `activate()` function from the previous section. Here, we will use it to calculate the activation for each node in a given layer. The `predict_row()` function must be replaced with a more elaborate version. The function takes a row of data and the network and returns the output of the network. We will define our network as a list of lists. Each layer will be a list of nodes and each node will be a list or array of weights. To calculate the prediction of the network, we simply enumerate the layers, then enumerate nodes, then calculate the activation and transfer output for each node. In this case, we will use the same transfer function for all nodes in the network, although this does not have to be the case. For networks with more than one layer, the output from the previous layer is used as input to each node in the next layer. The output from the final layer in the network is then returned. The `predict_row()` function below implements this.

```
def predict_row(row, network):
    inputs = row
    # enumerate the layers in the network from input to output
    for layer in network:
        new_inputs = list()
        # enumerate nodes in the layer
        for node in layer:
            # activate the node
            activation = activate(inputs, node)
            # transfer activation
            output = transfer(activation)
            # store output
            new_inputs.append(output)
        # output from this layer is input to the next layer
        inputs = new_inputs
    return inputs[0]
```
Program 28.17: Activation function for a network

That's about it. Finally, we need to define a network to use. For example, we can define an MLP with a single hidden layer with a single node as follows:

```
...
node = rand(n_inputs + 1)
layer = [node]
network = [layer]
```
Program 28.18: Create a one node network

This is practically a Perceptron, although with a sigmoid transfer function. Quite boring. Let's define an MLP with one hidden layer and one output layer. The first hidden layer will have 10 nodes, and each node will take the input pattern from the dataset (e.g. five inputs). The output layer will have a single node that takes inputs from the outputs of the first hidden layer and then outputs a prediction. We assume a bias term exists, hence the `+1` below.

```
...
n_hidden = 10
hidden1 = [rand(n_inputs + 1) for _ in range(n_hidden)]
output1 = [rand(n_hidden + 1)]
network = [hidden1, output1]
```
Program 28.19: One hidden layer and an output layer

We can then use the model to make predictions on the dataset.

```
...
yhat = predict_dataset(X, network)
```
Program 28.20: Generate predictions for dataset

Before we calculate the classification accuracy, we must round the predictions to class labels 0 and 1.

```
...
# round the predictions
yhat = [round(y) for y in yhat]
# calculate accuracy
score = accuracy_score(y, yhat)
print(score)
```

Program 28.21: Calculate classification accuracy

Tying this all together, the complete example of evaluating an MLP with random initial weights on our synthetic binary classification dataset is listed below.

```
from math import exp
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

# transfer function
def transfer(activation):
    # sigmoid transfer function
    return 1.0 / (1.0 + exp(-activation))

# activation function
def activate(row, weights):
    # add the bias, the last weight
    activation = weights[-1]
    # add the weighted input
    for i in range(len(row)):
        activation += weights[i] * row[i]
    return activation

# activation function for a network
def predict_row(row, network):
    inputs = row
    # enumerate the layers in the network from input to output
    for layer in network:
        new_inputs = list()
        # enumerate nodes in the layer
        for node in layer:
            # activate the node
            activation = activate(inputs, node)
            # transfer activation
            output = transfer(activation)
            # store output
            new_inputs.append(output)
        # output from this layer is input to the next layer
        inputs = new_inputs
    return inputs[0]

# use model weights to generate predictions for a dataset of rows
def predict_dataset(X, network):
    yhats = list()
    for row in X:
        yhat = predict_row(row, network)
```

```
        yhats.append(yhat)
    return yhats

# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,  ↵
    n_redundant=1, random_state=1)
# determine the number of inputs
n_inputs = X.shape[1]
# one hidden layer and an output layer, each perceptron has a bias term
n_hidden = 10
hidden1 = [rand(n_inputs + 1) for _ in range(n_hidden)]
output1 = [rand(n_hidden + 1)]
network = [hidden1, output1]
# generate predictions for dataset
yhat = predict_dataset(X, network)
# round the predictions
yhat = [round(y) for y in yhat]
# calculate accuracy
score = accuracy_score(y, yhat)
print(score)
```

Program 28.22: Develop an MLP model for classification

Running the example generates a prediction for each example in the training dataset, then prints the classification accuracy for the predictions.

> ⚠️ **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.

Again, we would expect about 50 percent accuracy given a set of random weights and a dataset with an equal number of examples in each class, and that is approximately what we see in this case.

```
0.499
```

Output 28.4: Result from Program 28.22

Next, we can apply the stochastic hill climbing algorithm to the dataset. It is very much the same as applying hill climbing to the Perceptron model, except in this case, a step requires a modification to all weights in the network. For this, we will develop a new function that creates a copy of the network and mutates each weight in the network while making the copy. The `step()` function below implements this.

```
def step(network, step_size):
    new_net = list()
    # enumerate layers in the network
    for layer in network:
        new_layer = list()
        # enumerate nodes in this layer
        for node in layer:
```

```
                # mutate the node
                new_node = node.copy() + randn(len(node)) * step_size
                # store node in layer
                new_layer.append(new_node)
            # store layer in network
            new_net.append(new_layer)
    return new_net
```
Program 28.23: Take a step in the search space

Modifying all weight in the network is aggressive. A less aggressive step in the search space might be to make a small change to a subset of the weights in the model, perhaps controlled by a hyperparameter. This is left as an extension. We can then call this new `step()` function from the `hillclimbing()` function.

```
def hillclimbing(X, y, objective, solution, n_iter, step_size):
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = step(solution, step_size)
        # evaluate candidate point
        candidte_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidte_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d %f' % (i, solution_eval))
    return [solution, solution_eval]
```
Program 28.24: Hill climbing local search algorithm

Tying this together, the complete example of applying stochastic hill climbing to optimize the weights of an MLP model for binary classification is listed below.

```
from math import exp
from numpy.random import randn
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# transfer function
def transfer(activation):
    # sigmoid transfer function
    return 1.0 / (1.0 + exp(-activation))

# activation function
def activate(row, weights):
    # add the bias, the last weight
    activation = weights[-1]
```

```
    # add the weighted input
    for i in range(len(row)):
        activation += weights[i] * row[i]
    return activation

# activation function for a network
def predict_row(row, network):
    inputs = row
    # enumerate the layers in the network from input to output
    for layer in network:
        new_inputs = list()
        # enumerate nodes in the layer
        for node in layer:
            # activate the node
            activation = activate(inputs, node)
            # transfer activation
            output = transfer(activation)
            # store output
            new_inputs.append(output)
        # output from this layer is input to the next layer
        inputs = new_inputs
    return inputs[0]

# use model weights to generate predictions for a dataset of rows
def predict_dataset(X, network):
    yhats = list()
    for row in X:
        yhat = predict_row(row, network)
        yhats.append(yhat)
    return yhats

# objective function
def objective(X, y, network):
    # generate predictions for dataset
    yhat = predict_dataset(X, network)
    # round the predictions
    yhat = [round(y) for y in yhat]
    # calculate accuracy
    score = accuracy_score(y, yhat)
    return score

# take a step in the search space
def step(network, step_size):
    new_net = list()
    # enumerate layers in the network
    for layer in network:
        new_layer = list()
        # enumerate nodes in this layer
        for node in layer:
            # mutate the node
            new_node = node.copy() + randn(len(node)) * step_size
            # store node in layer
            new_layer.append(new_node)
        # store layer in network
```

```
            new_net.append(new_layer)
        return new_net

    # hill climbing local search algorithm
    def hillclimbing(X, y, objective, solution, n_iter, step_size):
        # evaluate the initial point
        solution_eval = objective(X, y, solution)
        # run the hill climb
        for i in range(n_iter):
            # take a step
            candidate = step(solution, step_size)
            # evaluate candidate point
            candidte_eval = objective(X, y, candidate)
            # check if we should keep the new point
            if candidte_eval >= solution_eval:
                # store the new point
                solution, solution_eval = candidate, candidte_eval
                # report progress
                print('>%d %f' % (i, solution_eval))
        return [solution, solution_eval]

    # define dataset
    X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
        n_redundant=1, random_state=1)
    # split into train test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
    # define the total iterations
    n_iter = 1000
    # define the maximum step size
    step_size = 0.1
    # determine the number of inputs
    n_inputs = X.shape[1]
    # one hidden layer and an output layer
    n_hidden = 10
    hidden1 = [rand(n_inputs + 1) for _ in range(n_hidden)]
    output1 = [rand(n_hidden + 1)]
    network = [hidden1, output1]
    # perform the hill climbing search
    network, score = hillclimbing(X_train, y_train, objective, network, n_iter,
        step_size)
    print('Done!')
    print('Best: %f' % (score))
    # generate predictions for the test dataset
    yhat = predict_dataset(X_test, network)
    # round the predictions
    yhat = [round(y) for y in yhat]
    # calculate accuracy
    score = accuracy_score(y_test, yhat)
    print('Test Accuracy: %.5f' % (score * 100))
```

Program 28.25: Stochastic hill climbing to optimize a multilayer perceptron for classification

Running the example will report the iteration number and classification accuracy each time there is an improvement made to the model. At the end of the search, the performance of the

best set of weights on the training dataset is reported and the performance of the same model on the test dataset is calculated and reported.

> **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 optimization algorithm found a set of weights that achieved about 87.3 percent accuracy on the training dataset and about 85.1 percent accuracy on the test dataset.

```
...
>55 0.755224
>56 0.765672
>59 0.794030
>66 0.805970
>77 0.835821
>120 0.838806
>165 0.840299
>188 0.841791
>218 0.846269
>232 0.852239
>237 0.852239
>239 0.855224
>292 0.867164
>368 0.868657
>823 0.868657
>852 0.871642
>889 0.871642
>892 0.871642
>992 0.873134
Done!
Best: 0.873134
Test Accuracy: 85.15152
```
Output 28.5: Result from Program 28.25

28.5   Further Reading

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

APIs

`sklearn.datasets.make_classification` *API*.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

`sklearn.metrics.accuracy_score` *API*.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

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

28.6 Summary

In this tutorial, you discovered how to manually optimize the weights of neural network models. Specifically, you learned:

▷ How to develop the forward inference pass for neural network models from scratch.

▷ How to optimize the weights of a Perceptron model for binary classification.

▷ How to optimize the weights of a Multilayer Perceptron model using stochastic hill climbing.

Next, we will try another project to use optimization algorithm for feature selection.

Feature Selection using Stochastic Optimization

<div style="text-align: right; font-size: 3em; font-weight: bold; color: #999;">29</div>

Typically, a simpler and better-performing machine learning model can be developed by removing input features (columns) from the training dataset. This is called feature selection and there are many different types of algorithms that can be used. It is possible to frame the problem of feature selection as an optimization problem. In the case that there are few input features, all possible combinations of input features can be evaluated and the best subset found definitively. In the case of a vast number of input features, a stochastic optimization algorithm can be used to explore the search space and find an effective subset of features.

In this tutorial, you will discover how to use optimization algorithms for feature selection in machine learning. After completing this tutorial, you will know:

▷ The problem of feature selection can be broadly defined as an optimization problem.

▷ How to enumerate all possible subsets of input features for a dataset.

▷ How to apply stochastic optimization to select an optimal subset of input features.

Let's get started.

29.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Optimization for Feature Selection
2. Enumerate All Feature Subsets
3. Optimize Feature Subsets

29.2 Optimization for Feature Selection

Feature selection is the process of reducing the number of input variables when developing a predictive model. It is desirable to reduce the number of input variables to both reduce the computational cost of modeling and, in some cases, to improve the performance of the model. There are many different types of feature selection algorithms, although they can broadly be grouped into two main types: wrapper and filter methods. Wrapper feature selection methods create many models with different subsets of input features and select those features that

result in the best performing model according to a performance metric. These methods are unconcerned with the variable types, although they can be computationally expensive. RFE is a good example of a wrapper feature selection method. Filter feature selection methods use statistical techniques to evaluate the relationship between each input variable and the target variable, and these scores are used as the basis to choose (filter) those input variables that will be used in the model.

▷ **Wrapper Feature Selection**: Search for well-performing subsets of features.

▷ **Filter Feature Selection**: Select subsets of features based on their relationship with the target.

A popular wrapper method is the Recursive Feature Elimination, or RFE, algorithm. RFE works by searching for a subset of features by starting with all features in the training dataset and successfully removing features until the desired number remains. This is achieved by fitting the given machine learning algorithm used in the core of the model, ranking features by importance, discarding the least important features, and re-fitting the model. This process is repeated until a specified number of features remains.

The problem of wrapper feature selection can be framed as an optimization problem. That is, find a subset of input features that result in the best model performance. RFE is one approach to solving this problem systematically, although it may be limited by a large number of features. An alternative approach would be to use a stochastic optimization algorithm, such as a stochastic hill climbing algorithm, when the number of features is very large. When the number of features is relatively small, it may be possible to enumerate all possible subsets of features.

▷ **Few Input Variables**: Enumerate all possible subsets of features.

▷ **Many Input Features**: Stochastic optimization algorithm to find good subsets of features.

Now that we are familiar with the idea that feature selection may be explored as an optimization problem, let's look at how we might enumerate all possible feature subsets.

29.3   Enumerate All Feature Subsets

When the number of input variables is relatively small and the model evaluation is relatively fast, then it may be possible to enumerate all possible subsets of input variables. This means evaluating the performance of a model using a test harness given every possible unique group of input variables. We will explore how to do this with a worked example. First, let's define a small binary classification dataset with few input features. We can use the `make_classification()` function[1] to define a dataset with five input variables, two of which are informative, and 1,000 rows. The example below defines the dataset and summarizes its shape.

```
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
   n_redundant=3, random_state=1)
```

---

[1]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

```
# summarize the shape of the dataset
print(X.shape, y.shape)
```

Program 29.1: Define a small classification dataset

Running the example creates the dataset and confirms that it has the desired shape.

```
(1000, 5) (1000,)
```

Output 29.1: Result from Program 29.1

Next, we can establish a baseline in performance using a model evaluated on the entire dataset. We will use a `DecisionTreeClassifier`[2] as the model because its performance is quite sensitive to the choice of input variables. We will evaluate the model using good practices, such as repeated stratified k-fold cross-validation with three repeats and 10 folds. The complete example is listed below.

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=3, n_informative=2,
    n_redundant=1, random_state=1)
# define model
model = DecisionTreeClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report result
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Program 29.2: Evaluate a decision tree on the entire small dataset

Running the example evaluates the decision tree on the entire dataset and reports the mean and standard deviation classification accuracy.

> **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 model achieved an accuracy of about 80.5 percent.

```
Mean Accuracy: 0.805 (0.030)
```

Output 29.2: Result from Program 29.2

---

[2]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Next, we can try to improve model performance by using a subset of the input features. First, we must choose a representation to enumerate. In this case, we will enumerate a list of boolean values, with one value for each input feature: `True` if the feature is to be used and `False` if the feature is not to be used as input. For example, with the five input features the sequence `[True, True, True, True, True]` would use all input features, and `[True, False, False, False, False]` would only use the first input feature as input. We can enumerate all sequences of boolean values with the `length=5` using the `product()` Python function[3]. We must specify the valid values `[True, False]` and the number of steps in the sequence, which is equal to the number of input variables. The function returns an iterable that we can enumerate directly for each sequence.

```
...
# determine the number of columns
n_cols = X.shape[1]
best_subset, best_score = None, 0.0
# enumerate all combinations of input features
for subset in product([True, False], repeat=n_cols):
    ...
```

Program 29.3: Enumerate combinations of input features

For a given sequence of boolean values, we can enumerate it and transform it into a sequence of column indexes for each `True` in the sequence.

```
...
ix = [i for i, x in enumerate(subset) if x]
```

Program 29.4: Convert the sequence into column indexes

If the sequence has no column indexes (in the case of all `False` values), then we can skip that sequence.

```
if len(ix) == 0:
    continue
```

Program 29.5: Skip the sequence if no column

We can then use the column indexes to choose the columns in the dataset.

```
...
X_new = X[:, ix]
```

Program 29.6: Select columns

And this subset of the dataset can then be evaluated as we did before.

```
...
# define model
model = DecisionTreeClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

---

[3]https://docs.python.org/3/library/itertools.html#itertools.product

```
# evaluate model
scores = cross_val_score(model, X_new, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize scores
result = mean(scores)
```

Program 29.7: Evaluate the subset

If the accuracy for the model is better than the best sequence found so far, we can store it.

```
...
if best_score is None or result >= best_score:
    # better result
    best_subset, best_score = ix, result
```

Program 29.8: Check if it is better than the best so far

And that's it. Tying this together, the complete example of feature selection by enumerating all possible feature subsets is listed below.

```
from itertools import product
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=3, random_state=1)
# determine the number of columns
n_cols = X.shape[1]
best_subset, best_score = None, 0.0
# enumerate all combinations of input features
for subset in product([True, False], repeat=n_cols):
    # convert into column indexes
    ix = [i for i, x in enumerate(subset) if x]
    # check for now column (all False)
    if len(ix) == 0:
        continue
    # select columns
    X_new = X[:, ix]
    # define model
    model = DecisionTreeClassifier()
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X_new, y, scoring='accuracy', cv=cv, n_jobs=-1)
    # summarize scores
    result = mean(scores)
    # report progress
    print('>f(%s) = %f ' % (ix, result))
    # check if it is better than the best so far
    if best_score is None or result >= best_score:
        # better result
        best_subset, best_score = ix, result
```

```
# report best
print('Done!')
print('f(%s) = %f' % (best_subset, best_score))
```

Program 29.9: Feature selection by enumerating all possible subsets of features

Running the example reports the mean classification accuracy of the model for each subset of features considered. The best subset is then reported at the end of the run.

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 best subset of features involved features at indexes [2, 3, 4] that resulted in a mean classification accuracy of about 83.0 percent, which is better than the result reported previously using all input features.

```
>f([0, 1, 2, 3, 4]) = 0.813667
>f([0, 1, 2, 3]) = 0.827667
>f([0, 1, 2, 4]) = 0.815333
>f([0, 1, 2]) = 0.824000
>f([0, 1, 3, 4]) = 0.821333
>f([0, 1, 3]) = 0.825667
>f([0, 1, 4]) = 0.807333
>f([0, 1]) = 0.817667
>f([0, 2, 3, 4]) = 0.830333
>f([0, 2, 3]) = 0.819000
>f([0, 2, 4]) = 0.828000
>f([0, 2]) = 0.818333
>f([0, 3, 4]) = 0.830333
>f([0, 3]) = 0.821333
>f([0, 4]) = 0.816000
>f([0]) = 0.639333
>f([1, 2, 3, 4]) = 0.823667
>f([1, 2, 3]) = 0.821667
>f([1, 2, 4]) = 0.823333
>f([1, 2]) = 0.818667
>f([1, 3, 4]) = 0.818000
>f([1, 3]) = 0.820667
>f([1, 4]) = 0.809000
>f([1]) = 0.797000
>f([2, 3, 4]) = 0.827667
>f([2, 3]) = 0.755000
>f([2, 4]) = 0.827000
>f([2]) = 0.516667
>f([3, 4]) = 0.824000
>f([3]) = 0.514333
>f([4]) = 0.777667
Done!
f([0, 3, 4]) = 0.830333
```

Output 29.3: Result from Program 29.9

Now that we know how to enumerate all possible feature subsets, let's look at how we might use a stochastic optimization algorithm to choose a subset of features.

29.4   Optimize Feature Subsets

We can apply a stochastic optimization algorithm to the search space of subsets of input features. First, let's define a larger problem that has many more features, making model evaluation too slow and the search space too large for enumerating all subsets. We will define a classification problem with 10,000 rows and 500 input features, 10 of which are relevant and the remaining 490 are redundant.

```python
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=10000, n_features=500, n_informative=10,
    n_redundant=490, random_state=1)
# summarize the shape of the dataset
print(X.shape, y.shape)
```

Program 29.10: Define a large classification dataset

Running the example creates the dataset and confirms that it has the desired shape.

```
(10000, 500) (10000,)
```

Output 29.4: Result from Program 29.10

We can establish a baseline in performance by evaluating a model on the dataset with all input features. Because the dataset is large and the model is slow to evaluate, we will modify the evaluation of the model to use 3-fold cross-validation, i.e. fewer folds and no repeats. The complete example is listed below.

```python
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=500, n_informative=10,
    n_redundant=490, random_state=1)
# define model
model = DecisionTreeClassifier()
# define evaluation procedure
cv = StratifiedKFold(n_splits=3)
# evaluate model
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report result
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Program 29.11: Evaluate a decision tree on the entire larger dataset

Running the example evaluates the decision tree on the entire dataset and reports the mean and standard deviation classification accuracy.

> ⚠️ **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 model achieved an accuracy of about 91.3 percent. This provides a baseline that we would expect to outperform using feature selection.

```
Mean Accuracy: 0.913 (0.001)
```

Output 29.5: Result from Program 29.11

We will use a simple stochastic hill climbing algorithm as the optimization algorithm. First, we must define the objective function. It will take the dataset and a subset of features to use as input and return an estimated model accuracy from 0 (worst) to 1 (best). It is a maximizing optimization problem. This objective function is simply the decoding of the sequence and model evaluation step from the previous section. The `objective()` function below implements this and returns both the score and the decoded subset of columns used for helpful reporting.

```python
def objective(X, y, subset):
    # convert into column indexes
    ix = [i for i, x in enumerate(subset) if x]
    # check for now column (all False)
    if len(ix) == 0:
        return 0.0
    # select columns
    X_new = X[:, ix]
    # define model
    model = DecisionTreeClassifier()
    # evaluate model
    scores = cross_val_score(model, X_new, y, scoring='accuracy', cv=3, n_jobs=-1)
    # summarize scores
    result = mean(scores)
    return result, ix
```

Program 29.12: Objective function

We also need a function that can take a step in the search space. Given an existing solution, it must modify it and return a new solution in close proximity. In this case, we will achieve this by randomly flipping the inclusion/exclusion of columns in subsequence. Each position in the sequence will be considered independently and will be flipped probabilistically where the probability of flipping is a hyperparameter. The `mutate()` function below implements this given a candidate solution (sequence of booleans) and a mutation hyperparameter, creating and returning a modified solution (a step in the search space). The larger the `p_mutate` value (in the range 0 to 1), the larger the step in the search space.

```python
def mutate(solution, p_mutate):
    # make a copy
    child = solution.copy()
    for i in range(len(child)):
        # check for a mutation
        if rand() < p_mutate:
            # flip the inclusion
            child[i] = not child[i]
    return child
```

Program 29.13: Mutation operator

We can now implement the hill climbing algorithm. The initial solution is a randomly generated sequence, which is then evaluated.

```python
...
# generate an initial point
solution = choice([True, False], size=X.shape[1])
# evaluate the initial point
solution_eval, ix = objective(X, y, solution)
```

Program 29.14: Evaluate a random sequence

We then loop for a fixed number of iterations, creating mutated versions of the current solution, evaluating them, and saving them if the score is better.

```python
...
for i in range(n_iter):
    # take a step
    candidate = mutate(solution, p_mutate)
    # evaluate candidate point
    candidate_eval, ix = objective(X, y, candidate)
    # check if we should keep the new point
    if candidate_eval >= solution_eval:
        # store the new point
        solution, solution_eval = candidate, candidate_eval
    # report progress
    print('>%d f(%s) = %f' % (i+1, len(ix), solution_eval))
```

Program 29.15: Hill climbing

The `hillclimbing()` function below implements this, taking the dataset, objective function, and hyperparameters as arguments and returns the best subset of dataset columns and the estimated performance of the model.

```python
def hillclimbing(X, y, objective, n_iter, p_mutate):
    # generate an initial point
    solution = choice([True, False], size=X.shape[1])
    # evaluate the initial point
    solution_eval, ix = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = mutate(solution, p_mutate)
```

```
        # evaluate candidate point
        candidate_eval, ix = objective(X, y, candidate)
        # check if we should keep the new point
        if candidate_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidate_eval
        # report progress
        print('>%d f(%s) = %f' % (i+1, len(ix), solution_eval))
    return solution, solution_eval
```

Program 29.16: Hill climbing local search algorithm

We can then call this function and pass in our synthetic dataset to perform optimization for feature selection. In this case, we will run the algorithm for 100 iterations and make about five flips to the sequence for a given mutation, which is quite conservative.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=500, n_informative=10,
    n_redundant=490, random_state=1)
# define the total iterations
n_iter = 100
# probability of including/excluding a column
p_mut = 10.0 / 500.0
# perform the hill climbing search
subset, score = hillclimbing(X, y, objective, n_iter, p_mut)
```

Program 29.17: Perform the hill climbing search

At the end of the run, we will convert the boolean sequence into column indexes (so we could fit a final model if we wanted) and report the performance of the best subsequence.

```
...
ix = [i for i, x in enumerate(subset) if x]
print('Done!')
print('Best: f(%d) = %f' % (len(ix), score))
```

Program 29.18: Convert into column indexes

Tying this all together, the complete example is listed below.

```
from numpy import mean
from numpy.random import rand
from numpy.random import choice
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier

# objective function
def objective(X, y, subset):
    # convert into column indexes
    ix = [i for i, x in enumerate(subset) if x]
    # check for now column (all False)
    if len(ix) == 0:
```

```python
        return 0.0
    # select columns
    X_new = X[:, ix]
    # define model
    model = DecisionTreeClassifier()
    # evaluate model
    scores = cross_val_score(model, X_new, y, scoring='accuracy', cv=3, n_jobs=-1)
    # summarize scores
    result = mean(scores)
    return result, ix

# mutation operator
def mutate(solution, p_mutate):
    # make a copy
    child = solution.copy()
    for i in range(len(child)):
        # check for a mutation
        if rand() < p_mutate:
            # flip the inclusion
            child[i] = not child[i]
    return child

# hill climbing local search algorithm
def hillclimbing(X, y, objective, n_iter, p_mutate):
    # generate an initial point
    solution = choice([True, False], size=X.shape[1])
    # evaluate the initial point
    solution_eval, ix = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = mutate(solution, p_mutate)
        # evaluate candidate point
        candidate_eval, ix = objective(X, y, candidate)
        # check if we should keep the new point
        if candidate_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidate_eval
        # report progress
        print('>%d f(%s) = %f' % (i+1, len(ix), solution_eval))
    return solution, solution_eval

# define dataset
X, y = make_classification(n_samples=10000, n_features=500, n_informative=10,
    n_redundant=490, random_state=1)
# define the total iterations
n_iter = 100
# probability of including/excluding a column
p_mut = 10.0 / 500.0
# perform the hill climbing search
subset, score = hillclimbing(X, y, objective, n_iter, p_mut)
# convert into column indexes
ix = [i for i, x in enumerate(subset) if x]
```

```
print('Done!')
print('Best: f(%d) = %f' % (len(ix), score))
```

Program 29.19: Stochastic optimization for feature selection

Running the example reports the mean classification accuracy of the model for each subset of features considered. The best subset is then reported at the end of the run.

> ⚠️ **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 best performance was achieved with a subset of 239 features and a classification accuracy of approximately 91.8 percent. This is better than a model evaluated on all input features. Although the result is better, we know we can do a lot better, perhaps with tuning of the hyperparameters of the optimization algorithm or perhaps by using an alternate optimization algorithm.

```
...
>80 f(240) = 0.918099
>81 f(236) = 0.918099
>82 f(238) = 0.918099
>83 f(236) = 0.918099
>84 f(239) = 0.918099
>85 f(240) = 0.918099
>86 f(239) = 0.918099
>87 f(245) = 0.918099
>88 f(241) = 0.918099
>89 f(239) = 0.918099
>90 f(239) = 0.918099
>91 f(241) = 0.918099
>92 f(243) = 0.918099
>93 f(245) = 0.918099
>94 f(239) = 0.918099
>95 f(245) = 0.918099
>96 f(244) = 0.918099
>97 f(242) = 0.918099
>98 f(238) = 0.918099
>99 f(248) = 0.918099
>100 f(238) = 0.918099
Done!
Best: f(239) = 0.918099
```

Output 29.6: Result from Program 29.19

29.5   Further Reading

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

APIs

`sklearn.datasets.make_classification` *API*.
    https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_
    classification.html
`itertools.product` *API*.
    https://docs.python.org/3/library/itertools.html%5C#itertools.product

29.6 Summary

In this tutorial, you discovered how to use optimization algorithms for feature selection in machine learning. Specifically, you learned:

▷ The problem of feature selection can be broadly defined as an optimization problem.

▷ How to enumerate all possible subsets of input features for a dataset.

▷ How to apply stochastic optimization to select an optimal subset of input features.

Next, we will see how optimization algorithms can be used to tune hyperparameters.

Manually Optimize Machine Learning Model Hyperparameters

30

Machine learning algorithms have hyperparameters that allow the algorithms to be tailored to specific datasets. Although the impact of hyperparameters may be understood generally, their specific effect on a dataset and their interactions during learning may not be known. Therefore, it is important to tune the values of algorithm hyperparameters as part of a machine learning project. It is common to use naive optimization algorithms to tune hyperparameters, such as a grid search and a random search. An alternate approach is to use a stochastic optimization algorithm, like a stochastic hill climbing algorithm.

In this tutorial, you will discover how to manually optimize the hyperparameters of machine learning algorithms. After completing this tutorial, you will know:

▷ Stochastic optimization algorithms can be used instead of grid and random search for hyperparameter optimization.

▷ How to use a stochastic hill climbing algorithm to tune the hyperparameters of the Perceptron algorithm.

▷ How to manually optimize the hyperparameters of the XGBoost gradient boosting algorithm.

Let's get started.

30.1   Tutorial Overview

This tutorial is divided into three parts; they are:

1. Manual Hyperparameter Optimization
2. Perceptron Hyperparameter Optimization
3. XGBoost Hyperparameter Optimization

30.2   Manual Hyperparameter Optimization

Machine learning models have hyperparameters that you must set in order to customize the model to your dataset. Often, the general effects of hyperparameters on a model are known, but

how to best set a hyperparameter and combinations of interacting hyperparameters for a given dataset is challenging. A better approach is to objectively search different values for model hyperparameters and choose a subset that results in a model that achieves the best performance on a given dataset. This is called hyperparameter optimization, or hyperparameter tuning. A range of different optimization algorithms may be used, although two of the simplest and most common methods are random search and grid search.

▷ **Random Search**. Define a search space as a bounded domain of hyperparameter values and randomly sample points in that domain.

▷ **Grid Search**. Define a search space as a grid of hyperparameter values and evaluate every position in the grid.

Grid search is great for spot-checking combinations that are known to perform well generally. Random search is great for discovery and getting hyperparameter combinations that you would not have guessed intuitively, although it often requires more time to execute. Grid and random search are primitive optimization algorithms, and it is possible to use any optimization we like to tune the performance of a machine learning algorithm. For example, it is possible to use stochastic optimization algorithms. This might be desirable when good or great performance is required and there are sufficient resources available to tune the model.

Next, let's look at how we might use a stochastic hill climbing algorithm to tune the performance of the Perceptron algorithm.

30.3 Perceptron Hyperparameter Optimization

The Perceptron algorithm is the simplest type of artificial neural network. It is a model of a single neuron that can be used for two-class classification problems and provides the foundation for later developing much larger networks. In this section, we will explore how to manually optimize the hyperparameters of the Perceptron model. First, let's define a synthetic binary classification problem that we can use as the focus of optimizing the model. We can use the `make_classification()` function[1] to define a binary classification problem with 1,000 rows and five input variables. The example below creates the dataset and summarizes the shape of the data.

```
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# summarize the shape of the dataset
print(X.shape, y.shape)
```

Program 30.1: Define a binary classification dataset

Running the example prints the shape of the created dataset, confirming our expectations.

```
(1000, 5) (1000,)
```

Output 30.1: Result from Program 30.1

---

[1]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

The scikit-learn provides an implementation of the Perceptron model via the Perceptron class[2]. Before we tune the hyperparameters of the model, we can establish a baseline in performance using the default hyperparameters. We will evaluate the model using good practices of repeated stratified k-fold cross-validation via the `RepeatedStratifiedKFold` class[3].

The complete example of evaluating the Perceptron model with default hyperparameters on our synthetic binary classification dataset is listed below.

```python
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import Perceptron
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# define model
model = Perceptron()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report result
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Program 30.2: Perceptron default hyperparameters for binary classification

Running the example reports evaluates the model and reports the mean and standard deviation of the classification accuracy.

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 model with default hyperparameters achieved a classification accuracy of about 78.5 percent. We would hope that we can achieve better performance than this with optimized hyperparameters.

```
Mean Accuracy: 0.786 (0.069)
```

Output 30.2: Result from Program 30.2

Next, we can optimize the hyperparameters of the Perceptron model using a stochastic hill climbing algorithm. There are many hyperparameters that we could optimize, although we will focus on two that perhaps have the most impact on the learning behavior of the model; they are:

---

[2]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html
[3]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedStratified
KFold.html

▷ Learning Rate η_0 ("`eta0`").

▷ Regularization α ("`alpha`").

The learning rate controls the amount the model is updated based on prediction errors and controls the speed of learning. The default value of η_0 is 1.0. Reasonable values are larger than zero (e.g. larger than 10^{-8} or 10^{-10}) and probably less than 1.0 By default, the Perceptron does not use any regularization, but we will enable "*elastic net*" regularization which applies both L1 and L2 regularization during learning. This will encourage the model to seek small model weights and, in turn, often better performance.

We will tune the "α" hyperparameter that controls the weighting of the regularization, i.e. the amount it impacts the learning. If set to 0.0, it is as though no regularization is being used. Reasonable values are between 0.0 and 1.0. First, we need to define the objective function for the optimization algorithm. We will evaluate a configuration using mean classification accuracy with repeated stratified k-fold cross-validation. We will seek to maximize accuracy in the configurations.

The `objective()` function below implements this, taking the dataset and a list of config values. The config values (learning rate and regularization weighting) are unpacked, used to configure the model, which is then evaluated, and the mean accuracy is returned.

```python
def objective(X, y, cfg):
    # unpack config
    eta, alpha = cfg
    # define model
    model = Perceptron(penalty='elasticnet', alpha=alpha, eta0=eta)
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    # calculate mean accuracy
    result = mean(scores)
    return result
```

Program 30.3: Objective function

Next, we need a function to take a step in the search space. The search space is defined by two variables (`eta` and `alpha`). A step in the search space must have some relationship to the previous values and must be bound to sensible values (e.g. between 0 and 1). We will use a "`step_size`" hyperparameter that controls how far the algorithm is allowed to move from the existing configuration. A new configuration will be chosen probabilistically using a Gaussian distribution with the current value as the mean of the distribution and the step size as the standard deviation of the distribution. We can use the `randn()` NumPy function[4] to generate random numbers with a Gaussian distribution. The `step()` function below implements this and will take a step in the search space and generate a new configuration using an existing configuration.

---

[4] https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html

```
def step(cfg, step_size):
    # unpack the configuration
    eta, alpha = cfg
    # step eta
    new_eta = eta + randn() * step_size
    # check the bounds of eta
    if new_eta <= 0.0:
        new_eta = 1e-8
    if new_eta > 1.0:
        new_eta = 1.0
    # step alpha
    new_alpha = alpha + randn() * step_size
    # check the bounds of alpha
    if new_alpha < 0.0:
        new_alpha = 0.0
    # return the new configuration
    return [new_eta, new_alpha]
```

Program 30.4: Take a step in the search space

Next, we need to implement the stochastic hill climbing algorithm that will call our `objective()` function to evaluate candidate solutions and our `step()` function to take a step in the search space. The search first generates a random initial solution, in this case with `eta` and `alpha` values in the range 0 and 1. The initial solution is then evaluated and is taken as the current best working solution.

```
...
# starting point for the search
solution = [rand(), rand()]
# evaluate the initial point
solution_eval = objective(X, y, solution)
```

Program 30.5: Evaluate a random point

Next, the algorithm iterates for a fixed number of iterations provided as a hyperparameter to the search. Each iteration involves taking a step and evaluating the new candidate solution.

```
...
# take a step
candidate = step(solution, step_size)
# evaluate candidate point
candidate_eval = objective(X, y, candidate)
```

Program 30.6: Evaluate the candidate solution

If the new solution is better than the current working solution, it is taken as the new current working solution.

```
...
if candidate_eval >= solution_eval:
    # store the new point
    solution, solution_eval = candidate, candidate_eval
```

```
    # report progress
    print('>%d, cfg=%s %.5f' % (i, solution, solution_eval))
```
Program 30.7: Check if we should keep the new point

At the end of the search, the best solution and its performance are then returned. Tying this together, the `hillclimbing()` function below implements the stochastic hill climbing algorithm for tuning the Perceptron algorithm, taking the dataset, objective function, number of iterations, and step size as arguments.

```
def hillclimbing(X, y, objective, n_iter, step_size):
    # starting point for the search
    solution = [rand(), rand()]
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = step(solution, step_size)
        # evaluate candidate point
        candidate_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidate_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidate_eval
            # report progress
            print('>%d, cfg=%s %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]
```
Program 30.8: Hill climbing local search algorithm

We can then call the algorithm and report the results of the search. In this case, we will run the algorithm for 100 iterations and use a step size of 0.1, chosen after a little trial and error.

```
...
# define the total iterations
n_iter = 100
# step size in the search space
step_size = 0.1
# perform the hill climbing search
cfg, score = hillclimbing(X, y, objective, n_iter, step_size)
print('Done!')
print('cfg=%s: Mean Accuracy: %f' % (cfg, score))
```
Program 30.9: Perform the hill climbing search

Tying this together, the complete example of manually tuning the Perceptron algorithm is listed below.

```
from numpy import mean
from numpy.random import randn
from numpy.random import rand
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
```

```python
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import Perceptron

# objective function
def objective(X, y, cfg):
    # unpack config
    eta, alpha = cfg
    # define model
    model = Perceptron(penalty='elasticnet', alpha=alpha, eta0=eta)
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    # calculate mean accuracy
    result = mean(scores)
    return result

# take a step in the search space
def step(cfg, step_size):
    # unpack the configuration
    eta, alpha = cfg
    # step eta
    new_eta = eta + randn() * step_size
    # check the bounds of eta
    if new_eta <= 0.0:
        new_eta = 1e-8
    if new_eta > 1.0:
        new_eta = 1.0
    # step alpha
    new_alpha = alpha + randn() * step_size
    # check the bounds of alpha
    if new_alpha < 0.0:
        new_alpha = 0.0
    # return the new configuration
    return [new_eta, new_alpha]

# hill climbing local search algorithm
def hillclimbing(X, y, objective, n_iter, step_size):
    # starting point for the search
    solution = [rand(), rand()]
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = step(solution, step_size)
        # evaluate candidate point
        candidate_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidate_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidate_eval
            # report progress
            print('>%d, cfg=%s %.5f' % (i, solution, solution_eval))
```

```
        return [solution, solution_eval]

# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# define the total iterations
n_iter = 100
# step size in the search space
step_size = 0.1
# perform the hill climbing search
cfg, score = hillclimbing(X, y, objective, n_iter, step_size)
print('Done!')
print('cfg=%s: Mean Accuracy: %f' % (cfg, score))
```

Program 30.10: Manually search perceptron hyperparameters for binary classification

Running the example reports the configuration and result each time an improvement is seen during the search. At the end of the run, the best configuration and result are reported.

> ⚠️ **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 best result involved using a learning rate slightly above 1 at 1.004 and a regularization weight of about 0.002 achieving a mean accuracy of about 79.1 percent, better than the default configuration that achieved an accuracy of about 78.5 percent.

```
>0, cfg=[0.5827274503894747, 0.260872709578015] 0.70533
>4, cfg=[0.5449820307807399, 0.3017271170801444] 0.70567
>6, cfg=[0.6286475606495414, 0.17499090243915086] 0.71933
>7, cfg=[0.5956196828965779, 0.0] 0.78633
>8, cfg=[0.5878361167354715, 0.0] 0.78633
>10, cfg=[0.6353507984485595, 0.0] 0.78633
>13, cfg=[0.5690530537610675, 0.0] 0.78633
>17, cfg=[0.6650936023999641, 0.0] 0.78633
>22, cfg=[0.9070451625704087, 0.0] 0.78633
>23, cfg=[0.9253366187387938, 0.0] 0.78633
>26, cfg=[0.9966143540220266, 0.0] 0.78633
>31, cfg=[1.0048613895650054, 0.002162219228449132] 0.79133
Done!
cfg=[1.0048613895650054, 0.002162219228449132]: Mean Accuracy: 0.791333
```

Output 30.3: Result from Program 30.10

Now that we are familiar with how to use a stochastic hill climbing algorithm to tune the hyperparameters of a simple machine learning algorithm, let's look at tuning a more advanced algorithm, such as XGBoost.

30.4   XGBoost Hyperparameter Optimization

XGBoost is short for Extreme Gradient Boosting and is an efficient implementation of the stochastic gradient boosting machine learning algorithm. The stochastic gradient boosting algorithm, also called gradient boosting machines or tree boosting, is a powerful machine learning technique that performs well or even best on a wide range of challenging machine learning problems.

First, the XGBoost library must be installed. You can install it using pip, as follows:

```
sudo pip install xgboost
```

Once installed, you can confirm that it was installed successfully and that you are using a modern version by running the following code:

```
import xgboost
print("xgboost", xgboost.__version__)
```

Program 30.11: Show the version of XGBoost

Running the code, you should see the following version number or higher.

```
xgboost 1.0.1
```

Although the XGBoost library has its own Python API, we can use XGBoost models with the scikit-learn API via the `XGBClassifier` wrapper class[5]. An instance of the model can be instantiated and used just like any other scikit-learn class for model evaluation. For example:

```
...
model = XGBClassifier(use_label_encoder=False, eval_metric="logloss")
```

Program 30.12: Define model

Before we tune the hyperparameters of XGBoost, we can establish a baseline in performance using the default hyperparameters. We will use the same synthetic binary classification dataset from the previous section and the same test harness of repeated stratified *k*-fold cross-validation. The complete example of evaluating the performance of XGBoost with default hyperparameters is listed below.

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from xgboost import XGBClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
```

---

[5]https://xgboost.readthedocs.io/en/latest/python/python_api.html

```
# define model
model = XGBClassifier(use_label_encoder=False, eval_metric="logloss")
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report result
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Program 30.13: XGBoost with default hyperparameters for binary classification

Running the example evaluates the model and reports the mean and standard deviation of the classification accuracy.

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 model with default hyperparameters achieved a classification accuracy of about 84.9 percent. We would hope that we can achieve better performance than this with optimized hyperparameters.

```
Mean Accuracy: 0.849 (0.040)
```

Output 30.4: Result from Program 30.13

Next, we can adapt the stochastic hill climbing optimization algorithm to tune the hyperparameters of the XGBoost model. There are many hyperparameters that we may want to optimize for the XGBoost model. We will focus on four key hyperparameters; they are:

▷ Learning Rate (`learning_rate`)

▷ Number of Trees (`n_estimators`)

▷ Subsample Percentage (`subsample`)

▷ Tree Depth (`max_depth`)

The *learning rate* controls the contribution of each tree to the ensemble. Sensible values are less than 1.0 and slightly above 0.0 (e.g. 10^{-8}). The *number of trees* controls the size of the ensemble, and often, more trees is better to a point of diminishing returns. Sensible values are between 1 tree and hundreds or thousands of trees. The *subsample percentage* define the random sample size used to train each tree, defined as a percentage of the size of the original dataset. Values are between a value slightly above 0.0 (e.g. 10^{-8}) and 1.0. The *tree depth* is the number of levels in each tree. Deeper trees are more specific to the training dataset and perhaps overfit. Shorter trees often generalize better. Sensible values are between 1 and 10 or 20.

First, we must update the `objective()` function to unpack the hyperparameters of the XGBoost model, configure it, and then evaluate the mean classification accuracy.

```
def objective(X, y, cfg):
    # unpack config
    lrate, n_tree, subsam, depth = cfg
    # define model
    model = XGBClassifier(learning_rate=lrate, n_estimators=n_tree,
        subsample=subsam, max_depth=depth, use_label_encoder=False,
        eval_metric="logloss")
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    # calculate mean accuracy
    result = mean(scores)
    return result
```

Program 30.14: Objective function

Next, we need to define the `step()` function used to take a step in the search space. Each hyperparameter is quite a different range, therefore, we will define the step size (standard deviation of the distribution) separately for each hyperparameter. We will also define the step sizes in line rather than as arguments to the function, to keep things simple. The number of trees and the depth are integers, so the stepped values are rounded. The step sizes chosen are arbitrary, chosen after a little trial and error. The updated step function is listed below.

```
def step(cfg):
    # unpack config
    lrate, n_tree, subsam, depth = cfg
    # learning rate
    lrate = lrate + randn() * 0.01
    if lrate <= 0.0:
        lrate = 1e-8
    if lrate > 1:
        lrate = 1.0
    # number of trees
    n_tree = round(n_tree + randn() * 50)
    if n_tree <= 0.0:
        n_tree = 1
    # subsample percentage
    subsam = subsam + randn() * 0.1
    if subsam <= 0.0:
        subsam = 1e-8
    if subsam > 1:
        subsam = 1.0
    # max tree depth
    depth = round(depth + randn() * 7)
    if depth <= 1:
        depth = 1
    # return new config
    return [lrate, n_tree, subsam, depth]
```

Program 30.15: Take a step in the search space

Finally, the `hillclimbing()` algorithm must be updated to define an initial solution with appropriate values. In this case, we will define the initial solution with sensible defaults, matching the default hyperparameters, or close to them.

```
...
solution = step([0.1, 100, 1.0, 7])
```

Program 30.16: Set starting point for the search

Tying this together, the complete example of manually tuning the hyperparameters of the XGBoost algorithm using a stochastic hill climbing algorithm is listed below.

```python
from numpy import mean
from numpy.random import randn
from numpy.random import rand
from numpy.random import randint
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from xgboost import XGBClassifier

# objective function
def objective(X, y, cfg):
    # unpack config
    lrate, n_tree, subsam, depth = cfg
    # define model
    model = XGBClassifier(learning_rate=lrate, n_estimators=n_tree,
        subsample=subsam, max_depth=depth, use_label_encoder=False,
        eval_metric="logloss")
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    # calculate mean accuracy
    result = mean(scores)
    return result

# take a step in the search space
def step(cfg):
    # unpack config
    lrate, n_tree, subsam, depth = cfg
    # learning rate
    lrate = lrate + randn() * 0.01
    if lrate <= 0.0:
        lrate = 1e-8
    if lrate > 1:
        lrate = 1.0
    # number of trees
    n_tree = round(n_tree + randn() * 50)
    if n_tree <= 0.0:
        n_tree = 1
    # subsample percentage
    subsam = subsam + randn() * 0.1
```

```python
        if subsam <= 0.0:
            subsam = 1e-8
        if subsam > 1:
            subsam = 1.0
        # max tree depth
        depth = round(depth + randn() * 7)
        if depth <= 1:
            depth = 1
        # return new config
        return [lrate, n_tree, subsam, depth]

# hill climbing local search algorithm
def hillclimbing(X, y, objective, n_iter):
    # starting point for the search
    solution = step([0.1, 100, 1.0, 7])
    # evaluate the initial point
    solution_eval = objective(X, y, solution)
    # run the hill climb
    for i in range(n_iter):
        # take a step
        candidate = step(solution)
        # evaluate candidate point
        candidate_eval = objective(X, y, candidate)
        # check if we should keep the new point
        if candidate_eval >= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidate_eval
            # report progress
            print('>%d, cfg=[%s] %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]

# define dataset
X, y = make_classification(n_samples=1000, n_features=5, n_informative=2,
    n_redundant=1, random_state=1)
# define the total iterations
n_iter = 200
# perform the hill climbing search
cfg, score = hillclimbing(X, y, objective, n_iter)
print('Done!')
print('cfg=[%s]: Mean Accuracy: %f' % (cfg, score))
```

Program 30.17: *XGBoost manual hyperparameter optimization for binary classification*

Running the example reports the configuration and result each time an improvement is seen during the search. At the end of the run, the best configuration and result are reported.

⚠ | **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 best result involved using a learning rate of about 0.02, 52 trees, a subsample rate of about 50 percent, and a large depth of 53 levels. This configuration

resulted in a mean accuracy of about 87.3 percent, better than the default configuration that achieved an accuracy of about 84.9 percent.

```
>0, cfg=[[0.1058242692126418, 67, 0.9228490731610172, 12]] 0.85933
>1, cfg=[[0.11060813799692253, 51, 0.859353656735739, 13]] 0.86100
>4, cfg=[[0.11890247679234153, 58, 0.7135275461723894, 12]] 0.86167
>5, cfg=[[0.10226257987735601, 61, 0.6086462443373852, 17]] 0.86400
>15, cfg=[[0.11176962034280596, 106, 0.5592742266405146, 13]] 0.86500
>19, cfg=[[0.09493587069112454, 153, 0.5049124222437619, 34]] 0.86533
>23, cfg=[[0.08516531024154426, 88, 0.5895201311518876, 31]] 0.86733
>46, cfg=[[0.10092590898175327, 32, 0.5982811365027455, 30]] 0.86867
>75, cfg=[[0.099469211050998, 20, 0.36372573610040404, 32]] 0.86900
>96, cfg=[[0.09021536590375884, 38, 0.4725379807796971, 20]] 0.86900
>100, cfg=[[0.08979482274655906, 65, 0.3697395430835758, 14]] 0.87000
>110, cfg=[[0.06792737273465625, 89, 0.33827505722318224, 17]] 0.87000
>118, cfg=[[0.05544969684589669, 72, 0.2989721608535262, 23]] 0.87200
>122, cfg=[[0.050102976159097, 128, 0.204320396514893, 24]] 0.87200
>123, cfg=[[0.031493266763680444, 120, 0.2998819062922256, 30]] 0.87333
>128, cfg=[[0.023324201169625292, 84, 0.4017169945431015, 42]] 0.87333
>140, cfg=[[0.020224220443108752, 52, 0.5088096815056933, 53]] 0.87367
Done!
cfg=[[0.020224220443108752, 52, 0.5088096815056933, 53]]: Mean Accuracy: 0.873667
```

Output 30.5: Result from Program 30.17

30.5 Further Reading

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

APIs

sklearn.datasets.make_classification *API*.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html
sklearn.metrics.accuracy_score *API*.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html
numpy.random.rand *API*.
https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html
sklearn.linear_model.Perceptron *API*.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

Articles

Perceptron. Wikipedia.
https://en.wikipedia.org/wiki/Perceptron

XGBoost. Wikipedia.
    https://en.wikipedia.org/wiki/XGBoost

30.6   Summary

In this tutorial, you discovered how to manually optimize the hyperparameters of machine learning algorithms. Specifically, you learned:

▷ Stochastic optimization algorithms can be used instead of grid and random search for hyperparameter optimization.

▷ How to use a stochastic hill climbing algorithm to tune the hyperparameters of the Perceptron algorithm.

▷ How to manually optimize the hyperparameters of the XGBoost gradient boosting algorithm.

This is the final chapter of this book. Well done!

VII Appendix

Getting Help

<div style="text-align: right">

A

</div>

This is just the beginning of your journey with function optimization. As you start to work on projects and expand your existing knowledge of the techniques, you may need help. This appendix points out some of the best sources of help.

A.1   Page of Topics on Wikipedia

Wikipedia is a great place to start. All of the important topics are covered, the descriptions are concise, and the equations are consistent and readable. What is missing is the more human level descriptions such as analogies and intuitions. Nevertheless, when you have questions about linear algebra, I recommend stopping by Wikipedia first. Some good high-level pages to start on include:

- ▷ Mathematical Optimization
  https://en.wikipedia.org/wiki/Mathematical_optimization
- ▷ List of Optimization Algorithms
  https://en.wikipedia.org/wiki/List_of_algorithms#Optimization_algorithms

A.2   Textbooks

There are some good textboooks on the topic. You can use it as a reference. A good textbook can give you explanations in a great detail consistently. For the mathematical side of optimization, these are the recommended:

- ▷ Stephen Boyd and Lieven Vandenberghe, *Convex Optimization*, Cambridge, 2004.
  https://amzn.to/34mvCr1
- ▷ James C. Spall, *Introduction to Stochastic Search and Optimization*, Wiley, 2003.
  https://amzn.to/34JYN7m

Specific to machine learning, many books have a chapter or two on the optimization algorithms, such as

- ▷ Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016.
  https://amzn.to/3qSk3C2

▷ Christopher M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
https://amzn.to/36yvG9w

For the algorithms and implementation perspective, there are quite a number of books, such as

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

▷ Michel Gendreau and Jean-Yves Potvin, *Handbook of Metaheuristics*, 3rd ed., Springer, 2019.
https://amzn.to/2IIq0Qt

▷ GrÃľgoire Montavon, GeneviÃĺve Orr, and Klaus-Robert MÃijller, *Neural Networks: Tricks of the Trade*, 2nd ed., Springer, 2012.
https://amzn.to/3ac5S4Q

These are not mean to be exclusive. The bibliography section on the relevant pages in Wikipedia can usually bring you to some good references.

A.3 NumPy and SciPy Resources

In this book, we used NumPy and SciPy a lot. They are twins and their documentation is very well written. The following are some resources that you can learn more about how to use NumPy and SciPy.

▷ NumPy Reference
https://docs.scipy.org/doc/numpy/reference/

▷ Optimization (`scipy.optimize`)
https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html

▷ Optimization and root finding (`scipy.optimize`)
https://docs.scipy.org/doc/scipy/reference/optimize.html

If you are looking for a broader understanding on NumPy and SciPy usage, the below books provide a good starting reference:

▷ Wes McKinney, *Python for Data Analysis*, 2nd ed., O'Reilly, 2017.
http://amzn.to/2B1sfXi

▷ Juan Nunez-Iglesias, StÃĺfan van der Walt, and Harriet Dashnow, *Elegant SciPy*, O'Reilly, 2017.
http://amzn.to/2yujXnT

▷ Travis E. Oliphant, *Guide to NumPy*, 2nd ed., CreateSpace, 2015.
http://amzn.to/2j3kEzd

A.4 Ask Questions About Optimization

There are a lot of places that you can ask questions about optimization online given the current abundance of question-and-answer platforms. Below is a list of the top places I recommend

posting a question. Remember to search for your question before posting in case it has been asked and answered before.

▷ Optimization tag on the Mathematics Stack Exchange.
  https://math.stackexchange.com/?tags=optimization

▷ Mathematical Optimization tag on Stack Overflow.
  https://stackoverflow.com/questions/tagged/mathematical-optimization

▷ Mathematics and Machine Learning on Quora.
  https://www.quora.com/topic/Mathematics-and-Machine-Learning

▷ Math Subreddit.
  https://www.reddit.com/r/math/

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

▷ Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work.*

▷ Search for answers before asking questions.

▷ Provide complete code and error messages.

▷ Boil your code down to the smallest possible working example that demonstrates the issue.

A.6 Contact the Author

You are not alone. If you ever have any questions about deep learning, natural language processing, or this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

How to Setup Your Python Environment

<div style="text-align: right">**B**</div>

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda. After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, Mac OS X, and Linux platforms. I will demonstrate them on Windows, so you may see some Windows dialogs and file extensions.

Let's get started.

B.1   Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda

2. Install Anaconda

3. Start and Update Anaconda

4. Update scikit-learn Library

5. Install Deep Learning Libraries

B.2   Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

1. Visit the Anaconda homepage https://www.anaconda.com/
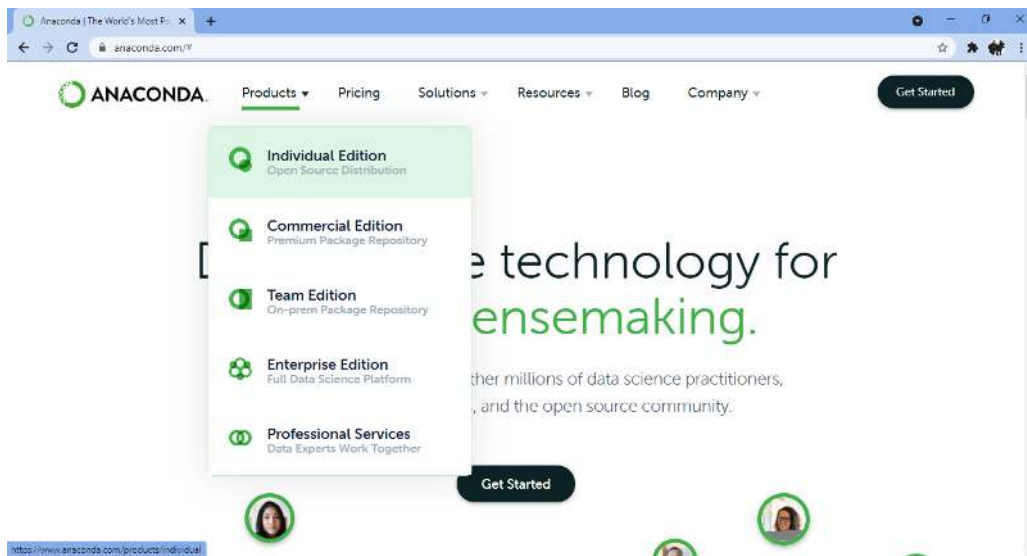
Figure B.1: Click "Products" and "Individual Edition"

2. Click "Products" from the menu and click "Individual Edition" to go to the download page <https://www.anaconda.com/products/individual-d/>.
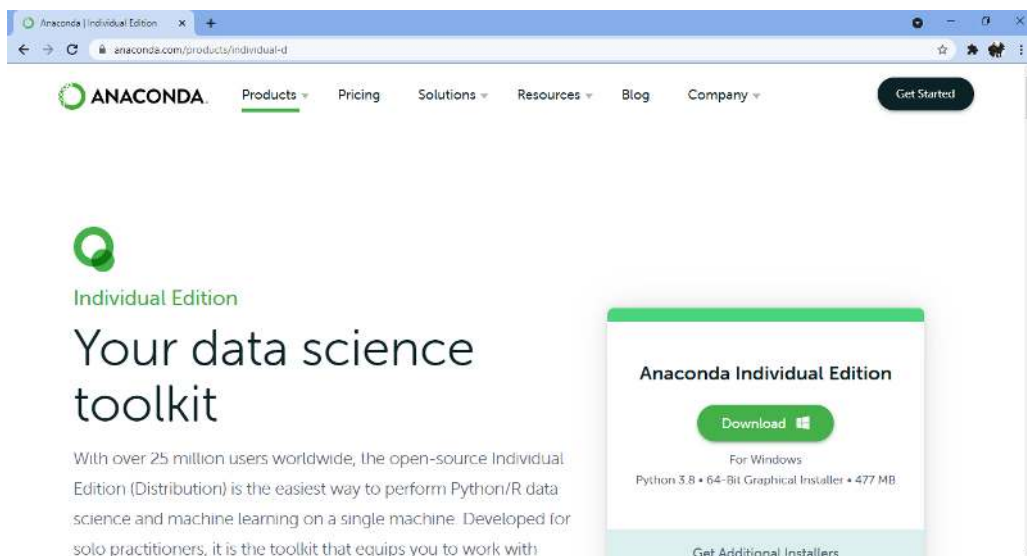
Figure B.2: Click Download

This will download the Anaconda Python package to your workstation. It will automatically give you the installer according to your OS (Windows, Linux, or MacOS). The file is about 480 MB. You should have a file with a name like:

```
Anaconda3-2021.05-Windows-x86_64.exe
```

B.3   Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

1. Double click the downloaded file.
2. Follow the installation wizard.
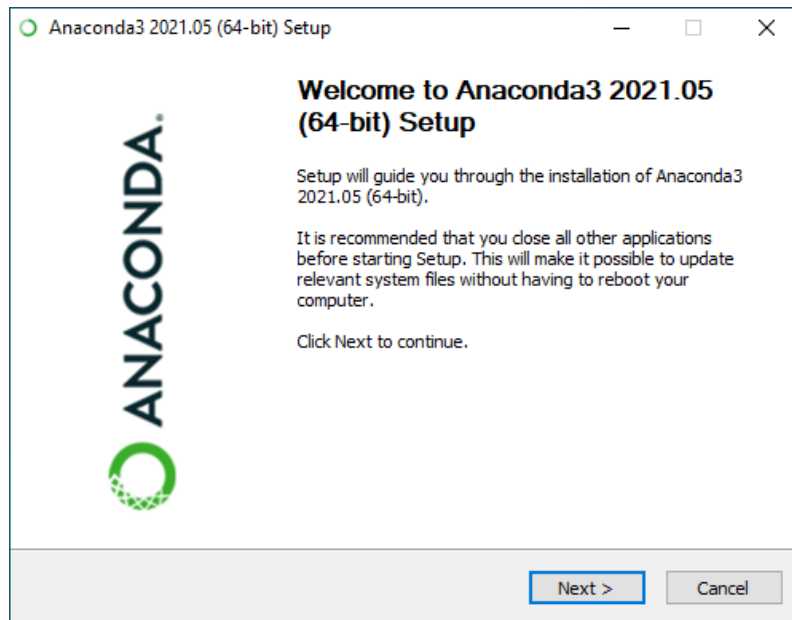
Figure B.3: Anaconda Python Installation Wizard

Installation is quick and painless. There should be no tricky questions or sticking points.
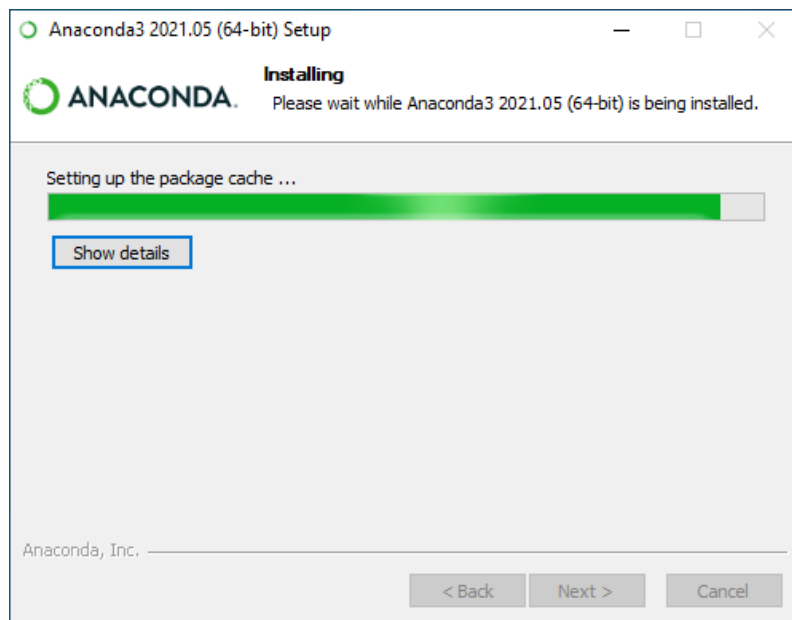
Figure B.4: Anaconda Python Installation Wizard Writing Files

The installation should take less than 10 minutes and take a bit more than 5 GB of space on your hard drive.

B.4   Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.
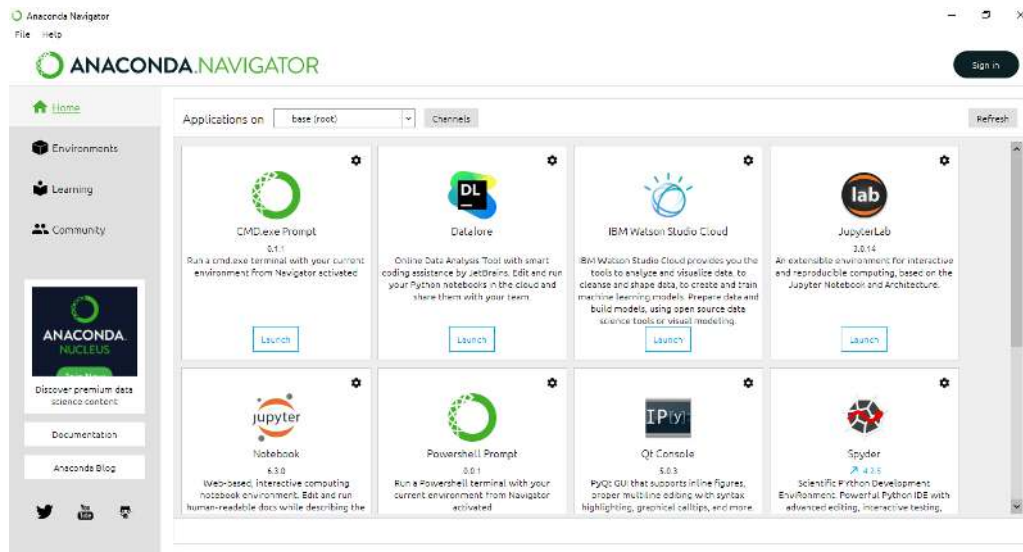
Figure B.5: Anaconda Navigator GUI

You can learn all about the Anaconda Navigator here: `https://docs.continuum.io/an aconda/navigator.html`. You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda[1]. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

1. Open a terminal or CMD.exe prompt (command line window).
2. Confirm conda is installed correctly, by typing:

```
conda -V
```

You should see the following (or something similar):

```
conda 4.10.1
```

---

[1]`https://conda.pydata.org/docs/index.html`

3. Confirm Python is installed correctly by typing:

```
python -V
```

You should see the following (or something similar):
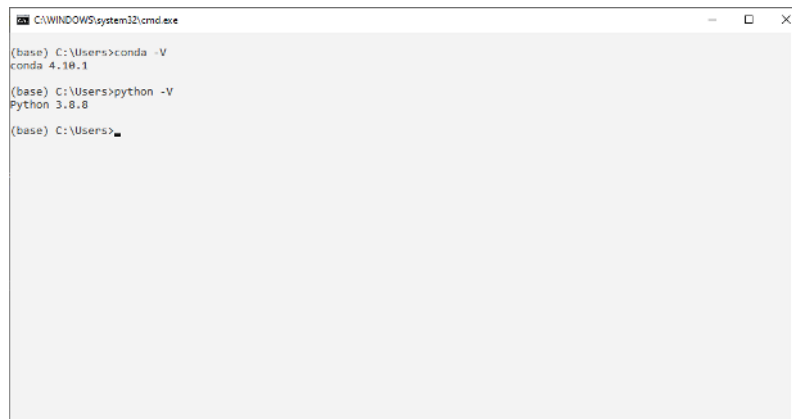
```
Python 3.8.8
```


Figure B.6: Confirm Conda and Python are Installed

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the "Further Reading" section.

4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

You may need to install some packages and confirm the updates.

5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type "python" and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```python
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
```

```
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Save the script as a file with the name: versions.py. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

You should see output like the following:

```
scipy: 1.7.1
numpy: 1.20.3
matplotlib: 3.4.2
pandas: 1.3.2
statsmodels: 0.12.2
sklearn: 0.24.2
```
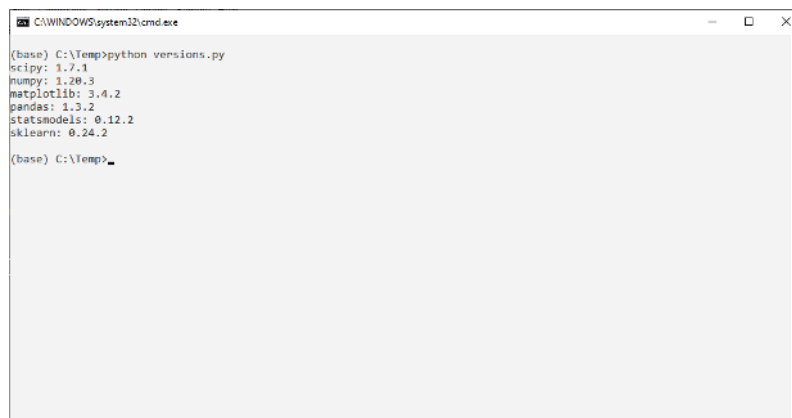

Figure B.7: Confirm Anaconda SciPy environment
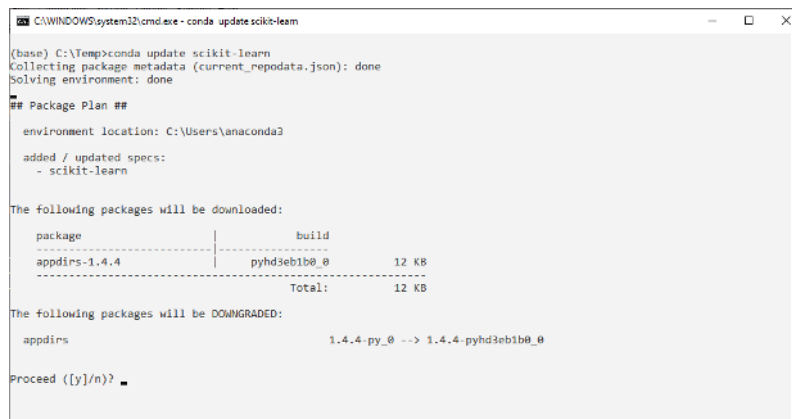
B.5 Update Library

In this step, we will see how can we update the library used for machine learning in Python.

If we ever found a new version of a library (such as scikit-learn) released, we can update it using the conda command; below is an example of updating scikit-learn to the latest version.

At the terminal, type:

```
conda update scikit-learn
```


Figure B.8: Update scikit-learn in Anaconda

Alternatively, you can update a library to a specific version by typing:

```
conda install scikit-learn=0.24.2
```

After you have the updated libraries, you can try a a scikit-learn tutorial, such as:

▷ Your First Machine Learning Project in Python Step-By-Step
https://machinelearningmastery.com/machine-learning-in-python-step-by-step/

B.6 Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: TensorFlow, and Keras.

1. Install the TensorFlow deep learning library and Keras by typing:

```
conda install tensorflow
```

Alternatively, you may choose to install using `pip` and a specific version of tensorflow for your platform.

See the installation instructions for tensorflow <https://www.tensorflow.org/install/pip>

2. Install Keras similarly by typing:

```
conda install keras
```

3. Confirm your deep learning environment is installed and working correctly.

   Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```python
# keras
import keras
print('keras: %s' % keras.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
```

Save the script to a file deep_versions.py. Run the script by typing:

```
python deep_versions.py
```

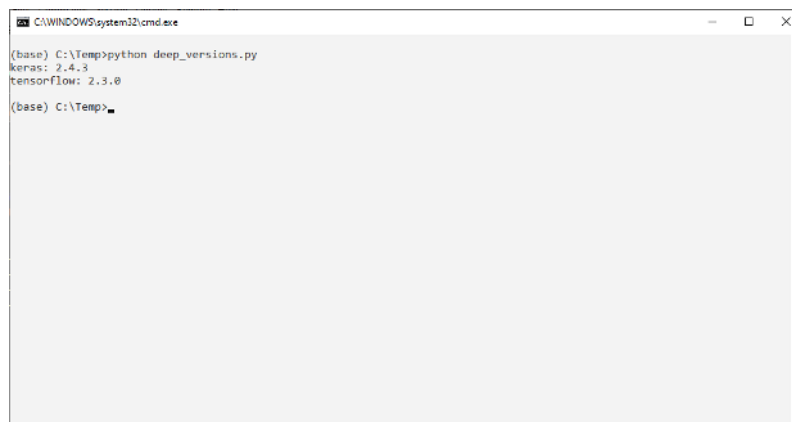You should see output like:

```
keras: 2.4.3
tensorflow: 2.3.0
```


Figure B.9: Anaconda Confirm Deep Learning Libraries

Try a Keras deep learning tutorial, such as:

▷ Develop Your First Neural Network in Python With Keras Step-By-Step
https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

B.7 Further Reading

This section provides some links for further reading.

▷ Anaconda Documentation
https://docs.continuum.io/

▷ Anaconda Documentation: Installation
https://docs.continuum.io/anaconda/install

▷ Conda
https://conda.pydata.org/docs/index.html

▷ Using conda
https://conda.pydata.org/docs/using/

▷ Anaconda Navigator
https://docs.continuum.io/anaconda/navigator.html

▷ Install TensorFlow
https://www.tensorflow.org/install/pip

▷ Keras Installation
https://keras.io/getting_started/

B.8 Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now 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

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable foundational skills in function optimization. You can now confidently:

▷ Understand the optimization algorithm in machine learning papers.

▷ Implement the optimization descriptions of machine learning algorithms.

▷ Describe the optimization operations of your machine learning models.

The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your optimization journey. I hope you keep learning and have fun as you continue to master machine learning.