Optimization Techniques for Machine Learning

Alassane KONE
(ISE 2019 – Senior Data Scientist)

February 2023

Chapter 1

ML Optimization Foundations

A gentle introduction to function optimization and its relationship with machine learning.

1. Why optimize?

What is Function Optimization ?

Function optimization is a subfield of mathematics, and in modern times is addressed using numerical computing methods.

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.

Machine learning is about developing predictive models. Whether one model is better than another, we have some evaluation metrics to measure a model's performance subject to a particular data set. In this sense, if we consider the parameters that created the model as the input, the inner algorithm of the model and the data set in concern as constants, and the metric that evaluated from the model as the output, then we have a function constructed.

The function here does not mean you need to explicitly define a function in the programming language. A conceptual one is sufficed. What we want to do next is to manipulate on the input and check the output until we found the best output is achieved. In case of machine learning, the best can mean:

- Highest accuracy, or precision, or recall
- Greatest F1 score in classification or R2 score in regression
- Least error, or log-loss

What is Function Optimization ?

Function optimization involves three elements:

- The input to the function (x): The input to the function to be evaluated, i.e. a candidate solution
- The objective function itself (f(x)): The objective function or target function that evaluates inputs.
- The output from the function (cost, y): The result of evaluating a candidate solution with the objective function, minimized or maximized

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.)

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.

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

2. Optimization in a Machine Learning Project

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**.

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.

In that case, Function inputs are sequences of transforms that are applied to data.

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.

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**.

In that case, Function inputs are algorithm hyperparameters.

Model Selection as Optimization

Model selection involves choosing one from among many candidate machine learning models for a predictive modeling problem.

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.

In that case, Function inputs are data transform, machine learning algorithm, and algorithm hyperparameters.

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.

3. How to choose an optimization algorithm ?

Optimization Algorithms Division

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.

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.

Differentiable Objective Function

A differentiable function 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.

▷ **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.

▷ **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.

Differentiable Objective Function

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**

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.

- ▷ **Fibonacci Search**
- ▷ **Golden Section Search**
- ▷ **Bisection Method**

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.

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).

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 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**

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.

▷ **Davidson-Fletcher-Powell**

▷ **Broyden-Fletcher-Goldfarb-Shanno (BFGS)**

▷ **Limited-memory BFGS (L-BFGS)**

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.

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**

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.

Examples of direct search algorithms include:

- ▷ **Cyclic Coordinate Search**
- ▷ **Powell’s Method**
- ▷ **Hooke-Jeeves Method**
- ▷ **Nelder-Mead Simplex Search****

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 can handle problems with deceptive local optima.

Stochastic optimization algorithms include:

- ▷ **Simulated Annealing**
- ▷ **Evolution Strategy**
- ▷ **Cross-Entropy Method**

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. Optimization in Scipy

Scipy Overview

SciPy is a scientific computing library for Python that provides a collection of functions and algorithms for numerical computing, scientific computing, and data analysis. It is built on top of the NumPy library and extends its functionality by providing additional modules for optimization, signal processing, linear algebra, integration, interpolation, statistics, and more.

SciPy is open source and free to use, and it has become one of the most popular libraries for scientific computing in Python. It is used by researchers, engineers, and data scientists to perform complex calculations, solve differential equations, optimize algorithms, and analyze data.

Application 1 : Nelder-Mean in Scipy

Let’s start with this function:

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

This is a function with two-dimensional input (x, y) and one-dimensional output.

1. Import *minimize* from scipy.optimize and numpy and *rand* from numpy.random and use the built-in function *help* to see what each function does.
2. Create a function called objective with takes as input an array x. (Note that x is a 2D-array and x[0] is the abscissa and x[1] is the ordinate. The function should return the calculation made by the function f.
3. Define the range for input to be $[-5; 5]$. This mean to create two variable r_min and r_max where r_min = -5 and r_max = 5.
4. Define in a variable called start, the starting point as a random sample from the domain. Use the imported rand function.
5. Using minimize, perform the search of the optimal solution using the objective function, the starting point and set the method argument to “nelder-mean”. Put the output in a variable called result.
6. Summarize the result i.e. show the “message”, the “nfev” and the “solution” from the variable result.
7. Print the image of the optimal solution by the function f.

Solution 1 : Nelder-Mean in Scipy

```
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))
```

Application 2 : Nelder-Mean in Scipy

Restart the previous application with this function:

$$f(x,y) = -20\, exp(-0.2 \times \sqrt{\left(0.5 \times (x^2 + y^2)\right)}\,) - exp(0.5 \times (cos(2\pi x) + cos(2\pi y))) + e + 20$$

1. Try repeat your code a few times and observe the output.
2. The global minimum is at [0,0]. However, Nelder-Mead most likely cannot find it because this function has many local minima.
3. This defined the Ackley function.

Application 2 : Nelder-Mean in Scipy

Restart the previous application with this function:

$$f(x,y) = -20\, exp(-0.2\ \times \sqrt{\left(0.5 \times (x^2 + y^2)\right)}\,) - \ exp(0.5\ \times (cos(2\pi x)\ +\ cos(2\pi y)))\ +\ e\ +\ 20$$

1. Try repeat your code a few times and observe the output.
2. The global minimum is at [0,0]. However, Nelder-Mead most likely cannot find it because this function has many local minima.
3. This defined the Ackley function.

The Ackley function is a mathematical function commonly used as a benchmark in optimization and search algorithms. It is named after its creator, Jeffery C. Ackley.

The Ackley function has the following properties:

- It is a multimodal function, meaning it has multiple local minima.
- The global minimum is located at (0,0) with a function value of 0.
- The function is symmetric about the origin, which means that the value of the function at (-x,-y) is the same as the value at (x,y).
- The function has a large flat region around the global minimum, which can make it difficult for optimization algorithms to find the minimum.

Application 3 : BFGS Algorithm in Scipy

We will use again the same function:

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

we can tell that the first-order derivative is: $\nabla f = [2x, 2y]$

1. Create a function called objective with takes as input an array x as previously and another function called derivative with takes as input an array x and return the gradient defined above.
2. Follow all the steps as in the application 1.
3. Using minimize, perform the search of the optimal solution using the objective function, the starting point and set the method argument to "BGFS" and the argument jac to derivative. Put the output in a variable called result.
4. Summarize the result i.e. show the "message", the "nfev" and the "solution" from the variable result.
5. Print the image of the optimal solution by the function f.

Solution 3 : BFGS in Scipy

1

```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))
```