Санкт-Петербургский политехнический университет Петра Великого
Институт металлургии, машиностроения и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня
Тема: Метод градиентного спуска

Выполнил студент гр. 3331506/70401 Архипов А.Е.

Преподаватель Ананьевский М.С.

«___»_____ 2020 г.

Санкт-Петербург
2020

Содержание

1. Формулировка задачи, которую решает алгоритм 3
2. Словесное описание алгоритма 3
3. Реализация алгоритма 4
4. Анализ алгоритма 6
5. Применение алгоритма 8
6. Заключение 8

1. Формулировка задачи, которую решает алгоритм

Градиентный спуск — один из наиболее популярных методов оптимизации в машинном обучении. Оптимизация — это процесс нахождения точек максимума/минимума некоторой функции. Задача оптимизации может быть представлена следующим образом:

Дано:

- множество $X = \{x_0, x_1, \ldots, x_m\} \subset R^n$
- функция $f : X \to \mathbb{R}$

Тогда решить задачу оптимизации означает одно из:

- Показать, что $X = \emptyset$
- Найти такие $x_0, x_1, \ldots, x_m \in X$, что $f(x_0, x_1, \ldots, x_m) = min\ f(x)$

В данной работе будет рассмотрен не совсем классический метод градиентного спуска, а его модернизированный вариант, под названием метод наискорейшего спуска. Программа позволяет оптимизировать функции с любым количеством переменных.

2. Словесное описание алгоритма

1. Задать начальное приближение x_0, и точность ε
2. Рассчитать градиент $\nabla f(\bar{x}^j)$, частные производные берутся с помощью численного дифференцирования центрально-разностной формуле первого порядка.

$$\frac{\partial f(x_1, \ldots, x_i, \ldots, x_n)}{\partial x_i} = \frac{f(x_1, \ldots, x_i + \Delta, \ldots, x_n) - f(x_1, \ldots, x_i - \Delta, \ldots, x_n)}{2\Delta}$$

3. Рассчитать единичный вектор направления функции $\bar{S}^j$ по формуле

$$\bar{S}^j = \frac{\nabla f(\bar{x}^j)}{\|\nabla f(\bar{x}^j)\|}$$

4. Итерационное нахождение экстремума функции

$$\bar{x}^{j+1} = \bar{x}^j \pm \lambda^j \cdot \bar{S}^j$$

5. Шаг λ^j проверяется и при необходимости корректируется в каждой итерации:

$$\text{если } f(\bar{x}^j) \geq \bar{S}^j, \text{ то } \lambda^{j+1} = \lambda^j / 2$$

6. Условие останова:

$$\lambda^j < \varepsilon$$

3. Реализация алгоритма

Листинг программы с реализацией алгоритма градиентного спуска и текстового пользовательского интерфейсом приведен ниже.

```
#include <iostream>
#include <cmath>

using namespace std;

typedef double D;

class Model
{
public:
    D *variable;

    Model();
    D Function();
};

Model :: Model()
{
    variable = new D[3];
    // initial guess
    variable[0]=1.3;
    variable[1]=1.;
    variable[2]=2.;
}

//Enter a function
D Model :: Function()
{
    return variable[0] * variable[1] + variable[0] * variable[0] +
variable[1] * variable[1];
}

class Optimization : public Model
{
public:
    void implementation_descent();
    void calculate_gradient(const int number_variables, D *gradient, D
delta);
};


void Optimization :: calculate_gradient(const int number_variables, D
*gradient, D delta)
{
    int variable_counter;
    D function_value_right, function_value_left, module_gradient = 0;

    for(variable_counter = 0; variable_counter < number_variables;
variable_counter++)
    {
        variable[variable_counter] += delta;
        function_value_right = Function();
        variable[variable_counter] -= delta;

        variable[variable_counter] -= delta;
        function_value_left = Function();
        variable[variable_counter] += delta;
```

```
            gradient[variable_counter] = (function_value_right -
function_value_left) / (2 * delta);
            module_gradient += gradient[variable_counter] *
gradient[variable_counter];
        }
        module_gradient = sqrt(module_gradient);

        for(variable_counter = 0; variable_counter < number_variables;
variable_counter++)
            gradient[variable_counter] /= module_gradient;

        gradient[number_variables] = Function();
    }

    void Optimization :: implementation_descent()
    {
        int variable_counter, number_iterations= 0;
        D function_value, step, eps;
        const int number_variables = 2;
        D *gradient = new D[number_variables + 1];

        step = 0.1;
        eps = 0.000001;

        while(step > eps)
        {
            calculate_gradient(number_variables, gradient, 0.0001);
            for(variable_counter = 0; variable_counter < number_variables;
variable_counter++)
                variable[variable_counter] -= step *
gradient[variable_counter];

            function_value = Function();
            if(function_value >= gradient[number_variables])
            {
                step /= 2.;
                for(variable_counter = 0; variable_counter <
number_variables; variable_counter++)
                    variable[variable_counter] += step *
gradient[variable_counter];
            }
            number_iterations++;
            cout << number_iterations << " " << variable[0] << "  " <<
variable[1] << "  " << variable[2]<< endl;
        }
    }


    int main()
    {
        Optimization fun;
        fun.implementation_descent();
    }
```

4. Анализ алгоритма

При анализе оптимизационных методов стоит обращать внимание не только на сложность каждой итерации, но и на скорость сходимость. И, как правило, об эффективности методе судят именно по скорости сходимости. В данной работе будет проведено сравнение времени работы и количества итераций у «классического» метода градиентного спуска и метода наискорейшего спуска.

Реализация «классического» метода градиентного спуска

```
void Optimization :: calculate_gradient(const int number_variables, D
*gradient, D delta)
{
    int variable_counter;
    D function_value_right, function_value_left, module_gradient = 0;

    for(variable_counter = 0; variable_counter < number_variables;
variable_counter++)
    {
        variable[variable_counter] += delta;
        function_value_right = Function();
        variable[variable_counter] -= delta;

        variable[variable_counter] -= delta;
        function_value_left = Function();
        variable[variable_counter] += delta;

        gradient[variable_counter] = (function_value_right -
function_value_left) / (2 * delta);
    }

    gradient[number_variables] = Function();
}

void Optimization :: implementation_descent()
{
    int variable_counter, number_iterations= 0;
    D function_value = variable[2];
    D step, eps;
    const int number_variables = 2;
    D *gradient = new D[number_variables + 1];
    D previous_fun = 0;

    step = 0.01;
    eps = 0.0000001;

    while(abs(function_value - previous_fun) > eps)
    {
        previous_fun = function_value;
        calculate_gradient(number_variables, gradient, 0.0001);
        for(variable_counter = 0; variable_counter < number_variables;
variable_counter++)
            variable[variable_counter] -= step * gradient[variable_counter];
        function_value = Function();
        number_iterations++;
```

4.1. Функция $f(x, y) = x^2 + \ x \cdot y + y^2$
Начальное приближение: x = 1,3; y = 1
Шаг у «классического» метода: 0,01
Точность: 0,0001

	Классический метод	Метод наискорейшего спуска
Кол-во итераций	420	42
Время (мс)	963	830
Значение функции	4,85008e-006	1,53998e-008

4.2. Функция $f(x, y) = -\cos(x) \cdot \cos(y) \cdot e^{x+y}$
Начальное приближение: x = 1,3; y = 1
Шаг у «классического» метода: 0,01
Точность: 0,0001

	Классический метод	Метод наискорейшего спуска
Кол-во итераций	131	22
Время (мс)	772	748
Значение функции	-2,40524	-2,40524

4.3. Функция $f(x, y) = x^3 + y^2 - 6xy - 39x + 18y + 20$
Начальное приближение: x = 1,3; y = 1
Шаг у «классического» метода: 0,01
Точность: 0,0001

	Классический метод	Метод наискорейшего спуска
Кол-во итераций	976	167
Время (мс)	772	830
Значение функции	-86	-86

5. Применение алгоритма

Градиентный спуск — метод численной оптимизации, который может быть использован во многих алгоритмах, где требуется найти экстремум функции — искусственные нейронные сети, SVM, k-средних, регрессии.

6. Заключение

Метод градиента вместе с его многочисленными модификациями является распространенным и эффективным методом поиска оптимума исследуемых объектов. Недостатком градиентного поиска (так же и рассмотренных выше методов) является то, что при его использовании можно обнаружить только локальный экстремум функции. Для отыскания других локальных экстремумов необходимо производить поиск из других начальных точек. Так же скорость сходимости градиентных методов существенно зависит также от точности вычислений градиента. Потеря точности, а это обычно происходит в окрестности точек минимума или в овражной ситуации, может вообще нарушить сходимость процесса градиентного спуска.