MINOR PROJECT

TITLE:-How to Solve an optimization problem using gradient descent with an example

INTRODUCTION

Gradient descent- is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function.

- 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.

OVERVIEW

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

Gradient Descent Optimization

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.

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

  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 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.

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.
    - x_new = x – alpha * 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 (*alpha*)**: Hyperparameter that controls how far to move in the search space against the gradient each iteration of the algorithm.

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.

The bounds can be defined along with an objective function as an array with a min and max value for each dimension. The rand() NumPy function can be used to generate a vector of random numbers in the range 0-1.

```
# generate an initial point
solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
```

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

```
# calculate gradient
gradient = derivative(solution) # Indent this line
```

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.

```
# take a step
solution = solution - step_size * gradient
```

We can then evaluate this point and report the performance.

```
# evaluate candidate point
solution_eval = objective(solution)
```

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

```
# run the gradient descent
for i in range(n_iter):
    # calculate gradient
    gradient = derivative(solution) # Indent this line
    # 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]
```

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.

```
# 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) # Indent this line
        # 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]
```

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.

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

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)
```

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.

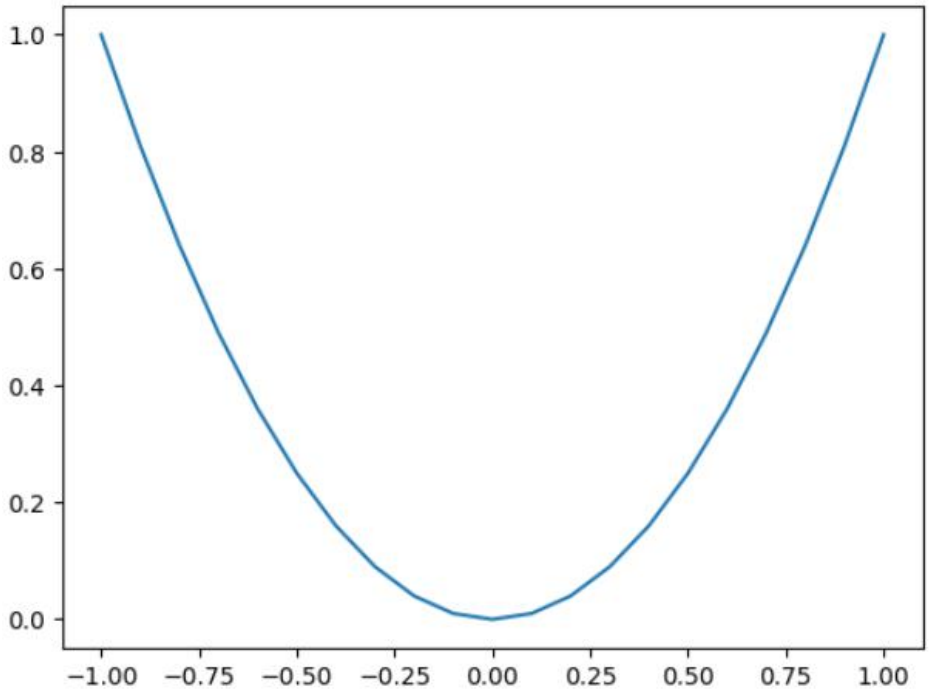
```
# create a line plot of input vs result
pyplot.plot(inputs, results)
# show the plot
pyplot.show()
```

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

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



Next, we can apply the gradient descent algorithm to the problem.

First, we need a function that calculates the derivative for this function.

```
# example of plotting a gradient descent search on a one-dimensional function
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 # Indent this line

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

# 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()
```

```
<ipython-input-8-c515c0669c64>:33: DeprecationWarning: Conversion of an array with ndim > 0
  print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
>0 f([0.2870601]) = 0.08240
>1 f([0.22964808]) = 0.05274
>2 f([0.18371846]) = 0.03375
>3 f([0.14697477]) = 0.02160
>4 f([0.11757982]) = 0.01383
>5 f([0.09406385]) = 0.00885
>6 f([0.07525108]) = 0.00566
>7 f([0.06020087]) = 0.00362
>8 f([0.04816069]) = 0.00232
>9 f([0.03852855]) = 0.00148
>10 f([0.03082284]) = 0.00095
>11 f([0.02465827]) = 0.00061
>12 f([0.01972662]) = 0.00039
>13 f([0.0157813]) = 0.00025
>14 f([0.01262504]) = 0.00016
>15 f([0.01010003]) = 0.00010
>16 f([0.00808002]) = 0.00007
>17 f([0.00646402]) = 0.00004
>18 f([0.00517122]) = 0.00003
>19 f([0.00413697]) = 0.00002
>20 f([0.00330958]) = 0.00001
>21 f([0.00264766]) = 0.00001
>22 f([0.00211813]) = 0.00000
>23 f([0.0016945]) = 0.00000
>24 f([0.0013556]) = 0.00000
>25 f([0.00108448]) = 0.00000
>26 f([0.00086759]) = 0.00000
>27 f([0.00069407]) = 0.00000
>28 f([0.00055525]) = 0.00000
>29 f([0.0004442]) = 0.00000
```