schulzd_Lab8

November 11, 2020

1 Lab 8 - Optimization Methods: Gradient Descent

David Schulz

1.1 Introduction

Optimization problems can be broken into two major components: 1. A cost function that describes the error between the model predictions and given data. This output of this function is dependent on the choice of model parameters and the provided training data. 2. An algorithm that adjusts parameters to either minimize (or maximize) a cost function. In our applications, we typically try to minimize the cost function (model error).

This lab will focus on the algorithm step (#2) through coding/implementation of a numerical gradient descent solver. Additionally, we will perform experiments where we apply this algorithm and explore some implications of solving optimization problems, starting parameter values, and global versus local optima. Gradient descent is an iterative algorithm and is defined by the following steps: 1. Set an appropriate cost function for the problem that you are solving. - Cost functions like those created in lab 06. - Could be any function with adjustable parameters. 2. Make an initial guess of the model parameters, 0. 3. For k iterations: - Compute the gradient, ( ), at current model parameters. - Choose a step size † - Update +1= − ( ) - If change in gradient is < tolerance stop – else go to step 3.

1.2 Questions

1. Reflect on the form and organization of our optimizer API. Specifically, discuss each of the methods and what role they serve. This discussion should include what arguments they accept, what the method returns, and why we might choose to separate out these specific methods into helper methods.
    - ___init___(self, step_size, max_iter, tol, delta): Initializes the Optimizer class and stores step_size, max_iter, tol, and delta for later use.
    - optimize(self, cost_func, starting_params): The main method of the class. Loops to find the smallest result of cost_func's cost() function by gradually adjusting the starting_params.
    - _calculate_change(self, old, new): Returns the euclidean distance between the two parameter vectors, old and new.
    - _gradient(self, cost_func, params): Calculates the gradient at the current state of the params based on the cost_func's cost() function.

- _update(self, param, gradient): Adjusts the param vector based on a percentage, the step_size, of the calculated gradient at the current state of the param vector.
- This class has been separated into helper methods to make it easier to debug and keep track of what does what at each step.

2. For experiment 1:
   - How many optima did you find? Hint: Discuss the significance of places where the derivative is equal to 0.
     - There are 2 optima. When the derivative is zero, the original function changes vertical directions, making it a local minimum or maximum.
   - When you used the optimizer you started at x = 0. How many optima did your optimizer return? Was it a minima or maxima? Was it a global or local optima? By looking at the gradient descent algorithm find what term pointed you toward the minimum. Describe how it did this. Can you think of a way to find the function's maxima?
     - It only returned one optima: the local minima. The computed gradient pointed it toward the minimum because of its sign. To find the maximum, we could try switching the sign of the gradient.
3. For experiment 2:
   - How many optima did you find?
     - There are 3 optima.
   - Describe the different starting locations that you used to solve for optima. Was the found optima different for any of these starting locations and were they the global or local optima? If it was, can you explain why the optimizer found different solutions?
     - The optima for the starting point at x=-2 was different than the one for x=3 and x=6. It was a local optima. This is because the optimizer only knows to look for the closest extrema, local or global.
4. For experiment 3:
   - How many optima did you find?
     - There is only one optima.
   - Look back at the heatmaps you generated in Lab06 for the gaussian distribution. Describe what the optimizer is doing using the heatmap visualization.
     - The optimizer is looking for that point on the grid that results in the smallest possible error.

1.3 Optimizer Testing

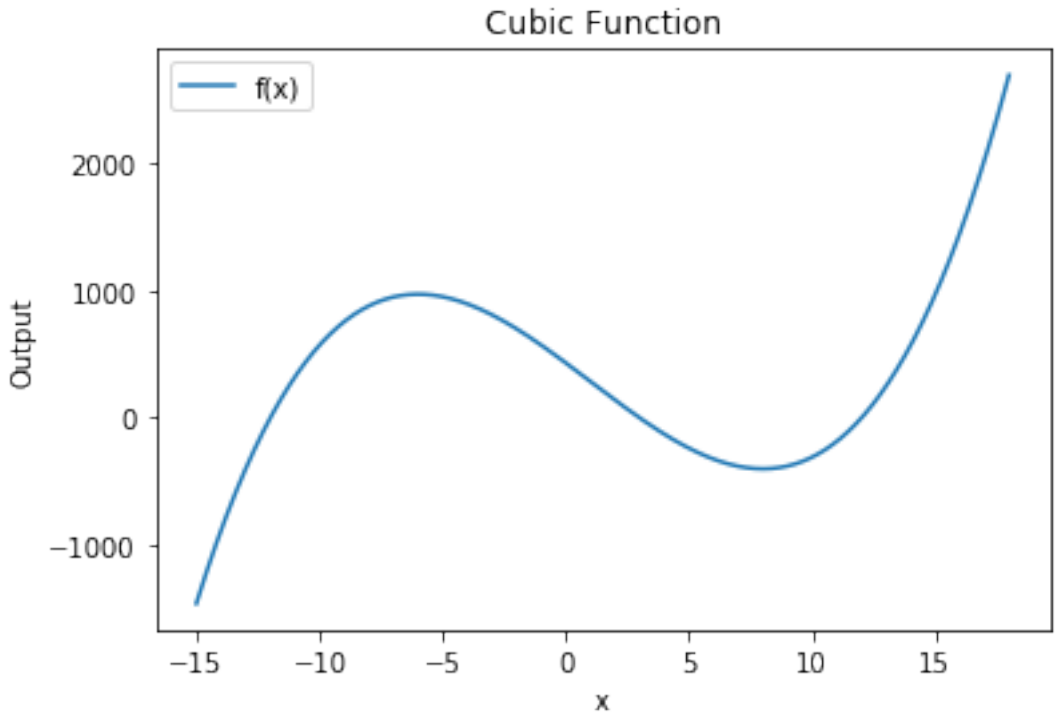
[1]:
```
from test_optim import TestOptimizer

tester = TestOptimizer()
tester.test_gradient_parabola()
tester.test_gradient_paraboloid()
tester.test_update_zero()
tester.test_update_nonzero()
tester.test_optimize_parabola()
tester.test_optimize_parabaloid()
print("All tests passed")
```

```
All tests passed
```

1.4 Experiment 1: Cubic Model

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

def cubic(x):
    return np.power(x, 3) - (3 * np.power(x, 2)) - (144 * x) + 432

x = np.arange(-15.0, 18.0, 0.01)
y = cubic(x)
plt.plot(x, y, label='f(x)')
plt.title('Cubic Function')
plt.xlabel('x')
plt.ylabel('Output')
plt.legend()
```

[2]:
```
<matplotlib.legend.Legend at 0x7f5c4b2f1ad0>
```

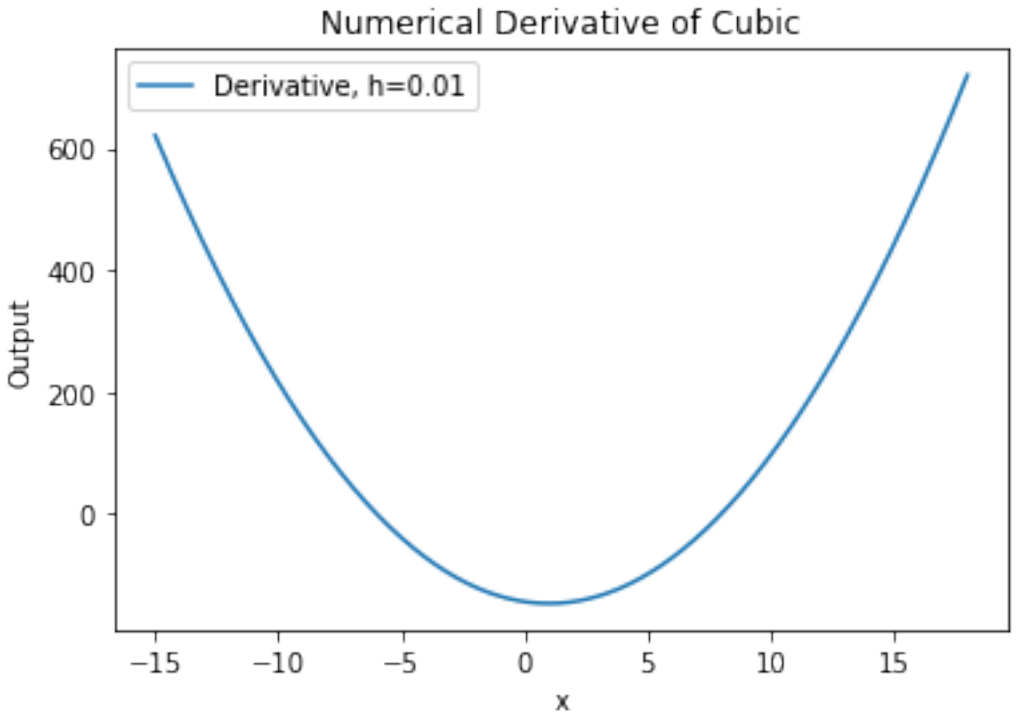


The function appears to have extrema at x=-6 and x=8.

[3]:
```
h = 0.01

cubic_deriv = (cubic(x+h) - cubic(x)) / h
plt.plot(x, cubic_deriv, label='Derivative, h=0.01')
```

```
plt.title('Numerical Derivative of Cubic')
plt.xlabel('x')
plt.ylabel('Output')
plt.legend()
```

[3]: <matplotlib.legend.Legend at 0x7f5c4a412310>



[4]:
```
ind = np.argpartition(np.abs(cubic_deriv), 2)
extrema_x = x[ind[:2]]
extrema_y = cubic_deriv[ind[:2]]
print("(" + str(extrema_x[0]) + ", " + str(extrema_y[0]) + ")")
print("(" + str(extrema_x[1]) + ", " + str(extrema_y[1]) + ")")
```

```
(-6.000000000000192, -0.2098999999930129)
(7.989999999999512, -0.20990000003848763)
```

[5]:
```
class CubicCostFunction:
    def cost(self, params):
        """
        Implements the cubic function:

        y = x**3 - 3x**2 - 144x + 432

        The extrema should be at x = 8 and x = -6
```

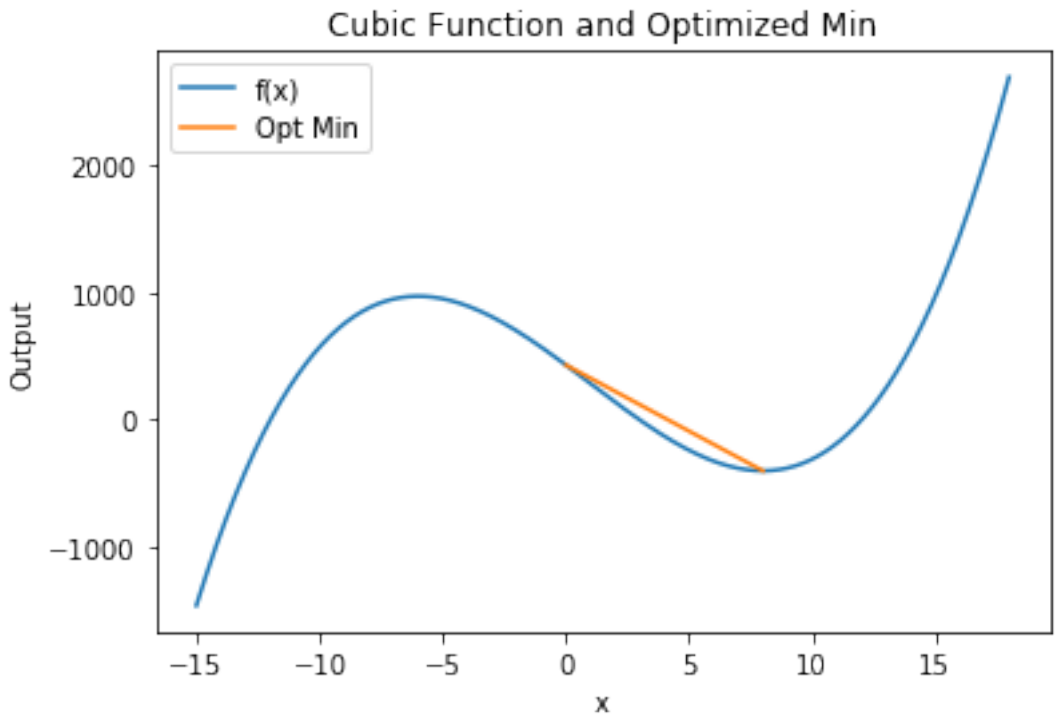
```
        """
        return np.power(params[0], 3) - (3 * np.power(params[0], 2)) - (144 *␣
↪params[0]) + 432
```

[6]:
```
from optim import Optimizer

cost_func = CubicCostFunction()
opt = Optimizer(0.01, 100, 1e-5, 1e-4)
start_params = np.zeros(1, dtype='float32')
opt_params, iters = opt.optimize(cost_func, start_params)
print("New params: " + str(opt_params))
print("Iterations: " + str(iters))
```

```
New params: [7.99993677]
Iterations: 27
```

[7]:
```
min_xs = np.array([0., opt_params[0]])
min_ys = np.array([cubic(0.), cubic(opt_params[0])])

plt.plot(x, y, label='f(x)')
plt.plot(min_xs, min_ys, label='Opt Min')
plt.title('Cubic Function and Optimized Min')
plt.xlabel('x')
plt.ylabel('Output')
plt.legend()
```

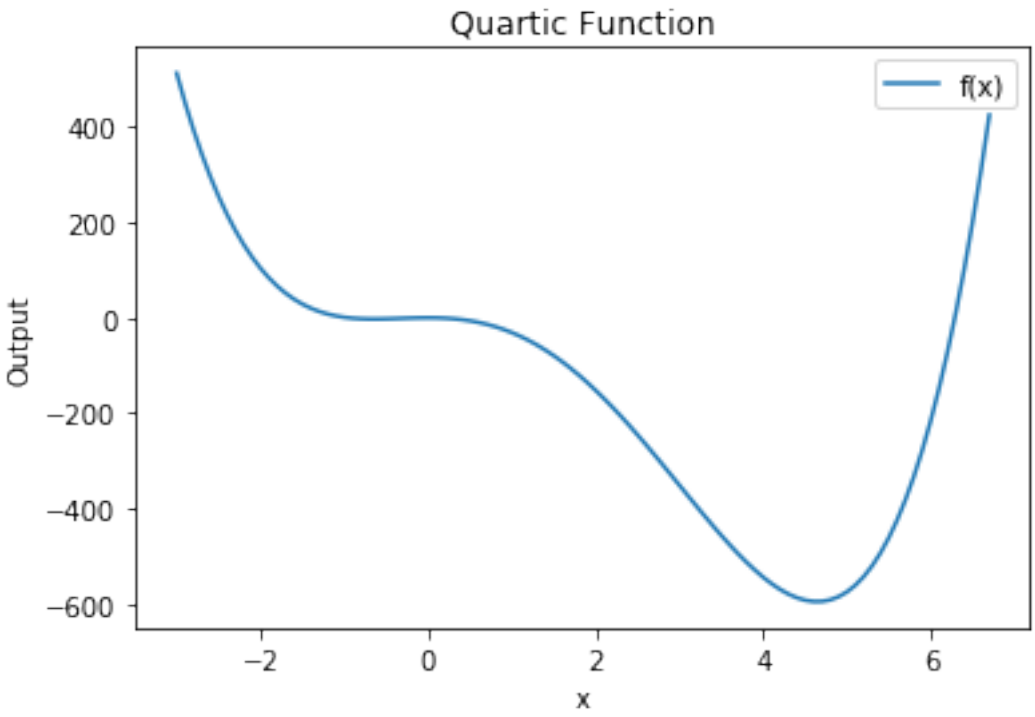
[7]:
```
<matplotlib.legend.Legend at 0x7f5c4a37d710>
```



1.5 Experiment 2: Quartic Model

[8]:
```
def quartic(x):
    return (3 * np.power(x, 4)) - (16 * np.power(x, 3)) - (18 * np.power(x, 2))

x = np.arange(-3.0, 6.75, 0.1)
y = quartic(x)
plt.plot(x, y, label='f(x)')
plt.title('Quartic Function')
plt.xlabel('x')
plt.ylabel('Output')
plt.legend()
```

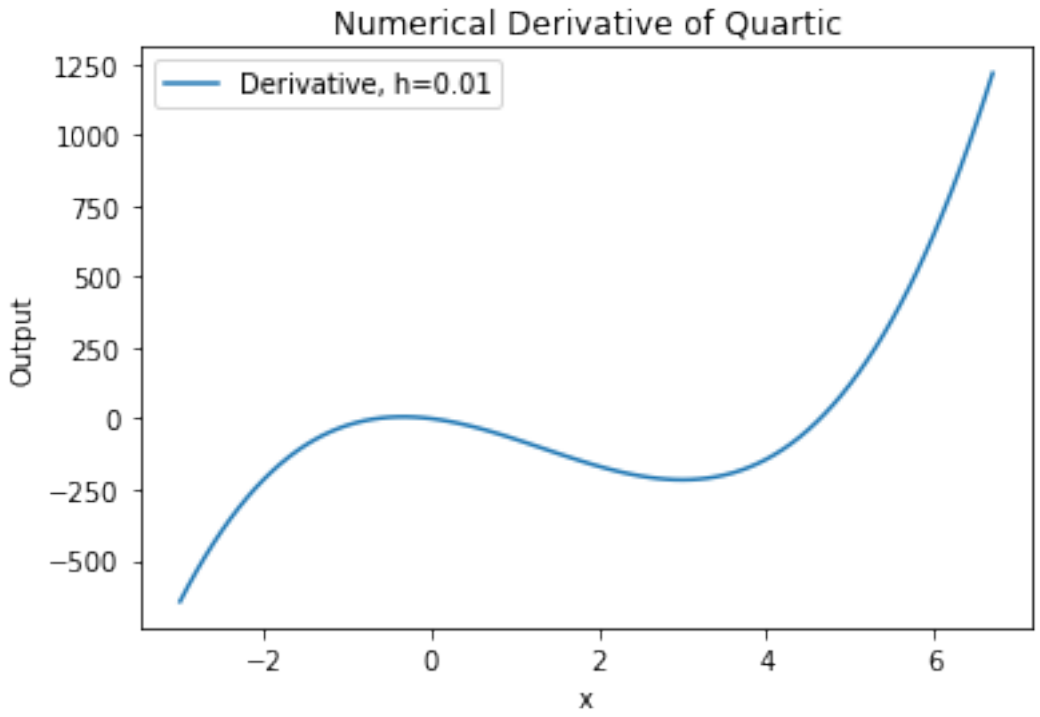
[8]:
```
<matplotlib.legend.Legend at 0x7f5c4a30a750>
```



The function appears to have extrema at around x=-1, x=0, and x=4.5

[9]:
```
h = 0.01

quart_deriv = (quartic(x+h) - quartic(x)) / h
plt.plot(x, quart_deriv, label='Derivative, h=0.01')
plt.title('Numerical Derivative of Quartic')
plt.xlabel('x')
plt.ylabel('Output')
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f5c4a265090>
```



[10]:
```
ind = np.argpartition(np.abs(quart_deriv), 3)
extrema_x = x[ind[:3]]
extrema_y = quart_deriv[ind[:3]]
print("(" + str(extrema_x[0]) + ", " + str(extrema_y[0]) + ")")
print("(" + str(extrema_x[1]) + ", " + str(extrema_y[1]) + ")")
print("(" + str(extrema_x[2]) + ", " + str(extrema_y[2]) + ")")
```

```
(2.6645352591003757e-15, -0.18159700000009718)
(-0.5999999999999979, 1.8984830000001285)
(-0.699999999999998, -2.1942369999998768)
```

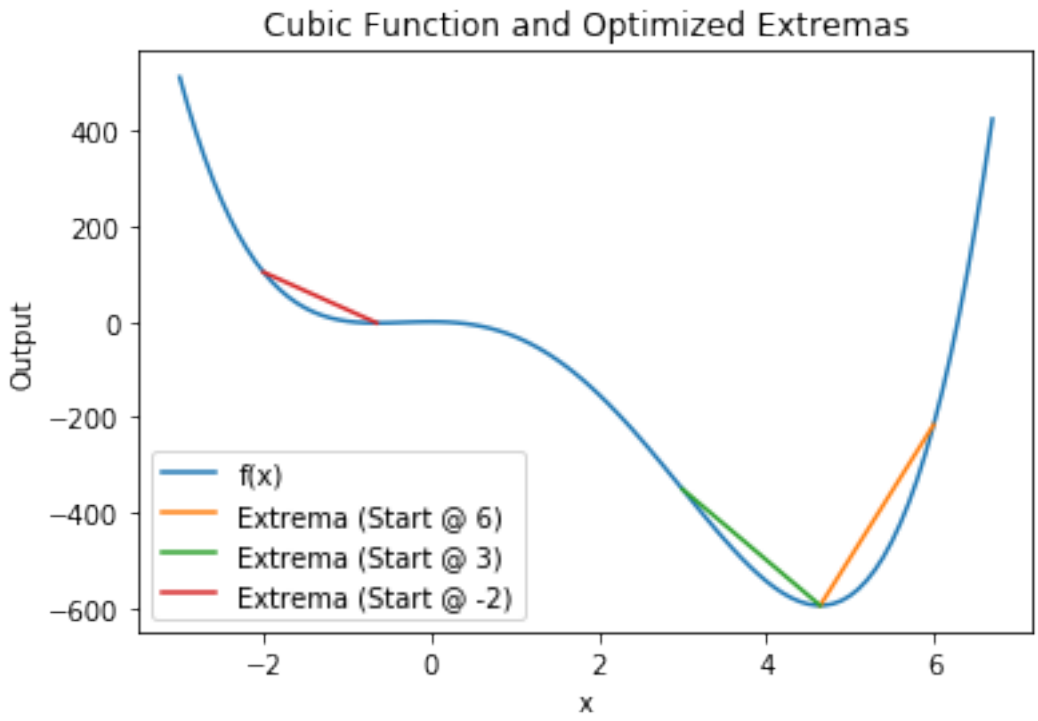
[11]:
```
class QuarticCostFunction:
    def cost(self, params):
        """
        Implements the quartic function:

        y = 3x**4 - 16x**3 - 18x**2

        The extrema should be at x=-0.646, x=0, and x=4.646
        """
        return (3 * np.power(params[0], 4)) - (16 * np.power(params[0], 3)) -␣
↪(18 * np.power(params[0], 2))
```

[12]:
```
cost_func = QuarticCostFunction()
opt = Optimizer(0.001, 1000, 1e-5, 1e-4)
start_params = np.array([6.0])
opt_params_6, iters = opt.optimize(cost_func, start_params)
print("New params (Start @ 6): " + str(opt_params_6))
print("Iterations (Start @ 6): " + str(iters))

start_params = np.array([3.0])
opt_params_3, iters = opt.optimize(cost_func, start_params)
print("New params (Start @ 3): " + str(opt_params_3))
print("Iterations (Start @ 3): " + str(iters))

start_params = np.array([-2.0])
opt_params_n2, iters = opt.optimize(cost_func, start_params)
print("New params (Start @ -2): " + str(opt_params_n2))
print("Iterations (Start @ -2): " + str(iters))
```

```
New params (Start @ 6): [4.64572056]
Iterations (Start @ 6): 30
New params (Start @ 3): [4.64568186]
Iterations (Start @ 3): 36
New params (Start @ -2): [-0.64603101]
Iterations (Start @ -2): 176
```

[13]:
```
min_xs_6 = np.array([6., opt_params_6[0]])
min_ys_6 = np.array([quartic(6.), quartic(opt_params_6[0])])
min_xs_3 = np.array([3., opt_params_3[0]])
min_ys_3 = np.array([quartic(3.), quartic(opt_params_3[0])])
min_xs_n2 = np.array([-2., opt_params_n2[0]])
min_ys_n2 = np.array([quartic(-2.), quartic(opt_params_n2[0])])

plt.plot(x, y, label='f(x)')
plt.plot(min_xs_6, min_ys_6, label='Extrema (Start @ 6)')
plt.plot(min_xs_3, min_ys_3, label='Extrema (Start @ 3)')
plt.plot(min_xs_n2, min_ys_n2, label='Extrema (Start @ -2)')
plt.title('Cubic Function and Optimized Extremas')
plt.xlabel('x')
plt.ylabel('Output')
plt.legend()
```

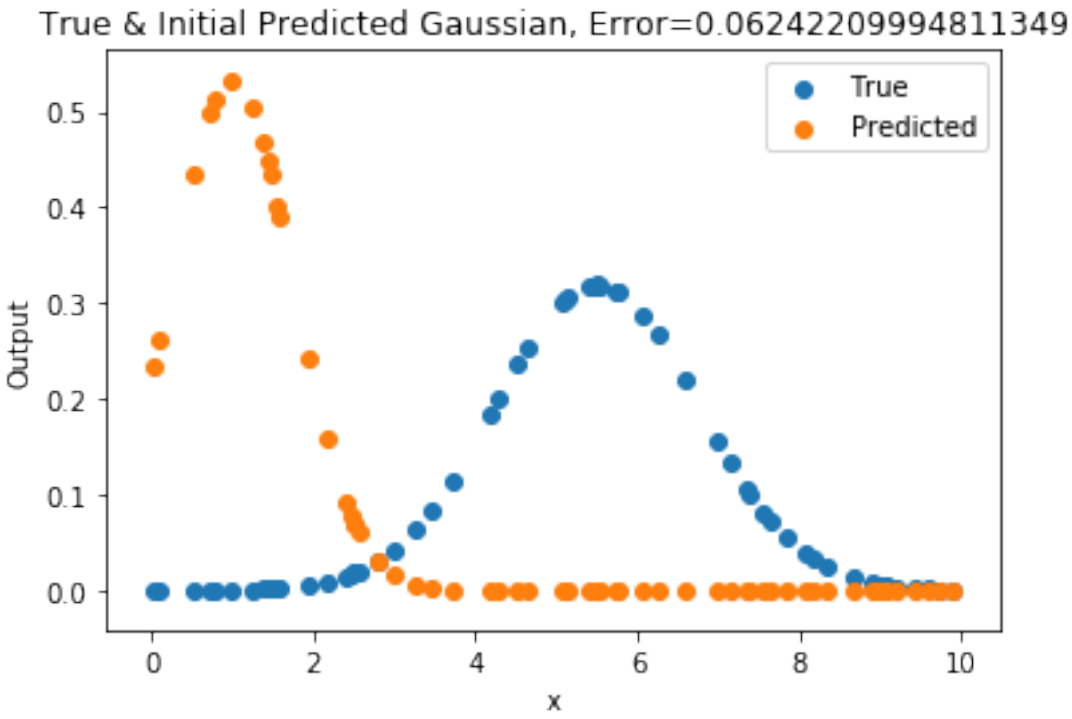
[13]:
```
<matplotlib.legend.Legend at 0x7f5c4a1e2890>
```



1.6 Experiment 3: Gaussian Model

[14]:
```
import pandas as pd
from cost_functions import GaussianCostFunction

gauss = pd.read_csv('gaussdist.csv', header=None)
gauss = gauss.to_numpy()

x = gauss[:, 0]
y = gauss[:, 1]

gcf = GaussianCostFunction(x, y)
start_params = np.array([1.0, 0.75])
pred = gcf._predict(x, start_params)
error = gcf.cost(start_params)

plt.xlabel('x')
plt.ylabel('Output')
plt.title('True & Initial Predicted Gaussian, Error=' + str(error))
plt.scatter(x, y, label='True')
plt.scatter(x, pred, label='Predicted')
plt.legend()
```

[14]: <matplotlib.legend.Legend at 0x7f5c492aa110>

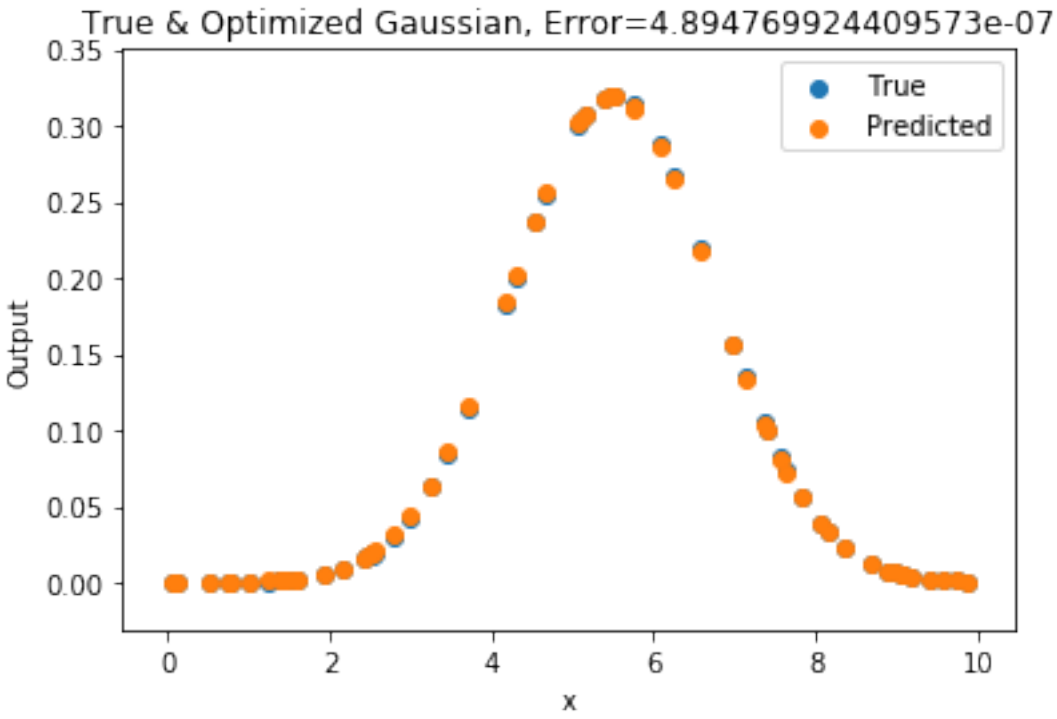


[15]:
```
opt = Optimizer(1, 5000, 1e-4, 1e-3)
opt_params, iters = opt.optimize(gcf, start_params)
print("New params: " + str(opt_params))
print("Iterations: " + str(iters))
```

```
New params: [5.49045906 1.2506877 ]
Iterations: 3031
```

[16]:
```
pred = gcf._predict(x, opt_params)
error = gcf.cost(opt_params)

plt.xlabel('x')
plt.ylabel('Output')
plt.title('True & Optimized Gaussian, Error=' + str(error))
plt.scatter(x, y, label='True')
plt.scatter(x, pred, label='Predicted')
plt.legend()
```

[16]: <matplotlib.legend.Legend at 0x7f5c483ad110>


True & Optimized Gaussian, Error=4.894769924409573e-07
True
Predicted
Output
x
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10