Applications of Gradient Descent

Serghie Lucas

May 27, 2024

Outline

The Algorithm

Before introducing the application we need to ensure that we have all the necessary tools for understanding it.

Definition

For any differentiable function f with starting point x_1, the basic gradient descent method is defined as follows

$$x_{i+1} = x_i - \eta_i f(x) \qquad i = 1, 2, ...$$

where the symbol η is a Greek letter pronounced "eta" and it represents the step size or more other referred to as learning rate, which varies with respect to i.

Lipschitz functions

To ensure effective analysis of the convergence of a function, it is essential to control the magnitude of the gradient across its domain. This control is created with the help of Lipschitz functions, which provide a way for bounding the rate of change of functions. By using these functions, we can establish an upper limit on the gradient.

Definition

We call a function $f : S \to \mathbb{R}$ a Lipschitz function if $\exists \beta$ constant such that $\forall x, y \in S$

$$|f(x) - f(y)| \leq \beta ||x - y||$$

Smooth functions

Definition

A function $f : S \to \mathbb{R}$, where $S \subseteq \mathbb{R}^n$ is said to be $\beta - smooth$ if the gradient of f is Lipschitz continuous, that is $\forall x, y \in S$

$$|\nabla f(x) - \nabla f(y)| \leq \beta ||x - y||$$

This definition states that as we take any two points x and y from the domain, the difference in the gradients does not vary more than the constant β allows it to, thus enabling us to control the curvature of the function.

Smooth functions

Theorem

Let f be a $\beta -$ smooth function on $S \subseteq \mathbb{R}^n$, then $\forall x, y \in S$

$$|f(y) - f(x) - \nabla f(x)^t (y - x)| \leq \frac{\beta}{2} ||y - x||^2$$

This lemma is very important since it allows us to choose any value for y. By combining this lemma with the iterative rule of the gradient descent algorithm we can choose $y = x - \frac{1}{\beta} \nabla f(x)$ to prove that the update step decreases the function proportional to the squared norm of the gradient, which essentially means that the update step regulates itself as the function approaches the minimum.

$$f(y) - f(x) \leq -\frac{1}{2\beta} ||\nabla f(x)||^2$$

Machine-Learning

The algorithm of gradient descent is implemented in a constantly growing number of industries, but probably the most common one is machine learning, where it is used to minimize certain cost functions in order to be able to make predictions.

In the following slides, we will analyze the gradient descent method and its convergence using different approaches for step sizes.

Appliaction

We start the algorithm by initializing our training set. For the sake of visualization, we will only take inputs with the number of features equal to two and one expected value.

```
# Data
X = np.array([(365, 28), (354, 28), (380, 53), (411, 61), (472, 88), (418, 60), (397, 55), (375, 35), (407, 61), (490, 100),
(403, 59), (423, 62), (398, 56), (475, 91), (338, 16), (475, 81), (343, 18), (386, 49), (360, 44), (363, 35),
(380, 44), (388, 58), (397, 62), (369, 50), (331, 16), (380, 49), (467, 78), (402, 53), (332, 24), (453, 87),
(400, 55), (444, 66), (365, 37), (412, 70), (457, 77), (406, 50), (319, 10), (425, 68), (322, 10), (390, 50),
(343, 26), (333, 29), (329, 19), (416, 64), (432, 77), (474, 99), (398, 52), (439, 77), (404, 62), (360, 26)])  # Features

y = np.array([
850, 785, 815, 914, 1288, 1040, 911, 778, 1021, 1230,
970, 1073, 1020, 1156, 774, 1157, 688, 872, 875, 923,
787, 922, 948, 798, 605, 886, 1214, 1004, 698, 1078,
991, 1070, 750, 1027, 1054, 968, 608, 953, 641, 821,
657, 772, 700, 996, 1052, 1231, 875, 1099, 1087, 709])  # Target variable
```

In order to account for the case when the linearity we need to add a bias column of 1's to the X set. After that, we can initialize the weights.

```
# Add bias term to features
X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Initialize weights for each method separately
W_const = np.zeros(3)
W_lipschitz = np.zeros(3)
W_adjusted = np.zeros(3)
```

Appliaction

As we cost function we have defined a simple MSE (mean squared error) and also its gradientMSE method.

```
def predict(X, W):
    return X.dot(W)

def MSE(errors):
    return (1/(2 * len(errors))) * sum(error**2 for error in errors)

def gradientMSE(m, X, errors):
    return (1/m) * X.T.dot(errors)
```

As mentioned earlier, we will compute the gradient descent method using 3 different types of step sizes. As it has been proven already, using gradient descent on β-smooth functions leads to faster convergence rates

1. Lipschitz Step

We start by defining the values of the parameters a and b, also the number of iterations, and proceed to call the `compute_lipschitz_constant` method.

```
# Parameters
a, b = -0.3, 0.3
iterations = 10  # Set the number of iterations to 50

# Compute Lipschitz constant learning rate
lipschitz_learning_rate = 1/compute_lipschitz_constant(len(y), X_b, y, a, b)
```

```
def compute_lipschitz_constant(m, X, y, a, b, num_points=10):
    x1_vals = np.linspace(a, b, num_points)
    x2_vals = np.linspace(a, b, num_points)
    y_vals = np.linspace(a, b, num_points)
    X1, X2, Y = np.meshgrid(x1_vals, x2_vals, y_vals)
    points = np.vstack([X1.ravel(), X2.ravel(), Y.ravel()]).T

    errors_matrix = np.apply_along_axis(lambda point: predict(X, point) - y, 1, points)
    gradients = np.apply_along_axis(lambda err: gradientMSE(m, X, err), 1, errors_matrix)

    grad_diff_matrix = np.linalg.norm(gradients[:, np.newaxis, :] - gradients[np.newaxis, :, :], axis=-1)
    point_diff_matrix = np.linalg.norm(points[:, np.newaxis, :] - points[np.newaxis, :, :], axis=-1)

    point_diff_matrix[point_diff_matrix == 0] = np.inf

    lipschitz_matrix = grad_diff_matrix / point_diff_matrix
    max_lipschitz = np.max(lipschitz_matrix)

    return max_lipschitz
```

Constant Step

This leads us to the second type of step size, the constant step size.

```
# Compute Lipschitz constant learning rate
lipschitz_learning_rate = 1/compute_lipschitz_constant(len(y), X_b, y, a, b)

# Set constant learning rates
constant_learning_rate = 0.01
```

Using these two step sizes we can define a gradient descent algorithm.

```
def gradient_descent_const_step(X, y, W, learning_rate, iterations):
    m = len(y)
    optimal_values = []
    mse_values = []

    for iteration in range(iterations):
        predictions = predict(X, W)
        errors = predictions - y
        gradients = gradientMSE(m, X, errors)
        W -= learning_rate * gradients
        optimal_values.append(W.copy())
        mse_values.append(MSE(errors))

    return W, optimal_values, mse_values
```

Adjusted Step

Based on the Lipschitz step size, we can also compute an updated step size using a backtracking algorithm.

```
def gradient_descent_adjusted_step(X, y, W, learning_rate, iterations):
    m = len(y)
    optimal_values = []
    mse_values = []

    for iteration in range(iterations):
        predictions = predict(X, W)
        errors = predictions - y
        gradients = gradientMSE(m, X, errors)
        learning_rate = backtracking_line_search(X, W, y, gradients, learning_rate)
        W -= learning_rate * gradients
        optimal_values.append(W.copy())
        mse_values.append(MSE(errors))

    return W, optimal_values, mse_values
```

The step is updated with respect to the rule from theorem 4

```
def backtracking_line_search(X, W, y, grad, alpha, beta=0.5, sigma=0.1):
    while MSE(X.dot(W - alpha * grad) - y) > MSE(X.dot(W) - y) - sigma * alpha * np.linalg.norm(grad)**2:
        alpha *= beta
    return alpha
```

Results

Finally, we can call the functions and train the model using the 3 step sizes.

```
# Train the models using different gradient descent methods
W_trained_with_constant_step, const_optimal_values, const_mse_values = gradient_descent_const_step(X_b, y, W_const, constant_learning_rate, itera
W_trained_with_lipschitz_step, lipschitz_optimal_values, lipschitz_mse_values = gradient_descent_const_step(X_b, y, W_lipschitz, lipschitz_learni
W_trained_with_adjusted_step, adjusted_optimal_values, adjusted_mse_values = gradient_descent_adjusted_step(X_b, y, W_adjusted, adjusted_learning

# Example of making a prediction
def make_prediction(new_data, W):
    new_data_b = np.c_[np.ones((new_data.shape[0], 1)), new_data]
    return predict(new_data_b, W)

# Example prediction
new_data = np.array([[6, 12]])
predicted_value_const = make_prediction(new_data, W_trained_with_constant_step)
predicted_value_lipschitz = make_prediction(new_data, W_trained_with_lipschitz_step)
predicted_value_adjusted = make_prediction(new_data, W_trained_with_adjusted_step)

print(f'Prediction with constant step: {predicted_value_const[0]}')
print(f'Prediction with Lipschitz step: {predicted_value_lipschitz[0]}')
print(f'Prediction with adjusted step: {predicted_value_adjusted[0]}')
```

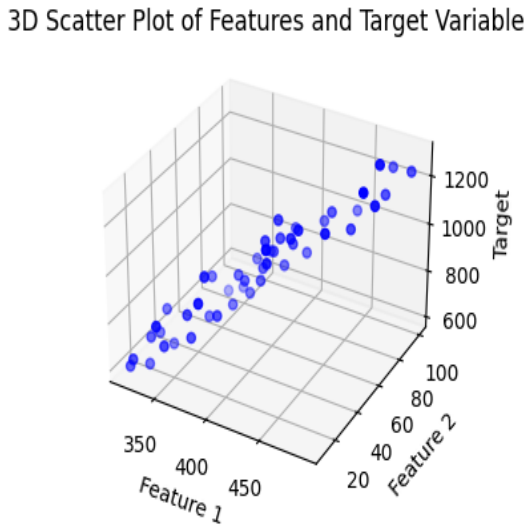
For a better analysis of the result, we also add some graphs to showcase the difference in convergence between the 3 methods.

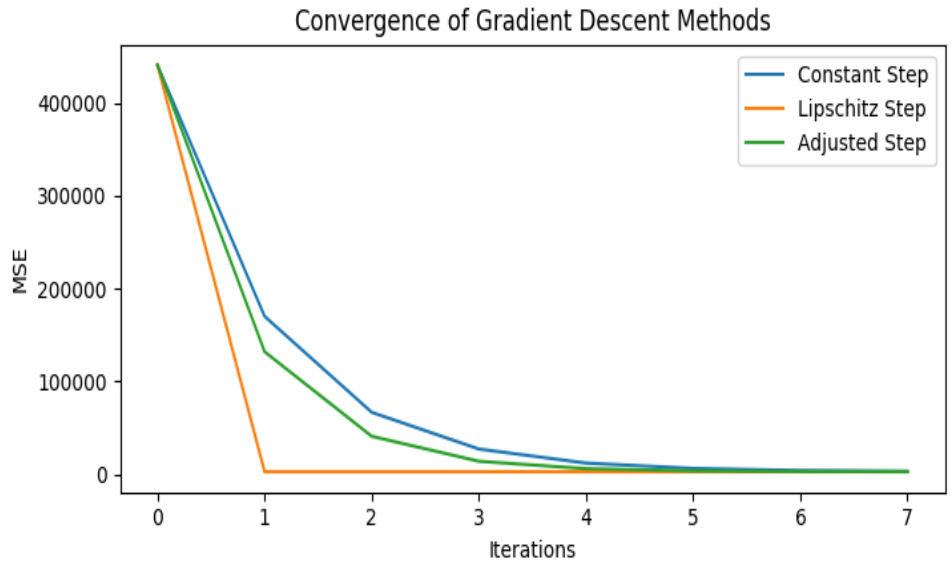
```
# Create a figure with two subplots
fig = plt.figure(figsize=(14, 6))

# 3D Scatter plot
ax1 = fig.add_subplot(121, projection='3d')
ax1.scatter(X[:, 0], X[:, 1], y, c='blue', marker='o')
ax1.set_xlabel('Feature 1')
ax1.set_ylabel('Feature 2')
ax1.set_zlabel('Target')
ax1.set_title('3D Scatter Plot of Features and Target Variable')
```

```
# Plot the convergence of the optimal values
ax2 = fig.add_subplot(122)
ax2.plot(const_mse_values, label='Constant Step')
ax2.plot(lipschitz_mse_values, label='Lipschitz Step')
ax2.plot(adjusted_mse_values, label='Adjusted Step')
ax2.set_xlabel('Iterations')
ax2.set_ylabel('MSE')
ax2.set_title('Convergence of Gradient Descent Methods')
ax2.legend()

plt.tight_layout()
plt.show()
```

Conclusions

Plotting the result we have obtained, one can clearly see the difference between the 3 methods of computation.





Conclusion

The results of our computation provide an obvious indication of the effectiveness of using gradient descent on β-smooth functions. By leveraging Lipschitz smoothness and controlling the step size, we observed a significant improvement in convergence rates compared to the traditional constant step approach.

In summary, our findings highlight the practical benefits of incorporating Lipschitz smoothness into gradient descent algorithms, paving the way for enhanced optimization in various fields.

Questions

Any questions?

Thank you for your attention!