Gradient Descent

Lecture 8

Centre for Data Science
Institute of Technical Education and Research
Siksha 'O' Anusandhan (Deemed to be University)
Bhubaneswar, Odisha, 751030

Overview

1. Introduction
2. The idea behind Gradient Descent
3. Estimating the Gradient
4. Using the Gradient
5. Choosing the Right Step Size
6. Using Gradient Descent to Fit Models
7. Minibatch and Stochastic Gradient Descent
8. References

Introduction

- When doing data science, we'll be trying to the find the best model for a certain situation.
- *"best"* will mean something like "minimizes the error of its predictions" or "maximizes the likelihood of the data".
- In other words, it will represent the solution to some sort of optimization problem.
- This means we'll need to solve a number of optimization problems. so our approach will be a technique called **gradient descent**.

The idea behind Gradient Descent

- Suppose we have some function f that takes as input a vector of real numbers and outputs a single real number. for example:

```python
from scratch.linear_algebra import Vector, dot
def sum_of_squares(v: Vector) -> float:
    """Computes the sum of squared elements in v"""
    return dot(v, v)
```

- We have to minimize or maximize such functions. That is, we need to find the input v that produces the largest (or smallest) possible value.
- The gradient gives the input direction in which the function most quickly increases.

The idea behind Gradient Descent

- So one approach to **maximize** a function is mentioned below:

Step 1 Pick a random starting point.
Step 2 Compute the gradient.
Step 3 Take a small step in the **direction of the gradient**.
Step 4 Repeat with the new starting point.

- Similarly you can find **minimum** of a function by below mentioned steps:

Step 1 Pick a random starting point.
Step 2 Compute the gradient.
Step 3 Take a small step in the **opposite direction of the gradient**.
Step 4 Repeat with the new starting point.

Estimating the Gradient

- If f is a function of one variable, its derivative at a point x measures how *f(x)* changes when we make a very small change to x.
- The derivative is defined as the limit of the difference quotients:

```
from typing import Callable
def difference_quotient(f: Callable[[float], float],x:
    float,h: float) -> float:
    return (f(x + h) - f(x)) / h
```

  as h approaches zero.
- The derivative is the slope of the tangent line at (x, f(x)), while the difference quotient is the slope of the not-quite-tangent line that runs through $(x + h, f(x + h))$. As h gets smaller and smaller, the not-quite-tangent line gets closer and closer to the tangent line.

Estimating the Gradient

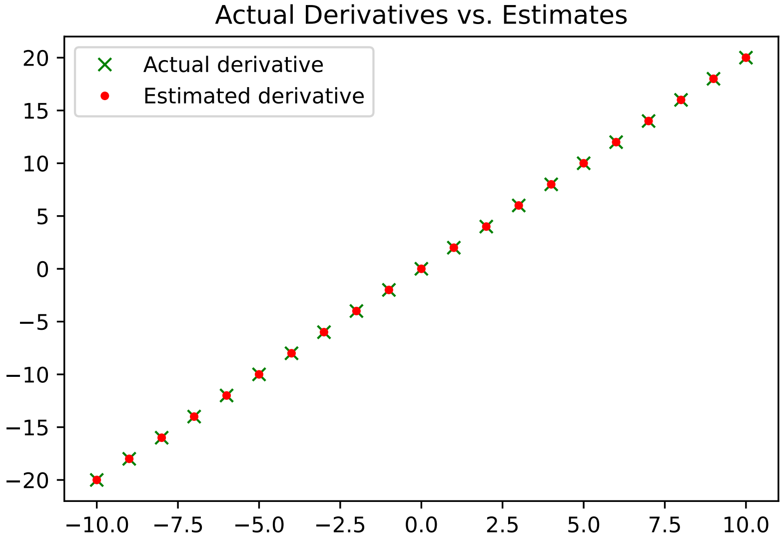
- For many functions it's easy to exactly calculate derivatives.For example

```
def square(x: float) -> float:
    return x * x
def derivative(x: float) -> float:
    return 2 * x
```

- What if you couldn't or didn't want to find the gradient?
- we can estimate derivatives by evaluating the difference quotient for a very small e.

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for
    x in xs]
```

Estimating the Gradient



Estimating the Gradient

- When f is a function of many variables, it has multiple partial derivatives, each indicating how f changes when we make small changes in just one of the input variables.

```python
def partial_difference_quotient(f: Callable[[Vector], float],v: Vector,i: int,h: float) -> float:
    """Returns the i-th partial difference quotient of f at v"""
    w = [v_j + (h if j == i else 0)for j, v_j in enumerate(v)]
    return (f(w) - f(v)) / h
def estimate_gradient(f: Callable[[Vector], float],v: Vector,h: float = 0.0001):
    return [partial_difference_quotient(f, v, i, h)for i in range(len(v))]
```

Using the Gradient

- Let's use gradients to find the minimum among all three-dimensional vectors.

Step 1 We'll just pick a random starting point.
Step 2 Take tiny steps in the opposite direction of the gradient
Step 3 Stop when we reach a point where the gradient is very small.

Using the Gradient

```python
import random
from scratch.linear_algebra import distance, add,
    scalar_multiply
def gradient_step(v: Vector, gradient: Vector, step_size:
    float) -> Vector:
    """Moves `step_size` in the `gradient` direction from `v`
    """
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)
def sum_of_squares_gradient(v: Vector) -> Vector:
    return [2 * v_i for v_i in v]
    # pick a random starting point
    v = [random.uniform(-10, 10) for i in range(3)]
    for epoch in range(1000):
        grad = sum_of_squares_gradient(v)
        v = gradient_step(v, grad, -0.01)
        print(epoch, v)
assert distance(v, [0, 0, 0]) < 0.001
```

Choosing the Right Step Size

- Although the rationale for moving against the gradient is clear, how far to move is not.
- Choosing the right step size is more of an art than a science.
- We have following options to choose step size:
  1. Using a fixed step size.
  2. Shrinking the step size over time.
  3. At each step, choosing the step size that minimizes the value of the objective function.
- we'll mostly just use a fixed step size.
- The last approach sounds great but is, in practice, a costly computation.
- The step size that "works" depends on the problem.
  - too small, and your gradient descent will take forever.
  - too big, and you'll take large steps that might make the function you care about get larger or even be undefined.

Using Gradient Descent to Fit Models

- we'll have some dataset and some model for the data that depends (in a differentiable way) on one or more parameters.
- we have a loss function that measures how well the model fits our data.
- Our loss function tells us how good or bad any particular model parameters are.
- we can use gradient descent to find the model parameters that make the loss as small as possible.

```python
# x ranges from -50 to 49, y is always 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
```

- In this case we know the parameters of the linear relationship between x and y, but imagine we'd like to learn them from the data.
- We'll use gradient descent to find the slope and intercept that minimize the average squared error.

Using Gradient Descent to Fit Models

```
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta
    predicted = slope * x + intercept # The prediction of the model.
    error = (predicted - y) # error is (predicted - actual).
    squared_error = error ** 2 # We'll minimize squared error
    grad = [2 * error * x, 2 * error] # using its gradient.
    return grad
```

- Imagine for some x our prediction is too large. In that case the error is positive.
- The second gradient term, 2 * error, is positive, which reflects the fact that small increases in the intercept will make the prediction even larger, which will cause the squared error to get even bigger.

Using Gradient Descent to Fit Models

- The first gradient term, 2 * error * x, has the same sign as x.
- If x is positive, small increases in the slope will again make the prediction (and hence the error) larger.
- If x is negative, though, small increases in the slope will make the prediction (and hence the error) smaller.
- The above mentioned computation was for a single data point.
- For the whole dataset we'll look at the mean squared error. And the gradient of the mean squared error is just the mean of the individual gradients.
- we will foloow following steps:

Step 1 Start with a random value for *theta*.
Step 2 Compute the mean of the gradients.
Step 3 Adjust *theta* in that direction
Step 4 Repeat.

Using Gradient Descent to Fit Models

```python
from scratch.linear_algebra import vector_mean
# Start with random values for slope and intercept
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
learning_rate = 0.001
for epoch in range(5000):
    # Compute the mean of the gradients
    grad = vector_mean([linear_gradient(x, y, theta) for x, y
     in inputs])
    # Take a step in that direction
    theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Minibatch and Stochastic Gradient Descent

- One drawback of the preceding approach is that we had to evaluate the gradients on the entire dataset before we could take a gradient step and update our parameters.
- For large datasets we can use a technique called minibatch gradient descent, in which we compute the gradient (and take a gradient step) based on a "minibatch" sampled from the larger dataset.

Minibatch and Stochastic Gradient Descent

```python
from typing import TypeVar, List, Iterator
T = TypeVar('T') # this allows us to type "generic" functions
def minibatches(dataset: List[T], batch_size: int, shuffle: bool = True) -> Iterator[List[T]]:
    """Generates `batch_size`-sized minibatches from the dataset"""
    # start indexes 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in range(0, len(dataset), batch_size)]
    if shuffle: random.shuffle(batch_starts) # shuffle the batches
    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

Minibatch and Stochastic Gradient Descent

- Now we can solve our problem again using minibatches:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta) for x, y in batch])
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Minibatch and Stochastic Gradient Descent

- Another variation is stochastic gradient descent, in which you take gradient steps based on one training example at a time:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta=gradient_step(theta,grad, -learning_rate)
    print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

- Stochastic gradient descent finds the optimal parameters in a much smaller number of epochs. But there are always tradeoffs.
- Basing gradient steps on small minibatches (or on single data points) allows you to take more of them, but the gradient for a single point might lie in a very different direction from the gradient for the dataset as a whole.

References

[1] Joel Grus. Data Science from Scratch. First Principles with Python. Second Edition. O'REILLY, May 2019.

Thank You