Introduction to Machine Learning (by Implementation)

Lecture 2: Gradient Descent

Ian J. Watson

University of Seoul

University of Seoul Graduate Course 2019

KOREA RESEARCH FELLOWSHIP
해외 우수신진연구자 유치사업

List comprehensions

- Review: it will often be useful to quickly make transformations from one list to another, applying some transformation to each element
- This can be done with a `for` loop and append:

```
oldList = [1, 2, 3, 4]
newList = []
for element in oldList:
    newList.append(element**2)
```

- But it can be done quicker with list comprehensions
  `newList = [element**2 for element in oldList]`
- Use enumerate if you need an index
  `[element if index == 0 else 0 for (index, element) in enumerate(oldList)]`
- If you don't need to use the element, can write _
  `[index for index, _ in enumerate(oldList)]`

Introduction

- So far, we've talked about python and done some introductory coding
- Today, we'll introduce and implement another concept that will be important going forward: gradient descent
- Often, we will have large complex models whose parameters we want to tune, in order to find the "best" model
  - "Best" will mean something like "minimizes the error of the model" or "maximizes the likelihood of the data"
- Therefore, we will need to a routine to find minima of functions
  - Generally, are models can be arbitrarily complex, and not amenable to analytical minimization (solving $\nabla f = 0$ directly)
- Gradient descent is a method to minimize a function numerically (as opposed to analytically)

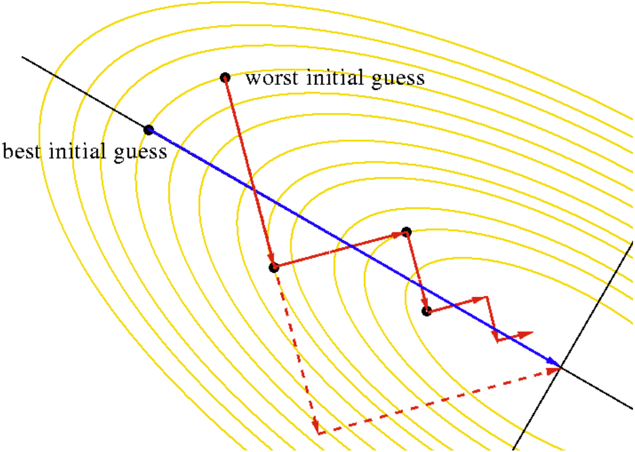
Analogy: Steepest descent

- A climber is trying to find his way down a mountain in deep fog, how should he proceed?
- One idea is to try to always go downhill the fastest way possible
- So, he figures out which direction has the steepest descent (ie which way is downhill), then takes a step in that direction
- After the step, he checks again, and takes another step
- He keeps proceeding in this manner until he cant go downhill anymore, he's reached the bottom

Gradient Descent

- From calculus, $\nabla f(\mathbf{x})$ gives the direction of largest increase of f at x (if its $\mathbf{0}$, we are at a minimum and done)
- Equivalently, $-\nabla f(\mathbf{x})$ gives direction of largest decrease, so $f(\mathbf{x} - \gamma \nabla f(\mathbf{x})) < f(\mathbf{x})$ (at least, for some γ small enough)
- We will define a sequence $\mathbf{x}_i$ to find the minimum:
  - Start with some random position $\mathbf{x}_0$
  - Iterate:
    - Find $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)$
    - Stop if $|f(\mathbf{x}_{n+1}) - f(\mathbf{x}_n)| < \epsilon$, i.e. we're not reducing further, so we're close to the minimum
  - Return the final $\mathbf{x}_n$
- γ_n can be different for each iteration, we'll find the best γ_n by checking several possible values
- ϵ is the *tolerance*, how close to a minima do we need to be before stopping (we could also check $|\nabla f(\mathbf{x}_n)| < \epsilon$)

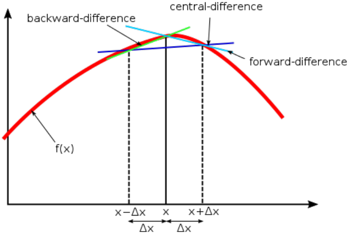
Example function



- Shows how the algorithm picks out different paths depending on starting point
- Lines are contours of equal value

Partial Derivative Estimation

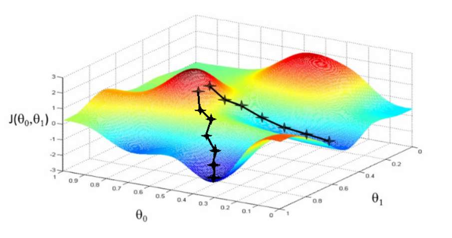
- To run gradient descent, we must find the partial derivatives
- Maybe we are given only the function, f, without the analytic form, or the derivative functions, what can we do then?
- We will have to come up with a numerical estimate for the derivative
- The simplest way to do this is to simply take a partial difference quotient:
  - $\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x}+h \cdot \mathbf{e}_i)-f(\mathbf{x})}{h}$
  - $\mathbf{e}_i$ is a unit vector in the x_i direction
  - As $h \to 0$, our estimate $\to \frac{\partial f}{\partial x_i}(\mathbf{x})$
- There are better (= converge faster) methods, but this suffices

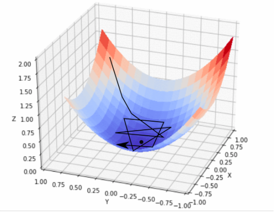


- Could take difference in forward or backward direction, or an average
  - Starting point for better estimates
- In limit as $\Delta x \to 0$, all the methods converge to the true derivative
  - So, just take the equation shown when building your code

Possible Issues

- Multiple minima: gradient descent will find the closest minimum, this is not necessarily the global minimum
    - Can be dealt with by testing multiple starting points and using techniques like simulated annealing





- Gradient descent also slowly converges near the minimum, can take a long time to go from near the minimum to the true minimum
    - Possible to detect and methods to speed up conversion
- A real numerical minimizer (e.g. MINUIT, used in particle physics libraries like ROOT) will take these considerations into account

Setup in code

- We will deal with functions of several variables
- The way we will represent this python is as follows:
- A function $f : \mathbb{R}^n \to \mathbb{R}$, will be a python function `f`, which takes a list of n floats, and returns a float
  - $f(x, y) = x^2 + y^2 \to$ `f = lambda x:  x[0]**2 + x[1]**2`
- We can then represent the basis vectors like $\mathbf{e}_i = [0, \ldots, 1, \ldots, 0]$ with the 1 in the i'th position
  - E.g. for $\mathbb{R}^2$, the basis vectors are `e0 = [1, 0]` and `e1 = [0, 1]`

Exercises

We will break up the algorithm into pieces to make it easier to test one by one. The functions should use the previous functions

- `step(v: List[float], direction: List[float], step_size: float) -> List[float]` increments the point `v` in the `direction` by `step_size`, i.e. $\texttt{step}(\mathbf{v}, \mathbf{d}, s) = \mathbf{v} + s \cdot \mathbf{d}$
- `move_point_along_ei(x: List[float], i: int, h: float) -> List[float]` takes the `x`, and shifts it by $h \cdot \mathbf{e}_i$, that is $\mathbf{x} + h \cdot \mathbf{e}_i$
- `partial_difference_quotient(f, v, i, h)` which estimates the `i`'th partial derivative of the function `f` at `v` with a step size of `h`, $\frac{f(\mathbf{x}+h\cdot\mathbf{e}_i)-f(\mathbf{x})}{h}$
- `estimate_gradient(f: Fn, v: List[float], h=0.00001)` which uses `partial_difference_quotient` to estimate ∇f at `v`
- Check that your `estimate_gradient` function gives sensible results, add some tests to test_gradient_descent.py
  - E.g. $f(x, y) = x^2 + y^2 \implies \nabla f(x, y) = (2x, 2y)$, does you function give a result close to the real gradient at $(x, y) = (0, 0)$, and $(x, y) = (1, 1)$?

Exercises (minimize)

- Write a function `minimize(f, df, x0, step_sizes, tol)` which implements the gradient descent algorithm
  - `f: Callable[[List[float]], float]` is the function we want to minimize
    - `Callable[a, b]` is the type of a function that takes in a (list of types) and returns a type `b`
  - `df: Callable[[List[float]], List[float]]` should be the gradient function, i.e. it takes a point $\mathbf{x}$ and returns $\nabla f(\mathbf{x})$
    - If you don't have a function for the exact gradient, you can use your estimate by passing `lambda x: estimate_gradient(f, x)` instead of `df`
  - `x0: List[float]` the initial position, point to start the gradient descent minimization from
  - `step_sizes: List[float]`, this is the list we choose γ_n from. Test each element of the list, and choose the one that gives the lowest value of `f`. Use default value `[10, 1, 0.1, 0.01, 0.001]`
  - `tol: float` the tolerance, when `f` from one step to the next changes by less than this amount, end the search. Use default value `1e-5`

Exercises (cont'd)

- Summary of the algorithm:
  - Find ∇f at x_n using `df` (first time through it will be)
  - Set $\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \nabla f$, where h is chosen from `step_sizes` to give the lowest value of the function
  - When when the function changes by less than `tol`, stop and return the previous estimate, not the new estimate for the step, otherwise loop
    - If $|f(\mathbf{x}_{n+1}) - f(\mathbf{x}_n)| <$ `tol`, return x_n
- Check the test_gradient_descent.py function `test_minimize` for examples of how the code should work
- Check that your `minimize` function finds a value close to the true minimum for some known functions, and add to the test file
  - Do you find $(0, 0)$ if you minimize $f(x, y) = x^2 + y^2$? Does it change if you use the real ∇f or your `estimate_gradient` version?