Lecture 11 – Gradient Descent

Optimisation CT5141

James McDermott

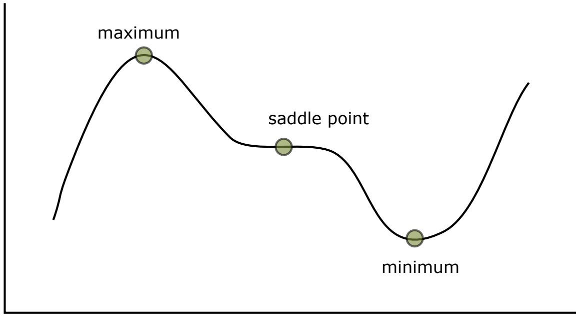
University of Galway

Overview

Not examinable

Note (2022-2023): we will skip all of the second-order gradient descent material, momentum, divergence, and order of convergence; and it is not examinable.

Reminder: good old-fashioned differentiation



- A minimum x of f will certainly have $f'(x) = 0$
- Sadly, the same is true of a maximum, and even a **saddle point**
- But the second derivative (the derivative of the derivative) $f''(x)$ will be negative, for a minimum
- No way to distinguish a local from global minimum.

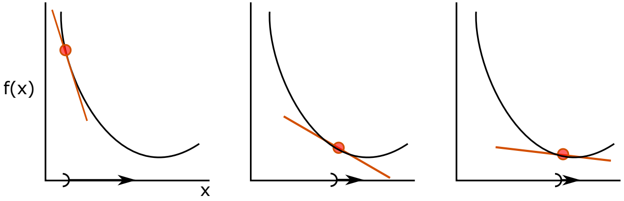
Gradient descent

GD is needed because:

- Sometimes solving for $f'(x) = 0$ and $f''(x) < 0$ is difficult/impossible
- We need a method which proceeds iteratively

1D GD is just:

- If we are at x, we step in the direction of $-f'(x)$



Reminder: GD for machine learning

- Tasks like regression, classification, clustering
- There is a loss **per point in the training set**
- This loss is **differentiable** wrt model parameters
- Total loss is a **sum** or **mean** over the dataset
- (Possibly plus a differentiable regularisation)

Reminder: GD for machine learning

- Tasks like regression, classification, clustering
- There is a loss **per point in the training set**
- This loss is **differentiable** wrt model parameters
- Total loss is a **sum** or **mean** over the dataset
- (Possibly plus a differentiable regularisation)

- E.g. MSE in regression: $f(w) = \frac{1}{N} \sum_i ((w_0 + w_1 x_i) - y_i)^2$
- We can calculate gradient $f'(w)$ (exercise!)
- At each iteration, step by $-f'(w)$ to descend

Reminder: GD for machine learning

- Tasks like regression, classification, clustering
- There is a loss **per point in the training set**
- This loss is **differentiable** wrt model parameters
- Total loss is a **sum** or **mean** over the dataset
- (Possibly plus a differentiable regularisation)

- E.g. MSE in regression: $f(w) = \frac{1}{N}\sum_i((w_0 + w_1 x_i) - y_i)^2$
- We can calculate gradient $f'(w)$ (exercise!)
- At each iteration, step by $-f'(w)$ to descend

- Also **stochastic GD**: one batch of the dataset at each step.

GD in multiple dimensions

The **partial derivative**: one derivative **per dimension**. Suppose we are in 2D (two real decision variables), so $f : \mathbb{R}^2 \to \mathbb{R}$.

Notation:

$$\frac{\partial f}{\partial x_0}$$

(also written f_{x_0}), is the **partial derivative** of f **with respect to** x_0**.**

Calculate f_{x_0} by differentiating wrt x_0, pretending x_1 is a constant.

GD in multiple dimensions

The **partial derivative**: one derivative **per dimension**. Suppose we are in 2D (two real decision variables), so $f : \mathbb{R}^2 \to \mathbb{R}$.

Notation:

$$\frac{\partial f}{\partial x_0}$$

(also written f_{x_0}), is the **partial derivative** of f **with respect to** x_0**.**

Calculate f_{x_0} by differentiating wrt x_0, pretending x_1 is a constant.

Example: if $f(x_0, x_1) = x_0^3 x_1 + x_1^3$, then

$$\frac{\partial f}{\partial x_0} = 3x_0^2 x_1$$

GD in multiple dimensions

Out of all possible directions from point x, there must be some direction(s) which give the **steepest descent**, i.e. largest downward step in f-value. The partial derivatives give us this direction.

The gradient ∇

The partial derivatives are collected up into a vector called the **gradient** or **Jacobian** of f, notated

$$\nabla f = (\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_{n-1}})$$

This is a function from $\mathbb{R}^n \to \mathbb{R}^n$, or we can think of it as a vector of functions, each from $\mathbb{R}^n \to \mathbb{R}$.

We can evaluate it at a particular point. E.g. if $f(x_0, x_1) = x_0^3 x_1 + x_1^3$, then $\nabla f = (3x_0^2 x_1, x_0^3 + 3x_1^2)$, and $\nabla f(2, 0) = (0, 8)$.

Avoiding a misunderstanding

By the way, the symbol ∇ isn't a "standalone" symbol – don't try to divide by it!

E.g. $\nabla f(2,0)$ doesn't mean $\nabla(f(2,0))$.

(No such thing exists: $f(2,0)$ is just a number, so it has no gradient.)

Instead $\nabla f(2,0)$ means $(\nabla f)(2,0)$ – some people write it this way to emphasise that ∇f is an object by itself. It's a function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$.

Steepest descent in n dimensions

The negative of the gradient is the direction of steepest descent.

1. Set $x^{(1)} := (x_0, x_1, \ldots, x_{n-1})$, an initial point
2. Set $x^{(i+1)} := x^{(i)} - \alpha \nabla f(x^{(i)})$ (α is learning rate)
3. If $||x^{(i+1)} - x^{(i)}|| <$ tol, or maxits has been exceeded, return $x^{(i+1)}$
4. Otherwise go to 2.

(We use superscripts $x^{(i)}$ to indicate the *i*-th iteration.)

Observe $x^{(i)}$ and $-\alpha \nabla f(x^{(i)})$ are each vectors of n components.

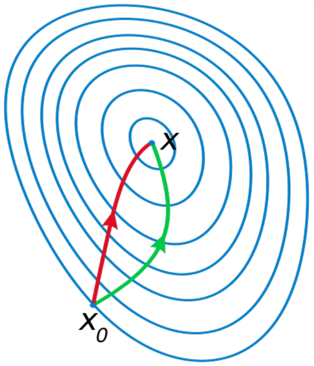
Overview

Steepest is not best

- From a point x, there are many vectors p which point in a descent direction, ie where $f(x+p) < f(x)$ (unless x is already a local minimum).
- The steepest descent is a particular choice of p.
- Some sophisticated variant algorithms use other vectors instead and do better.

Second-order methods

The first derivative tells us the direction of steepest descent. But by looking at the **second derivative** as well, we can actually choose a better direction. No surprise because second derivative was used in good old-fashioned differentiation as well.



Wiki

E.g. Newton's method **approximates** f locally by a quadratic (i.e. a Taylor expansion) and uses the **second derivative** to choose a **better step**:

$$x^{(i)} \mathrel{-}= f'(x^{(i)})/f''(x^{(i)})$$

If f is actually quadratic, then the approximation is exact and we find the global optimum in one step.

Tangent: Newton's method (for root-finding)

Finding roots (zeros) of equations is **closely related** to optimisation. You might have studied Newton's method for root-finding:

$$x^{(i)} -= f(x^{(i)})/f'(x^{(i)})$$

It uses $f(x)$ and $f'(x)$, whereas Newton's method for **minimisation** uses $f'(x)$ and $f''(x)$.

Second derivative in n dimensions

The first derivative generalises in n dimensions to the gradient ∇f (a vector of length n), a.k.a. Jacobian.

Similarly the second derivative generalises to a **matrix** of size $(n \times n)$. This object is called the **Hessian** and denoted H_f.
The i, j-th entry is defined as $(H_f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} = f_{x_i, x_j}$, that is the second derivative with respect to i and then j. For example if $f(x_1, x_2) = x_1^2 x_2$:

$$H_f = \begin{bmatrix} 2x_2 & 2x_1 \\ 2x_1 & 0 \end{bmatrix}$$

Second derivative in n dimensions

The first derivative generalises in n dimensions to the gradient ∇f (a vector of length n), a.k.a. Jacobian.

Similarly the second derivative generalises to a **matrix** of size $(n \times n)$. This object is called the **Hessian** and denoted H_f.
The i, j-th entry is defined as $(H_f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} = f_{x_i, x_j}$, that is the second derivative with respect to i and then j. For example if $f(x_1, x_2) = x_1^2 x_2$:

$$H_f = \begin{bmatrix} 2x_2 & 2x_1 \\ 2x_1 & 0 \end{bmatrix}$$

By **Clairaut's theorem**, the Hessian is symmetric.

Newton's method in multiple dimensions

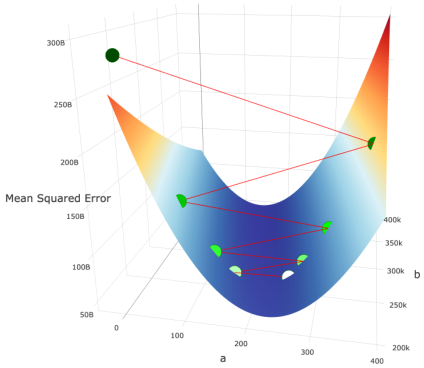
In 1D, we take a step $\delta = -f'(x)/f''(x)$. The generalisation to multiple dimensions is not quite obvious.

We would like to take a step $\delta = -\nabla f(x)/H_f(x)$, where δ is a vector of size n. But we can't divide by a matrix!

We could multiply by its inverse H_f^{-1}, i.e. set $\delta := -H_f(x)^{-1}\nabla f(x)$, but matrix inversion is slow. In practice, at each iteration we solve $H_f(x)\delta = -\nabla f(x)$ for δ, and that is our step. This turns out faster.

1. Set $x^{(i)} := (x_0, x_1, \ldots, x_{n-1})$, an initial point
2. Solve $H_f(x^{(i)})\delta = -\nabla f(x^{(i)})$ for δ
3. Set $x^{(i+1)} := x^{(i)} + \delta$
4. If $||x^{(i+1)} - x^{(i)}|| <$ tol, or if maxits exceeded, return $x^{(i+1)}$
5. Otherwise go to 2.

Overshoot and divergence



Source

- In any GD method, it’s possible that we could take a step that is too large
- We would **overshoot**
- It’s likely that the **next** step would overshoot in the opposite direction
- The algorithm will quickly **diverge**, e.g. reaching infinite x
- Usual solution: decrease the learning rate.

Momentum

In some methods, especially stochastic GD, we add a **momentum** term:

- it tends to smooth out the noisy behaviour of the stochastic GD
- (compare to momentum in PSO).

Adaptive learning rate

In some methods, the algorithm changes the learning rate over time in response to its own movements.

- avoid the user having to set it carefully
- usually introduces another hyperparameter instead.

Choosing an algorithm

There are many possible approaches to gradient descent. Which is best?

- In machine learning, people usually just run experiments to see what is best in practice: GD, SGD, BFGS, Adam, Adaboost, RMSProp…

Order of convergence

- In “classical” optimisation, we compare algorithms by talking about **convergence**. That is:
    - Is the method guaranteed to converge to a unique point?
    - How fast? Some methods have “quadratic convergence”.

These questions are beyond our scope.

Overview

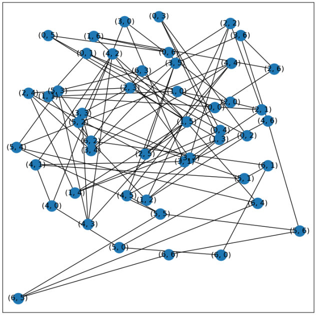
GD outside ML

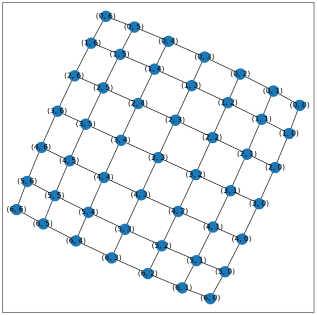
GD is used a lot in ML.

But GD is **not only** used in ML.

Graph Layout

We visualise a graph by drawing the nodes and edges. But **where to position the nodes**?





This is the interesting and practical problem of **graph layout**.

Graph Layout

We **could** use a metaheuristic here, and maybe we could create a constructive heuristic. But a better approach involves using a gradient. It will also give us great insight into how GD works.

Force-directed model

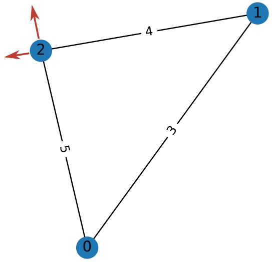
In a **force-directed** model, we imagine **physical forces** pushing the nodes around in order to achieve a good layout.

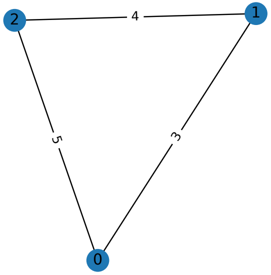
1. Spring forces
2. Magnetic repulsion forces

Spring forces

Suppose every edge has a **desired length**. This may be given as part of the graph. If not, let the desired length be 1.

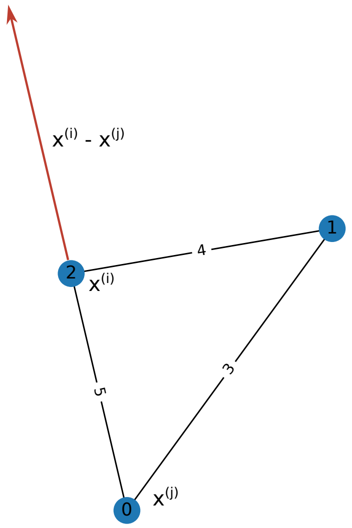
- Then imagine a spring of that length connecting nodes n_i and n_j. It tries to expand or contract to the desired length, exerting forces on the nodes.





Spring forces case 1: too close

If the nodes are **too close**, i.e. their **current** distance is less than the desired distance, then node n_i should move **away** from node n_j (and vice versa). Let their current positions be $x^{(i)}$ and $x^{(j)}$. Then the vector $x^{(i)} - x^{(j)}$ points in the right direction as shown. It has length $||x^{(i)} - x^{(j)}||$.



Calculating the step

However, the amount of movement should be **proportional to how wrong the distance is**. If the current distance is **almost right**, we should take only a small step. So, multiply the direction by the difference in lengths, $(D_{ij} - C_{ij})$.

where D is the desired distance and C the current distance, $C_{ij} = ||x^{(i)} - x^{(j)}||$

We know that in this case, $D_{ij} > C_{ij}$, so the term is positive and does not change the direction of the vector.

Calculating the step

However, the amount of movement should be **proportional to how wrong the distance is**. If the current distance is **almost right**, we should take only a small step. So, multiply the direction by the difference in lengths, $(D_{ij} - C_{ij})$.

where D is the desired distance and C the current distance, $C_{ij} = ||x^{(i)} - x^{(j)}||$

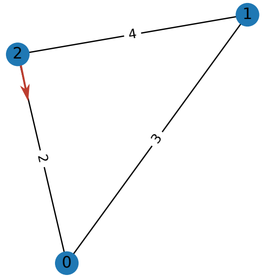
We know that in this case, $D_{ij} > C_{ij}$, so the term is positive and does not change the direction of the vector.

Finally, multiply by a learning rate α as always in GD.

We should set

$$x^{(i)} \mathrel{+}= \alpha(D_{ij} - C_{ij})(x^{(i)} - x^{(j)})$$

Spring forces case 2: too far apart



Suppose instead the current distance is **greater than** the desired distance, then the nodes should instead move **closer**. The vector $x^{(i)} - x^{(j)}$ now points in the wrong direction (by 180^o).

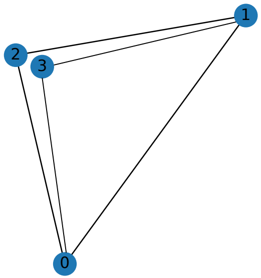
But in this case, $D_{ij} < C_{ij}$, so the middle term is negative, reversing the direction as needed.

So the same formula works! We set

$$x^{(i)} \mathrel{+}= \alpha(D_{ij} - C_{ij})(x^{(i)} - x^{(j)})$$

Magnetic repulsion

Although the above seems complete, it turns out that nodes which are **not connected by an edge** often get stuck too close to each other.
The solution is an extra force applied to every pair of nodes.



Imagine two particles both with negative magnetic polarity (e.g. electrons). If they get close, they push each other away. For $x^{(i)}$, the vector $x^{(i)} - x^{(j)}$ points in the right direction. But the strength of the force drops as the square of the distance (an **inverse square law**, just like gravity):

$$x^{(i)} \mathrel{+}= \alpha(x^{(i)} - x^{(j)})/C_{ij}^2$$

Springs: edges only!

The spring update step applies only to any pair of nodes connected by an edge

The magnetic repulsion step applies to every pair of nodes.

Aside: complexity

The magnetic repulsion step applies to every pair of nodes.

What does this tell us about computational complexity?

We never wrote an objective!

- We never wrote down an objective or calculated a true gradient
- … but this approach is still GD, because we are using:
- steps in the **right direction**;
- steps whose **size** is proportional to how bad the current solution is;
- a **learning rate**.

Energy metaphors

Energy metaphors are common in optimisation – e.g. simulated annealing is an energy metaphor.

The idea is: we have a physical system, e.g.:

- a set of springs attached to points
- a ball rolling on a hill
- a set of particles which can move around
- a set of particles with magnetic orientation, which can rotate.

Any point in the search space corresponds to some **configuration** of the system, e.g. a position of the ball, or positions of all the particles.

A configuration has some amount of energy (analogy: objective value). Nature will try to dissipate the energy, e.g. push particles around, to try to find a lower-energy solution.

Energy and equilibrium

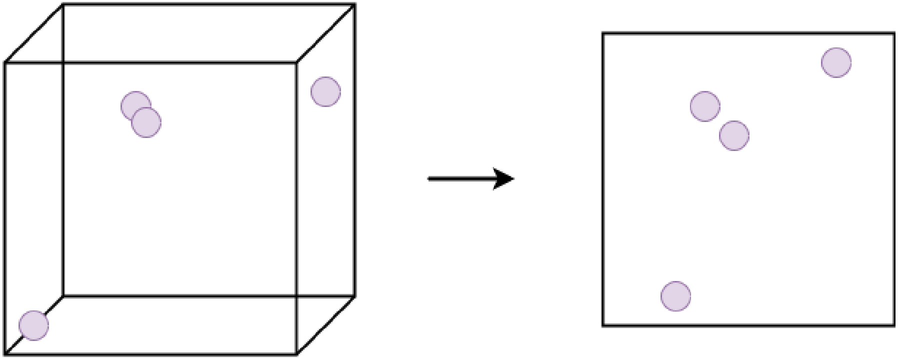
Nature can become stuck in local optima (**annealing** of metals is an attempt to avoid this).

When the physical system reaches an optimum, it will be **in equilibrium** – no longer moving.

A system in equilibrium is not moving, so it has a derivative of zero. No surprise, because optima have derivative zero in good old-fashioned differentiation!

Multi-dimensional scaling (MDS)

MDS is a dimensionality reduction technique



Given m points in n dimensions, find m locations in 2D such that **inter-point distances are preserved**.

This is just like the spring forces in graph layout! We can easily adapt our graph layout code to solve this problem.

Overview

`scipy.optimize.minimize`

```
scipy.optimize.minimize(fun, x0, args=(),
  method=None, jac=None, hess=None, hessp=None,
  bounds=None, constraints=(), tol=None,
  callback=None, options=None)
```

- `fun`: objective
- `x0`: initial value
- `args`: extra args if needed for `fun`, `jac`, `hess`
- `method`: optimization approach, e.g. `Newton-CG`, `BFGS`
- `jac`: **Jacobian**, i.e. function to get vector of partial derivatives
- `hess`: **Hessian**, i.e. function to get matrix of second partial derivatives
- `bounds`: box constraints, usable e.g. by `BFGS`
- `tol`: stopping criterion
- `callback`: function to e.g. save or print information at each step.

`scipy.optimize.minimize`

We don’t **have to** provide the gradient and Hessian (e.g. if it’s hard to calculate them.)

If we don’t, some algorithms, e.g. `BFGS`, will approximate them numerically. Of course, this costs more CPU.

Sympy

Sometimes, a tool like Sympy might help to calculate a derivative for us.

```
# sympy symbols and sympy funs, not math or numpy
x_1 = sympy.Symbol("x_1")
x_2 = sympy.Symbol("x_2")
f = sympy.sin(x_1) + sympy.cos(10 + x_1*x_2)

# differentiate with respect to x_1
df_dx_1 = sympy.diff(f, x_1)
df_dx_2 = sympy.diff(f, x_2)
```

Basin-hopping algorithm

- Many local optima, so GD doesn't find global optimum
- Approach: **basin-hopping** is a **two-stage** algorithm:

1. GD to local optimum
2. Use a large mutation, hoping to reach a new basin of attraction, and go to 1.

```
scipy.optimize.basinhopping(func,
                            x0, niter=100, T=1.0, ...)
```

- T: temperature to control optional **annealing** behaviour

More software

`scipy.optimize` also has linear programming, scalar optimisation (pretty interesting), and even root-finding.

Optional readings

A very nice short set of notes on the main ideas of gradient-based optimisation:

http://fa.bianp.net/teaching/2018/eecs227at/introduction.html

Overview