Line search. Matrix calculus. Automatic differentiation.

Daniil Merkulov

Optimization for ML. Faculty of Computer Science. HSE University

Line search

Problem

Suppose, we have a problem of minimization of a function $f(x) : \mathbb{R} \to \mathbb{R}$ of scalar variable:

$$f(x) \to \min_{x \in \mathbb{R}}$$

Problem

Suppose, we have a problem of minimization of a function $f(x) : \mathbb{R} \to \mathbb{R}$ of scalar variable:

$$f(x) \to \min_{x \in \mathbb{R}}$$

Sometimes, we refer to the similar problem of finding minimum on the line segment $[a, b]$:

$$f(x) \to \min_{x \in [a,b]}$$

Problem

Suppose, we have a problem of minimization of a function $f(x) : \mathbb{R} \to \mathbb{R}$ of scalar variable:

$$f(x) \to \min_{x \in \mathbb{R}}$$

Sometimes, we refer to the similar problem of finding minimum on the line segment $[a, b]$:

$$f(x) \to \min_{x \in [a,b]}$$

Example

Typical example of line search problem is selecting appropriate stepsize for gradient descent algorithm:

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$
$$\alpha = \text{argmin } f(x_{k+1})$$

Problem

Suppose, we have a problem of minimization of a function $f(x) : \mathbb{R} \to \mathbb{R}$ of scalar variable:

$$f(x) \to \min_{x \in \mathbb{R}}$$

Sometimes, we refer to the similar problem of finding minimum on the line segment $[a, b]$:

$$f(x) \to \min_{x \in [a,b]}$$

Example

Typical example of line search problem is selecting appropriate stepsize for gradient descent algorithm:

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$
$$\alpha = \text{argmin } f(x_{k+1})$$

The line search is a fundamental optimization problem that plays a crucial role in solving complex tasks. To simplify the problem, let's assume that the function, $f(x)$, is *unimodal*, meaning it has a single peak or valley.

Unimodal function

Definition

Function $f(x)$ is called **unimodal** on $[a, b]$, if there is $x_* \in [a, b]$, that $f(x_1) > f(x_2) \quad \forall a \leq x_1 < x_2 < x_*$ and $f(x_1) < f(x_2) \quad \forall x_* < x_1 < x_2 \leq b$

Unimodal function

Definition

Function $f(x)$ is called **unimodal** on $[a, b]$, if there is $x_* \in [a, b]$, that $f(x_1) > f(x_2) \quad \forall a \leq x_1 < x_2 < x_*$ and $f(x_1) < f(x_2) \quad \forall x_* < x_1 < x_2 \leq b$

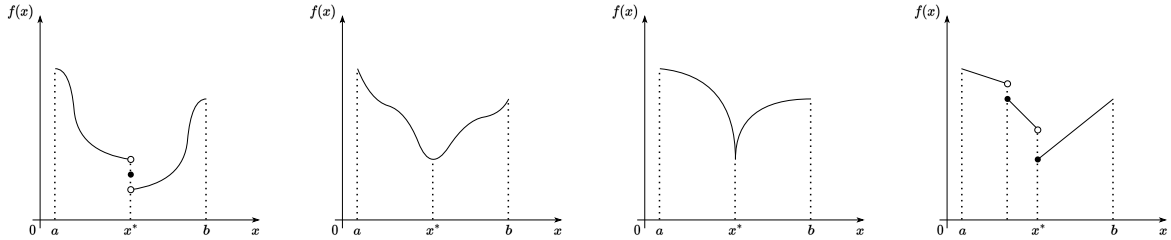


Figure 1: Examples of unimodal functions

Key property of unimodal functions

Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$

Key property of unimodal functions

Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$
- if $f(x_1) \geq f(x_2) \to x_* \in [x_1, b]$

Key property of unimodal functions

Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$
- if $f(x_1) \geq f(x_2) \to x_* \in [x_1, b]$

Key property of unimodal functions

Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$
- if $f(x_1) \geq f(x_2) \to x_* \in [x_1, b]$

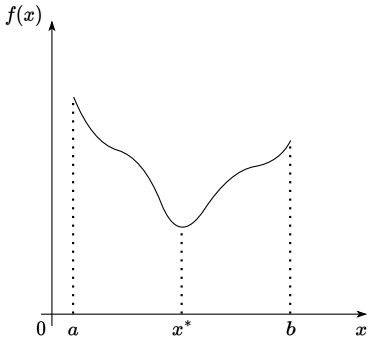
Proof Let's prove the first statement. On the contrary, suppose that $f(x_1) \leq f(x_2)$, but $x^* > x_2$. Then necessarily $x_1 < x_2 < x^*$ and by the unimodality of the function $f(x)$ the inequality: $f(x_1) > f(x_2)$ must be satisfied. We have obtained a contradiction.

Key property of unimodal functions

Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$
- if $f(x_1) \geq f(x_2) \to x_* \in [x_1, b]$

Proof Let's prove the first statement. On the contrary, suppose that $f(x_1) \leq f(x_2)$, but $x^* > x_2$. Then necessarily $x_1 < x_2 < x^*$ and by the unimodality of the function $f(x)$ the inequality: $f(x_1) > f(x_2)$ must be satisfied. We have obtained a contradiction.

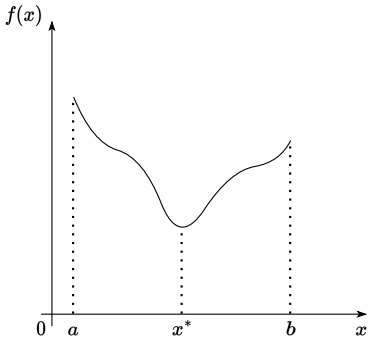


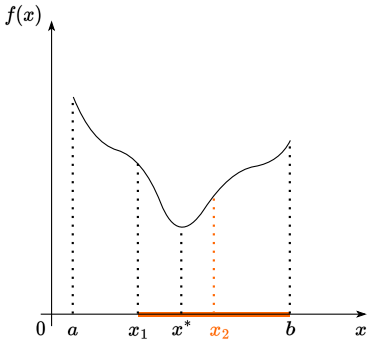
Key property of unimodal functions

Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$
- if $f(x_1) \geq f(x_2) \to x_* \in [x_1, b]$

Proof Let's prove the first statement. On the contrary, suppose that $f(x_1) \leq f(x_2)$, but $x^* > x_2$. Then necessarily $x_1 < x_2 < x^*$ and by the unimodality of the function $f(x)$ the inequality: $f(x_1) > f(x_2)$ must be satisfied. We have obtained a contradiction.



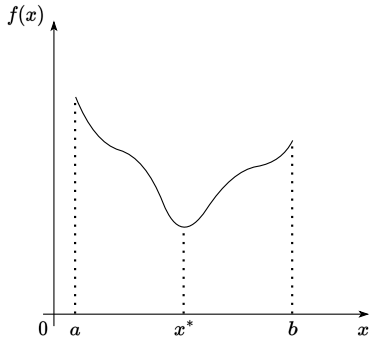


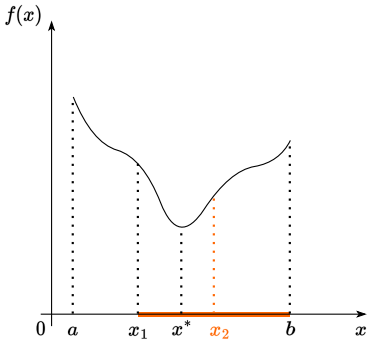
Key property of unimodal functions

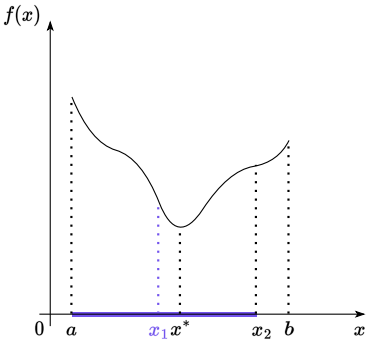
Let $f(x)$ be unimodal function on $[a, b]$. Than if $x_1 < x_2 \in [a, b]$, then:

- if $f(x_1) \leq f(x_2) \to x_* \in [a, x_2]$
- if $f(x_1) \geq f(x_2) \to x_* \in [x_1, b]$

Proof Let's prove the first statement. On the contrary, suppose that $f(x_1) \leq f(x_2)$, but $x^* > x_2$. Then necessarily $x_1 < x_2 < x^*$ and by the unimodality of the function $f(x)$ the inequality: $f(x_1) > f(x_2)$ must be satisfied. We have obtained a contradiction.







Dichotomy method

We aim to solve the following problem:

$$f(x) \to \min_{x \in [a,b]}$$

We divide a segment into two equal parts and choose the one that contains the solution of the problem using the values of functions, based on the key property described above. Our goal after one iteration of the method is to halve the solution region.

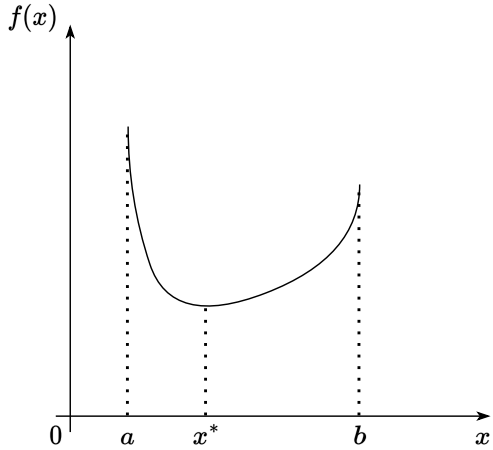


Figure 2: Dichotomy method for unimodal function

Dichotomy method

We measure the function value at the middle of the line segment

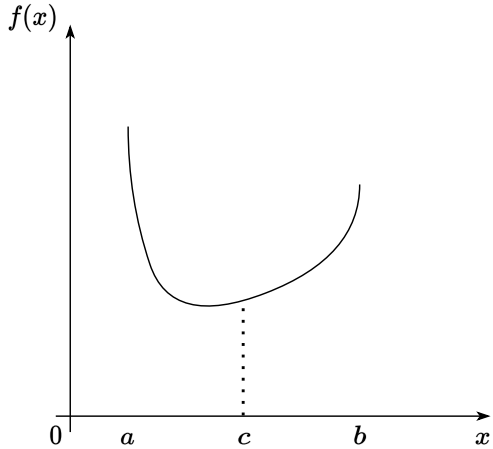


Figure 3: Dichotomy method for unimodal function

Dichotomy method

In order to apply the key property we perform another measurement.

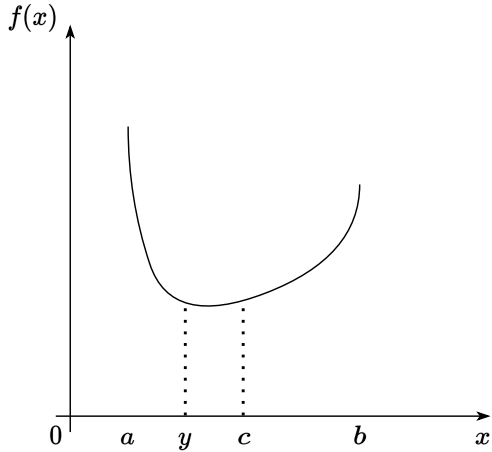


Figure 4: Dichotomy method for unimodal function

Dichotomy method

We select the target line segment. And in this case we are lucky since we already halved the solution region. But that is not always the case.

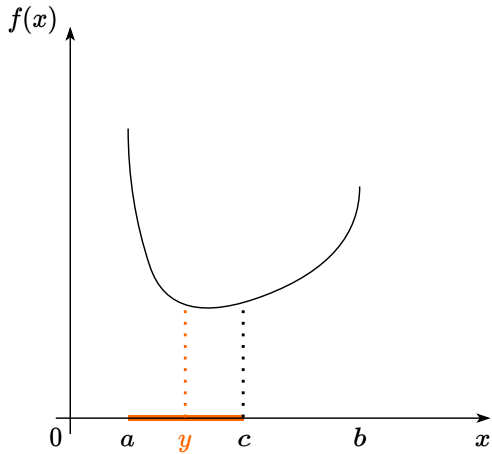


Figure 5: Dichotomy method for unimodal function

Dichotomy method

Let's consider another unimodal function.

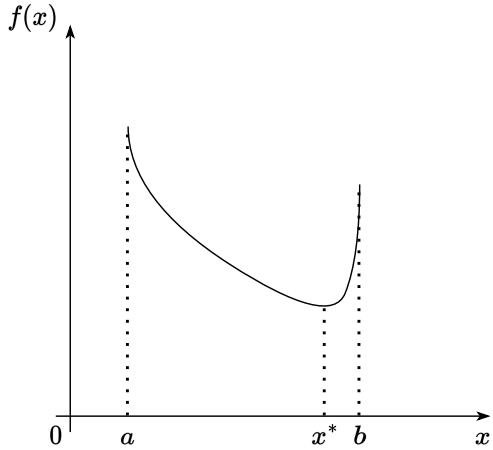


Figure 6: Dichotomy method for unimodal function

Dichotomy method

Measure the middle of the line segment.

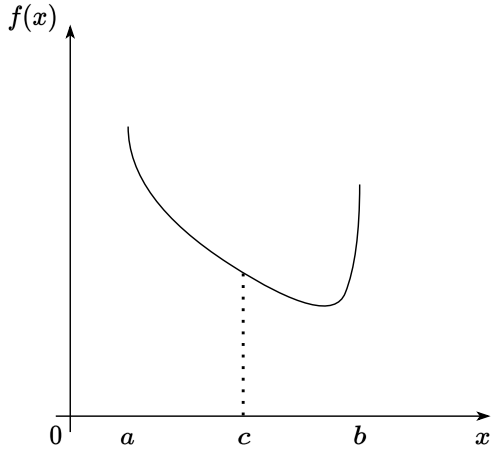


Figure 7: Dichotomy method for unimodal function

Dichotomy method

Get another measurement.

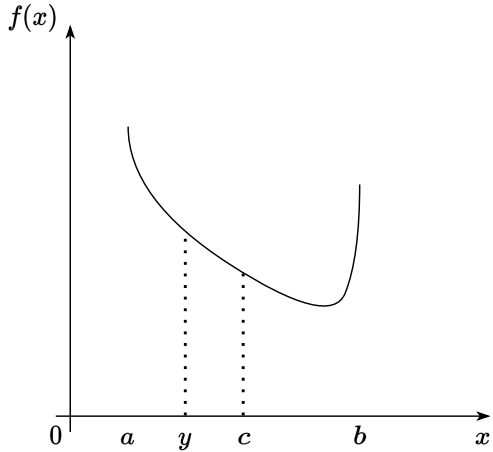


Figure 8: Dichotomy method for unimodal function

Dichotomy method

Select the target line segment. You can clearly see, that the obtained line segment is not the half of the initial one. It is $\frac{3}{4}(b-a)$. So to fix it we need another step of the algorithm.

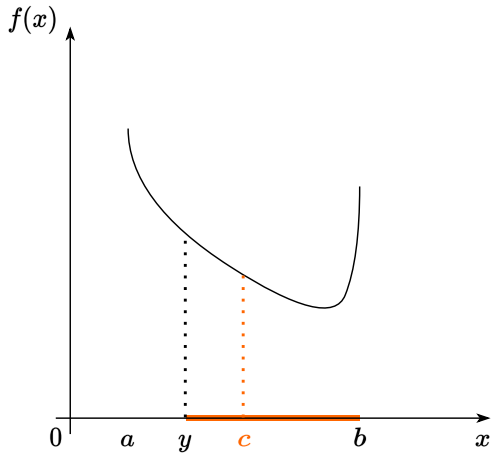


Figure 9: Dichotomy method for unimodal function

Dichotomy method

After another additional measurement, we will surely get $\frac{2}{3}\frac{3}{4}(b-a) = \frac{1}{2}(b-a)$

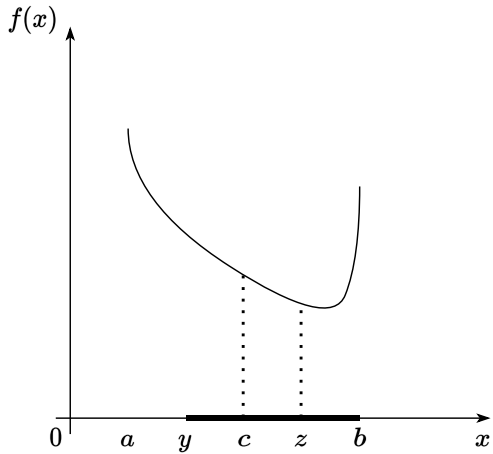


Figure 10: Dichotomy method for unimodal function

Dichotomy method

To sum it up, each subsequent iteration will require at most two function value measurements.

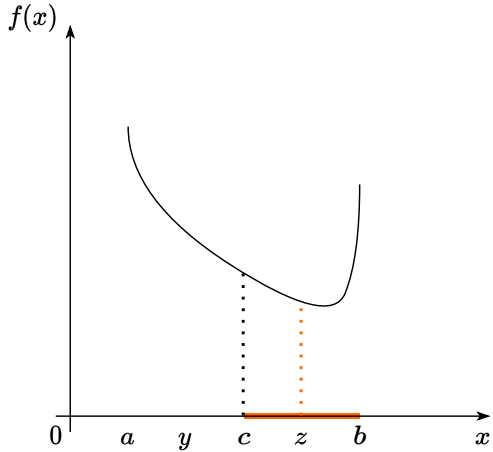


Figure 11: Dichotomy method for unimodal function

Dichotomy method. Algorithm

```
def binary_search(f, a, b, epsilon):
    c = (a + b) / 2
    while abs(b - a) > epsilon:
        y = (a + c) / 2.0
        if f(y) <= f(c):
            b = c
            c = y
        else:
            z = (b + c) / 2.0
            if f(c) <= f(z):
                a = y
                b = z
            else:
                a = c
                c = z
    return c
```

Dichotomy method. Bounds

The length of the line segment on $k+1$-th iteration:

$$\Delta_{k+1} = b_{k+1} - a_{k+1} = \frac{1}{2^k}(b-a)$$

Dichotomy method. Bounds

The length of the line segment on $k+1$-th iteration:

$$\Delta_{k+1} = b_{k+1} - a_{k+1} = \frac{1}{2^k}(b-a)$$

For unimodal functions, this holds if we select the middle of a segment as an output of the iteration x_{k+1}:

$$|x_{k+1} - x_*| \leq \frac{\Delta_{k+1}}{2} \leq \frac{1}{2^{k+1}}(b-a) \leq (0.5)^{k+1} \cdot (b-a)$$

Dichotomy method. Bounds

The length of the line segment on $k+1$-th iteration:

$$\Delta_{k+1} = b_{k+1} - a_{k+1} = \frac{1}{2^k}(b-a)$$

For unimodal functions, this holds if we select the middle of a segment as an output of the iteration x_{k+1}:

$$|x_{k+1} - x_*| \leq \frac{\Delta_{k+1}}{2} \leq \frac{1}{2^{k+1}}(b-a) \leq (0.5)^{k+1} \cdot (b-a)$$

Note, that at each iteration we ask oracle no more, than 2 times, so the number of function evaluations is $N = 2 \cdot k$, which implies:

$$|x_{k+1} - x_*| \leq (0.5)^{\frac{N}{2}+1} \cdot (b-a) \leq (0.707)^N \frac{b-a}{2}$$

Dichotomy method. Bounds

The length of the line segment on $k+1$-th iteration:

$$\Delta_{k+1} = b_{k+1} - a_{k+1} = \frac{1}{2^k}(b-a)$$

For unimodal functions, this holds if we select the middle of a segment as an output of the iteration x_{k+1}:

$$|x_{k+1} - x_*| \leq \frac{\Delta_{k+1}}{2} \leq \frac{1}{2^{k+1}}(b-a) \leq (0.5)^{k+1} \cdot (b-a)$$

Note, that at each iteration we ask oracle no more, than 2 times, so the number of function evaluations is $N = 2 \cdot k$, which implies:

$$|x_{k+1} - x_*| \leq (0.5)^{\frac{N}{2}+1} \cdot (b-a) \leq (0.707)^N \frac{b-a}{2}$$

By marking the right side of the last inequality for ε, we get the number of method iterations needed to achieve ε accuracy:

$$K = \left\lceil \log_2 \frac{b-a}{\varepsilon} - 1 \right\rceil$$

Golden selection

The idea is quite similar to the dichotomy method. There are two golden points on the line segment (left and right) and the insightful idea is, that on the next iteration one of the points will remain the golden point.

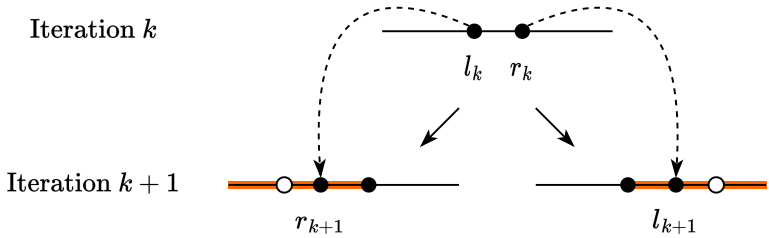


Figure 12: Key idea, that allows us to decrease function evaluations

Golden selection. Algorithm

```
def golden_search(f, a, b, epsilon):
    tau = (sqrt(5) + 1) / 2
    y = a + (b - a) / tau**2
    z = a + (b - a) / tau
    while b - a > epsilon:
        if f(y) <= f(z):
            b = z
            z = y
            y = a + (b - a) / tau**2
        else:
            a = y
            y = z
            z = a + (b - a) / tau
    return (a + b) / 2
```

Golden selection. Bounds

$$|x_{k+1} - x_*| \leq b_{k+1} - a_{k+1} = \left(\frac{1}{\tau}\right)^{N-1} (b-a) \approx 0.618^k (b-a),$$

where $\tau = \frac{\sqrt{5}+1}{2}$.

- The geometric progression constant **more** than the dichotomy method - 0.618 worse than 0.5

Golden selection. Bounds

$$|x_{k+1} - x_*| \leq b_{k+1} - a_{k+1} = \left(\frac{1}{\tau}\right)^{N-1} (b-a) \approx 0.618^k (b-a),$$

where $\tau = \frac{\sqrt{5}+1}{2}$.

- The geometric progression constant **more** than the dichotomy method - 0.618 worse than 0.5
- The number of function calls **is less** than for the dichotomy method - 0.707 worse than 0.618 - (for each iteration of the dichotomy method, except for the first one, the function is calculated no more than 2 times, and for the gold method - no more than one)

Successive parabolic interpolation

Sampling 3 points of a function determines unique parabola. Using this information we will go directly to its minimum. Suppose, we have 3 points $x_1 < x_2 < x_3$ such that line segment $[x_1, x_3]$ contains minimum of a function $f(x)$. Then, we need to solve the following system of equations:

Successive parabolic interpolation

Sampling 3 points of a function determines unique parabola. Using this information we will go directly to its minimum. Suppose, we have 3 points $x_1 < x_2 < x_3$ such that line segment $[x_1, x_3]$ contains minimum of a function $f(x)$. Then, we need to solve the following system of equations:

$$ax_i^2 + bx_i + c = f_i = f(x_i), i = 1, 2, 3$$

Note, that this system is linear, since we need to solve it on a, b, c. Minimum of this parabola will be calculated as:

Successive parabolic interpolation

Sampling 3 points of a function determines unique parabola. Using this information we will go directly to its minimum. Suppose, we have 3 points $x_1 < x_2 < x_3$ such that line segment $[x_1, x_3]$ contains minimum of a function $f(x)$. Then, we need to solve the following system of equations:

$$ax_i^2 + bx_i + c = f_i = f(x_i), i = 1, 2, 3$$

Note, that this system is linear, since we need to solve it on a, b, c. Minimum of this parabola will be calculated as:

$$u = -\frac{b}{2a} = x_2 - \frac{(x_2 - x_1)^2(f_2 - f_3) - (x_2 - x_3)^2(f_2 - f_1)}{2\left[(x_2 - x_1)(f_2 - f_3) - (x_2 - x_3)(f_2 - f_1)\right]}$$

Note, that if $f_2 < f_1, f_2 < f_3$, than u will lie in $[x_1, x_3]$

Successive parabolic interpolation. Algorithm [1]

```
def parabola_search(f, x1, x2, x3, epsilon):
    f1, f2, f3 = f(x1), f(x2), f(x3)
    while x3 - x1 > epsilon:
        u = x2 - ((x2 - x1)**2*(f2 - f3) - (x2 - x3)**2*(f2 - f1))/(2*((x2 - x1)*(f2 - f3) - (x2 - x3)*(f2 - f1)))
        fu = f(u)

        if x2 <= u:
            if f2 <= fu:
                x1, x2, x3 = x1, x2, u
                f1, f2, f3 = f1, f2, fu
            else:
                x1, x2, x3 = x2, u, x3
                f1, f2, f3 = f2, fu, f3
        else:
            if fu <= f2:
                x1, x2, x3 = x1, u, x2
                f1, f2, f3 = f1, fu, f2
            else:
                x1, x2, x3 = u, x2, x3
                f1, f2, f3 = fu, f2, f3
    return (x1 + x3) / 2
```

[1] The convergence of this method is superlinear, but local, which means, that you can take profit from using this method only near some neighbour of optimum. *Here* is the proof of superlinear convergence of order 1.32.

Inexact line search

Sometimes it is enough to find a solution, which will approximately solve out problem. This is very typical scenario for mentioned stepsize selection problem

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$
$$\alpha = \text{argmin } f(x_{k+1})$$

Inexact line search

Sometimes it is enough to find a solution, which will approximately solve out problem. This is very typical scenario for mentioned stepsize selection problem

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$
$$\alpha = \text{argmin } f(x_{k+1})$$

Consider a scalar function $\phi(\alpha)$ at a point x_k:

$$\phi(\alpha) = f(x_k - \alpha \nabla f(x_k)), \alpha \geq 0$$

Inexact line search

Sometimes it is enough to find a solution, which will approximately solve out problem. This is very typical scenario for mentioned stepsize selection problem

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$
$$\alpha = \text{argmin } f(x_{k+1})$$

Consider a scalar function $\phi(\alpha)$ at a point x_k:

$$\phi(\alpha) = f(x_k - \alpha \nabla f(x_k)), \alpha \geq 0$$

The first-order approximation of $\phi(\alpha)$ near $\alpha = 0$ is:

$$\phi(\alpha) \approx f(x_k) - \alpha \nabla f(x_k)^\top \nabla f(x_k)$$

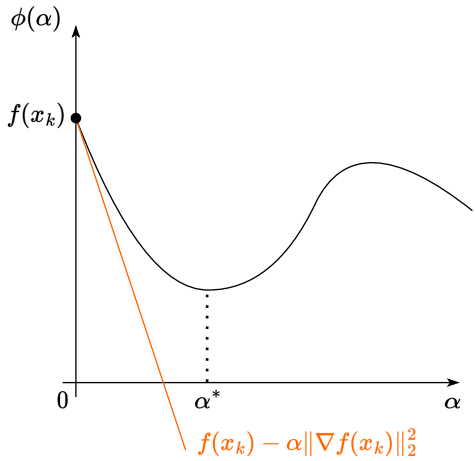


Figure 13: Illustration of Taylor approximation of $\phi_0^I(\alpha)$

Inexact line search. Sufficient Decrease

The inexact line search condition, known as the Armijo condition, states that α should provide sufficient decrease in the function f, satisfying:

$$f(x_k - \alpha \nabla f(x_k)) \leq f(x_k) - c_1 \cdot \alpha \nabla f(x_k)^\top \nabla f(x_k)$$

Inexact line search. Sufficient Decrease

The inexact line search condition, known as the Armijo condition, states that α should provide sufficient decrease in the function f, satisfying:

$$f(x_k - \alpha \nabla f(x_k)) \leq f(x_k) - c_1 \cdot \alpha \nabla f(x_k)^\top \nabla f(x_k)$$

for some constant $c_1 \in (0, 1)$. Note that setting $c_1 = 1$ corresponds to the first-order Taylor approximation of $\phi(\alpha)$. However, this condition can accept very small values of α, potentially slowing down the solution process. Typically, $c_1 \approx 10^{-4}$ is used in practice.

Inexact line search. Sufficient Decrease

The inexact line search condition, known as the Armijo condition, states that α should provide sufficient decrease in the function f, satisfying:

$$f(x_k - \alpha \nabla f(x_k)) \leq f(x_k) - c_1 \cdot \alpha \nabla f(x_k)^\top \nabla f(x_k)$$

for some constant $c_1 \in (0, 1)$. Note that setting $c_1 = 1$ corresponds to the first-order Taylor approximation of $\phi(\alpha)$. However, this condition can accept very small values of α, potentially slowing down the solution process. Typically, $c_1 \approx 10^{-4}$ is used in practice.

Example

If $f(x)$ represents a cost function in an optimization problem, choosing an appropriate c_1 value is crucial. For instance, in a machine learning model training scenario, an improper c_1 might lead to either very slow convergence or missing the minimum.

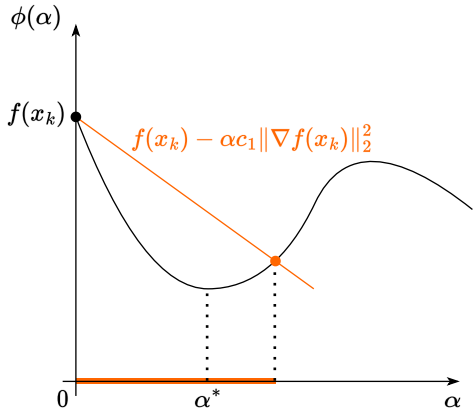


Figure 14: Illustration of sufficient decrease condition with coefficient c_1

Inexact line search. Goldstein Conditions

Consider two linear scalar functions $\phi_1(\alpha)$ and $\phi_2(\alpha)$:

$$\phi_1(\alpha) = f(x_k) - c_1\alpha\|\nabla f(x_k)\|^2$$

$$\phi_2(\alpha) = f(x_k) - c_2\alpha\|\nabla f(x_k)\|^2$$

Inexact line search. Goldstein Conditions

Consider two linear scalar functions $\phi_1(\alpha)$ and $\phi_2(\alpha)$:

$$\phi_1(\alpha) = f(x_k) - c_1\alpha\|\nabla f(x_k)\|^2$$

$$\phi_2(\alpha) = f(x_k) - c_2\alpha\|\nabla f(x_k)\|^2$$

The Goldstein-Armijo conditions locate the function $\phi(\alpha)$ between $\phi_1(\alpha)$ and $\phi_2(\alpha)$. Typically, $c_1 = \rho$ and $c_2 = 1 - \rho$, with $\rho \in (0.5, 1)$.

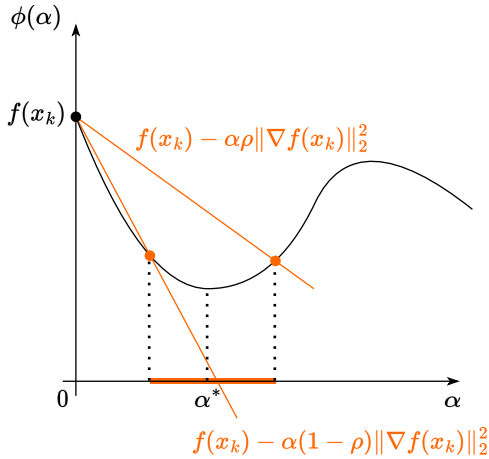


Figure 15: Illustration of Goldstein conditions

Inexact line search. Curvature Condition

To avoid excessively short steps, we introduce a second criterion:

$$-\nabla f(x_k - \alpha \nabla f(x_k))^\top \nabla f(x_k) \geq c_2 \nabla f(x_k)^\top (-\nabla f(x_k))$$

Inexact line search. Curvature Condition

To avoid excessively short steps, we introduce a second criterion:

$$-\nabla f(x_k - \alpha \nabla f(x_k))^\top \nabla f(x_k) \geq c_2 \nabla f(x_k)^\top (-\nabla f(x_k))$$

for some $c_2 \in (c_1, 1)$. Here, c_1 is from the Armijo condition. The left-hand side is the derivative $\nabla_\alpha \phi(\alpha)$, ensuring that the slope of $\phi(\alpha)$ at the target point is at least c_2 times the initial slope $\nabla_\alpha \phi(\alpha)(0)$. Commonly, $c_2 \approx 0.9$ is used for Newton or quasi-Newton methods. Together, the sufficient decrease and curvature conditions form the Wolfe conditions.

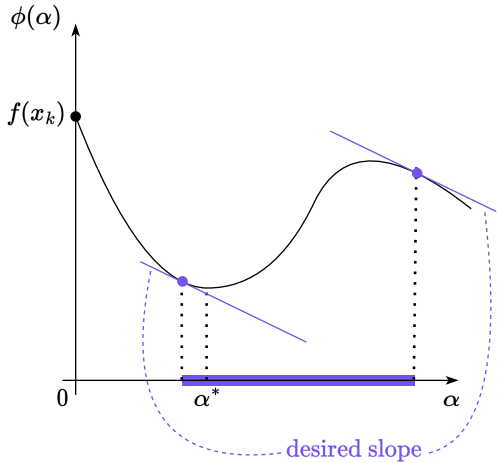


Figure 16: Illustration of curvature condition

Inexact line search. Wolfe Condition

$$-\nabla f(x_k - \alpha \nabla f(x_k))^\top \nabla f(x_k) \geq c_2 \nabla f(x_k)^\top (-\nabla f(x_k))$$

Together, the sufficient decrease and curvature conditions form the Wolfe conditions.

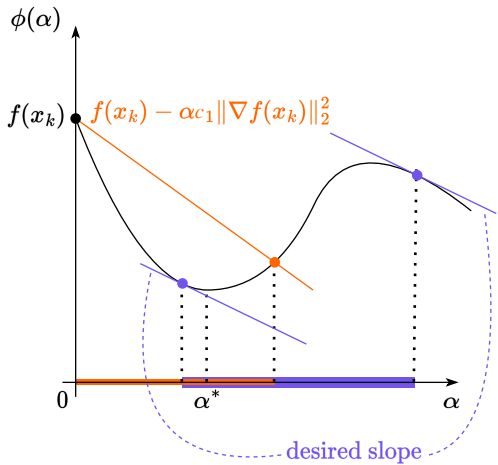


Figure 17: Illustration of Wolfe condition

Backtracking Line Search

Backtracking line search is a technique to find a step size that satisfies the Armijo condition, Goldstein conditions, or other criteria of inexact line search. It begins with a relatively large step size and iteratively scales it down until a condition is met.

Backtracking Line Search

Backtracking line search is a technique to find a step size that satisfies the Armijo condition, Goldstein conditions, or other criteria of inexact line search. It begins with a relatively large step size and iteratively scales it down until a condition is met.

Algorithm:

1. Choose an initial step size, α_0, and parameters $\beta \in (0, 1)$ and $c_1 \in (0, 1)$.

Backtracking Line Search

Backtracking line search is a technique to find a step size that satisfies the Armijo condition, Goldstein conditions, or other criteria of inexact line search. It begins with a relatively large step size and iteratively scales it down until a condition is met.

Algorithm:

1. Choose an initial step size, α_0, and parameters $\beta \in (0, 1)$ and $c_1 \in (0, 1)$.
2. Check if the chosen step size satisfies the chosen condition (e.g., Armijo condition).

Backtracking Line Search

Backtracking line search is a technique to find a step size that satisfies the Armijo condition, Goldstein conditions, or other criteria of inexact line search. It begins with a relatively large step size and iteratively scales it down until a condition is met.

Algorithm:

1. Choose an initial step size, α_0, and parameters $\beta \in (0, 1)$ and $c_1 \in (0, 1)$.
2. Check if the chosen step size satisfies the chosen condition (e.g., Armijo condition).
3. If the condition is satisfied, stop; else, set $\alpha := \beta\alpha$ and repeat step 2.

Backtracking Line Search

Backtracking line search is a technique to find a step size that satisfies the Armijo condition, Goldstein conditions, or other criteria of inexact line search. It begins with a relatively large step size and iteratively scales it down until a condition is met.

Algorithm:

1. Choose an initial step size, α_0, and parameters $\beta \in (0, 1)$ and $c_1 \in (0, 1)$.
2. Check if the chosen step size satisfies the chosen condition (e.g., Armijo condition).
3. If the condition is satisfied, stop; else, set $\alpha := \beta\alpha$ and repeat step 2.

Backtracking Line Search

Backtracking line search is a technique to find a step size that satisfies the Armijo condition, Goldstein conditions, or other criteria of inexact line search. It begins with a relatively large step size and iteratively scales it down until a condition is met.

Algorithm:

1. Choose an initial step size, α_0, and parameters $\beta \in (0, 1)$ and $c_1 \in (0, 1)$.
2. Check if the chosen step size satisfies the chosen condition (e.g., Armijo condition).
3. If the condition is satisfied, stop; else, set $\alpha := \beta\alpha$ and repeat step 2.

The step size α is updated as

$$\alpha_{k+1} := \beta\alpha_k$$

in each iteration until the chosen condition is satisfied.

Example

In machine learning model training, the backtracking line search can be used to adjust the learning rate. If the loss doesn't decrease sufficiently, the learning rate is reduced multiplicatively until the Armijo condition is met.

Numerical illustration

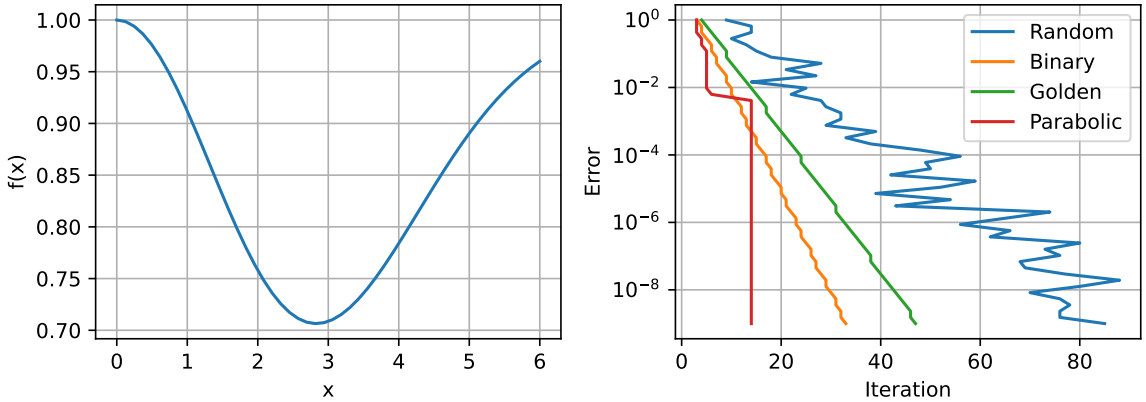


Figure 18: Comparison of different line search algorithms

Open In Colab ♣

Matrix calculus

Gradient

Let $f(x) : \mathbb{R}^n \to \mathbb{R}$, then vector, which contains all first-order partial derivatives:

$$\nabla f(x) = \frac{df}{dx} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

Gradient

Let $f(x) : \mathbb{R}^n \to \mathbb{R}$, then vector, which contains all first-order partial derivatives:

$$\nabla f(x) = \frac{df}{dx} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

named gradient of $f(x)$. This vector indicates the direction of the steepest ascent. Thus, vector $-\nabla f(x)$ means the direction of the steepest descent of the function in the point. Moreover, the gradient vector is always orthogonal to the contour line in the point.

Example

For the function $f(x, y) = x^2 + y^2$, the gradient is:

$$\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

This gradient points in the direction of the steepest ascent of the function.

Question

How does the magnitude of the gradient relate to the steepness of the function?

Hessian

Let $f(x) : \mathbb{R}^n \to \mathbb{R}$, then matrix, containing all the second order partial derivatives:

$$f''(x) = \nabla^2 f(x) = \frac{\partial^2 f}{\partial x_i \partial x_j} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

Hessian

Let $f(x) : \mathbb{R}^n \to \mathbb{R}$, then matrix, containing all the second order partial derivatives:

$$f''(x) = \nabla^2 f(x) = \frac{\partial^2 f}{\partial x_i \partial x_j} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

In fact, Hessian could be a tensor in such a way: $(f(x) : \mathbb{R}^n \to \mathbb{R}^m)$ is just 3d tensor, every slice is just hessian of corresponding scalar function $\left(\nabla^2 f_1(x), \ldots, \nabla^2 f_m(x)\right)$.

Example

For the function $f(x, y) = x^2 + y^2$, the Hessian is:

$$H_f(x, y) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

This matrix provides information about the curvature of the function in different directions.

Question

How can the Hessian matrix be used to determine the concavity or convexity of a function?

Schwartz theorem

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function. If the mixed partial derivatives $\frac{\partial^2 f}{\partial x_i \partial x_j}$ and $\frac{\partial^2 f}{\partial x_j \partial x_i}$ are both continuous on an open set containing a point a, then they are equal at the point a. That is,

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(a) = \frac{\partial^2 f}{\partial x_j \partial x_i}(a)$$

Schwartz theorem

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function. If the mixed partial derivatives $\frac{\partial^2 f}{\partial x_i \partial x_j}$ and $\frac{\partial^2 f}{\partial x_j \partial x_i}$ are both continuous on an open set containing a point a, then they are equal at the point a. That is,

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(a) = \frac{\partial^2 f}{\partial x_j \partial x_i}(a)$$

Given the Schwartz theorem, if the mixed partials are continuous on an open set, the Hessian matrix is symmetric. That means the entries above the main diagonal mirror those below the main diagonal:

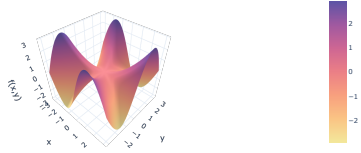
$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i} \quad \nabla^2 f(x) = (\nabla^2 f(x))^T$$

This symmetry simplifies computations and analysis involving the Hessian matrix in various applications, particularly in optimization.

Schwartz counterexample

$$f(x,y) = \begin{cases} \frac{xy(x^2-y^2)}{x^2+y^2} & \text{for } (x,\, y) \neq (0,\, 0), \\ 0 & \text{for } (x,y) = (0,0). \end{cases}$$

Counterexample ♣



One can verify, that $\frac{\partial^2 f}{\partial x \partial y}(0,0) \neq \frac{\partial^2 f}{\partial y \partial x}(0,0)$, although the mixed partial derivatives do exist, and at every other point the symmetry does hold.

Jacobian

The extension of the gradient of multidimensional $f(x) : \mathbb{R}^n \to \mathbb{R}^m$ is the following matrix:

$$J_f = f'(x) = \frac{df}{dx^T} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_n} & \frac{\partial f_2}{\partial x_n} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

This matrix provides information about the rate of change of the function with respect to its inputs.

Question

Can we somehow connect those three definitions above (gradient, jacobian, and hessian) using a single correct statement?

Example

For the function

$$f(x, y) = \begin{bmatrix} x + y \\ x - y \end{bmatrix},$$

the Jacobian is:

$$J_f(x, y) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Question

How does the Jacobian matrix relate to the gradient for scalar-valued functions?

Summary

$$f(x) : X \to Y; \qquad \frac{\partial f(x)}{\partial x} \in G$$

X	Y	G	Name
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$f'(x)$ (derivative)
$\mathbb{R}^n$	$\mathbb{R}$	$\mathbb{R}^{\ltimes}$	$\frac{\partial f}{\partial x_i}$ (gradient)
$\mathbb{R}^n$	$\mathbb{R}^m$	$\mathbb{R}^{n \times m}$	$\frac{\partial f_i}{\partial x_j}$ (jacobian)
$\mathbb{R}^{m \times n}$	$\mathbb{R}$	$\mathbb{R}^{m \times n}$	$\frac{\partial f}{\partial x_{ij}}$

Differentials

Theorem

Let $x \in S$ be an interior point of the set S, and let $D : U \to V$ be a linear operator. We say that the function f is differentiable at the point x with derivative D if for all sufficiently small $h \in U$ the following decomposition holds:

$$f(x + h) = f(x) + D[h] + o(\|h\|)$$

If for any linear operator $D : U \to V$ the function f is not differentiable at the point x with derivative D, then we say that f is not differentiable at the point x.

Differentials

After obtaining the differential notation of df we can retrieve the gradient using the following formula:

$$df(x) = \langle \nabla f(x), dx \rangle$$

Differentials

After obtaining the differential notation of df we can retrieve the gradient using the following formula:

$$df(x) = \langle \nabla f(x), dx \rangle$$

Then, if we have a differential of the above form and we need to calculate the second derivative of the matrix/vector function, we treat "old" dx as the constant dx_1, then calculate $d(df) = d^2 f(x)$

$$d^2 f(x) = \langle \nabla^2 f(x) dx_1, dx \rangle = \langle H_f(x) dx_1, dx \rangle$$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$
- $d\left(\det X\right) = \det X \langle X^{-T}, dX\rangle$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$
- $d\left(\det X\right) = \det X \langle X^{-T}, dX\rangle$
- $d\left(\mathrm{tr}\ X\right) = \langle I, dX\rangle$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$
- $d\left(\det X\right) = \det X \langle X^{-T}, dX\rangle$
- $d\left(\mathrm{tr}\ X\right) = \langle I, dX\rangle$
- $df(g(x)) = \dfrac{df}{dg} \cdot dg(x)$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$
- $d\left(\det X\right) = \det X \langle X^{-T}, dX\rangle$
- $d\left(\mathrm{tr}\ X\right) = \langle I, dX\rangle$
- $df(g(x)) = \dfrac{df}{dg} \cdot dg(x)$
- $H = \left(J(\nabla f)\right)^T$

Differential properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^T) = (dX)^T$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$
- $d\left(\det X\right) = \det X \langle X^{-T}, dX\rangle$
- $d\left(\mathrm{tr}\ X\right) = \langle I, dX\rangle$
- $df(g(x)) = \dfrac{df}{dg} \cdot dg(x)$
- $H = \left(J(\nabla f)\right)^T$
- $d(X^{-1}) = -X^{-1}(dX)X^{-1}$

Matrix calculus. Example 1

Example

Find $df, \nabla f(x)$, if $f(x) = \langle x, Ax \rangle - b^T x + c$.

Matrix calculus. Example 2

Example

Find $df, \nabla f(x)$, if $f(x) = \ln\langle x, Ax\rangle$.

1. It is essential for A to be positive definite, because it is a logarithm argument. So, $A \in \mathbb{S}^n_{++}$Let's find the differential first:

$$df = d\left(\ln\langle x, Ax\rangle\right) = \frac{d\left(\langle x, Ax\rangle\right)}{\langle x, Ax\rangle} = \frac{\langle dx, Ax\rangle + \langle x, d(Ax)\rangle}{\langle x, Ax\rangle} =$$

$$= \frac{\langle Ax, dx\rangle + \langle x, Adx\rangle}{\langle x, Ax\rangle} = \frac{\langle Ax, dx\rangle + \langle A^T x, dx\rangle}{\langle x, Ax\rangle} = \frac{\langle (A + A^T)x, dx\rangle}{\langle x, Ax\rangle}$$

Matrix calculus. Example 2

Example

Find $df, \nabla f(x)$, if $f(x) = \ln\langle x, Ax\rangle$.

1. It is essential for A to be positive definite, because it is a logarithm argument. So, $A \in \mathbb{S}^n_{++}$Let's find the differential first:

$$df = d\left(\ln\langle x, Ax\rangle\right) = \frac{d\left(\langle x, Ax\rangle\right)}{\langle x, Ax\rangle} = \frac{\langle dx, Ax\rangle + \langle x, d(Ax)\rangle}{\langle x, Ax\rangle} =$$
$$= \frac{\langle Ax, dx\rangle + \langle x, Adx\rangle}{\langle x, Ax\rangle} = \frac{\langle Ax, dx\rangle + \langle A^T x, dx\rangle}{\langle x, Ax\rangle} = \frac{\langle (A + A^T)x, dx\rangle}{\langle x, Ax\rangle}$$

2. Note, that our main goal is to derive the form $df = \langle \cdot, dx\rangle$

$$df = \left\langle \frac{2Ax}{\langle x, Ax\rangle}, dx \right\rangle$$

Hence, the gradient is $\nabla f(x) = \dfrac{2Ax}{\langle x, Ax\rangle}$

Matrix calculus. Example 3

Example

Find $df, \nabla f(X)$, if $f(X) = \langle S, X \rangle - \log \det X$.

Automatic differentiation

Problem

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

. . .

Such problems typically arise in machine learning, when you need to find optimal hyperparameters w of an ML model (i.e. train a neural network). You may use a lot of algorithms to approach this problem, but given the modern size of the problem, where d could be dozens of billions it is very challenging to solve this problem without information about the gradients using zero-order optimization algorithms. That is why it would be beneficial to be able to calculate the gradient vector $\nabla_w L = \left(\frac{\partial L}{\partial w_1}, \dots, \frac{\partial L}{\partial w_d} \right)^T$. Typically, first-order methods perform much better in huge-scale optimization, while second-order methods require too much memory.

Finite differences

The naive approach to get approximate values of gradients is **Finite differences** approach. For each coordinate, one can calculate the partial derivative approximation:

$$\frac{\partial L}{\partial w_k}(w) \approx \frac{L(w + \varepsilon e_k) - L(w)}{\varepsilon}, \quad e_k = (0, \dots, \underset{k}{1}, \dots, 0)$$

Question

If the time needed for one calculation of $L(w)$ is T, what is the time needed for calculating $\nabla_w L$ with this approach?
** Answer ** $2dT$, which is extremely long for the huge scale optimization. Moreover, this exact scheme is unstable, which means that you will have to choose between accuracy and stability.

Theorem

There is an algorithm to compute $\nabla_w L$ in $\mathcal{O}(T)$ operations. [2]

[2] Linnainmaa S. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

Forward mode automatic differentiation

To dive deep into the idea of automatic differentiation we will consider a simple function for calculating derivatives:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

Let's draw a *computational graph* of this function:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

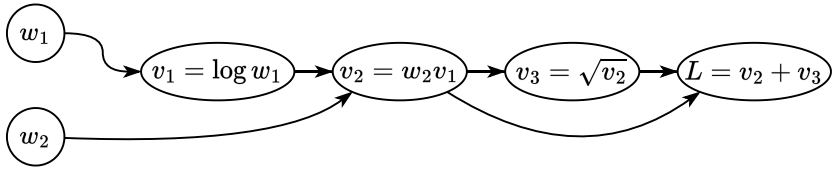


Figure 19: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Forward mode automatic differentiation

To dive deep into the idea of automatic differentiation we will consider a simple function for calculating derivatives:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

Let's draw a *computational graph* of this function:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

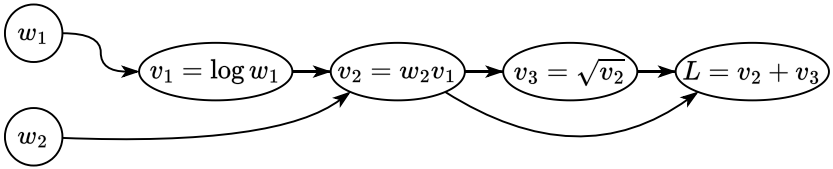


Figure 19: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Let's go from the beginning of the graph to the end and calculate the derivative $\dfrac{\partial L}{\partial w_1}$.

Forward mode automatic differentiation

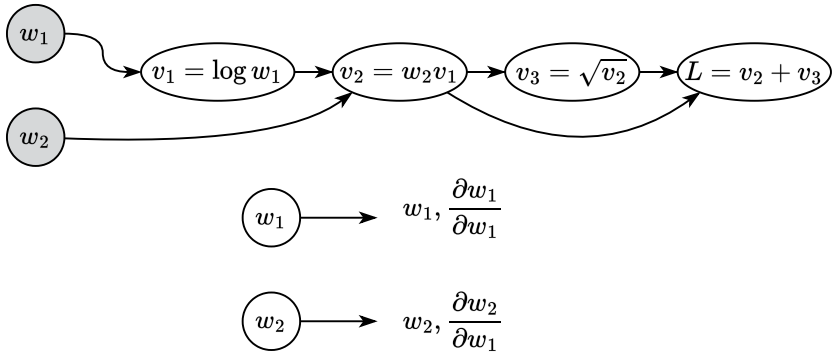


Figure 20: Illustration of forward mode automatic differentiation

Function

$w_1 = w_1, w_2 = w_2$

Derivative

$\frac{\partial w_1}{\partial w_1} = 1, \frac{\partial w_2}{\partial w_1} = 0$

Forward mode automatic differentiation

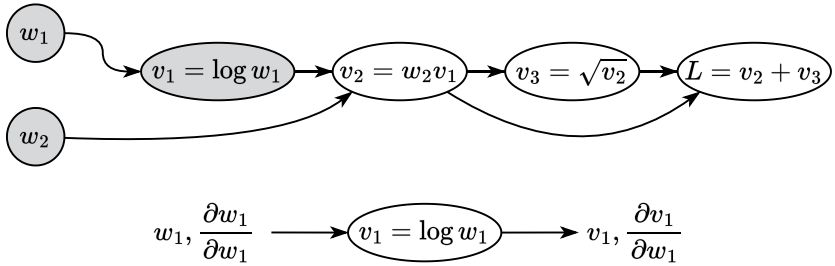


Figure 21: Illustration of forward mode automatic differentiation

Function

$v_1 = \log w_1$

Derivative

$\frac{\partial v_1}{\partial w_1} = \frac{\partial v_1}{\partial w_1}\frac{\partial w_1}{\partial w_1} = \frac{1}{w_1}1$

Forward mode automatic differentiation

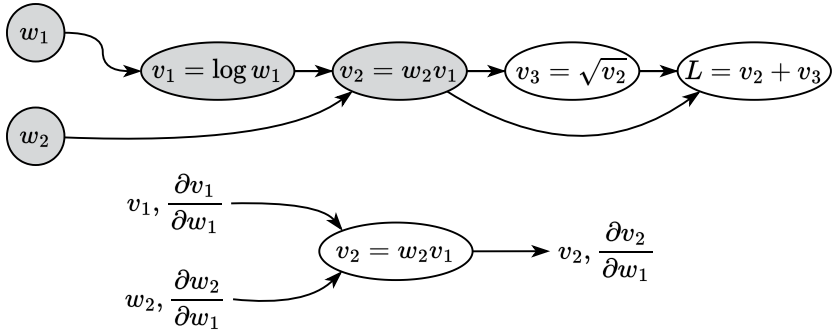


Figure 22: Illustration of forward mode automatic differentiation

Function

$v_2 = w_2 v_1$

Derivative

$$\frac{\partial v_2}{\partial w_1} = \frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial w_1} + \frac{\partial v_2}{\partial w_2}\frac{\partial w_2}{\partial w_1} = w_2 \frac{\partial v_1}{\partial w_1} + v_1 \frac{\partial w_2}{\partial w_1}$$

Forward mode automatic differentiation

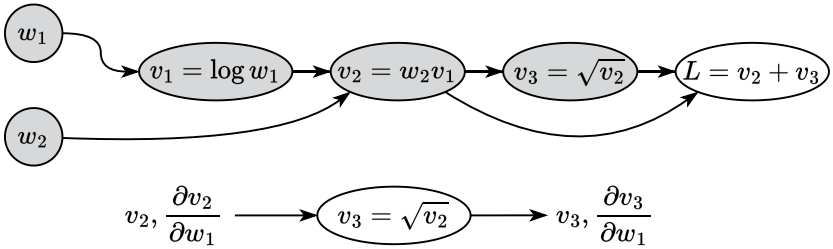


Figure 23: Illustration of forward mode automatic differentiation

Function

$v_3 = \sqrt{v_2}$

Derivative

$\frac{\partial v_3}{\partial w_1} = \frac{\partial v_3}{\partial v_2}\frac{\partial v_2}{\partial w_1} = \frac{1}{2\sqrt{v_2}}\frac{\partial v_2}{\partial w_1}$

Forward mode automatic differentiation

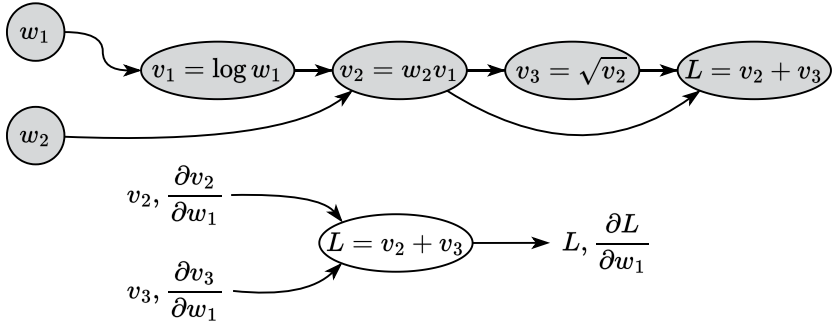


Figure 24: Illustration of forward mode automatic differentiation

Function

$L = v_2 + v_3$

Derivative

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial w_1} + \frac{\partial L}{\partial v_3}\frac{\partial v_3}{\partial w_1} = 1\frac{\partial v_2}{\partial w_1} + 1\frac{\partial v_3}{\partial w_1}$$

Make the similar computations for $\dfrac{\partial L}{\partial w_2}$

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

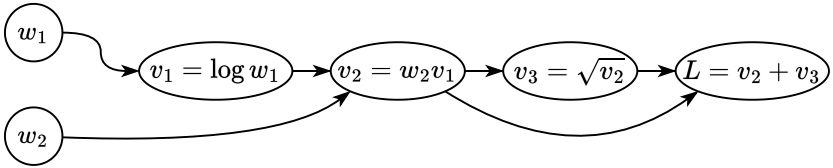


Figure 25: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Forward mode automatic differentiation example

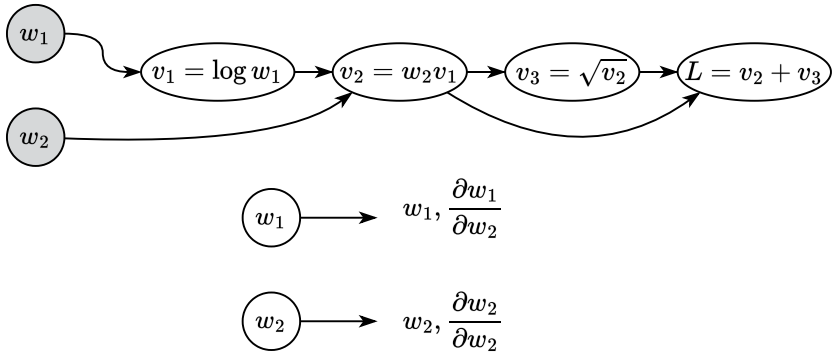


Figure 26: Illustration of forward mode automatic differentiation

Function

$w_1 = w_1, w_2 = w_2$

Derivative

$\frac{\partial w_1}{\partial w_2} = 0, \frac{\partial w_2}{\partial w_2} = 1$

Forward mode automatic differentiation example

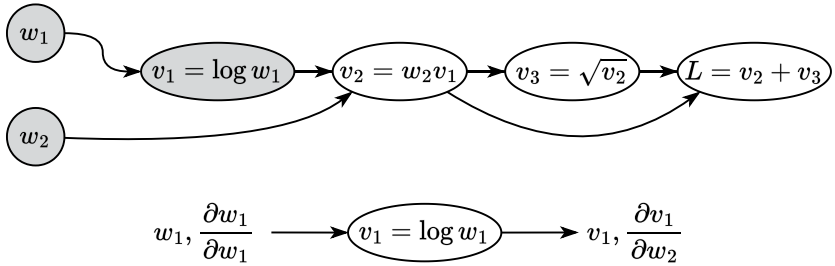


Figure 27: Illustration of forward mode automatic differentiation

Function

$v_1 = \log w_1$

Derivative

$\frac{\partial v_1}{\partial w_2} = \frac{\partial v_1}{\partial w_2}\frac{\partial w_2}{\partial w_2} = 0 \cdot 1$

Forward mode automatic differentiation example

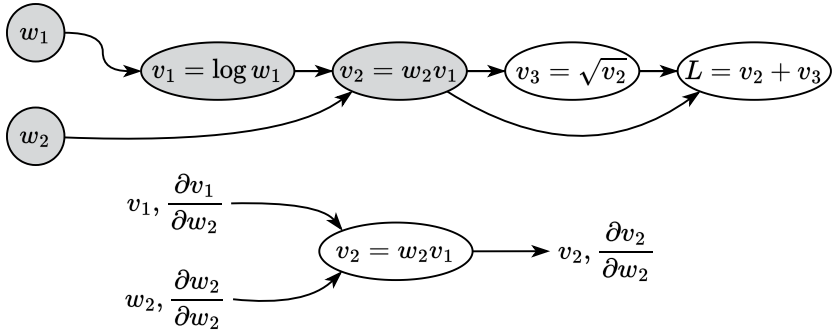


Figure 28: Illustration of forward mode automatic differentiation

Function

$v_2 = w_2 v_1$

Derivative

$$\frac{\partial v_2}{\partial w_2} = \frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial w_2} + \frac{\partial v_2}{\partial w_2}\frac{\partial w_2}{\partial w_2} = w_2 \frac{\partial v_1}{\partial w_2} + v_1 \frac{\partial w_2}{\partial w_2}$$

Forward mode automatic differentiation example

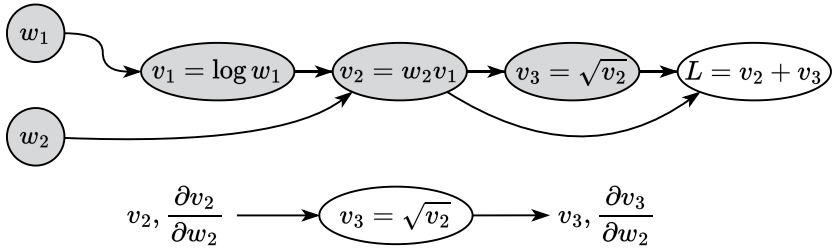


Figure 29: Illustration of forward mode automatic differentiation

Function

$v_3 = \sqrt{v_2}$

Derivative

$\frac{\partial v_3}{\partial w_2} = \frac{\partial v_3}{\partial v_2}\frac{\partial v_2}{\partial w_2} = \frac{1}{2\sqrt{v_2}}\frac{\partial v_2}{\partial w_2}$

Forward mode automatic differentiation example

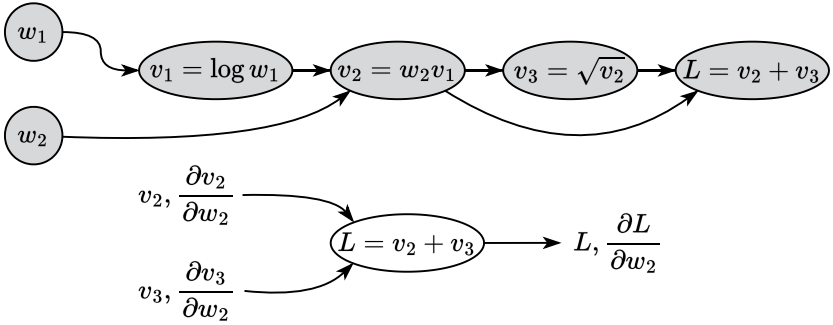


Figure 30: Illustration of forward mode automatic differentiation

Function

$L = v_2 + v_3$

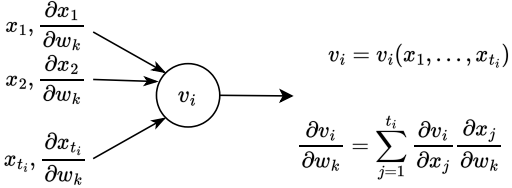
Derivative

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial w_2} + \frac{\partial L}{\partial v_3}\frac{\partial v_3}{\partial w_2} = 1\frac{\partial v_2}{\partial w_2} + 1\frac{\partial v_3}{\partial w_2}$$

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$



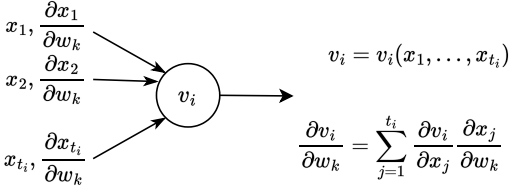
- For $i = 1, \ldots, N$:

Illustration of forward chain rule to calculate the

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$



- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

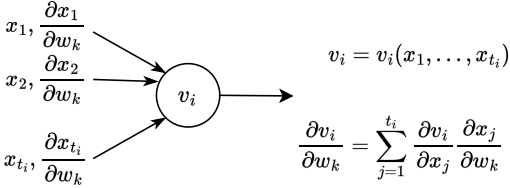
$$v_i = v_i(x_1, \ldots, x_{t_i})$$

Illustration of forward chain rule to calculate the

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$



- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

$$v_i = v_i(x_1, \ldots, x_{t_i})$$

  - Compute the derivative $\overline{v_i}$ using the forward chain rule:

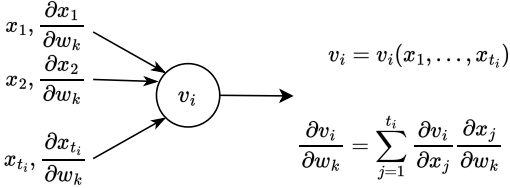
$$\overline{v_i} = \sum_{j=1}^{t_i} \frac{\partial v_i}{\partial x_j} \frac{\partial x_j}{\partial w_k}$$

Illustration of forward chain rule to calculate the

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$



- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

$$v_i = v_i(x_1, \ldots, x_{t_i})$$

  - Compute the derivative $\overline{v_i}$ using the forward chain rule:

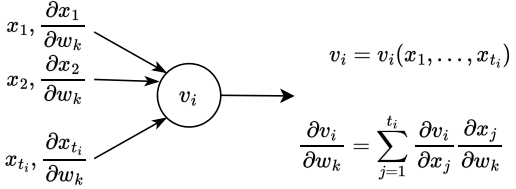
$$\overline{v_i} = \sum_{j=1}^{t_i} \frac{\partial v_i}{\partial x_j} \frac{\partial x_j}{\partial w_k}$$

Illustration of forward chain rule to calculate the

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$



- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

$$v_i = v_i(x_1, \ldots, x_{t_i})$$

  - Compute the derivative $\overline{v_i}$ using the forward chain rule:

$$\overline{v_i} = \sum_{j=1}^{t_i} \frac{\partial v_i}{\partial x_j} \frac{\partial x_j}{\partial w_k}$$

Note, that this approach does not require storing all intermediate computations, but one can see, that for calculating the derivative $\dfrac{\partial L}{\partial w_k}$ we need $\mathcal{O}(T)$ operations. This means, that for the whole gradient, we need $d\mathcal{O}(T)$ operations, which is the same as for finite differences, but we do not have stability issues, or inaccuracies now (the formulas above are exact).

Illustration of forward chain rule to calculate the

Backward mode automatic differentiation

We will consider the same function with a computational graph:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

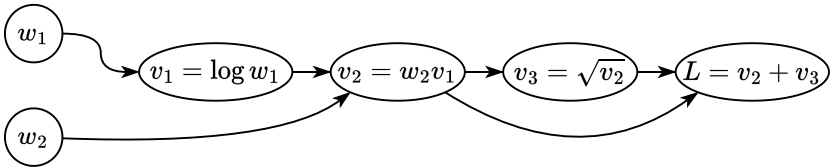


Figure 32: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Backward mode automatic differentiation

We will consider the same function with a computational graph:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

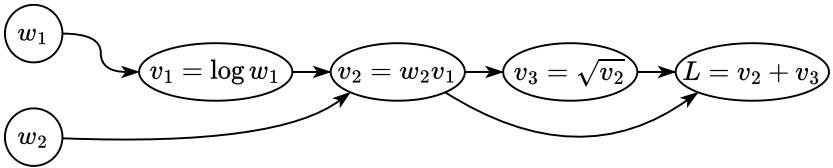


Figure 32: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Assume, that we have some values of the parameters w_1, w_2 and we have already performed a forward pass (i.e. single propagation through the computational graph from left to right). Suppose, also, that we somehow saved all intermediate values of v_i. Let's go from the end of the graph to the beginning and calculate the derivatives $\dfrac{\partial L}{\partial w_1}, \dfrac{\partial L}{\partial w_1}$:

Backward mode automatic differentiation

Question

Note, that for the same price of computations as it was in the forward mode we have the full vector of gradient $\nabla_w L$. Is it a free lunch? What is the cost of acceleration?

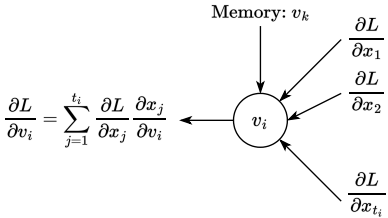
Answer

Note, that for using the reverse mode AD you need to store all intermediate computations from the forward pass. This problem could be somehow mitigated with the gradient checkpointing approach, which involves necessary recomputations of some intermediate values. This could significantly reduce the memory footprint of the large machine-learning model.

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \dots, \frac{\partial v_N}{\partial w_d}\right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$

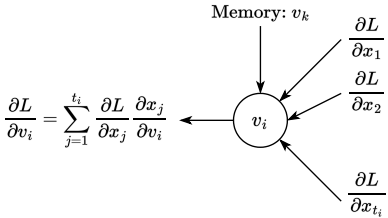


- **FORWARD PASS**
  For $i = 1, \dots, N$:

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \dots, \frac{\partial v_N}{\partial w_d}\right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$



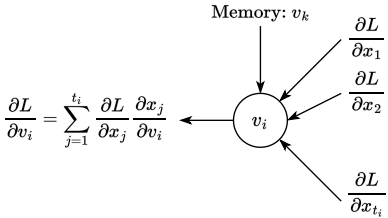
- **FORWARD PASS**

  For $i = 1, \dots, N$:
  - Compute and store the values of v_i as a function of its parents (inputs)

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \dots, \frac{\partial v_N}{\partial w_d}\right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$



- **FORWARD PASS**

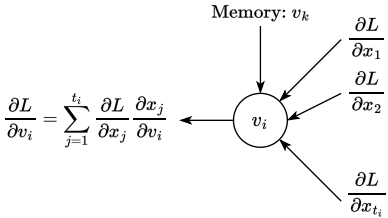
  For $i = 1, \dots, N$:
  - Compute and store the values of v_i as a function of its parents (inputs)
- **BACKWARD PASS**

  For $i = N, \dots, 1$:

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \dots, \frac{\partial v_N}{\partial w_d}\right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$



- **FORWARD PASS**

  For $i = 1, \dots, N$:
  - Compute and store the values of v_i as a function of its parents (inputs)
- **BACKWARD PASS**

  For $i = N, \dots, 1$:
  - Compute the derivative $\overline{v_i}$ using the backward chain rule and information from all of its children (outputs) $(x_1, \dots, x_{t_i})$:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \sum_{j=1}^{t_i} \frac{\partial L}{\partial x_j} \frac{\partial x_j}{\partial v_i}$$

Choose your fighter

Which of the AD modes would you choose (forward/ reverse) for the following computational graph of primitive arithmetic operations? Suppose, you are needed to compute the jacobian $J = \left\{ \frac{\partial L_i}{\partial w_j} \right\}_{i,j}$

Note, that the reverse mode computational time is proportional to the number of outputs here, while the forward mode works proportionally to the number of inputs there. This is why it would be a good idea to consider the forward mode AD.

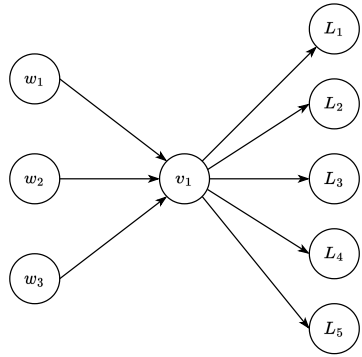


Figure 34: Which mode would you choose for calculating gradients there?

Choose your fighter

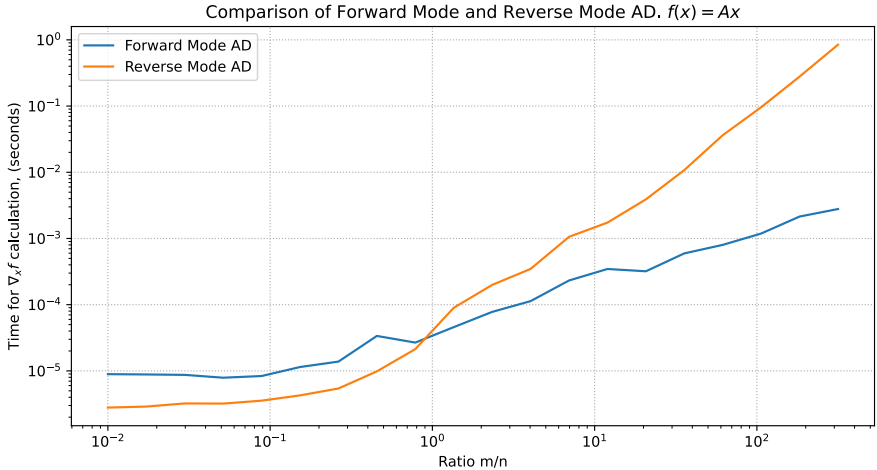


Figure 35: This graph nicely illustrates the idea of choice between the modes. The $n = 100$ dimension is fixed and the graph presents the time needed for Jacobian calculation w.r.t. x for $f(x) = Ax$

Choose your fighter

Which of the AD modes would you choose (forward/ reverse) for the following computational graph of primitive arithmetic operations? Suppose, you are needed to compute the jacobian $J = \left\{ \frac{\partial L_i}{\partial w_j} \right\}_{i,j}$. Note, that G is an arbitrary computational graph

It is generally impossible to say it without some knowledge about the specific structure of the graph G. Note, that there are also plenty of advanced approaches to mix forward and reverse mode AD, based on the specific G structure.

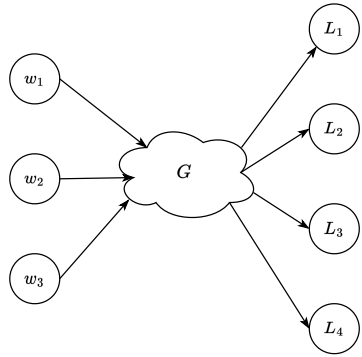


Figure 36: Which mode would you choose for calculating gradients there?

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.

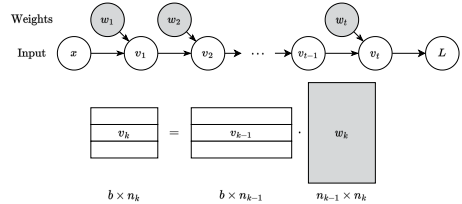


Figure 37: Feedforward neural network architecture

BACKWARD

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:

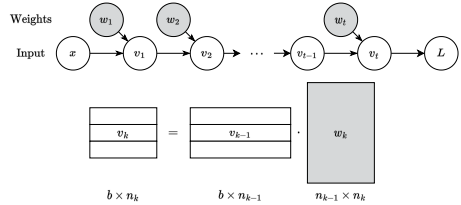


Figure 37: Feedforward neural network architecture

BACKWARD

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.

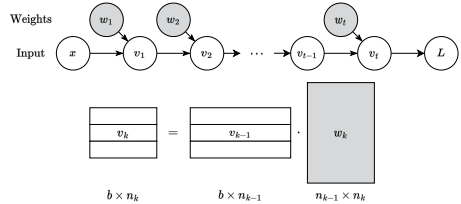


Figure 37: Feedforward neural network architecture

BACKWARD

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

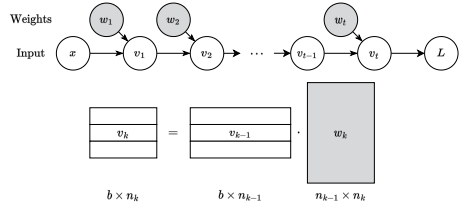


Figure 37: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$

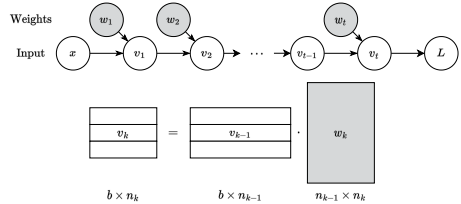


Figure 37: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$
- For $k = t, t-1, \ldots, 1$:

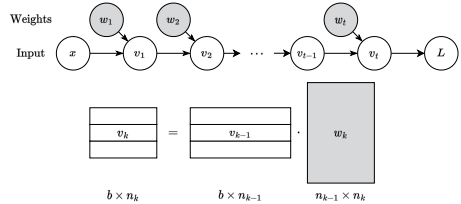


Figure 37: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$
- For $k = t, t-1, \ldots, 1$:
  - $\underset{b \times n_k}{\dfrac{\partial L}{\partial v_k}} = \underset{b \times n_{k+1}}{\dfrac{\partial L}{\partial v_{k+1}}} \underset{n_{k+1} \times n_k}{\dfrac{\partial v_{k+1}}{\partial v_k}}$

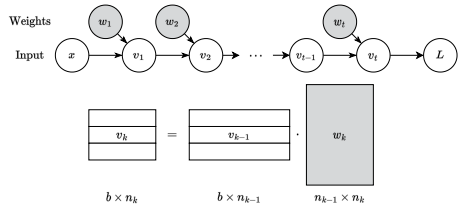


Figure 37: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$
- For $k = t, t-1, \ldots, 1$:
  - $$\underset{b \times n_k}{\frac{\partial L}{\partial v_k}} = \underset{b \times n_{k+1}}{\frac{\partial L}{\partial v_{k+1}}} \underset{n_{k+1} \times n_k}{\frac{\partial v_{k+1}}{\partial v_k}}$$
  - $$\underset{b \times n_{k-1} \cdot n_k}{\frac{\partial L}{\partial w_k}} = \underset{b \times n_{k+1}}{\frac{\partial L}{\partial v_{k+1}}} \cdot \underset{n_{k+1} \times n_{k-1} \cdot n_k}{\frac{\partial v_{k+1}}{\partial w_k}}$$

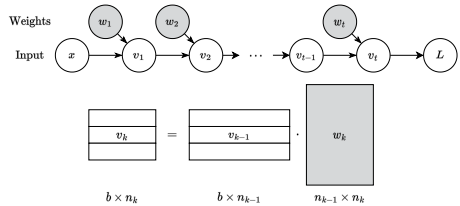


Figure 37: Feedforward neural network architecture

What automatic differentiation (AD) is NOT:

- AD is not a finite differences

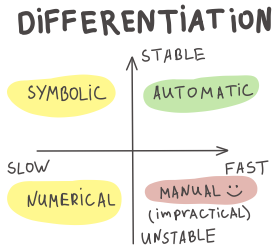


Figure 38: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative

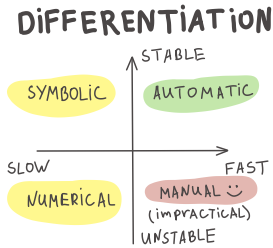


Figure 38: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule

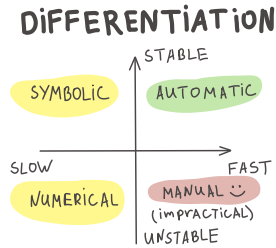


Figure 38: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule
- AD is not just backpropagation

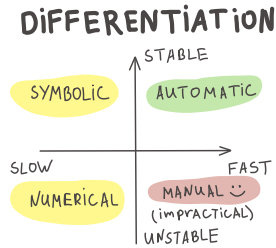


Figure 38: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule
- AD is not just backpropagation
- AD (reverse mode) is time-efficient and numerically stable

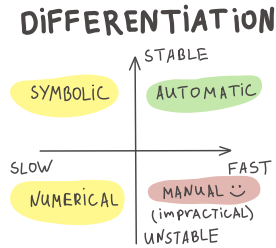


Figure 38: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule
- AD is not just backpropagation
- AD (reverse mode) is time-efficient and numerically stable
- AD (reverse mode) is memory inefficient (you need to store all intermediate computations from the forward pass).

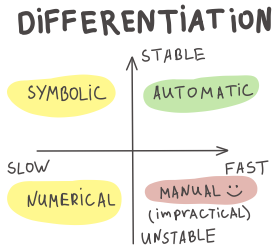


Figure 38: Different approaches for taking derivatives

Code

Open In Colab ♣