Automatic Differentiation (AD)

Why Do We Need Derivatives in Machine Learning?

At the core of training most machine learning models is **optimization**. We want to find the model parameters θ that minimize a loss function $L(\theta)$.

A very popular and effective family of optimization algorithms is **gradient-based optimization**. The most famous example is Gradient Descent.

- The update rule for Gradient Descent is:

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta L(\theta)$$

- $\nabla_\theta L(\theta)$ is the **gradient** of the loss function. It's a vector of partial derivatives that tells us the direction of the steepest ascent.
- To minimize the loss, we need to move in the opposite direction of the gradient.

Key takeaway: Computing derivatives (gradients) accurately and efficiently is absolutely critical for modern machine learning.

Four Ways to Compute Derivatives

There are four main approaches to get the derivatives we need:

1. **Manual Differentiation:** Using pen and paper to derive the gradient formula.
2. **Numerical Differentiation:** Approximating the derivative using finite differences.
3. **Symbolic Differentiation:** Manipulating mathematical expressions using rules of calculus.
4. **Automatic Differentiation (AD):** Breaking down the computation into a sequence of elementary operations and applying the chain rule.

Comparison of Methods

Method	Advantages	Drawbacks
Manual	- Can be highly efficient if simplified well - Good for understanding	- Prone to human error - Time-consuming and tedious - Must be re-derived if the model changes
Numerical	- Easy to implement - Works on any function (black box)	- It's an approximation, not exact - Suffers from floating-point precision issues - Computationally very expensive
Symbolic	- Provides an exact analytical expression - High precision	- Can lead to "expression swell" (very large formulas) - Can be computationally inefficient to evaluate - Requires the whole expression upfront
Automatic	- Exact derivatives (up to machine precision) - Computationally efficient (often comparable to the original function evaluation) - Implemented in all modern ML frameworks	- Can have a higher memory footprint - Implementation can be complex (but we use libraries!)

Table: A comparison of the four main differentiation methods.

The Problems with Numerical Differentiation I

Numerical differentiation approximates a derivative using the finite difference formula, inspired by the definition of a derivative:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for a small } h > 0$$

1. High Computational Cost

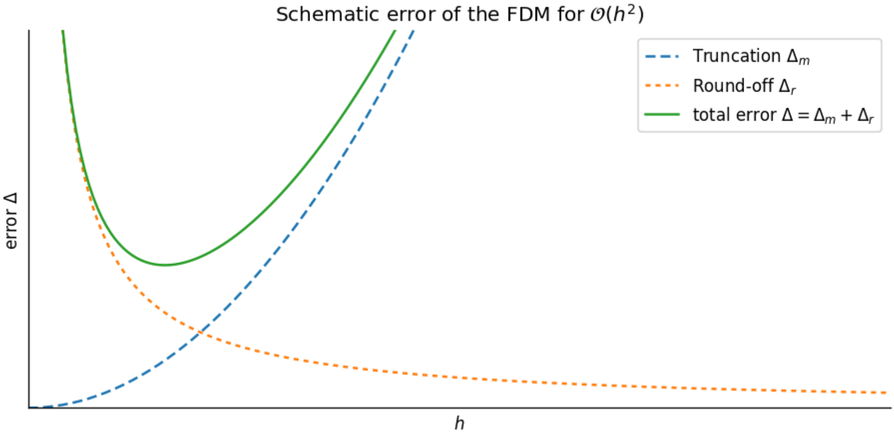
- To compute the gradient ∇f of a function $f : \mathbb{R}^n \to \mathbb{R}$, we need to compute a partial derivative for each of the n input variables.
- This requires $n + 1$ **function evaluations**. For a deep learning model with millions of parameters (n), this is completely infeasible.
- For the **Jacobian** of a function $f : \mathbb{R}^n \to \mathbb{R}^m$, we would need to compute n partial derivatives for each of the m outputs. This would cost $n \times m$ **function evaluations**!

The Problems with Numerical Differentiation II

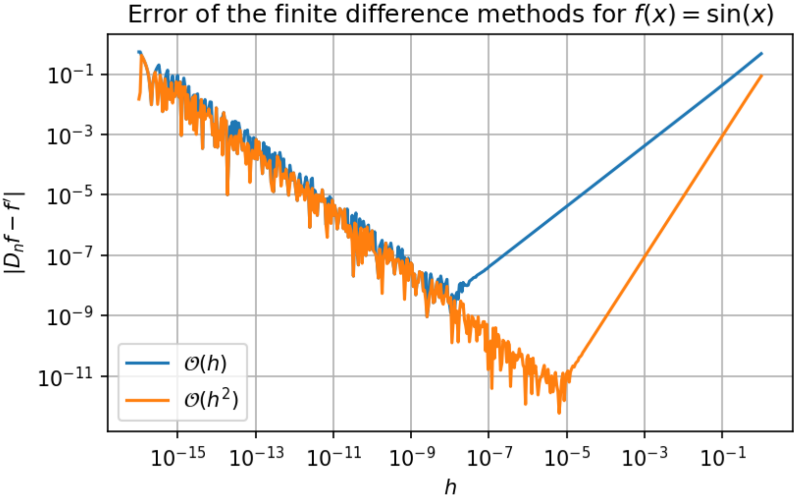
2. Floating Point Arithmetic Issues

- There is a fundamental trade-off in choosing the step size h:
  - **Truncation Error:** From the Taylor series approximation, this error is proportional to h ($O(h)$). A smaller h reduces this error.
  - **Round-off Error:** If h is too small, $f(x+h)$ becomes very close to $f(x)$. Since computers have finite precision, the subtraction $f(x+h) - f(x)$ can lose significant precision, leading to a large error that is proportional to $1/h$.
- Finding the optimal h that balances these two errors is difficult and problem-dependent.

Round-off and Truncation errors I



Round-off and Truncation errors II



▸ Link

What Automatic Differentiation is NOT

It's crucial to understand the distinctions because the name can be misleading.

AD is NOT Numerical Differentiation

AD does not approximate the derivative with finite differences. It computes the derivative value **exactly** (up to machine precision) by propagating derivative values, not by evaluating the function at perturbed points. There is no step-size h to worry about.

AD is NOT Symbolic Differentiation

AD does not build up a giant symbolic expression for the derivative and then evaluate it. Instead, it works with **concrete numerical values**. It breaks down a complex function into a sequence of elementary operations (like $+$, -, *, $/$, sin, exp) and applies the chain rule step-by-step to the intermediate numerical results. This avoids the "expression swell" problem of symbolic methods.

The Wengert List: A Formal Trace

A function $f : \mathbb{R}^n \to \mathbb{R}^m$ can be decomposed into a formal sequence of elementary operations, known as a **Wengert list** or evaluation trace. This list is constructed using a set of intermediate variables v_i. We distinguish three types of variables based on their role in the computation:

Input Variables These are the independent variables of the function. They initialize the list.

$$v_{i-n} = x_i, \quad \text{for } i = 1, \dots, n$$

Intermediate Variables These are the results of each elementary operation, which depend only on previously computed variables.

$$v_i = \phi_i(v_j, v_k, \dots), \quad \text{where } j, k, \dots < i$$

Output Variables These are the dependent variables, which are simply the last m intermediate variables calculated.

$$y_{m-i} = v_{l-i}, \quad \text{for } i = m-1, \dots, 0$$

(where l is the index of the last intermediate variable).

Decomposing Functions: A Wengert List Example

Any complex function can be broken down into a sequence of elementary operations. This sequence is called a **Wengert list** or an evaluation trace.

Let's consider the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2$.

Its Wengert list can be written as:

Trace	**Description**
$v_{-1} = x_1$	Input 1
$v_0 = x_2$	Input 2
$v_1 = \ln(v_{-1})$	Natural log of input 1
$v_2 = v_{-1} \times v_0$	Product of inputs
$v_3 = v_1 + v_2$	Sum of intermediate results
$y = v_3$	Final output

This list provides a step-by-step recipe for evaluating the function.

From List to Graph: The Computational Graph

The Wengert list directly defines a **Directed Acyclic Graph (DAG)**, where nodes are variables (v_i) and edges show dependencies.

For $f(x_1, x_2) = \ln(x_1) + x_1 x_2$:

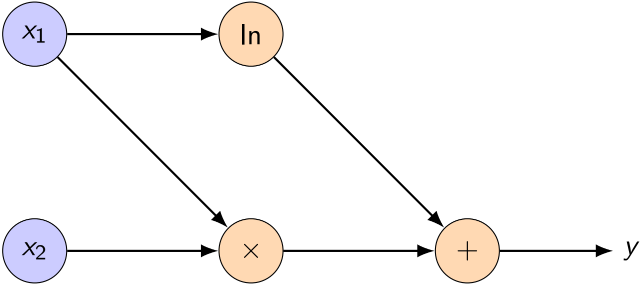
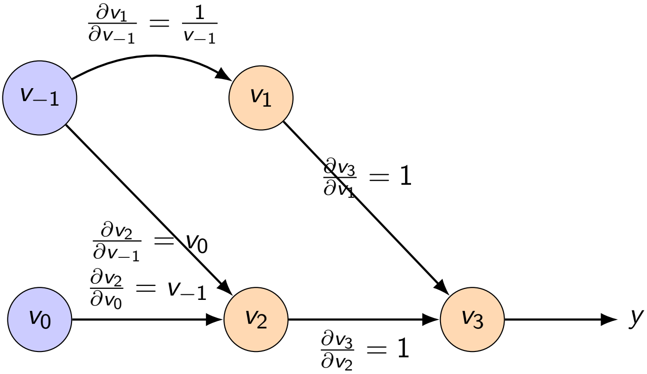


Figure: Computational graph showing dependencies for each operation.

Each edge in this graph represents a simple, local relationship that is easy to differentiate.

Derivatives on the Edges

The power of the graph representation is that each edge corresponds to a simple partial derivative.


$\frac{\partial v_1}{\partial v_{-1}} = \frac{1}{v_{-1}}$
v_{-1}
v_1
$\frac{\partial v_3}{\partial v_1} = 1$
$\frac{\partial v_2}{\partial v_{-1}} = v_0$
$\frac{\partial v_2}{\partial v_0} = v_{-1}$
v_0
v_2
v_3
y
$\frac{\partial v_3}{\partial v_2} = 1$

Automatic Differentiation (AD) works by combining these simple local derivatives using the chain rule.

Forward Mode Automatic Differentiation

Forward mode propagates derivative values **forward** through the graph. We compute $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$ at each step.

Let's find $\frac{\partial f}{\partial x_1}$ at $(x_1, x_2) = (2, 5)$. We seed with $\dot{x}_1 = 1$ and $\dot{x}_2 = 0$.

Primal Trace	**Value**	**Tangent Trace** $(\dot{v}_i = \frac{\partial v_i}{\partial x_1})$
$v_{-1} = x_1$	2	$\dot{v}_{-1} = \dot{x}_1 = 1$
$v_0 = x_2$	5	$\dot{v}_0 = \dot{x}_2 = 0$
$v_1 = \ln(v_{-1})$	$\ln(2)$	$\dot{v}_1 = \dot{v}_{-1}/v_{-1} = 1/2$
$v_2 = v_{-1} \times v_0$	10	$\dot{v}_2 = \dot{v}_{-1} v_0 + v_{-1} \dot{v}_0 = 1 \cdot 5 + 2 \cdot 0 = 5$
$v_3 = v_1 + v_2$	$\ln(2) + 10$	$\dot{v}_3 = \dot{v}_1 + \dot{v}_2 = 0.5 + 5 = 5.5$

The result is $\frac{\partial f}{\partial x_1}(2, 5) = 5.5$. To find $\frac{\partial f}{\partial x_2}$, we would need another pass.

Reverse Mode AD (Backpropagation)

Reverse mode propagates derivatives **backward** from the output. It is more efficient for getting the full gradient of a scalar function.
We compute $\bar{v}_i = \frac{\partial y}{\partial v_i}$, starting with $\bar{y} = 1$.

Core Idea

Reverse Mode computes the derivative of one output with respect to ALL inputs in a single backward pass.

Forward vs. Reverse Mode: Which to Choose?

The choice between forward and reverse mode depends entirely on the dimensions of your function, i.e., the number of inputs (n) and outputs (m).

Aspect	**Forward Mode**	**Reverse Mode**
Computes	One column of Jacobian ($J_f e_k$) per pass	One row of Jacobian ($e_k^T J_f$) per pass
Cost for Full Jacobian	$O(n) \times \text{cost}(f)$	$O(m) \times \text{cost}(f)$
Best For	Tall Jacobians $\mathbf{n \ll m}$	Wide Jacobians $\mathbf{n \gg m}$
Example Use Case	Finding tangent of a curve in 3D space. ($f : \mathbb{R}^1 \to \mathbb{R}^3$)	**Machine Learning**: Differentiating a scalar loss ($m = 1$) w.r.t. millions of parameters (n).

The Machine Learning Rule of Thumb

Since we almost always differentiate a **scalar loss function** ($m = 1$) with respect to a **large number of parameters** ($n \gg 1$), **Reverse Mode (Backpropagation) is the default choice**.

Computational Complexity and Memory

Let $ops(f)$ be the number of operations to compute $f : \mathbb{R}^n \to \mathbb{R}^m$. The cost to compute the full Jacobian is:

Mode	**Number of Operations for Jacobian**
Forward	$\approx n \times c \cdot ops(f)$
Reverse	$\approx m \times c \cdot ops(f)$

Where c is a small constant, typically $c \in [2, 3]$.

The Drawback of Reverse Mode: Memory

- Reverse mode must store the entire computational graph (or a "tape" of all intermediate variables) from the forward pass to use during the backward pass.
- The memory requirement grows in proportion to the number of operations in the function.
- For very large models (like in deep learning), this can be a significant challenge, requiring techniques like checkpointing to manage memory.

Matrix-Free Products

Often we don't need the full Jacobian, but its product with a vector. AD computes these "matrix-free" products very efficiently.

Forward Mode: Jacobian-Vector Product ($J_f v$)

- We can compute the product of the Jacobian J_f with a vector $v \in \mathbb{R}^n$ in a **single forward pass**.
- We simply "seed" the derivative inputs with the vector v: set $\dot{x} = v$.
- The resulting derivative output vector $\dot{y}$ is the desired product: $\dot{y} = J_f v$.

Reverse Mode: Transposed-Jacobian-Vector Product ($J_f^T v$)

- We can compute the product of the transposed Jacobian J_f^T with a vector $v \in \mathbb{R}^m$ in a **single reverse pass**.
- We "seed" the derivative outputs with the vector v: set $\bar{y} = v$.
- The resulting derivative input vector $\bar{x}$ is the desired product: $\bar{x} = J_f^T v$.

Computing the Hessian Matrix

The Hessian matrix H_f contains all second-order partial derivatives. It's the Jacobian of the gradient function: $H_f = J(\nabla f)$.

Computing the full Hessian is possible but often expensive. For $f : \mathbb{R}^n \to \mathbb{R}$:

- We can apply Reverse Mode to the gradient computation. Since the gradient is a function from $\mathbb{R}^n \to \mathbb{R}^n$, this would take n reverse passes.
- The total cost is $O(n^2)$, which is infeasible for large n.

The Efficient Approach: Hessian-Vector Products

In optimization, we often don't need the full Hessian, but its product with a vector, $H_f v$. This can be computed efficiently by combining forward and reverse modes.

Hessian-Vector Products: The Efficient Way

We can compute $H_f v$ with a cost similar to computing just the gradient. This is often called the "reverse-on-forward" trick.

Consider a new function $g(x) = \nabla f(x) \cdot v$. This is the directional derivative of f along v. The gradient of this new function, $\nabla g(x)$, is our target:

$$\nabla g(x) = \nabla(\nabla f(x) \cdot v) = H_f v$$

We compute this in two steps:

Step 1: Forward Pass Compute the value of $g(x)$ using a **single forward pass** of AD on the original function $f(x)$. We seed the pass with our vector v: set $\dot{x} = v$. The scalar result is $g(x)$.

Step 2: Reverse Pass Compute the gradient of the function defined by Step 1 using a **single reverse pass**. The result of this pass is $\nabla g(x)$, which is exactly $H_f v$.

This powerful technique is the backbone of many modern second-order optimization methods.

Reverse Mode: The Full Gradient in One Pass

Let's compute the full gradient of $f(x_1, x_2) = \ln(x_1) + x_1 x_2$ at $(2, 5)$.

Phase 1: Forward Pass (Evaluate and store values)

- $v_{-1} = 2,\ v_0 = 5,\ v_1 = \ln(2),\ v_2 = 10,\ v_3 = \ln(2) + 10.$

Phase 2: Reverse Pass (Propagate adjoints $\bar{v}_i$)

Reverse Step	**Adjoint Calculation** ($\bar{v}_i = \frac{\partial y}{\partial v_i}$)
Initialize output: $\bar{v}_3 = \bar{y} = 1$	Initializes all $\bar{v}_i = 0$ except for the output.
$v_3 = v_1 + v_2$	$\bar{v}_1 = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = 1 \cdot 1 = 1$ $\bar{v}_2 = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1 \cdot 1 = 1$
$v_2 = v_{-1} \times v_0$	$\bar{v}_{-1} \mathrel{+}= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = 1 \cdot v_0 = 5$ $\bar{v}_0 \mathrel{+}= \bar{v}_2 \frac{\partial v_2}{\partial v_0} = 1 \cdot v_{-1} = 2$
$v_1 = \ln(v_{-1})$	$\bar{v}_{-1} \mathrel{+}= \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = 1 \cdot (1/v_{-1}) = 0.5$
Final Gradients	$\frac{\partial f}{\partial x_1} = \bar{v}_{-1} = 5 + 0.5 = 5.5$ $\frac{\partial f}{\partial x_2} = \bar{v}_0 = 2$

A Tool for Forward Mode: Dual Numbers

Forward mode can be implemented elegantly using **dual numbers**.
A dual number is an expression $z = a + b\epsilon$, where $a, b \in \mathbb{R}$ and ϵ is a symbol with the property:

$$\epsilon^2 = 0, \quad (\epsilon \neq 0)$$

The connection to derivatives comes from the Taylor series expansion:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon + \frac{f''(x)}{2!}\epsilon^2 + \dots$$

Since $\epsilon^2 = 0$, all higher terms vanish, leaving the fundamental identity:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon$$

This means we can compute $f(x)$ and $f'(x)$ simultaneously.

Dual Numbers in Action: A Complete Example

Let's compute $f(x)$ and $f'(x)$ for $f(x) = \frac{x^2}{\cos(x)}$ at $x = \pi$.

1. **Seed the input**: $x = \pi + 1\epsilon$.
2. **Compute Numerator**: $x^2 = (\pi + \epsilon)^2 = \pi^2 + 2\pi\epsilon + \epsilon^2 = \pi^2 + 2\pi\epsilon$.
3. **Compute Denominator**:
   $\cos(\pi + \epsilon) = \cos(\pi) - \sin(\pi)\epsilon = -1 - (0)\epsilon = -1$.
4. **Perform Division**: $\frac{\pi^2+2\pi\epsilon}{-1}$

$$\begin{aligned} &= (\pi^2 + 2\pi\epsilon) \times (-1) \\ &= -\pi^2 - 2\pi\epsilon \end{aligned}$$

5. **Extract Results**: The final dual number is $-\pi^2 - 2\pi\epsilon$.
   - The real part is the function value: $f(\pi) = -\pi^2$.
   - The dual part is the derivative value: $f'(\pi) = -2\pi$.

Dual Number Properties (with Proofs)

Sum: $(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$

- *Proof*: Follows from re-grouping terms. Corresponds to $(f + g)' = f' + g'$.

Product: $(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon$

- *Proof*: $(a + b\epsilon)(c + d\epsilon) = ac + ad\epsilon + bc\epsilon + bd\epsilon^2 = ac + (ad + bc)\epsilon$. Corresponds to the product rule $(fg)' = f'g + fg'$.

Composite Function $(h(x) = g(f(x)))$:

- *Proof*:

$$\begin{aligned} h(x + \epsilon) &= g(f(x + \epsilon)) \\ &= g(f(x) + f'(x)\epsilon) \\ &= g(f(x)) + g'(f(x)) \cdot f'(x)\epsilon \\ &= h(x) + h'(x)\epsilon \end{aligned}$$

This automatically implements the chain rule.

The Meaning of Epsilon: A Matrix Representation

ϵ is not a "small number"; it is an abstract algebraic object. We can give it a concrete meaning using matrices.

We can represent the dual number $a + b\epsilon$ as the 2×2 matrix:

$$a + b\epsilon \quad \iff \quad \begin{pmatrix} a & b \\ 0 & a \end{pmatrix}$$

This is equivalent to $a\mathbf{I} + b\mathbf{E}$, where $\mathbf{I}$ is the identity matrix and $\mathbf{E}$ is:

$$\mathbf{E} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Let's check the property $\mathbf{E}^2 = \mathbf{0}$:

$$\mathbf{E}^2 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

The matrix $\mathbf{E}$ is not the zero matrix, but its square is. This provides a perfect model for the algebra of dual numbers.

Extensions: Higher-Order and Multivariable Derivatives

Higher-Order Derivatives via Truncated Taylor Series

- We can extend the dual number idea to compute higher-order derivatives.
- Consider an algebra where $\epsilon^3 = 0$ but $\epsilon^2 \neq 0$. The Taylor expansion becomes:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon + \frac{f''(x)}{2!}\epsilon^2$$

- We can represent a number as a triplet (a_0, a_1, a_2) corresponding to $a_0 + a_1\epsilon + a_2\epsilon^2$.
- Seeding an input as $x \to (x, 1, 0)$ and evaluating f with the appropriate arithmetic rules yields a result $(f(x), f'(x), f''(x)/2)$.
- This allows computing the value, first, and second derivative all in one forward pass, at the cost of more complex arithmetic.

Functions of More Variables

- To get the gradient of $f(x_1, \ldots, x_n)$, we perform n forward passes.
- In pass k, we seed the inputs as $x_k + 1\epsilon$ and $x_j + 0\epsilon$ for $j \neq k$.
- The dual part of the final result will be $\frac{\partial f}{\partial x_k}$.

Examples I

Example 1: Second Derivative with $\epsilon^3 = 0$ Find $f(2)$, $f'(2)$, and $f''(2)$ for $f(x) = x^3$.
Let $x = 2 + 1\epsilon + 0\epsilon^2$.

$$\begin{aligned} x^2 &= (2+\epsilon)^2 = 4 + 4\epsilon + \epsilon^2 \\ x^3 &= x \cdot x^2 = (2+\epsilon)(4 + 4\epsilon + \epsilon^2) \\ &= 8 + 8\epsilon + 2\epsilon^2 + 4\epsilon + 4\epsilon^2 + \epsilon^3 \\ &= 8 + 12\epsilon + 6\epsilon^2 \end{aligned}$$

Comparing this to $f(x) + f'(x)\epsilon + \frac{f''(x)}{2}\epsilon^2$:

- $f(2) = 8$
- $f'(2) = 12$
- $f''(2)/2 = 6 \implies f''(2) = 12$

Examples II

Example 2: Gradient Find the gradient of $f(x_1, x_2) = x_1 \cos(x_2)$ at $(2, \pi)$.

- **Pass 1 ($\partial/\partial x_1$)**: Seed $(2 + 1\epsilon, \pi + 0\epsilon)$.
  $f = (2 + \epsilon)\cos(\pi) = (2 + \epsilon)(-1) = -2 - \epsilon$. Result $\implies \frac{\partial f}{\partial x_1} = -1$.
- **Pass 2 ($\partial/\partial x_2$)**: Seed $(2 + 0\epsilon, \pi + 1\epsilon)$.
  $f = (2)\cos(\pi + \epsilon) = 2(\cos(\pi) - \sin(\pi)\epsilon) = 2(-1 - 0\epsilon) = -2$. Result $\implies \frac{\partial f}{\partial x_2} = 0$.

The gradient is $(-1, 0)$.